INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

ISTNanosat-1 Heart

Processing and digital communications unit

João André Henriques Ferreira

Dissertation submitted to obtain the Master Degree in

Communication Networks Engineering

Jury

Chairman:	Prof. Doutor Paulo Jorge Pires Ferreira
Supervisor:	Prof. Doutor Rui Manuel Rodrigues Rocha
Co-Supervisor:	Prof. Doutor Moisés Simões Piedade
Members:	Prof. Doutor Mário Serafim dos Santos Nunes

October 2012

Acknowledgments

First of all, I would like to thank Professor Doutor Rui Manuel Rocha and Professor Doutor Moisés Simões Piedade for giving me the opportunity to join the ISTNanosat project. The constant supervision from Professor Rui Rocha with his scientific background on embedded systems design and networking architectures allied with Professor Moisés Piedade knowledge on electronics constraints and expected behaviours, were very useful during all project phases.

I would also like to thank all the AMSAT-CT engineers for bringing me a lot of technical information on space related technology through very interesting discussions. I also want to highlight my colleagues on GEMS group, specially the Ph.D candidate José Catela for his useful knowledge and availability helping me on one of the implementation board programming and its details, sometimes until late evening.

To all my course colleagues for the healthy acquaintanceship even during the long hard work nights, specially to Bruno Henriques for his valuable friendship since my first hour at IST.

I'd like to thank my girlfriend Susana Rodrigues for her support and patience specially during the stress periods and to my grandparents for all the attention and affection. Finally, I want to express my deep thankfulness and gratitude to my parents for the unconditional support since the beginning. Thank you very much for your valued endeavour.

Abstract

The ISTNanosat-1 is a double CubeSat which is being developed by students and teachers from IST/TULisbon. This nano-satellite is composed by different subsystems that are fitted together into the flight module. This flight module carry all the required technology to maximize the spacecraft mission lifetime and provide the required logistics to the intended scientific mission aboard. The scientific experiment is responsible to make some measurements related to the Flyby Anomaly phenomenon and will take 1U (10cm cube) of free cargo space.

The Heart unit, whose software was developed in this dissertation, presents a solution to manage the remote interactions with the Ground Station through the space-link. This unit is also responsible for satellite housekeeping. The solution comprises two different subsystems: Digital communications and Command & Data Handling.

The digital communication subsystem implements the required network functionalities e.g. allowing a ground operator to invoke remote commands in a reliable way onto a specific subsystem or in the satellite as a whole. This subsystem also provides the satellite general health status information (telemetry) periodically in the downlink taking into account maximum Ground Stations compatibility. Since the ISTNanosat will carry a tiny camera aboard, the Communications solution also implements a transport protocol that enable the imagery transmission from the spacecraft, abstracting all the details on this process. All the network functionalities were implemented taking into account the intrinsic characteristics typically found on Low Earth Orbit space-links such as: intermittent and disruptive connections; low and very asymmetric throughputs.

The Command & Data Handling is a critical subsystem due to its responsibilities as the on-board information orchestrator. To enhance its operation correctness it relies on a Real-Time Operating System to implement the satellite housekeeping and internal communications tasks when it enters in the safe-mode profile.

Finally, the Heart unit solution was deployed using a AT91RM9200 and the moteISTs5++ as prototype platforms. The final solution was submitted to a set of performance and quality tests highlighting the solution compact design, low power consumption and ARM processor under-utilization even in stressful cases.

Keywords

ISTNanosat, Cubesat, embedded systems, space communications, redundant subsystems

Resumo

O ISTNanosat-1 é um CubeSat duplo que tem vindo a ser desenvolvido por alunos e professores no IST. Este nano-satélite é composto por vários sub-sistemas instalados no módulo de voo. Este módulo transporta toda a tecnologia necessária para maximizar o tempo da missão e disponibilizar a logística necessária à missão cientifica. A experiência cientifica é responsável por fazer algumas medições relacionadas com o fenómeno anormal *Flyby* e ocupará 1U (cubo de 10cm) de espaço útil dentro do satélite.

A unidade *Heart* cujo software foi desenvolvido nesta dissertação, apresenta uma solução para gestão das interações remotas com a estação terrestre através do *link* espacial. Esta unidade é também responsável pela gestão interna do satélite. A solução engloba dois sub-sistemas: Comunicações digitais e sistema de comando e tratamento de dados.

O sub-sistema de comunicação digital implementa as funcionalidades de rede necessárias e.g. permitir ao operador na Terra invocar comandos remotamente de uma forma fiável num sub-sistema especifico ou no satélite em geral. Este sub-sistema também disponibiliza informação sobre o estado de operação do satélite (telemetria) periodicamente no *downlink*, tendo em conta a máxima compatibilidade com as estações terrestres. Como o ISTNanosat irá transportar uma pequena câmera a bordo, o sub-sistema de comunicações implementa um protocolo de transporte que permite a transmissão de imagens do satélite, abstraindo todos os detalhes deste processo. Todas as funcionalidades de rede foram implementadas tendo em atenção as características das ligações tipicamente encontradas em *Low Earth Orbit*, tais como: ligações intermitentes e com grande perturbação; *throughputs* baixos e muito assimétricos.

O sub-sistema de comando e tratamento de dados é considerado crítico devido às suas responsabilidades como orquestrador principal da informação a bordo. Para melhorar a correcção da sua operação este sub-sistema utiliza um sistema operativo de tempo real para implementar as tarefas de gestão do satélite e comunicações internas quando este entra em modo de segurança.

Finalmente, a solução foi ainda instalada sobre as plataformas de prototipagem AT91RM9200 e moteISTs5++. Esta foi ainda sujeita a um conjunto de testes de desempenho e qualidade que destacam o seu design compacto, baixo consumo energético e sub-utilização do processador ARM mesmo sobre grande stress.

Palavras Chave

ISTNanosat, Cubesat, sistemas embebidos, comunicações espaciais, sub-sistemas redundantes

Contents

List of Figures

List of Tables

List of Acronyms

ADC Analog-to-Digital Converter

ADCS Attitude Determination and Control System

ADU Application Data Units

APRS Automatic Packet Reporting System

ARM Advanced RISC Machine

ARP Address Resolution Protocol

ARQ Automatic Repeat reQuest

AX.25 Amateur X.25

BER Bit Error Rate

BP Bundle Protocol

C&DH Command & Data Handling

CAN Controller Area Network

CCSDS Consultative Committee for Space Data Systems

CDMA Code Division Multiple Access

CDS CubeSat Design Specification

CEPT European Conference of Postal and Telecommunications Administrations

CIGS Copper indium gallium (di)selenide

CL Convergence Layer

COM Communications

COTS Commercial Off-The-Shelf

CRC CubeSat Reconfigurable Computer

CSP CubeSat Space Protocol

CW Continuous Wave

DAC Digital-to-Analog Converter

DET Direct Energy Transfer

DICE Dynamic Ionosphere Cubesat Experiment

DLSAP Data-Link Service Access Point

DMC Disaster Monitoring Constellation

DTN Delay Tolerant Network

EABI Embedded-Application Binary Interface

EBI External Bus Interface

EID Endpoint Identifier

EPS Electrical Power System

EXT2 second EXTended filesystem

FCC Federal Communications Commission

FCS Frame-Check Sequence

FDD Frequency-Division Duplexing

FDMA Frequency Division Multiple Access

FEC Forward Error Correction

FPGA Field-Programmable Gate Array

GaAs Gallium Arsenide

GPS Global Positioning System

GS Ground Station

GSN Ground Station Network

HAL Hardware Abstraction Layer

HDLC High-Level Data Link Control

HMAC Hash-based Message Authentication Code

I^{2}C Inter-Integrated Circuit

IDE Integrated Development Environment

IEC International Electrotechnical Commission

IPC Inter-process communication

ISR Interrupt Service Routine

ISL Inter-Satellite Links

ISM Industrial, Scientific and Medical

IPN InterPlaNetary

ITU-R ITU Radiocommunication Sector

JTAG Joint Test Action Group

LANL Los Alamos National Laboratory

LEO Low Earth Orbit

Li-Po Lithium-ion Polymer

LPM Low-Power Mode

LTP Licklider Transmission Protocol

LV Launch Vehicle

MAC Media Access Control

MCU Microcontroller Unit

MEMS Micro Electro-Mechanical Systems

MMU Memory Management Unit

MPPT Maximum Power Point Tracking

MSP Mixed-Signal Processors

MTU Maximum Transmission Unit

NiCd Nickel–Cadmium

OS Operating System

OSCAR Orbiting Satellite Carrying Amateur Radio

P-POD Poly Picosatellite Orbital Deployer

PID Protocol IDentifier

PN Pseudo-Noise

PPT Peak Power Tracking

PriSIS Primary Satellite Interface Software

RAM Random Access Memory

RDP Reliable Datagram Protocol

RTC Real Time Clock

RTOS Real-Time Operating System

SBC Single-board Computer

SCPS Space Communications Protocol Specifications

SCPS-FP SCPS File Transfer Protocol

SCPS-NP SCPS Network Protocol

SCPS-SP SCPS Security Protocol

SCPS-TP SCPS Transport Protocol

SDK Software Development Kit

SDR Software-Defined Radio

SIL Safety Integrity Level

SNACK Selective Negative ACKnowledgment

SPA Space Plug-and-Play Avionics

SPI Serial Peripheral Interface

SRAM Static Random-Access Memory

SRLL Simple Radio Link Layer

SSETI Student Space Exploration and Technology Initiative

SSDL Space Systems Development Laboratory

SSID Secondary Station Identifier

T-CSP Tolerant-CSP

TFTP Trivial File Transfer Protocol

TNC Terminal Node Controller

UART Universal Asynchronous Receiver/Transmitter

UTJ Ultra Triple Junction

UHF Ultra High Frequency

UI Unnumbered Information

URI Uniform Resource Identifier

USART Universal Synchronous/Asynchronous Receiver Transmitter

VHF Very High Frequency

XTEA eXtended Tiny Encryption Algorithm

1

Introduction

Space exploration is always related with large investments from governments and/or private institutions. This represents a limitation to technological advances, since the development is confined to some organizations with the required resources. In mid-1960 the Orbiting Satellite Carrying Amateur Radio (OSCAR) group was created, having as a major goal the construction and launch of amateur satellites. Two years later, the first amateur satellite called OSCAR I was launched. In 1969, the OSCAR project merged with *COMSAT Amateur Radio Club*, forming the *Radio Amateur Satellite Corporation* - also known as AMSAT - thus enabling the OSCAR 5 launch [4]. In 1981 the UO-9 - UoSAT-OSCAR 9 - or *University of Surrey's UoSAT-1* was launched. This satellite, marked the beginning of universities's mostly funded projects [4] [5]. Since then, dozens of projects have been developed within the academic community. It is estimated that, on average, 12 satellites are launched by universities per year [5].

In 1999 the CubeSat project was started. This collaborative project, initially between the *California Polytechnic State University* (Cal Poly) and the *Standford University's* Space Systems Development Laboratory (SSDL), aims at the standardization of pico-satellites design. This standardization increases the space accessibility by allowing cost reduction, development time decrease, and keeping frequent launches. A Cubesat is a cube with 10cm - (10x10x10cm) with up to 1.33kg - or 1U [6]. These cubes, can

be grouped easily in order to form larger satellites, for example 2U (10x10x20cm) or 3U (10x10x30cm). It is estimated that in 2010, 250 CubeSats were built in 1U, 2U and 3U formats [7]. The satellites developed under this specification - CubeSat Design Specification (CDS) - in addition to carry all necessary technology to its orbit operation, they can be designed to transport more specific scientific experiments. One clear example of this is the NASA GeneSat-1, which has a bacteria miniature laboratory inside with the capacity of detecting proteins - products of specific genetic activity [8]. CubeSats are commonly delivered in Low Earth Orbit (LEO), defined as 160-2000km above the Earth's surface [9].

To encourage the development of academic projects, some technical support and founding initiatives starting to appear. These initiatives, mainly focus on launch opportunities into rocket's remaining free cargo or collective launching coordination. One example of this projects is NASA ELaNa[1]. With the easy launch of very small and cheap units it is possible to start engineering nano and pico satellite constellations (also known as Swarms) where each unit is apart hundreds of kilometres. The Swarm[2] project from Student Space Exploration and Technology Initiative (SSETI) aims to deploy this concept in a real scenario.

Apart from the CubeSat community developments, other working groups have been formed in order to solve more generic problems. For example, to address the problems associated with long distance communications, keeping in mind the lack of efficiency of terrestrial protocols in such scenarios, new communication paradigms were forced to emerge; a very significant example is Delay Tolerant Network (DTN). This new communication approach, has been catapulted by the *Delay Tolerant Networking Research Group*[3]. The DTN proposed architecture is an evolution from the initial proposed InterPlaNetary (IPN) Internet architecture [10]. "The IPN is a member of a family of emerging Delay Tolerant Networks" [11]. The *Internet Society IPN Special Interest Group*[4] is responsible for IPN developments. The IPN is already included in the *NASA Mars mission program* [12].

In 2010, the ISTNanosat-1 project was born, presenting itself as a candidate to be the first Portuguese satellite entirely made inside an academic context. In this project, minimum use of Commercial Off-The-Shelf (COTS) components is planned or, in other words, preference will be given to academic developed technology. This nano-satellite [5] will be built over CubeSat specifications in a 2U structure. One Cubesat unit (1U) will be used to accommodate the Flight Module. This flight module encompass all the required spacecraft subsystems, responsible for e.g. satellite position determination and actuation, communications, energy gathering and storage. The Heart Unit is responsible for both digital communication processing and central decision logic orchestration. The Heart functionalities

[1] `http://www.nasa.gov/offices/education/centers/kennedy/technology/elana_feature.html` accessed on 15-09-2012

[2] `http://sseti.net/swarm/` accessed on 15-09-2012

[3] `http://www.dtnrg.org` accessed on 15-09-2012

[4] `http://www.ipnsig.org` accessed on 15-09-2012

[5] Satellite with a wet mass between 1 and 10 kg

are implemented across two different subsystems, the Communications (COM) and Command & Data Handling (C&DH). The ISTNanosat-1 will also carry one scientific experiment that will study the flyby anomaly[6] phenomenon.

The ISTNanosat-1 has already demonstrated the intent to integrate collaborative projects such as the QB50 project[7]. The participation on such projects will increase the ISTNanosat-1 success hypothesis since they provide, for example, launch opportunities and knowledge exchange between other universities and entities.

1.1 Motivation and Objectives

The Heart unit is responsible for ISTnanosat-1 control, operation management and digital communications handling. The whole solution encompasses two different subsystems, the Command & Data Handling (C&DH) and the Communications (COM). The C&DH subsystem is responsible for processing data gathered from on-board components, such as magnetometers, sun sensors, gyroscopes etc. acting as a critical subsystem due to its responsibilities in keeping the satellite in proper operation. The COM subsystem is responsible for digital communications through the space link (link between satellite and ground station). This subsystem should take into account the high processing requirements imposed by the required exported network functionalities.

Taking into account the large budget involved in a spatial project, it is necessary to develop very robust and reliable systems, in order to avoid jeopardizing the entire investment due to a particular component fault. With this requirement in mind, the Heart module architecture needs to be redundant being adaptable enough to allow different faulty scenarios, in order to keep the general satellite performance as good as possible, avoiding that specific faults can turn into global failures.

During the development and deployment phases, it is necessary to keep in mind the project costs and the physical tight constraints expected such as the lack of energy or cargo space aboard. Here, following, as much as possible, software development standards for critical scenarios one can enhance the developed solution final quality.

Beyond the imposed constraints to this space navigation device, it is important to endue the communications with tolerance to delay, disruptions, lack of bi-directionality or highly asymmetric links, etc.. Thus, the proposed network stack solution needs to take these peculiarities into account.

Finally, the Heart unit solution needs be tested in a quasi-realistic environment. The designed test set will provide performance results, allowing conclusions about Heart feasibility.

[6] `http://istnanosat.ist.utl.pt/index.php?n=Main.Mission` accessed on 15-09-2012
[7] `https://qb50.eu/` accessed on 15-05-2012

1.2 Contributions

The dissertation contributions can be listed as follows:

Portable COM subsystem software - The developed COM software solution can run on top of different hardware platforms. This portability was achieved by extending a meta-package system, which allows a full parametrized embedded system.

Network protocol stack extended - Since the used network stack on space-link lacks of intended link-layer protocol support, the main network protocol implementation was extended by adding a new layer two protocol driver.

New transport protocol for file transmission over space-link - In order to provide reliable imagery transmission from ISTNanosat-1 to Earth Ground Station (GS) a new simple transport protocol designated as Tolerant-CSP (T-CSP) was developed. It is used on top of the deployed network stack code to perform the required fragmentation and segment recovery functionalities.

Unified embedded communications service - The COM software solution relies in one developed onboard network application (designated as Primary Satellite Interface Software (PriSIS)) which unifies all the satellite communications and remote GS command processing. The GS software was also developed.

Compact and energy aware solution - This work presents a C&DH and COM hardware/software solution which meet the imposed power requirements. This was achieved mainly due to a large effort on overall functionality code compression.

RTOS support for C&DH prototyping board - One free Real-Time Operating System (RTOS) port with an open-source toolchain was tailored to support the C&DH development hardware platform. This was achieved by developing a new Hardware Abstraction Layer (HAL) for the used RTOS port.

1.3 Dissertation organization

This dissertation is composed of 6 chapters. The second chapter brings a summary of the state of the art on CubeSat emergent technologies and concepts.

The third chapter details the conceptual view on Heart unit architecture. It also enumerates the requirements and goals to be achieved in its design. The hardware and software design directives are also presented.

In the chapter 4, the implementation details and problems are addressed.

Chapter 5 documents the tests performed and its results are discussed and analysed. The performed tests aims the performance evaluation as well as code quality assessment.

Finally in chapter 6, the work conclusions are discussed, presenting some future work improvements.

2

State of the art

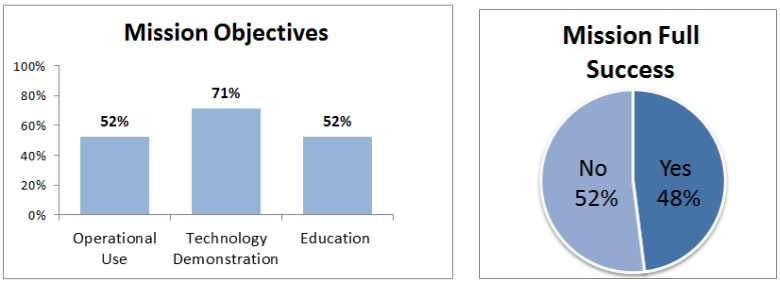
With the launch of the first artificial satellite - Sputnik I [13] - in October 4^{th}, 1957, the *space age* began [4]. Somehow influenced by this event and with prior knowledge about radio electric sciences radio amateurs launched the first amateur spacecraft in 1961. This satellite, named OSCAR I, was very successful and demonstrated that amateurs are capable of designing, building and tracking satellites. The amateur's ability to coordinate with government launch agencies and collect/process scientific and engineering related information was also proven [4].

2.1 Cubesat Generations and Missions

Since the first OSCAR was launched, many amateur satellites were built relying on different design strategies. With the help of the CubeSat project, the non-consensual design methodologies were standardised and became widely adopted by amateur satellite developers. The first CubeSat generation should have ended when the viability of this standard was proven. Projects like Delfi-C^3 or CUTE-I (OSCAR 55) are successful examples of this standardization process. After the first CubeSat generation a second one followed which focus on on-board enhanced technologies, e.g. more accurate navigation

systems or better power handling. The NASA NanoSail-D2 demonstrated the successful deployment of solar sail propulsion [14], clearly fitting it into this generation. The future generation will perhaps improve the interaction ability in large distributed systems - *swarms* - and/or open the possibility for CubeSats to integrate IPN network(s). The QB50 project, aims to be unique in establishing a space network at around 300 km altitude [15], using the CubeSat standard. This project can be considered as one of the third generation precursors.

These very small - nano and pico class - satellites are used to accomplish a wide spectrum of objectives. About 70% are used for technology demonstration although only 14% have only this as an objective. Operational use, like scientific measurements or radio communications experiments, is another common mission goal. About 52% of missions take these kind of purposes into account. More than a half (52%), are used for educational purposes. The planned mission duration is, on average, eight months and if the launch is successful, around 48% of the missions fully succeed [1]. Figure 2.1 summarises these facts.



(a) **(b)**

Figure 2.1: Mission objectives and full success rate, adapted from [1]

According to Figure 2.1(b), it is possible to conclude that pico-nano satellite missions are risky and difficult, but pursue challenging objectives. Also, the Figure 2.1(a) shows that nano-satellites can be used as new technologies test platform.

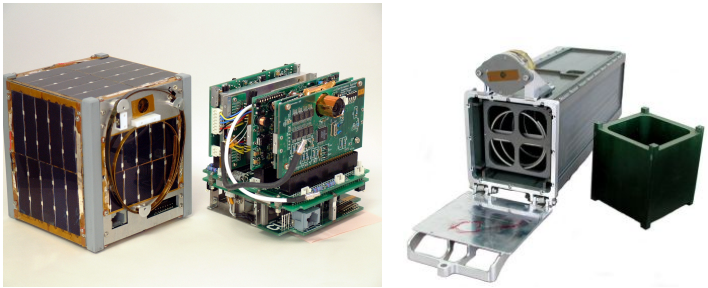
2.2 Common subsystems

CubeSats are typically structured into task independent modules called subsystems. Those are the Physical Structure, C&DH, COM, Attitude Determination and Control System (ADCS) and Electrical Power System (EPS).

2.2.1 Physical Structure

The CubeSat primary structure typically represents 15% to 20% of the total satellite mass [16]. This structure needs to be built bearing in mind the requirements described in the CubeSat Design Specification (CDS) document. Those requirements, emphasize the CubeSats design conventions, such as dimensions, mass[1], wear and electrical isolation. Suggestions for the main structure and rails material - Aluminum 7075 or 6061-T6 - are also given [6]. The Aluminium is often used due to its mass, ductility, strength, low-cost, and ease of manufacturing [16]. CubeSat developers are encouraged to comply with the Poly Picosatellite Orbital Deployer (P-POD). This structure - P-POD - is one of the CubeSat standard deploy mechanisms which acts as physical interface between the spacecraft and the Launch Vehicle (LV) [17].

Figure 2.2(a)[2] illustrates a CubeSat in/outside view and Figure 2.2(b)[3] shows the P-POD interface.



(a) **(b)**

Figure 2.2: CubeSat (a) and P-POD (b)

In order to interconnect the different subsystems inside the physical structure, the PC/104 bus is commonly used.

Besides the widely use of the single unit CubeSat form factor (1U), other geometries are used in nano/pico satellite missions. As Figure 2.3 [1] illustrates, structures based on sphere, cylinder, prism, cone, rectangular and other 1U CubeSat extensions are also adopted options, although less used.

[1] e.g. single units shall not exceed 1.33kg and triple unit CubeSats, 4.0kg

[2] Retrieved from `http://www.space.t.u-tokyo.ac.jp/cubesat/news/img/0212311.jpg` in 28-11-2011

[3] Retrieved from `http://www.pe0sat.vgnet.nl/wp-content/uploads/2011/11/P-Pod-Launcher.jpg` in 28-11-2011

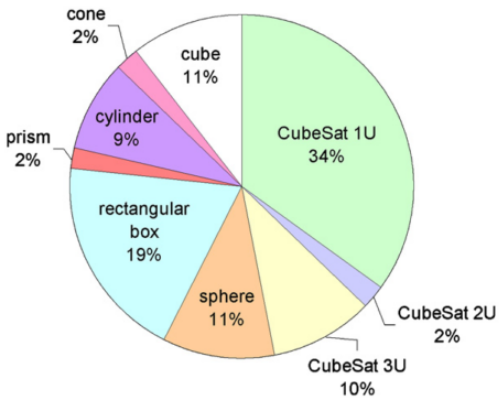


Figure 2.3: Pico-Nano satellite used form factors

2.2.2 Electrical Power System - EPS

All CubeSats require some type of electrical power management system - EPS. This system is considered critical since the most common cause of problems resides in power subsystem failures [18]. The EPS is subdivided into the following modules [19]:

Energy Harvesting - Responsible for gathering energy. Solar cells are considered the preferred power source collector due to its cost and reliability [19]. The most used type of cells is based on Gallium Arsenide (GaAs), since it provides higher conversion efficiency, up to 30%, and is widely available. About 60% of pico/nano satellite mission use GaAs, 14% use Silicon, 2% use Copper indium gallium (di)selenide (CIGS) and about 24% don't use any type of solar cell at all. [1]. As an example, in a 1U CubeSat surface (10x10x10cm) with the commercial 28.3% efficient Spectrolabs solar cells, 12.27W can be generated and in a larger form factor like 2U it is possible to gather 20.45W [20].

Energy storage - Responsible for preserving the captured energy in excess for future use. Rechargeable batteries are the most common solution to store the spare energy produced by the Energy Harvesting module. This component is vital to maintain the spacecraft operationality when solar energy radiation is not available - e.g. during an eclipse[4]. The main battery type used is based on Li-ion - 66% -, followed by Nickel–Cadmium (NiCd) - 16% and Lithium-ion Polymer (Li-Po) - 12%. The only nano-satellite known which does not carry any battery is Delfi-C^3 [1].

Power Distribution - Is responsible for providing power switching, fault detection, correction and isolation mechanisms [19].

[4]For example, when the satellite goes into Earth's shadow and stops receiving solar radiation.

Power regulation and control - The most used energy conversion methods - raw available power from solar cells to power on the spacecraft bus - are Direct Energy Transfer (DET) and Peak Power Tracking (PPT). DET is a very simple and reliable method and takes the power at a predetermined voltage point on the current-voltage (IV) characteristic of solar cells and shunts excessive power. The PPT method just follows the IV-curve from the open-circuit voltage with DC-DC converters, but can lead to problems if there is an extremely large instantaneous current surge [1].

The Maximum Power Point Tracking (MPPT) is the most elegant method, since it will retrieve the maximum power from solar cells, but it is also more complex. About 47% of missions use PPT, 46% use DET and only 7% use MPPT as conversion method [1] [21].

2.2.3 Attitude Determination and Control System - ADCS

The ADCS have two main functions, measuring and determining the spacecraft orientation - Attitude Determination - and guiding it according to a given direction - Control System [22]. Both functions are vital to maintain reliable power generation and maximize the communications performance [1] because they highly depend on proper satellite[5] directivity.

Two major families of attitude control techniques coexist: passive and active. The passive ones, take advantage of basic physical principles and/or forces occurring in the spacecraft, while the active schemes directly sense the attitude and supply a torque command to change it as required [23]. Most of the first CubeSats used passive stabilization mechanisms [16]; however today, the active solutions are becoming more interesting. About 40% of the nano-pico satellite missions use active control and almost the same amount uses passive mechanisms. Slightly more than 20% do not use any attitude control [1].

The technologies used to modify the spacecraft attitude - Attitude Control mechanisms - are summarized as follows:

Reaction Wheel - This actuator allows the satellite to change its angular momentum without using rockets or other reaction devices. The AAUsat-2 and CanX-2 launched in 2008 were the first CubeSats with this kind of technology. It can be built with DC-micro-motors and an inertia wheel [22].

Magnetorquers - Simple but with low accuracy when compared to momentum exchanged devices. It acts by producing a controllable magnetic momentum which interacts with the Earth's magnetic field to produce the mechanical torque. Magnetic torques are light, simple and low-power consuming [22].

Thruster - Used to produce torques onto the satellite with low thrust propellant or electric systems.

[5] e.g. antennas and solar panels

Its expensive, heavy and require extra space to store the propellant. Its usage is not common in nano-satellites but they are starting to gather developers attention [22].

Momentum Wheel - It is a small circular flywheel and can be used to change the satellite attitude due to the preservation of the angular momentum [24]. This actuator is mainly used for spacecraft gyroscopic stabilization[6] [22].

Spin-stabilized - The basic principle behind this reaction mechanism is routing high pressure gas into a couple of small tubes whose exhaust ports, located at the edge of the CubeSat, produce opposite tangential thrust vectors around the spacecraft spin axis [25].

Fluid Damper - This mechanism can be implemented using a toroid filled with viscous liquid containing a ball bearing. Due to spacecraft oscillations this ball is forced to travel through the liquid where the vibrational energy is dissipated by viscous friction. This method can be useful to attenuate the transients induced by deployments [26].

Three-axis Stability and Control was already used, e.g. in BeeSat, but it is technically and operationally more complex than spinners [27]. This technique can be achieved by torquing the CubeSat using three microweels or magnetic coils and a gas cylinder powered microjet [25].

Fig. 2.4 illustrates the usage of on-board attitude actuators in pico/nano satellite missions, where is possible to conclude that the most used actuators are the Passive Magnetic followed by Magnetorquers.

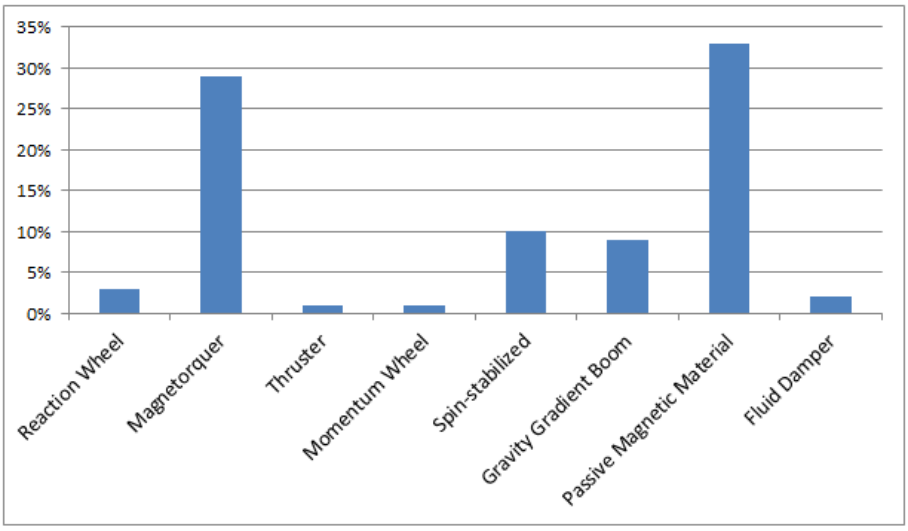


Figure 2.4: Attitude actuators usage, adapted from [1] .

The satellite orientation/position measurements can be obtained from embedded sensors, such as:

[6] Adjusting the angular or rotational movements

Sun Sensors - The most widely used mechanism in attitude determination is Sun position measurement [22]. The measurements made by these sensors are constrained by sun exposure availability. In order to maximize the sensor exposure efficiency, they tend to be integrated into the satellite solar panels. The CSTB1 is an example of this technique demonstration[7].

Earth Sensors - This mechanism helps the orientation of the spacecraft relative to Earth, where the detection of thermal radiation in the infra-red range is one of the strategies used. Another possible approach is the thermal noise detection from a radio receiver. With horizon-crossing sensors higher accuracy can be achieved, where each of the three orthogonal sensors scans a ring of space and the found crossings of the horizon [28].

Magnetometers - These simple and light sensors, measures the local magnetic field and can be found as Micro Electro-Mechanical Systems (MEMS). This sensors can be subjected to relevant errors in special conditions like sun storms [22].

Rate Gyros/Gyroscopes - These sensors measure the spacecraft angular rate relative to inertial space [23] and not the orientation itself [22]. Rate sensors are mainly used during fast attitude changes, where numerical differentiation of reference measurements are no longer reliable [28].

Star Sensors - The Star tracker is an optical device that measure the stars positions using photocells. These measurements are compared with a local database in order to determine the current attitude. Although its good accuracy level, its size and weight typically represents an issue to nanosatellites [22].

Figure 2.5 illustrates the on-board attitude sensors usage in pico/nano satellite missions. It is possible to conclude that Sun sensor and Magnetometer are the most used ones.

[7] `https://directory.eoportal.org/web/eoportal/satellite-missions/c-missions/cstb1` accessed on 02-10-2012

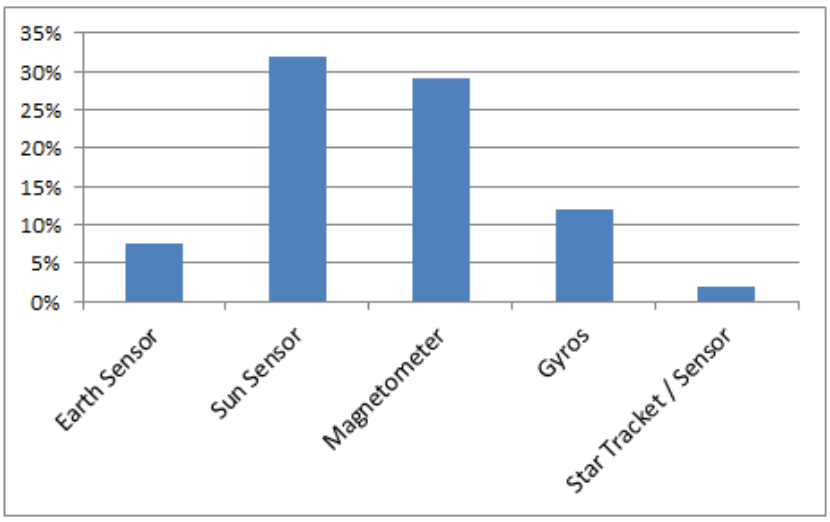


Figure 2.5: Attitude sensors usage, adapted from [1].

2.2.4 Command & Data Handling - C&DH

This subsystem is responsible for data storage (e.g. telemetry collected from on-board sensors), autonomous operations management, and other operational events [29]. The latter comprise control experiment execution and/or monitor and control of other subsystems, like ADCS, EPS, etc. [30]. The C&DH Subsystem is the spacecraft's central decision unit.

Typically the C&DH relies on a Microcontroller Unit (MCU) to process data.

In order to connect different subsystems to C&DH, interfaces like Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I^2C), Universal Synchronous/Asynchronous Receiver Transmitter (USART) [31], Space Plug-and-Play Avionics (SPA) and Controller Area Network (CAN) can be used [1]. The I^2C is one of the most used and have the following advantages: it is widely available; it implements a simple protocol and requires simple hardware; requires low power (typically 10mW) independent from the number of nodes. However, I^2C have some drawbacks such as low robustness, lack of bus protection against bus capture, and no built-in error detection and correction (only acknowledgement) [32].

Section 2.3 brings out more detailed information about the C&DH subsystem.

2.2.5 Communications - COM

The communications involved in LEO satellites (such as CubeSats) have short communication opportunity windows (up to 15 minutes) relative to ground stations in a given area (footprint). The large communication distances - from 500 to 900km [27] - are also an issue due to propagation delays involved.

The usage of store-and-forward mechanisms and Ground Station Network (GSN) such as GENSO[8] are commonly used to overcome these limitations.

Communication subsystem handles the connections with ground stations and other satellites. Information about spacecraft internal conditions, tracking guides (beacons), commands and generic information (e.g. photos), are sent through the space links. Without the ability to communicate the spacecraft becomes space junk in most cases. Hence, the mission success highly depends on the COM's reliability.

This subsystem is mainly composed by transceivers, antennas - monopole or dipole - and sometimes a MCU to handle the digital information. Both MCU and transceivers are very important parts of this subsystem. There are three main transceiver types: COTS, modified COTS and custom-build. The use of unmodified COTS simplifies the design but sometimes they require proprietary device specific protocols and modulations. They are expensive, heavy, big and have thermal dissipation problems to be used in CubeSats. In order to overcome these drawbacks some modifications (modified COTS) are made by CubeSat developers, sometimes with the help from manufacturers. The Custom-build type allows tighter requirements and functionality control, but due to difficulties in RF Board design it is less successful. The Delfi-C^3 carry its own transceiver built at transistor level [33].

Most of the CubeSats use frequencies allocated to radio amateur communication bands, but this is not restrictive. Sensitive payloads that wish to use frequencies outside the amateur bands can ask Federal Communications Commission (FCC) [16], European Conference of Postal and Telecommunications Administrations (CEPT) or ITU Radiocommunication Sector (ITU-R) for such assignment. The Industrial, Scientific and Medical (ISM) bands are another alternative to radio amateur frequencies, where the 2.4GHz band is sometimes used (e.g. GeneSat-1 or MAST [34]). The Ultra High Frequency (UHF) band (from 300 MHz to 3 GHz) with digital modulation, namely GMSK, MSK, AFSK, FSK or BPSK, is widely used with typical rates ranging from 1200 bit/s to 80 kbit/s. Inside the amateur bands, the 437 MHz is the most desirable band to ensure the downlink[9] communication. The Very High Frequency (VHF) (from 30 to 300 MHz) and S-bands (from 2 to 4 GHz) are also used. VHF allows data rates up to 9600 bit/s while S-Band may permit data-rates in the order of 256 kbit/s. About 75% of pico-nano satellites missions use UHF, $\sim$20% use VHF and $\sim$15% uses S-Band [1].

2.3 Command & Data Handling and Communication subsystems

This section will focus simultaneously on architectures of both C&DH and COM subsystems together due to its main implementation similarities.

[8] http://www.genso.org/

[9] link between satellite and the Ground Station

2.3.1 Hardware architectures

It is possible to characterize a particular subsystem by its processing architecture. The processing architectures used in CubeSats can be summarized as follows:

Microcontroller-based - This strategy relies on a MCU (or several) to assure the required on-board intelligence. A MCU is a self-contained system with a processor, memory and peripherals. In many cases it contains all that is needed to run a piece of software [35]. Vast solutions from different vendors are available. MCUs like PIC, MSP430, ARM and AVR are widely seen in CubeSat missions. For example, the Jugnu project considers two MCU, MSP430 and ARM7 core AT91SAM7x, in its setup [36].

Microprocessor-based - This approach relies on a general purpose processor, such as Intel 80C186, interconnecting external peripherals and memory [35]. The Trailblazer-2 CubeSat is an example of this design strategy where it is planned that C&DH subsystem will use a Texas Instruments OMAP3530 processor [37].

SBC-based - Single-board Computer (SBC) as the name suggests is a computer completely built on a circuit board. An example of SBC usage is the QuakeSat project. This satellite relies on Diamond Systems Prometheus SBC, which contains a low power ZFx86 CPU-on-a-chip, Ethernet, serial ports, IDE, analog-digital I/O and PC/104 support [38]. The Cool LiteRunner 2 SBC from Lippert Embedded, is another SBC option for CubeSats [39]. Probably the most common COTS manufacturer for CubeSat platforms is Pumpkin, which sells SBC solutions based on e.g. MSP430 or dsPIC MCUs.

Specific purpose - This hardware architecture is used in missions which have designed the processing unit according strictly to the mission's requirements or in order to serve very peculiar purposes. For example, the Los Alamos National Laboratory (LANL) CubeSat Reconfigurable Computer (CRC) project employs reconfigurable computing using Field-Programmable Gate Array (FPGA) [40].

Different hardware and software strategies are used in CubeSats. The major processing units used have low power consumption profiles, such as TI MSP430. These low power processing MCUs, impose a very restrictive and requirement-specific programming strategies.

Some projects cope with system faults using redundancy strategies over several subsystems, such as EPS by duplicating the battery and/or solar cell sets [41], or in C&DH for example, by giving the processing responsibility to other subsystems (e.g. Delfi-n3Xt plan implement the basic C&DH functions at COM MCU) [32].

2.3.2 Software architectures and operating systems

Besides the processing architectures described in the last section, a couple of software design strategies are possible to follow.

C programming - This approach allows the developer to build the entire solution from scratch with high hardware control. Also, since the entire software solution is coded taking into account strictly the intended specifications, the code becomes cleaner and therefore smaller and faster.

General Purpose Embedded systems - Some CubeSats use generic OS's, such as Linux, running in its MCUs. With this philosophy the code for low-level management of hardware, drivers, communication and processes can be reused/tailored. The support for modern languages, such as Python, can also be granted [42]. General purpose systems require a more robust MCU than in a clean slate approach using simple C programming.

Real Time Operating System - RTOS deals with rigid time constraints, where the processing must be done within defined periods. There are two main RTOS types, soft and hard. Soft RTOS allows task prioritization where real-time tasks are assigned to higher scheduling priority. Beyond this, Hard RTOS must guarantee that real-time tasks are computed within its acceptance time limits. Task preemption, minimal interrupt and dispatch delay must be also employed in the task's scheduler to meet the real-time tasks requirements [43]. Commercial and opensource RTOS are used in CubeSats. The most common commercial solution is Salvo[10] RTOS from Pumpkin Inc.. Other free RTOS systems such as FreeRTOS[11], Contiki[12] or ChibiOS/RT[13] can also be used.

Apart from pure C programming, proprietary and opensource RTOS for constrained environments, such as Salvo (KySat-1), FreeRTOS (PW-Sat) or µCLinux (UWE-1), are becoming very popular in CubeSat standard. Both the RTOS COTS vendors [44] and the main opensource projects, like FreeRTOS, tend to ignore software standardization like IEEE std 1003.13-1998. This standard aims the definition of Real-time environments based on ISO/IEC 9945 standards [45]. Also, it seems none of the most popular Operating Systems for CubeSats are ready for any software certification such as DO-178B[14]. These standards/certifications, even non mandatory for CubeSat projects, can be useful to avoid software design problems and control the software's life cycle. The C and assembly languages are commonly used in CubeSat software development. The use of programming languages for safety-critical environments, such as ADA[15], are very incipient.

[10] http://www.pumpkininc.com/content/salvo.htm

[11] http://www.freertos.org/

[12] http://www.contiki-os.org/

[13] http://www.chibios.org/

[14] Software Considerations in Airborne Systems and Equipment Certification

[15] Structured, statically typed, imperative, wide-spectrum, and object-oriented Programming language used in e.g. avionics, rockets, banking, etc.

2.4 Communication protocols

The digital protocols used in CubeSats are quite basic. The most widely used is the Amateur Packet Radio Link Layer Protocol (AX.25), sometimes together with some improvements such as Simple Radio Link Layer (SRLL). Network protocols such as CubeSat Space Protocol (CSP) also exist. This layer 3 protocol allows remote operation over a specific subsystem from the GS. Other protocols such as Saratoga or Delay Tolerant Network (DTN) may be a possible solution for cubesat applications. Actuality the extra overhead imposed by these protocols discourage its use in the common cubesat links.

2.4.1 Amateur X.25 (AX.25)

This protocol is based on X.25 and specifies both physical and data link layers from the OSI stack model [2]. The AX.25 is very similar to X.25 except in the address fields size (which was expanded to accommodate the nodes callsign), and the incorporation of UI frames. This type of frames brings a solution for broadcast-oriented communications such as those over a radio medium. Note that X.25 are intended to be used in point-to-point links. The link error detection and framing functions are adopted from High-Level Data Link Control (HDLC) [46]. AX.25 supports connection-oriented and connectionless modes [2].

Figure 2.6 illustrates the AX.25 functionalities mapped to correspondent OSI layers.

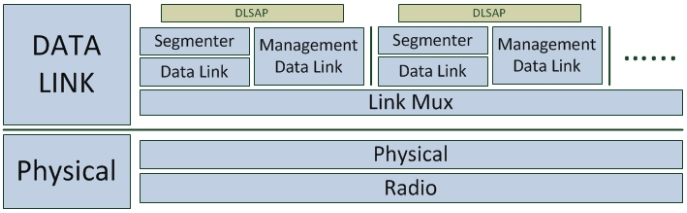


Figure 2.6: AX.25 Protocol reference, adapted from [2]

Data-Link Service Access Point (DLSAP) serves as service interface to upper layers. The Segmenter, Data Link and Management Data Link modules represent each AX.25 link, which will be multiplexed through physical link by Link Mux. The Segmenter function splits the information in small chunks if the original data exceeds the frame payload size. The Data Link component provides connection management (establish and release). The Management Data-link provides parameter negotiation ability between stations and, finally, the physical module handles the radio transmitter/receiver functionality [2].

This protocol uses three general frame formats, Information frame (I frame), shown in Fig. 2.7, Unnumbered frame (U frame) and Supervisory frame (S frame).

Flag	Address	Control	PID	Info	FCS	Flag
01111110	112/224 Bits	8/16 Bits	8 Bits	N*8 Bits	16 Bits	01111110

Figure 2.7: I frame format [2].

The address field indicates the destination/origin address, each one formed by an amateur callsign (6 ASCII characters maximum), Secondary Station Identifier (SSID) (4 bits) and another 4 reserved and control bits. This field can accommodate 2 (112 bits) or 4 (224 bits) callsigns. The 112 bits wide address field is used when the communication is done directly between source and destination peers, while the 224 bit length field is used when the communication is done using repeaters in-between [2]. If 255 bytes are considered for Maximum Transmission Unit (MTU) - maximum data inside the Info field - without using repeaters, the UI frame efficiency is $\sim$92.7%. In the design phase of this protocol, some concerns were raised about excessive frame overhead due to complete callsign transmission - 14 bytes only for source and destination addresses. This design option proved to be a good decision, because it avoids the need for centralized address assignment process and allows a straightforward way for node identification [46]. The control field indicates the frame type and the Protocol IDentifier (PID), the carried protocol. The Frame-Check Sequence (FCS) is used by the receiver to validate the frame correctness. U and S frames have the same format as the I frame, except the omission of the PID field.

The AX.25 protocol is widely used but has poor performance when facing error-prone channels [47]. This lack of performance over noisy links has special impact in these low bandwidth scenarios where retransmissions are not desirable. The FX.25 extension proposes the addition of a Forward Error Correction (FEC) to the standard AX.25 packet, allowing error correction in the receiver. The FX.25 frame structure keeps interoperability with pure AX.25 systems.

2.4.2 Simple Radio Link Layer (SRLL)

This protocol was developed to compensate the lack of error correction functionalities in AX.25. It specifies fixed length frames - 28 bytes - and has the capacity to recover up to three lost bits per frame [48]. The use of fixed size frames avoids errors in frame detection caused by bit flipping. A 32bit Pseudo-Noise (PN) code is used for frame detection [49]. Fig. 2.8 depicts the SRLL frame format.

Figure 2.8: SRLL frame format

The processed data field is composed by user data and a generated error correction code (8 bytes) based on Hamming code. This field is later subjected to an interleaving operation [49].

This protocol is used, for example, in Cute-1.7+APDI (CO-56) and Cute-1.7+APDII (CO-65) [33].

2.4.3 CubeSat Space Protocol (CSP)

The CSP[16] provides the functionalities associated to layers three and four of the OSI model [50]. Its main feature is the existence of a routed network approach in which every satellite subsystem (or the Terminal Node Controller (TNC) connected to the mission control computer located at the ground station) can be a network node. Each network node is addressed by a unique identifier, representing the location of the node[17]. The routing information is pre-programmed in the source code of each node before the deployment. In the latest CSP version the address was increased in order to allow 32 distinct addresses [51].

Fig. 2.9 represents the concept, where a unique identifier is assigned to each CubeSat subsystem (COM - 1; C&DH - 2; ADCS - 3; EPS - 4). Also, the TNC and the control computer have unique addresses. For example, if one CSP packet is received by the COM with source address CSP - 9 (TNC) and with destination address CSP - 2 (C&DH) the COM subsystem, based on this information one can conclude two important facts: The packet arrives from ground segment (9-15); and needs to be routed through the local bus (I^2C in Fig. 2.9) to some node attached to the space segment (C&DH in this case).

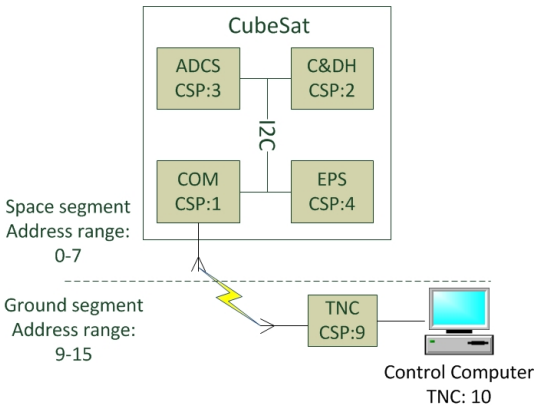


Figure 2.9: CSP routed network

The CSP implementation maintained by Gomspace, keeps both layers, Router Core (OSI layer 3) and Transport extensions (layer 4), as open-source under LGPL licence. On one hand, the Router Core layer is responsible for making routing decisions and buffer management. On the other hand, the Transport layer provides a UDP functionality extension which consists of a direct implementation of RFC 908 (Reliable Datagram Protocol (RDP)) and RFC 1151 (RDP version 2). The RDP gives the missing packet re-ordering and retransmission capabilities [51].

[16] `https://github.com/GomSpace/libcsp`

[17] e.g. space segment (address range 0-7) or ground segment (address range 9-15)

Fig. 2.10 shows the CSP packet structure.

Figure 2.10: CSP packet

The overhead imposed by the CSP header is 4 bytes and no trail is used. The options field carries information about the usage of CRC, Hash-based Message Authentication Code (HMAC) and eXtended Tiny Encryption Algorithm (XTEA) security mechanisms and finally RDP.

2.4.4 Delay Tolerant Network (DTN) architecture and Bundle Protocol (BP)

Taking into account the unsuitability of the terrestrial-oriented protocols for space communications, a DTN architecture (RFC 4838) was proposed. These protocols do not provide resilience to long end-to-end path loss, heterogeneous networks, long propagation delays, lack of directional paths or high bit error rates. The approach taken by the DTN architecture is to transport Application Data Units (ADU) into DTN generic data containers - ”bundles” - in order to be forwarded by DTN nodes. Each DTN node is uniquely identified by its Endpoint Identifier (EID), which is syntactically represented in a Uniform Resource Identifier (URI) format (described in RFC3986) [10][18]. The DTN nodes also have persistent bundle storage, e.g. disk, flash memory, etc., to allow a more robust store-and-forward mechanism against disruption caused by lack of connectivity or spacecraft problems.

The entire forwarding intelligence of this overlay network - DTN - sits on top of a wide range of possible underlying protocols. The DTN infrastructure interacts with this underlying protocols using a convergence layer, which acts like a protocol specific interface [10].

Fig. 2.11 exemplifies this layer interaction in presence of a heterogeneous environment. The connection between DTN Node1 (e.g. control computer located in ground station) and Node2 (e.g. TNC) can be, for example, IP over Ethernet while the connection between Node2 and Node3 (e.g. CubeSat) is a space link using AX.25. When the control application installed at the Node1 sends a bundle containing, for example, an attitude command to Node3 it forwards the bundle to the BP layer. The BP layer will route the bundle (based on its destination EID) to the respective next-hop (Node2 in this case) using the appropriate Convergence Layer (CL) (TCPCL). The Node2 will receive the bundle in its BP coming from the same CL as Node1. Then the Node2 BP will perform a new route decision (in this case, forward it to the Node3 via AX.25 link using the AX.25CL). If the link between Node2 and 3 becomes very disruptive the BP implementation at Node2 can, for example, automatically schedule new a retransmission.

[18] e.g. dtn://istnanosat-1.int/communications

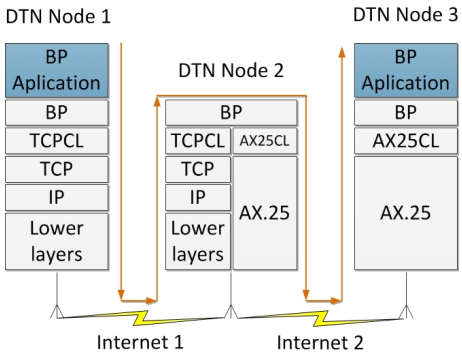


Figure 2.11: Example of DTN usage on a heterogeneous environment.

The BP brings the DTN architecture services (communication adaptability in stressed environments) by allowing [52]:

- Custody-based retransmission;
- Intermittent connectivity adaptation;
- Scheduled, predicted and opportunistic connectivity;
- Late binding of overlay endpoints identifiers to Internet addresses;

Some BP software implementations are already under development, such as DTN2[19], ION[20], Postellation[21] or IBR-DTN[22].

2.4.5 Saratoga

Saratoga is a simple UDP-based transfer protocol and was designed to transfer, in a reliable way, data over disruptive and very asymmetric connections. Since 2004, Saratoga has been used to transfer image files over IP from Disaster Monitoring Constellation (DMC) satellites to Earth. The motivation behind this protocol design is the need to fully use the downlink capacity in a short time window of communication opportunities. It implements the negative-ack Automatic Repeat reQuest (ARQ) as loss recovery method and FEC, both to ensure the transmission reliability. It can be used over IPv4 or IPv6. [3].

Each Saratoga node acts like a simple file server. Different file and directory operations are available, such as: pull - ”download”; push - ”upload”, directory listing and deletion request. Nodes announce its presence by sending a BEACON packet over reserved IPv4 multicast address. The file operations

[19] http://sourceforge.net/projects/dtn/files/DTN2/
[20] https://ion.ocp.ohiou.edu/
[21] http://postellation.viagenie.ca/
[22] http://www.ibr.cs.tu-bs.de/projects/ibr-dtn/

between nodes are made using transactions [3]. Fig. 2.12 illustrates the exchanged messages in one _get_ transaction.

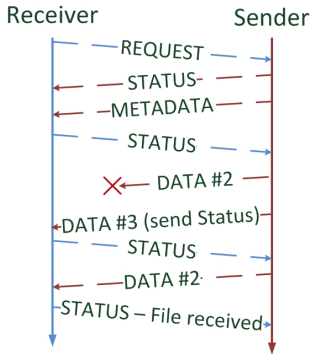


Figure 2.12: Example of saratoga transaction _get_ , adapted from [3].

Alternatively, Saratoga can be used to exchange DTN bundles between DTN nodes acting as a convergence layer (described in subsection 2.4.4). When the UDP convergence layer is used in BP, it is presumed that each bundle will fit into a single UDP packet and no reliable delivery is given. Saratoga is an option (apart from Licklider Transmission Protocol (LTP)) to give support to bundle fragmentation with reliable delivery [53].

2.4.6 Consultative Committee for Space Data Systems (CCSDS) - Standards recommendations

The CCSDS organization formulates recommendations to address generic problems found in space data systems. Standard recommendations are detailed in a so called blue book, where the main goal is to promote systems interoperability and cost reduction. Individual projects can select specific subset of features to meet their requirements [54] such as Packet telemetry or Space Communications Protocol Specifications (SCPS)[23]. On one hand, the packet telemetry recommendation describes data structures used to transport information from space vehicle to data sinks on the ground [55]. On the other hand, the SCPS is a protocol stack where the primary objective is the reliable information transfer from/to space end systems [56]. SCPS protocol stack is composed by SCPS Network Protocol (SCPS-NP), SCPS Security Protocol (SCPS-SP), SCPS Transport Protocol (SCPS-TP), and SCPS File Transfer Protocol (SCPS-FP) [57].

Some CubeSat projects, such as Dynamic Ionosphere Cubesat Experiment (DICE) or BeeSat, use the CCSDS standards.

[23] http://www.scps.org/

2.4.7 Discussion

Nowadays CubeSats tend not to use very sophisticated/complex communication protocols. Most of the CubeSat projects rely on the simple AX.25 and a Continuous Wave (CW) beacon for communications. Only a few experiments with SRLL were made (in Cute-1.7+APDI and II CubeSats). Also, some projects support the CCSDS standard. In practice, the use of IP based solutions on CubeSats are very incipient or even not used at all.

Besides some developments in CSP implementation source code, its usage in real scenarios remains immature and it is waiting for the AAUSAT3 launch to prove its design viability.

Protocols such as BP or Saratoga are only seen in larger satellites, such as UK-DMC, and represent a big challenge to CubeSat due to this standard typical limitations (available processing power, data storage, throughput, etc.) to process these exigent protocols.

Apart from the solutions for Earth-Satellite link, some research was made for Inter-Satellite Links (ISL), but the applicability for current solutions for CubeSats is not clear. For example, proposals based on modifications of the traditional IEEE 802.11 were already made [58].

Since CubeSats operates over a inherently broadcast/shared medium it is necessary to employ communication coordination to allow collision resilience and low level station addressing. The Media Access Control (MAC) strategy widely used in CubeSat scenarios is based on Frequency Division Multiple Access (FDMA) / Frequency-Division Duplexing (FDD). The FDMA/FDD technique assigns two unique frequencies to each peer, one for downlink and another for uplink. Obviously this strategy requires cooperation between all frequency assignment entities (described in Subsection 2.2.5) and the CubeSat developers to ensure the uniqueness of these frequencies. Besides the FDMA access method, the Code Division Multiple Access (CDMA) is starting to gain some attention, but the research on this technique in CubeSats is incipient.

Fig. 2.13 illustrates the protocols described in this section, according to its OSI reference.

2.5 Trends/Hot topics

Today CubeSats are becoming very interesting platforms for technology research and demonstration. One particular CubeSat project can rely on knowledge from several science fields spread among multidisciplinary working teams. A lot of research has been done in order to improve the CubeSat navigation systems and in the scientific experiments carried by these satellites.

The M-Cubed and the CINEMA are examples of cutting edge technology projects in science experiment advances. The M-Cubed (Michigan Multipurpose MiniSatellite) Project has as main goal the raise of the CubeSats technology limit in Earth imaging field. The CINEMA relies on CubeSat standard for housing a very advanced miniaturized sensors for space weather monitoring [59].

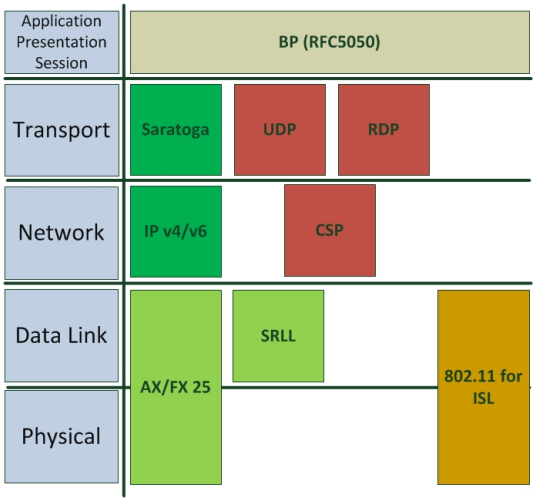


Figure 2.13: Protocol reference.

There are some projects like PhoneSat or CubeSail that aim at the enhancement of CubeSat fundamental subsystems. The NASA PhoneSat demonstrates the viability of general purpose hardware usage. This satellite relies all of its operations on a commercial Android smartphone[24] integrated into the CubeSat chassis. The CubeSail will try to be the first solar sail demonstration using a CubeSat structure [60].

2.6 Conclusion

The CubeSat platform is becoming popular in academic projects that intent to develop new solutions for spatial sciences. The advances made in all subsystems regarding the first CubeSat projects are perceptible. It is also evident that the low available power budget, strongly limits the satellite functionalities design. This limitation is related to the small satellite surface (low solar radiation exposure) and the EPS available technology - mainly in solar cells efficiency. Some vendors - e.g. ClydeSpace - are developing some new energy systems solutions, which use high efficiency cells and deployable solar panels.

In the C&DH and COM subsystems, a lot of engineering have been made, but redundant solutions are incipient. The redundant functionalities are hard to implement because the component duplication mean more power consumption and more complexity. Also, the available space inside the CubeSat limits the use of duplicated components. The size problem can be minimized using philosophies like those proposed in this document for ISTNanosat-1 Heart unit - task migrations between different subsystems.

The available throughput in CubeSat space links also imposes a big limitation to new communication protocols (such as BP) testing and development. Besides this, there is no substantial research available

[24] http://open.nasa.gov/plan/phonesat/

for ISL solutions. With the new opportunities for formation flying, such as QB50 project[25], aiming at an international cubesat network of 50 satellites, the ISL questions are becoming very interesting for CubeSats.

[25] https://qb50.eu/project.php

3

Heart Architecture

ISTNanosat-1 Heart is the satellite central intelligence unit. It is responsible for onboard data processing and handling the communications with Earth Ground Stations. It may be able to run the necessary procedures for satellite housekeeping and other functional tasks. These procedures can be, for example, image gathering, collect internal system performance information, or inject well formatted telemetry information into the radio link.

The developed solution should take into account the high robustness patterns associated with these very expensive (time and money) projects. This is important, because a simple system fault can have a high cost.

3.1 Requirements and design goals

The project requirements are described hereafter. The first five items discuss the non-functional requirements where the expected general satellite characteristics are presented. The remaining functional requirements describe some specific system details.

Hardware suitability for space environments - The hardware solution should be robust against the

expected environment up to 2000km[1] above Earth surface. For example, large thermal amplitudes may occur due to, for example, high solar radiation exposure. The Heart unit may also be prepared for unexpected power interruptions which imply, for example, automatic system boot process when the energy is re-established.

Available satellite payload capacity - The developed solution should be able to run on top of small embedded systems in order to minimize the use of scarce satellite volume and weight capacity. Therefore the Heart unit should have a light and compact design.

Energy consumption - The energy consumption of the whole Heart system is a crucial aspect. The available on-board energy is very scarce. The solution should employ strategies that allow a hardware/software compact design minimizing the required power consumption. The ISTNanosat-1 project envisages $\sim$ 900mW for both C&DH and COM subsystem.

Reliability of the developed code - The developed software should be bug free. This means that maximum compile time checks should be done as well as runtime operation validation. The runtime operation checks includes memory leaks, input validations, buffer overflows, race conditions, etc.. Also the developed applications should be fast and compact. The code solution should be well documented to allow work re-utilization from other projects/subsystems.

Heart redundancy - The on-board software solution should be flexible enough in order to raise the Hearth success hypothesis when facing such highly error prone environments. This means that the Heart Unit should employ a redundant strategy which allows as much as possible the minimization of failure impact caused by some unexpected problems, such as lack of power energy or some hardware glitch. The main philosophy of this redundant design should be in a graceful degradation basis, when more and better functionalities are offered when everything works as expected, and minor functions are available when some problems occur.

Safety-critical operation - The C&DH is the subsystem responsible for taking the principal decisions aboard, e.g. change the spacecraft attitude or change the general operational behaviour. This crucial role should be taken into account in its design.

Remote operation - The GS should be able to invoke remote tasks on the satellite in a reliable way (acknowledged). The remote tasks can be for example beacon operation toggle, or image/file transfer request. Both satellite side and GS side software should be developed.

Imagery gathering - The ISTNanosat-1 plans the use of a tiny camera on-board, installed on the ADCS subsystem. It is important to endue the COM subsystem with the required features to allow the download of the satellite stored images to the GS.

[1] Low Earth Orbit (LEO) typical maximum altitude

Satellite traceability from GS and health report - The satellite should be traceable by a wide range of GS. The most common strategy to employ this feature is periodically send a CW - Analog morse code signal. This beacon also carries the on-board telemetry encoded. This telemetry can also be transmitted using packet radio over the space-link containing the overall system status and performance.

3.2 Design specifications

The Heart unit has the following pre-defined specifications. They ensure the Heart unit compatibility and design coherence with the global ISTNanosat project, which aims a redundant approach with compatible communications between onboard subsystems.

Redundant connections - The Heart unit shall implement redundant connections between the C&DH and the ADCS subsystem. This is useful when the primary communication bus stops working unexpectedly. Therefore, the C&DH must use the secondary link to access sensor/actuator elements keeping (the best as it can) proper satellite attitude when such error occur.

Redundant processors - In order to avoid complete operation cessation due to a particular processor glitch, the Heart unit must include two processors in its design. These processors can be different in terms of characteristics.

Redundant access to space-link communications - The Heart unit design shall implement redundant connections to space-link communications. These redundant connections allow basic satellite communication even when one of the subsystems stops working.

Standard inter-subsystem protocol - The communication between subsystems must take into account the compatibility with the protocols already defined by ISTNanosat-1 project. The major protocol used for inter-subsystem communications is the I^2C.

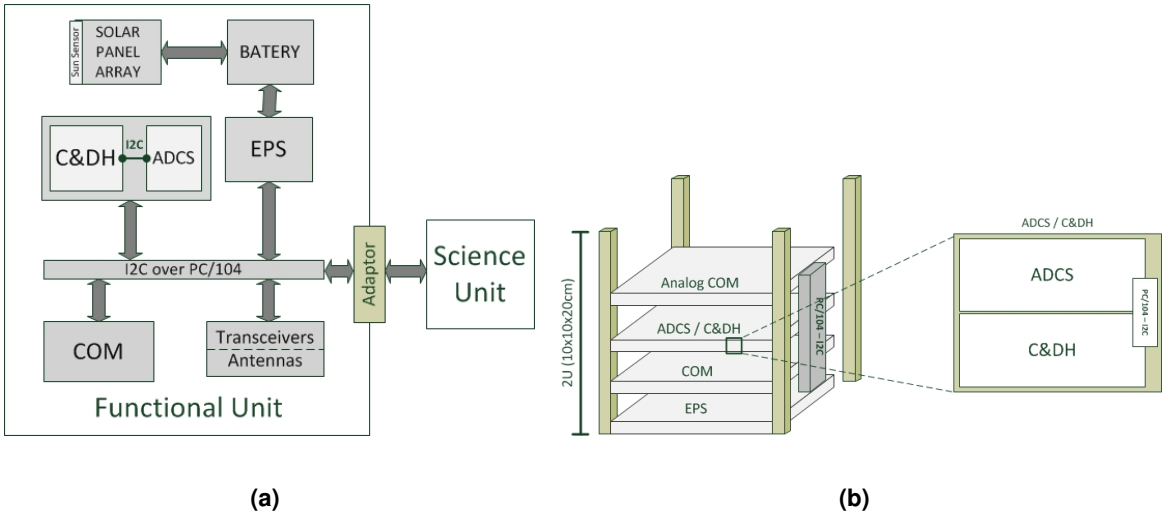
3.3 ISTNanosat-1 general architecture

This section will provide a brief overview of the foreseen ISTNanosat-1 general architecture, giving a global perspective of the Heart Unit in the satellite and its interconnections with the remaining subsystems.

The ISTNanosat-1 is composed by two single cubesats grouped together, forming a 2U structure. One cubesat (1U) will carry the Flight Module, which is responsible for accommodate all the required

technology for proper satellite operation. Taking advantage of this complete navigation system, the ISTNanosat-1 will also carry one Science Module, which will be responsible for some scientific measurements related to the Flyby Anomaly[2]. As the functional unit, the scientific unit also have 1U reserved.

The Flight Module will carry the following subsystems: Sensor/actuator - ADCS; Energy - EPS; Analog COM; C&DH and COM. The ISTNanosat-1 Heart Unit encompass two distinct subsystems: the digital communications subsystem - COM - and the central intelligence subsystem - C&DH. Fig. 3.1(a) represents the foreseen ISTNanosat-1 architecture, where the Fig. 3.1(b) shows the same architecture in 3D view.



(a) **(b)**

Figure 3.1: ISTNanosat-1 generic architecture. 3.1(a) - flat view, 3.1(b) - 3D view

As Fig. 3.1 illustrates, every subsystem is inter-connected using the I^2C protocol. This protocol is transported on top of a PC/104 mechanical bus. The I^2C protocol is used here because is an economic and efficient way for inter-subsystem communication allowing a multi-master operation. Besides this, the direct connection between the C&DH and ADCS can be used to avoid the primary communication bus. This secondary connection will allow direct sensor data acquisition and actuation interaction from C&DH, which can be useful if the primary bus (I^2C over PC/104 interface) stops working. This means that C&DH and ADCS have two redundant connections between them, one using the PC/104 system bus and another using direct connection, both with the I^2C protocol.

The ADCS subsystem will carry the following set of sensors:

Gyroscope - Responsible for spacecraft angular momentum measurements. These measurements enhances the precision of the position prediction process.

[2]More detail about the scientific experiment can be found here: `http://istnanosat.ist.utl.pt/index.php?n=Main.Mission` - accessed in 06-07-2012

Horizon sensor - The spacecraft orientation relative to Earth is very useful for example, to keep a good communication path towards the available GS. The Earth sensor will provide useful guides to maximize e.g. the communications directivity.

Magnetometer - The proprieties about the magnetic fields around and inside the spacecraft are a valuable information in the attitude determination process.

Sun-sensor - The Sun is the primary satellite energy source. Its major function is to provide reliable readings about the direction of the solar exposure. The solar sensor integration into the solar cells is still an open question.

Tiny CMOS camera - The use of an ultra small camera is under research. This camera will take pictures from the satellite viewpoint;

GPS Receiver - Like the tiny camera, the use of a Global Positioning System (GPS) receiver is under study. Some technical issues exist in common GPS receivers for terrestrial usage such as unsuitability due to its operational altitude and speed limitation (e.g. to avoid illegal ballistic missile guidance) .

For attitude control, the following set of actuators are foreseen:

Magnetorquer - The satellite attitude correction process can be aided by this actuator, which acts generating a magnetic torque in the cubesat.

Reaction Wheel - This actuator is used when a small cubesat rotation correction is required.

All the sensors and actuators are controlled by a dedicated, very low power, MCU. This interface will abstract the details associated to all ADCS functionalities e.g. Analog-to-Digital Converter (ADC) / Digital-to-Analog Converter (DAC), allowing a digital-based interaction (get or set digital data) from another subsystem.

The EPS subsystem is responsible for all energy management. It is connected to the battery pack that stores the harvested energy from the solar cells. These solar cells need a high efficiency ratio. Having the Commercial[3] Ultra Triple Junction (UTJ) cells with 28.3% on orbit efficiency, up to 135.3mW/cm^2 can be generated. All the six spacecraft faces must be coated, as much as possible, with solar cells and no deployable/external panels are predicted in the design since they are expected to increase the satellite drag, an undesirable effect that shorts the mission lifetime.

The Analog COM subsystem (refer Fig 3.2 for a more detailed view) entails the transceivers and antennas. It is responsible for signal modulation/demodulation and transmission/reception. The use of 145MHz ($\lambda \approx$ 2m) in the uplink (satellite reception) and 435MHz ($\lambda \approx$ 70cm) for downlink (satellite

[3] `http://www.spectrolab.com/DataSheets/TNJCell/utj3.pdf` accessed on 08-09-2012

transmission) frequencies in half duplex mode are currently under study. The operation on higher frequencies, such as those found in the S-Band, are also valid but this question remains under research. With the S-Band use, high bit rates can be achieved since a larger bandwidth is available.

As Fig. 3.1(b) shows, all these subsystems are stacked up together and inter-connected with a common bus. In the case of ISTNanosat this physical bus is used to support mainly the I^2C serial data.

3.4 Heart unit hardware architecture

The Heart unit architecture comprises two processor boards and the corresponding software running on top of them. This dissertation will rely on a subset of the pre-existing ISTNanosat-1 hardware conceptual architecture as reference to develop the intended software functionalities. Either the developed software and the conceptual hardware architecture are driven by three main aspects: low power, compact form factor, and redundancy. The conceptual hardware architecture is illustrated in Fig. 3.2.

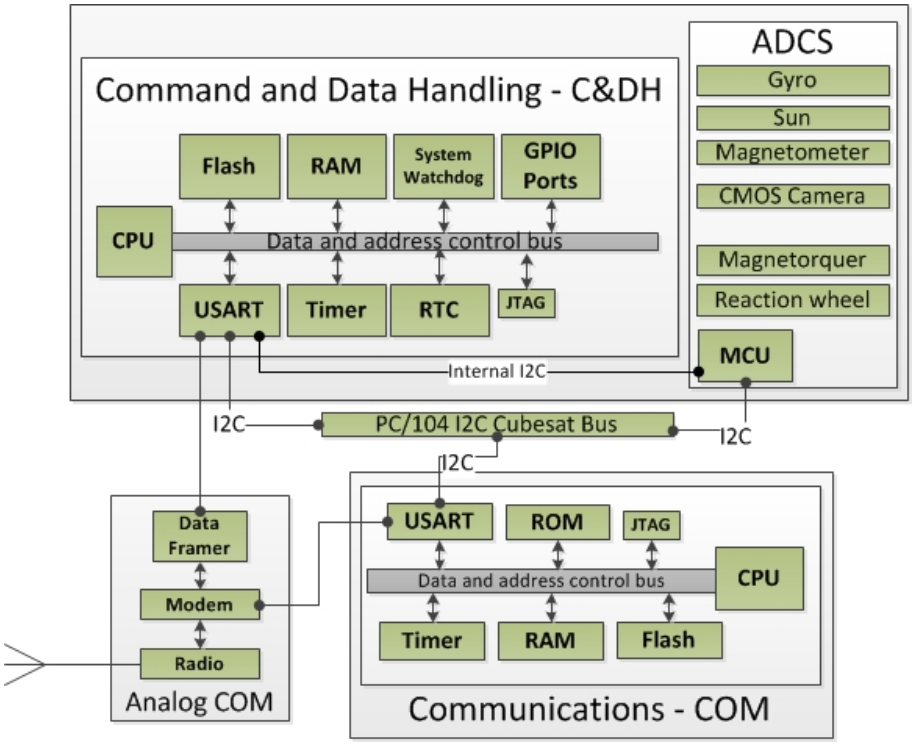


Figure 3.2: Heart Unit conceptual hardware architecture as defined in the ISTNanosat project.

The COM subsystem is responsible for processing all in/output space-link digital data and to route some information to another subsystems like the C&DH. The first design decision taken for this subsystem concerns the necessary processing power to support the demanding functionalities, such as receive the GS commands and ensure its reliability, send the telemetry beacon and allow the imagery

transmission to Earth. These functionalities require interaction with different types of interfaces and processing space-link network protocols. Fortunately, all these functionalities share a common behaviour: an heavy but sporadic energy consumption. With this in mind, from all the available power allocated to the Heart unit (950mW) the COM subsystem will have $\sim$ 80% of it (750mW). This 80% is a rough estimation based on the foreseen high demanding requirements in terms of processing power, e.g. space-link network stack processing and remote command execution. This power budget allows the utilization of a general purpose MCU. Such MCU permits software flexibility in terms of code re-utilization, e.g. from open-source community, and a more wide spectrum choice of possible OS types and operational philosophies.

The C&DH subsystem is responsible for satellite housekeeping tasks and for the safety-critical operational decisions as well as the remaining subsystems management. This subsystem must have an ultra low power consumption profile to allow basic satellite operation when facing unfavourable scenarios. For example, if EPS batteries are defective or long eclipse periods occur, the C&DH can keep the basic satellite functions: maintain attitude and telemetry transmission and simple remote command reception. The C&DH gets the remaining Heart unit power budget, $\sim$ 15% or 150mW.

Fig 3.2 shows also that besides the COM and C&DH primary connection through the system bus (I^2C over PC/104), they are also both connected to the Analog COM subsystem. This subsystem have three distinct functions. The Data Framer, with dedicated hardware and software, is responsible for formatting the serial data into a specific network protocol. The Framer also conceals from the remain subsystems the intrinsic protocol mechanisms like handshakes, signalling, retransmissions, etc.. This functional block in/output a simple serial data stream to other subsystems. The modem component translates the digital data into appropriate analog signals to be later transmitted over the air through the radio transceiver and antennas. The Data Framer and Modem blocks together are usually called Terminal Node Controller (TNC).

This Analog COM subsystem allows two type of interactions, one using the Data Framer and another using the Modem directly. The COM subsystem direct connection to the modem implies that COM has to implement the framing functionality on its own. This protocol agnostic interface is more flexible because it allows the use of any network protocol stack on the space-link. Therefore, the COM software implementation is free to use the most suitable protocol stack. When the framing operations are done by the accessing subsystem, the Data Framer can be switched off in order to save energy.

The C&DH subsystem is also connected to the Analog COM subsystem. This connection is a secondary one and was only implemented for redundancy reasons. In normal operational mode, both C&DH and COM have the necessary conditions to operate, being the COM responsible for the space-link management. In this mode, the C&DH subsystem will be free to process its satellite housekeeping tasks and interact with ADCS subsystem. Moreover, when there are no conditions to keep all the subsystems

in proper operation, the C&DH ensures a simple communication profile (only beacon transmission and GS command reception) using all the Analog COM function chain, following a low duty cycle strategy. In such scenario, known as safe-mode profile, the COM unit is switched off to minimize the overall energy consumption. In this case, the C&DH relies on one very basic Data Framer to perform the space-link data encapsulation process. This Data Framer may have a very basic protocol functions subset implemented in a dedicated hardware, thus freeing the C&DH of being overburden with this heavy task.

From the Heart Unit perspective, it is assumed that some external subsystem (e.g. EPS) will trigger the Safe-mode profile when it is needed. Table 3.1 summarizes this conceptual redundancy strategy.

Table 3.1: Operational profiles

Operational profile	C&DH functions	COM functions	Active connections
Normal	• General satellite housekeeping • On-board subsystems management	• Process and executes GS commands • Send telemetry beacon • Imagery transmission	• C&DH ↔ COM • COM ↔ Analog Modem
Safe-mode	• Same as Normal profile; • Send beacon over the space-link • Process and execute basic commands from GS	Off	• C&DH ↔ Data Frammer

Table 3.1 shows that the image transmission function is not supported in safe-mode profile, mainly because this is considered a secondary satellite functionality. Also, a simple command service is supported by the C&DH through the Data Framer for reasons that were above discussed.

Now let's take a closer look on the C&DH and COM hardware platforms that will support the software modules developed in this dissertation. The purpose of this conceptual description is not the description of the hardware development but rather to provide a guideline for software development and a starting point for the future hardware implementation.

- **COM hardware platform**

  The requirements already described for this subsystem suggests the use of a general purpose 32-bit MCU, capable of supporting off-the-shelf operating systems with a reasonable processing power and energy consumption ratio. The selected CPU architecture was the 32-bit Advanced RISC Machine (ARM) due to its power efficiency characteristics [61] and global support from many embedded systems vendors. The successful BeeSat-1 mission, with almost 3 years on orbit, carries an ARM-7 that somehow demonstrates the viability of this architecture in CubeSat missions. The ARM also incorporates a Memory Management Unit (MMU), which allows the use of popular OS such as Linux. This MMU is responsible for handling the memory access by the CPU, imple-

menting for example, virtual to physical (and vice versa) address translation, memory protection or cache control.

In what concern system memory, there is a need for non-volatile memory such as flash memory to store onboard data and volatile memory as operating system working memory.

Finally, regarding the project specifications a USART must be used to enable I^2C communications outside the COM subsystem.

- **C&DH hardware platform**

  According to the Heart unit hardware architecture, the C&DH must have a low power consumption profile. To meet this requirement, the ultra low power TI MSP430 architecture was selected. This MCU, one of the best options when low power constraints are envisaged, allows operation in different Low-Power Modes (LPMs), which brings a useful way to employ energy-aware software solutions. Also, this processor type is commonly used in CubeSat platforms, mainly through the CubeSat-kit sold by Pumpkin, a popular CubeSat COTS vendor. A lot of different well succeeded projects, such as Delfi-C^3, HawkSat-1 or e-st@r, proved the TI MSP430 suitability for this kind of demanding scenarios. Also, a lot of OS, mostly with real-time characteristics, support this architecture. This fact brings also a large OS selection versatility.

  The remaining components are basically the same as those in COM subsystem but with more restrictions. There are several operating system solutions that can easily be adapted for this architecture, a lot of them with low memory footprint[4]. This is useful because, even with small memory capacity, it is possible to reuse and tailor public available code as necessary. In what concerns the platform communications, and taking into account the Heart unit hardware conceptual architecture and specifications, three USART are needed: one for the PC/104 connection, other for the ADCS direct connection and another to access the Analog COM when the satellite is in safe-mode. Additionally, the C&DH board should have a Timer, a Real Time Clock (RTC) and a system watchdog.

3.5 Heart unit software architecture

The presented Heart architecture was designed according three main vectors: low power, ability to run in constraint hardware, and resilience against lack of resources (such as energy).

In this architecture it was assumed that some external subsystem is in charge for Heart normal or safe profile switch. It also relies on the assumption that there is a Data Framer on the Analog COM subsystem that can perform the necessary layer 2 framing when the safe-mode is running. This assumption leads

[4] Amount of required memory needed to run a specific program

to a redundant Data Framer implementation, one in the COM subsystem when normal profile is running, and another in the Analog COM subsystem when the safe-mode profile is activated. The rationale behind this function redundancy is having a versatile layer 2 implementation (over software) that can smooth the upper layers interaction in COM. The hardware implemented Data Framer will serve the C&DH when Heart runs on the safe-mode profile.

The next subsections will focus on the software architecture developed for both C&DH and COM subsystems. It starts by discussing the developed COM functionalities as well as the base system details. Then, the C&DH software details are presented.

3.5.1 COM Network protocols

Taking into account the need for a reliable remote command operation, the possibility to transmit images from the satellite, and the telemetry service, some conceptual decisions were made.

Fig. 3.3 shows the interactions between each service and the corresponding network layers. The beacon service directly uses the AX.25 layer 2 protocol, because it is a very simple service. The reliable remote command service uses the CSP protocol which is later encapsulated over AX.25 frames. To ensure the imagery transmission to GS the Tolerant-CSP (T-CSP) is used. All these services are implemented in a module called Primary Satellite Interface Software (PriSIS) which runs on top of the OS. This very important application serves as the satellite data gateway when the satellite runs on the Normal profile.

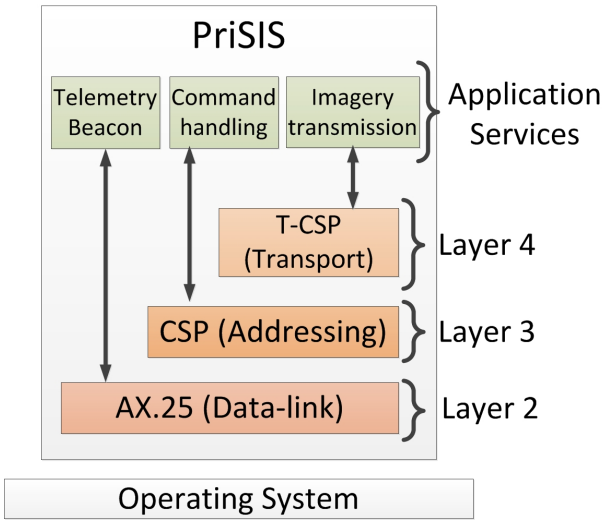


Figure 3.3: PriSIS module with each supported service.

The rationale behind this network protocol stack for these services is discussed hereafter.

- **Telemetry -** The first protocol design decision had the specific objective of maximizing the number

of potential GS on Earth that can receive the broadcast of this basic health satellite information (telemetry). To achieve this compatibility one widely deployed protocol must be used to ensure, as much as possible, the communication standardization. The protocol used for digital telemetry in downlink (satellite to GS) is AX.25, recurring exclusively to Unnumbered Information (UI) frames. This AX.25 protocol subset is conceptual simple and being the basis of the amateur packet radio service is widely supported by most of the radio amateur GS. The AX.25 -UI is also the basis of a widely used packet radio service (mostly radio amateurs) called Automatic Packet Reporting System (APRS)[5], which raises the number of individual operators that are capable of decoding AX.25 frames. The UI frames have a synchronization field in the beginning and in the end of every frame. This important characteristic avoids the framing processing errors. Another interesting fact is that this frames allow unacknowledged/asynchronous mode operation, which is useful in this very disruptive medium because they may suffer from bandwidth asymmetries (up and downlink) and high packet loss rates. The UI frame efficiency is not very high ($\approx$ 93 %) but the described intrinsic advantages makes this trade-off acceptable and the least worst option.

Other different frames are available in AX.25 protocol, but only for connected mode operation. The connected mode is undesirable for this scenario because it relies on time constraints to detect transmission errors. These timeouts can lead to link under-utilization when, for example, a node stops its transmission waiting for the error trigger mechanism. The connection mode also highly depends on both up and downlink conditions, which is still an undesirable behaviour in the presence of highly asymmetric duplex links. Finally, this mode requires extra control message exchange between nodes implying a high overhead in the space-link.

The telemetry information is directly encoded into the AX.25 UI frame payload in a human-readable format (ASCII). Every piece of telemetry data is delimited by a special character ':'. Fig. 3.4 shows the encoded telemetry packet used to transmit the COM health status. The system clock data provides a time reference for the remaining values.

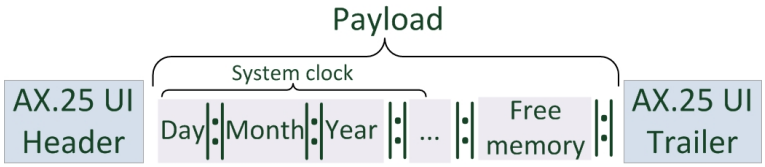


Figure 3.4: Telemetry information encoded into AX.25 UI Frame

Any kind of information with dynamic size can be encoded using this special character approach. The receiver does not need to know a priori each field length; it should only know the information format semantics. In other words, the receptor only needs to know what kind of information is

[5]System which allows real-time tactical information reporting, such as GPS coordinates, over radio networks.

inside each field. The use of this special character to separate every piece of information have little efficiency penalty, but without this and keeping the information in ASCII format, each field must have a fixed size. Another alternative is to employ an external mechanism that informs the receiver about each field length and its contents. This last alternative was discarded because it will affect the compatibility requirement.

Taking into account both viable approaches for telemetry encoding (one using the character as separator and other using fixed length fields) in the worst case scenario, i.e. using the least possible useful information in each field, the efficiency of using fixed length fields is $\approx$ 43%, while the adopted solution features $\approx$ 56%. It is also important to note the character approach has a downside since this special character cannot be transmitted as telemetry information because it is reserved.

Another aspect to be taken into account is the frame fragmentation. This implies a trade-off between much fragmentation, which affects the overall transmission efficiencies, and large MTUs that raise the probability of data corruption. Therefore, the correct choice for a MTU value should be related with the link expected Bit Error Rate (BER). Since the ISTNanosat-1 uplink/downlink channels are not yet defined, a fixed MTU of 256 bytes is used. This MTU value represents the maximum payload possible inside an UI frame.

- **Remote command reception and processing -** The remote task invocation from GS operators on Earth raises a problem since each command can have other subsystem (rather than COM) as destination. Two possible approaches can be used to address this routing functionality:

  1- To implement a command dispatcher, which looks inside the UI frame payload, parse it, and compute the required forwarding logic passing the payload content to the destination subsystem. This option seems to be not very elegant as it lacks standardization, versatility and transparency if a plug-n-play architecture is followed for ISTNanosat. It also suffer from scaling problems when multiple subsystem are added and multiple applications are installed in each subsystem.

  2- To use a small layer 3 protocol which is layer 2 technology agnostic. This network protocol also takes care of the routing to different subsystems (nodes) supporting different applications. With the use of a Layer 3 protocol, the PriSIS software can route a specific command thought the satellite internal bus, delegating the packet directly to another subsystem without processing its content. To address this functionality the CSP protocol is used. In this solution, the CSP packets are encapsulated using AX.25 UI frames. The utilization of this layer 2 protocol under CSP enhances the error resilience by forcing a maximum CSP packet length (recall that CSP have variable length packets) and providing error detection using its FCS mechanism. The CSP does not support AX.25 protocol directly. So, an extra driver had to be developed to support this solution.

Since each subsystem has more than one remote executable task, each command message is composed by the remote task unique numeric identification, the local GS counter and, optionally, some arguments. After the correct command processing by PriSIS, it sends back to the GS operator another CSP packet containing the received GS counter and the respective success/error code. In the particular case of loosing a PriSIS response, the GS operator can later issue another command asking the satellite to send a list of the previously executed remote tasks. Fig. 3.5 illustrates a typical bidirectional communication.

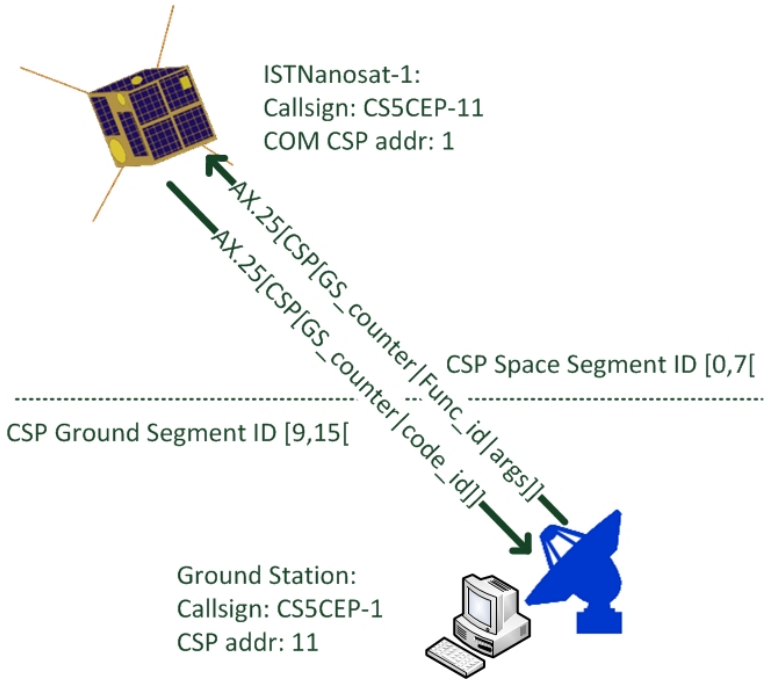


Figure 3.5: Remote command invocation using CSP over AX.25

Here, the GS operator issues a satellite command, targeting the COM subsystem identified by the CSP id 1. The GS software encapsulates the command into a CSP packet which, in turn, is encapsulated into an AX.25 UI frame and sent through the space-link. The COM subsystem replies using the same network stack with the received GS counter and its response code. This response code can be an acknowledgement or other specified code.

- **Imagery gathering -** In order to allow the correct transmission of onboard images taken by the foreseen tiny camera installed, a layer 4 protocol must be employed. The rationale behind the use of such a transport protocol is the lack of available payload size inside a CSP packet, due to the imposed AX.25 MTU restrictions. Every image with more than 253 bytes[6] needs to be segmented into multiple CSP packets. This segmentations process also needs to take into account

[6] 256 bytes AX.25 MTU value and 4 bytes CSP header overhead (recall the CSP overhead from Fig.2.10)

an efficient mechanism for missing packet retransmission, to overcome the issues imposed by a particular segment lost in the downlink. Since the CSP protocol does not have one transport layer implementation that performs fragmentation and packet recovery, a new solution called Tolerant-CSP (T-CSP) was engineered. The Tolerant prefix has to do with the underlying asynchronous transmission mode provided by the AX.25 UI logic. This UI mode is very useful here as it does not have any time-based constraints. Thus, the segment transmission process can proceed continuously without any acknowledgement (here optimistic can be synonym of tolerant). The T-CSP rely on Selective Negative ACKnowledgment (SNACK) logic to ensure packet recovery. The SNACK approach is adequate for this particular scenario because the space-link may suffer from high throughput asymmetry and delay. This asymmetry tends to be favourable (more throughput) for downlink, being important to use a parsimonious ACK mechanism to avoid the utilization of the uplink channel as much as possible.

In Fig 3.6 the T-CSP segment format is shown, where 8 bits are allocated for file identification, 16 for segment number and another 16 for total segments in the file being transmitted.

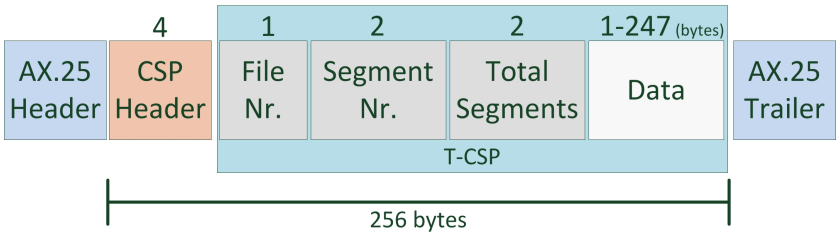


Figure 3.6: T-CSP segment over CSP packet

Therefore this structure allows up to 255 possible different files and 65535 segments per file. The ”File nr.” field in the T-CSP segment identifies the file being transported. This field is useful when, for example, a remote task asking for download all aboard images in a row is triggered. The largest file that can be transported with this mechanism can have:

$$\frac{2^{16} \times 247}{1024^2} = 15.44\ Mbytes.$$

This value is reasonable regarding the actual typical birates (9600 and 1200 bit/s) on such links. When new bitrates become available the overhead becomes less critical and the T-CSP fields can be stretched, allowing the transmission of large files.

The efficiency of this complete stack, in the best case (the 29 bytes overhead is used to transport 247 bytes payload), is $\approx 90\%$. Since the T-CSP solution is not used in critical satellite communications (only in file transmission) this efficiency ratio is not dramatic. Fig 3.7 demonstrates a successful file transmission from the PriSIS (communications handler software aboard the COM subsystem) to the GS on Earth.

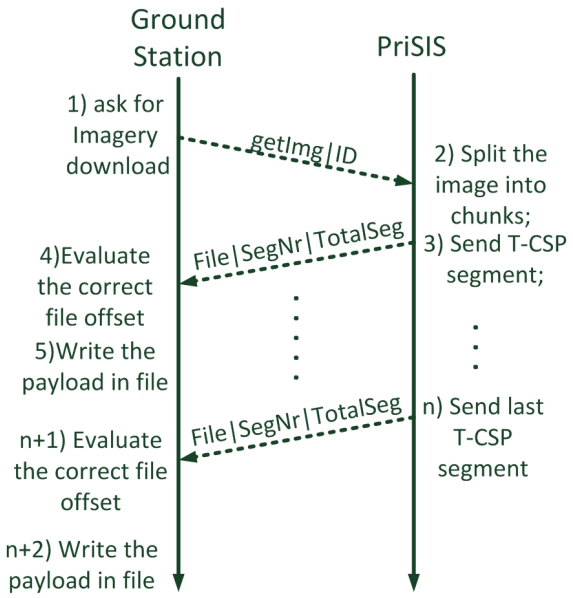


Figure 3.7: T-CSP Fragmentation mechanism

In the first step, the GS software (which implements the T-CSP protocol) sends a CSP packet in an AX.25 UI frame asking the execution of the functionality with code ”getImg” with the argument ”ID” identifying the specific image to fetch. Then, in the step 2), the PriSIS software loads the image from the local file system and splits its content into multiple chunks to be later sent one by one using the T-CSP container (step 3 to n). Every time the GS receives a T-CSP segment, it computes the correct file offset for the incoming payload based on the segment number and store the content in a proper place. Note that segments may not arrive sequentially (e.g. a segment retransmission can occur), so the receiver must know which file part is receiving. After the reception of all segments (step n+1) the GS software is able to display the saved image.

Fig. 3.8(a) gives an example of what happens when some segments are lost and 3.8(b) demonstrates some problems associated with large verification windows (later discussed).

In Fig. 3.8(a), the correct CSP packet reception in COM by PriSIS (asking for a particular image), triggers the T-CSP file transmission. Then, assuming the first segment is lost and PriSIS does not know about this failure, it continues sending every segment sequentially. When the segment 2 is correctly received by the GS software (step 2), it checks if there are x missing segments before the segment 2 (in this case check if 1 was correctly received). So, it sends a CSP packet asking for that specific lost segment (segment 1 in this case). Immediately after this, some unexpected link disruption occurs and all the next segments are lost, including segment 1 retransmission. When the GS software detects this heavy failure it triggers a timeout and reports to the operator this failure, reporting the number of segments that were correctly received and how many segments

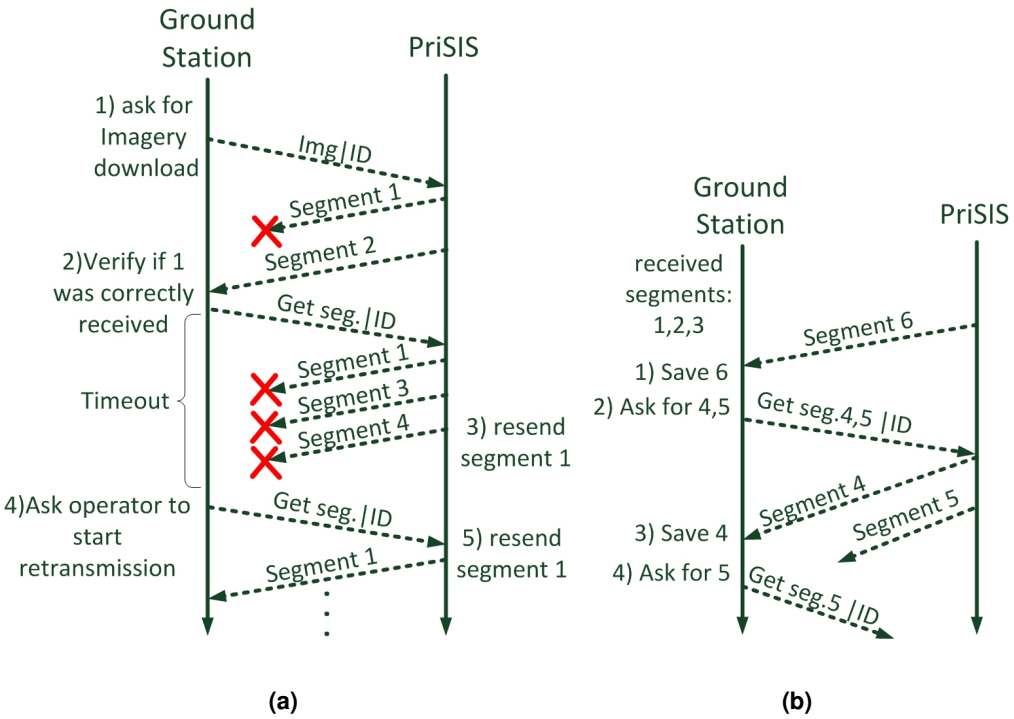


Figure 3.8: Fig 3.8(a) - T-CSP Segment retransmission mechanism; Fig. 3.8(b) - Verification window size

are missing at that particular moment. With this information, the operator have the opportunity to decide if it is good to proceed with the retransmission process now or later. For example, if the satellite contact opportunity window closed, the operator can delay the retransmission process to wait for another satellite pass, keeping the already received segments in memory. After some delay (or not) the GS issues another segment request and finally the missing segment is received. Obviously, if too many segments were lost, the operator should cancel de retransmission because this process, which ideally benefits the downlink usage, will be reverted due to high the retransmission message rate. With all segments correctly received, the GS software can now display the correct image to the operator.

The timeout value should be selected taking into account the time that one complete AX.25 UI frame takes to be transmitted. Regarding the typical bitrates, each frame takes $\approx$ 0.23 seconds at 9600 bit/s and $\approx$ 1.8 seconds at 1200 bit/s to be transmitted. The propagation delay can be ignored taking into account the distances involved in LEO.

The Fig. 3.8(b) shows how the x (back verification window) value affect the T-CSP protocol efficiency. This x value should be small (1), to avoid PriSIS memory overflow when multiple CSP packets are sent to it, asking for T-CSP segment retransmission. In the Fig.3.8(b) example, suppose that $x = 2$, and in step 1 the segments 1,2 and 3 were already received. When the segment

6 arrives, the GS software will inspect if segment 5 an 4 was correctly received. In this case, they are missing, so the GS asks PriSIS for them. As soon as segment 4 arrives (step 3) the T-CSP protocol will inspect again for the last 2 segments and sends the CSP packet for the segment 5 again (which is unnecessary in this case).

The motivation behind this tunable x parameter, as window back verification size, is to inspect how much high proactive-based approach (large windows) impacts the convergence speed and uplink utilization.

The T-CSP protocol can be also be used in the opposite way to upload e.g. configuration files to the satellite.

Finally, Fig. 3.9 resumes the final protocol architecture involved in the space-link when both Normal and Safe-mode profile are used.

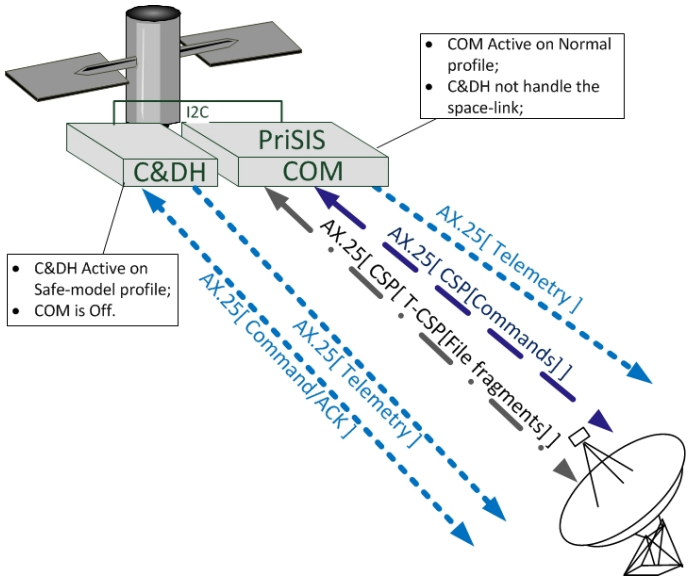


Figure 3.9: Exported C&DH and COM communication options using the available function profiles

The image shows the simple AX.25 telemetry in the downlink broadcast by COM when normal-mode is activated, and broadcast by C&DH when safe-mode is running. The remote command service provided by COM using the CSP packets on top of AX.25 UI frames is highlighted as well as the transport protocol interaction with either GS and PriSIS for image transmission. Finally, the simple command service in C&DH over a simple command/acknowledge is shown when the Heart unit enters safe-mode.

3.5.2 System software

This subsection will cover the software architecture for the underlying operating systems and auxiliary processes, e.g. boot loader. Firstly the COM base system is described followed by a discussion on the

strategy chosen for C&DH Operating System.

COM Operating System

This subsystem uses the Linux kernel as OS nucleus and GNU software as userland tools. The use of Linux speeds up the functionality development, since a lot of code can be re-used from the open-source community. Also, CubeSat developers and HAM operators are among the very active GNU/Linux on-line community which provide a good resource of experience from past CubeSat projects and documentation. The economic factor is also important when choosing Linux, since it is free and therefore it will lower the project costs. Both kernel and user-land software are split into two different files, the kernel image and the compressed root FileSystem (rootFS) image, both stored in non volatile memory (Flash). The COM OS boot loader implements the required image loading process to ensure e.g. the robustness of the kernel image before loading it. This specific OS loading process in described in Figure 3.10.

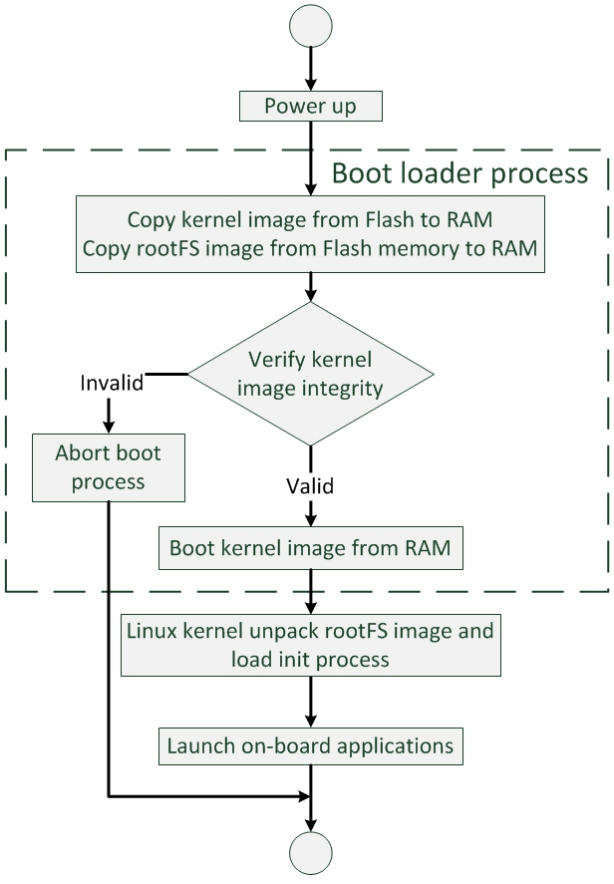


Figure 3.10: COM Operating System (OS) loading process

As soon as power is available, the boot loader will copy both images (kernel and rootFS) from Flash memory to RAM to be later used. Running entire OS from RAM allows a faster and energy efficient

data access with a minor performance penalty imposed by the initial copy time. Next, the boot process will perform the CRC calculations against the kernel image to attest the software integrity in order to avoid executing corrupted code, which can lead to unexpected satellite behaviours. This evaluation is only done against the kernel image, because it is the critical part of this entire software system. If the integrity check detect errors in the kernel image, the boot process aborts and waits for a external decision e.g. EPS energy re-initiation. This strategy avoids constant useless boot retries, which leads to energy waste. If the kernel image checks succeeds, the boot process will launch the Linux kernel, which later decompress the rootfs image and triggers the normal Linux init process. Afterwards, the entire COM base system is ready to host the on-board applications.

The Linux kernel abstracts the subsystem interaction using the generic serial and I^2C drivers. The rootFS contains the needed tools to interact with this kernel drivers from userland side.

This COM base system allows a light and compact OS solution to serve as host for the described PriSIS module.

C&DH Software

Taking into account the described requirements, specially the C&DH safety-critical tasks, it was settled that the OS aboard this subsystem will be a RTOS. A RTOS will enhance the overall system correctness enabling time-constraint task prioritization. This is important for this subsystem since it is in charge of the overall satellite management, namely receive and send information to ADCS in order to understand and react on the CubeSat attitude. The RTOS allows a strict task differentiation. For example if the OS scheduler takes two different concurrent tasks, one sending the satellite beacon (if the satellite is running on safe-mode profile) and another monitoring the satellite attitude, the higher priority task always retains processor control when it is expected no matter whether the low priority task have finished its computations. In this example, a good choice probably will be to allocate more priority to the task that will handle the attitude than to the beacon task. Furthermore, it is important that the high priority task uses the processor time always and when it is strictly expected. Since task prioritization is not enough *per si* (soft Real Time) and a tight control over each task timeline are needed, a hard RTOS will be used on this subsystem.

The RTOS tend to have very low footprints and allows close hardware programming. Both of these aspects are desirable characteristic for this subsystem. There are some important aspects to take into account when choosing an RTOS:

- Task switching delay;
- Maximum tasks supported;
- Amount memory required;
- Task priority (shared or unique) and management;
- Interrupt adaptability from external devices;

- Idle state existence or requirement.

Besides the technical questions just refereed it is also important to keep the C&DH implementation costs low and for that reason an open-source/free RTOS will be more interesting. Beyond the RTOS it is also important to choose a compatible opensource Software Development Kit (SDK). This SDK must include the toolchain - compilers, linkers, assemblers, etc. - and the Integrated Development Environment (IDE)/debugger, which allows a full opensource development chain.

The more relevant free RTOS available that actually support the TI Mixed-Signal Processors (MSP)430 MCU are Contiki, ChibiOS and FreeRTOS. None of these RTOS have a quality certification such as International Electrotechnical Commission (IEC) 61508 Safety Integrity Level (SIL). This awareness raises the attention for the software implementation and test phases. Among these RTOS solutions the FreeRTOS is used, because it has almost all the TI MSP430 families ported (which gives versatility for later implementation phase) with different toolchain's, including the opensource ones. The FreeRTOS also has the largest/active community which brings more and better/reviewed documentation and code available.

The FreeRTOS typically occupies from 6 to 10 Kbytes memory footprint, which is quite acceptable for almost all MSP430 MCUss. The FreeRTOS has a pre-emptive priority scheduler, where tasks with the same priority are served in a round robin time slicing basis. In addition, the FreeRTOS also provides the necessary Inter-process communication (IPC) mechanisms, such as queues and semaphores.

In order to meet the C&DH power requirements the FreeRTOS operates using the LPM available in MSP430 to lower the overall C&DH power consumption. When the satellite is in safe mode profile it uses the internal MSP430 USART to send/receive serial data to the Analog subsystem (more specifically to the Data Framer). A dedicated beacon task is used to send the beacon periodically and the GS command handler task which process the received commands from GS operators. As a design option, the task that processes the received remote commands have higher priority than the beacon task to ensure that commands can take effect as quickly as possible. The interconnection to the remain subsystem is also handled by a non-critical task which processes the sent and received I^2C commands.

4

Implementation

The tight Heart unit requirements and the corresponding architecture, clearly suggests that the implementation phase should employ a compact, reliable and portable software solution. These three main drives directly affect the implementation strategy used. The mindset was to try to develop, as much as possible, reusable software. The possibility to reuse public available third-party components was also taken into account. This portability on either the code developed and in the code reused enables a modular software architecture. Other subsystem developers or even other projects can also benefit from this modularity. In order to minimize the hardware requirements imposed by the produced software solution, it was necessary to rely on strategies that allow low overhead systems having, for example, the possibility to use shared and compact libraries.

4.1 Prototyping boards for software development

To enable the software development on both C&DH and COM subsystems it was necessary to choose a rapid prototyping hardware. Taking into account the Heart architecture, its requirements and specifications, two different hardware platforms were chosen, one for the COM, and another for the

C&DH units. The hardware characteristics of each board were selected so that they were not very different from the envisaged final hardware architecture.

4.1.1 COM hardware platform

The COM subsystem is built around a 32-bit ARM processor, capable of supporting the intended general purpose operating system and the required PriSIS application. An AT91RM9200 based board was used, namely the AT91RM9200-DK board. Fig. 4.1 shows the prototyping platform used.

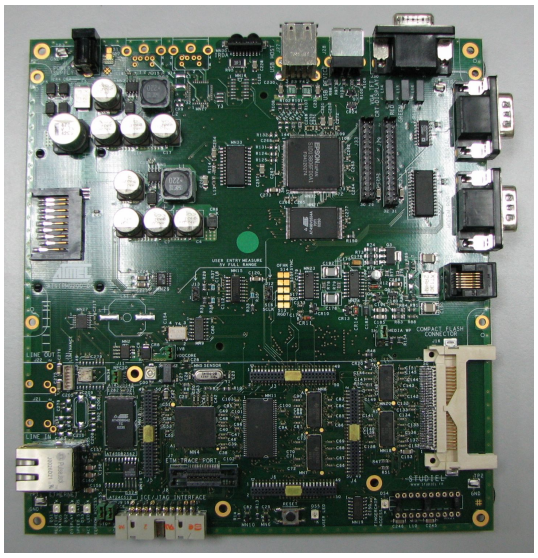


Figure 4.1: AT91RM9200 Development Kit

This Development Kit comes with a lot of unnecessary peripherals (according to the COM design approach). Therefore the relevant specifications of this board are:

- **AT91RM9200 MCU** - This Micro-controller have an ARM920T CPU core, capable of 200 MIPS at 180 MHz. Furthermore, it has 16 Kbyte of data cache and the same amount of instruction cache. In addition, it has 16 Kbytes of SRAM and 128 Kbytes of ROM memory. External Bus Interface (EBI), SPI, I^2C interfaces and generic USART ports are also available. The MCU manufacturer argues that the maximum consumption (measured running a full performance algorithm, which puts the processor at full speed) at 25ºC is 45mW (1.8V and 25mA) [62].

- **2 Mbyte parallel flash (AT49BV)** - This flash memory features a 70ns of access time. It consumes about 20mA at 3V in Active mode (60mW) and 10μA in standby.

- **8 Mbyte serial DataFlash (AT45DB)** - This flash memory operates at 2.7V and consumes 4mA in active mode (10.8mW) and 2μA in standby.

- **USART** - Serial communications interface driver with I^2C. Additionally, two DB9 interfaces are available.

The interconnected components above described result on a conceptual hardware architecture illustrated by Fig. 4.2.

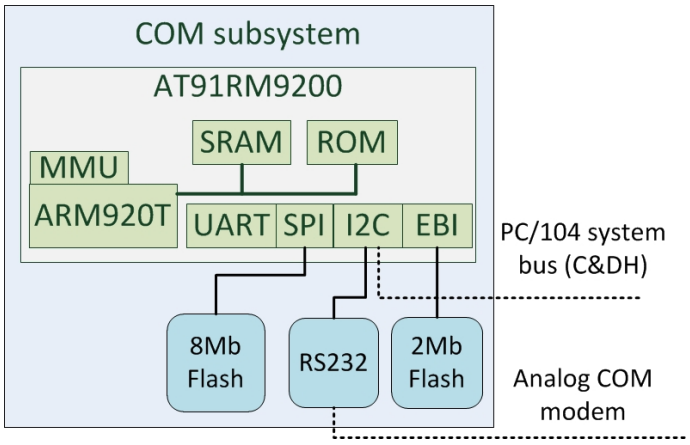


Figure 4.2: COM subsystem prototype

The entire solution depicted in Fig. 4.2 consumes $\approx$ 150mW if all the components are being used. Therefore this prototyping board meet the power requirements previously settled.

4.1.2 C&DH hardware platform

Taking into account the C&DH architecture, the hardware platform used was a moteist++s5/1011. This board, illustrated in Fig. 4.3, meets the intended hardware specifications for the C&DH subsystem.

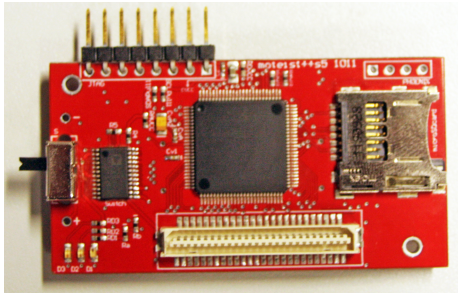


Figure 4.3: Moteist++s5/1011 top view

This platform uses as MCU the TI MSP430F5438A. This ultra-low power micro-controller relies on a 16-bit CPU running with a 25 MHz system clock. It also has 256 Kbytes of Flash memory, 16 Kbytes of SRAM and multiple external peripheral connections such as I^2C, SPI or Universal Asynchronous Receiver/Transmitter (UART). In the moteist++s5 platform, these external connection peripherals are

available through the DF9M/Hirose physical bus. The MSP430F543xA have one active mode and six software selectable Low-Power Mode (LPM) of operation. When the MCU enters on any of these LPM it can be awake back to active mode by e.g. an interrupt event. The power consumption depends on the time spent by the MCU on each LPM. In the lowest power consumption profile (LPM-0) the MCU consumes 73μA using 3V at 25ºC (0.105 mW). The MCU top consumption occurs when it enters on the active mode operating at 25MHz. In this case it spends 9.60mA at 3V (28.8mW).

According to the architecture, the C&DH should be connected to the Analog COM subsystem using a serial line. This was done by using a Crossbow MDA100 sensor board connected to the moteist++s5 DF9M/Hirose port. Since the serial line operates at 5V, in this case, the Analog COM subsystems was connected with the MDA100 through a MAXIM low-power MAX3222 RS232 transceiver to achieve the required voltage levels. The I^2C connection with the COM subsystem is also done using the exported available ports on the MDA100.

Analysing the Heart unit as a whole, with both C&DH and COM subsystems connected and regarding the previous information on each board power consumption, it is possible to conclude that this hardware setup should spend from ≈150mW to ≈180mW as maximum power consumption values. Fig. 4.4 shows the expected Heart unit power consumption as a function of the amount of time C&DH spends on either LPM-0 and Active Modes. Here, the maximum consumption of the COM subsystem was only considered because it is hard to predict its dynamic power consumption.

Time spent in LPM-0 (%)	Time spent in AM (%)	Total (mW)
0	100	180
30	70	170
100	0	150

Figure 4.4: Heart unit overall power consumption.

4.1.3 Software solution

The Heart unit is composed by two different subsystems, each of these is responsible for different functionalities. Here, we consider the different applications developed on each subsystem, focusing on the major implementation decisions taken. We start by presenting the COM operating system giving the necessary details to understand how can PriSIS run on top of a compact and clean environment. After this, the required applications and the underlying network stack implementation is discussed. Finally, a special attention is given to C&DH where an understanding on the developed real-time based

applications is provided.

Two different processor architectures are used, MSP430 in the C&DH and ARM in the COM. Since the host system processor architecture (x86) differs from both target processor architectures, all the embedded software needs to be cross-compiled. Note that the host system is where the code is produced, assembled, linked, etc. and the target system is where the resulted binaries actually run.

Anex A gives extra information on the steps required to install software in a production machine. Also some notes are given to enable its cross-compilation targeting the different subsystem platforms.

4.1.3.A COM base system

The COM operating system includes three different components: the boot-loader, the Linux Kernel and the userland environment also known as root FileSystem (rootFS). The first implementation issue was what would be the best approach capable of minimizing the hardware resources and, at the same time, enabling a versatile platform for the PriSIS module. Versatility, here, means possible third-party library code reuse and special kernel module loading e.g. Linux AX.25 native support.

Two main reasonable options are available. On one hand, it is possible to tailor and implement an out-of-the-box Linux distribution for embedded systems, such as OpenWRT; the problem with such approach is the required time that must be spent to remove all the useless pre-defined software which comes by default within this Linux distributions. This type of operating systems typically are designed to run on specific scenarios such as domestic wireless routers. Also, these distributions implement the system configuration typically through a couple of system tools such as run-time package managers. All these facts are considered undesirable for this scenario because they mean extra unnecessary run-time overhead aboard. On the other hand, it is possible to rely on simple frameworks, such as Buildroot[1], which allows complete embedded system parametrization through a set of meta-information files (e.g. Makefiles and patches) in compile time. It also provides a simple approach to new developed software integration in a portable way. With this in mind, the buildroot option seemed to be a reasonable choice for the rootFS.

The Linux kernel version used was the 2.6.38 vanilla source[2]. This version was used mainly because this is the last kernel version where it is possible to apply the AT91 support patch[3]. This patch is important because it enables/enhances the support of some of the required AT91RM9200 MCU hardware on Linux. Those features are related with e.g. DataFlash operation, I^2C and SPI bug corrections.

The Linux kernel natively supports some relevant features for PriSIS module. Therefore, the following kernel features were enabled:

[1] `http://buildroot.uclibc.org/` - accessed on 8-10-2012

[2] Unmodified kernel source from kernel.org

[3] `http://maxim.org.za/at91_26.html` - accessed in 08-09-2012

- AX.25 layer 2 protocol;
- Use ARM Embedded-Application Binary Interface (EABI) feature;
- Support for bzip2 compressed rootFS;
- Activate the I^2C, SPI and second EXTended filesystem (EXT2) support.

Probably, the most important feature here is the native AX.25 support as a loadable kernel module. This kernel module implements all the required AX.25 framing functions. To access these framing functionalities from userland, two options exist: one through a kernel created virtual interface, and another via the AX.25-library as Fig. 4.5 illustrates.

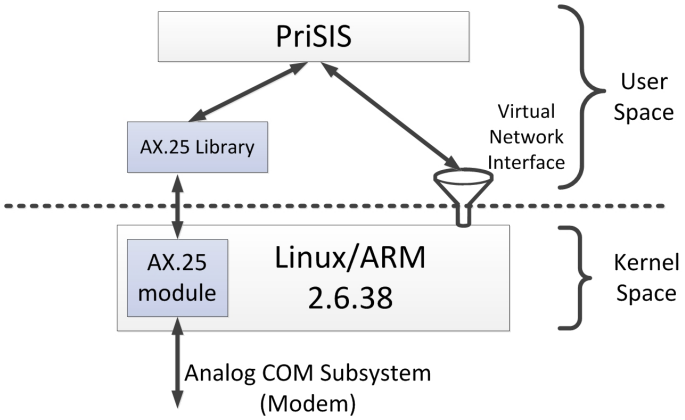


Figure 4.5: AX.25 kernel module interactions.

This kernel module was activated using the build-in strategy. This will automatically include the module into a single kernel image file, avoiding the dynamic module load process. Since this kernel image strictly has only the required functionalities, the dynamic module load process is unnecessary and it would bring unnecessary module organization care.

The support for compressed rootFS image is also enabled. With this functionality the kernel is able to load a compressed rootFS from the storage medium. This is useful as it allows extra storage space saving.

To raise the compatibility between different toolchains, both the Linux kernel and the rootFS were compiled using the Embedded-Application Binary Interface (EABI). This enables a code standard format among different objects produced by different compilers. This is important because it enhances the modularity between kernel and rootFS compiled images.

The final Linux/ARM kernel image occupies 1.55 Mbytes of flash memory storage.

At the end of a correct Linux kernel boot operation, the userland software (rootFS) is loaded. This rootFS image comprehends all the necessary software, ranging from basic system management tools, such as GNU coreutils[4], to peripheral configuration handlers, such as AX.25-tools. The buildroot infras-

[4]Basic file, shell and text manipulation utilities

tructure simplifies and automatizes the intended rootFS cross-compilation process with a high level of customization. It relies on a combination of Makefiles and patches executed in chain and activated by the required configuration.

The most important configurations for COM rootFS are the following:

- Enable the EABI interface and μClibc usage;
- Instruct the compiler (GCC) for code size optimization;
- Use shared libraries;
- Instruct for EXT2 bzip2 compressed image output;
- Ask for SPI and I^2C-tools packages installation;
- Configure the target architecture (ARM) and processor variant (920T);

The highly configurable μClibc use is important here because it is specially designed to run on embedded systems, dealing with space constraints better than the typical general purpose glibc. It also supports shared libraries, which is a good strategy for code redundancy removal, thus allowing even more space saving. Some experiments with static linked libraries were made and the impact on storage requirements were high. This means that the produced software that will run on top of this rootFS (e.g. PriSIS) will also need to be dynamic linked to avoid this waste of storage space due to library code duplication.

The I^2C-tools package will help the I^2C bus runtime debug and the setserial will enable the serial interface configuration.

Note that buildroot infrastructure will produce its own toolchain, based on μClibc which will be useful to compile the rootFS. This toolchain will be also used later to compile the COM software that is not compiled by the buildroot framework.

The EXT2 file system was chosen due to its high maturity on embedded systems, where the bzip2 compression option allows in this case 81.25% of storage saving compared with an uncompressed image.

To enable the interaction with the exported AX.25 kernel driver functionalities from userland, as described in Fig. 4.5, the AX.25 library and AX.25-tools[5] are required. Since this software is not currently automatically supported by the buildroot packages, it was necessary to extend the buildroot framework to support the AX.25 protocol by creating the required buildroot packages. These packages instruct the framework about the steps required to compile this software. A buildroot package is composed by a set of patches and meta-information. The package is later integrated into the buildroot package system to allow the required software to be compiled together with all the remaining rootFS software. In alternative to this packet method, it is possible to cross-compile the original library code outside the framework and copy the resulted binaries into the rootFS image. This is not an elegant method because it compromises

[5] `www.linux-ax25.org` - accessed on 20-08-2012

two important aspects: versatility, as every time the rootFS must be recompiled the binaries must be copied into the produced image again; also the external compilation process must take care about the correct library paths (that are installed into the rootFS image), which can be very complex to maintain when multiple dependencies are required. More important than this is the portability penalty. If the AX.25 library and tools are externally compiled and if something needs to be later changed in the underlying system (e.g the processor architecture change from ARM-9 to MIPS) then the library should reflect this changes (e.g. change the proper toolchain). Using the Buildroot packaging system the compilation is done according to the entire system specification transparently, which is a more portable, time effective and organized process.

To create the intended AX.25-library buildroot package, one mk (containing the compilation information) file and a patch was produced. Since the AX.25-library original code[6] is not directly compilable by the actual buildroot configuration, one patch had to be created. This code patch allows the `setpgrp()` function availability verification bypass on the original configure script. Since this system is intended to be single-user and no extra worry with file permissions are needed, the lack of this function that changes the process group id it is not important and therefore can be ignored. Fig. 4.6 illustrates the major steps performed by the created AX.25-library buildroot package.

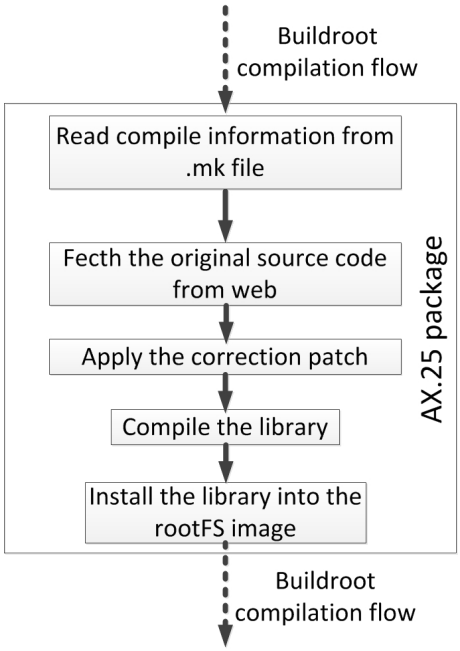


Figure 4.6: AX.25 package integrated into buildroot framework.

The same strategy was followed for the AX.25-tools software, except the patch use which was not necessary. These created buildroot packages were submitted to the buildroot project to make the soft-

[6]Original code can be found on `http://www.linux-ax25.org/` - accessed on 13-10-2012

ware widely available to all buildroot users and developers.

The resulting final rootFS image takes 1.5Mbyte of storage capacity.

Finally, the boot-loader used was the Das u-boot [7]. This boot-loader can perform the functionalities described in chapter 3, namely those depicted in Fig. 3.10. Other boot-loader alternatives exist, but since this one supports the intended design features with simple configuration, it was the one that was used.

4.1.3.B Beacon software and Primary Satellite Interface Software (PriSIS)

Here, we highlight the most important implementation strategies and decisions taken on both the beacon software and PriSIS. The discussion follows a bottom up approach in terms of network layers. Fig. 4.7 shows a complete overview of the entire PriSIS solution. This figure is an extension of Fig. 3.3 on chapter 3. The beacon software, using only the layer 2 functions is the most simple one and it is the firstly discussed. After this, the remote command implementation, which uses both layer 2 and 3 is presented. Finally, the imagery service implementation is described.

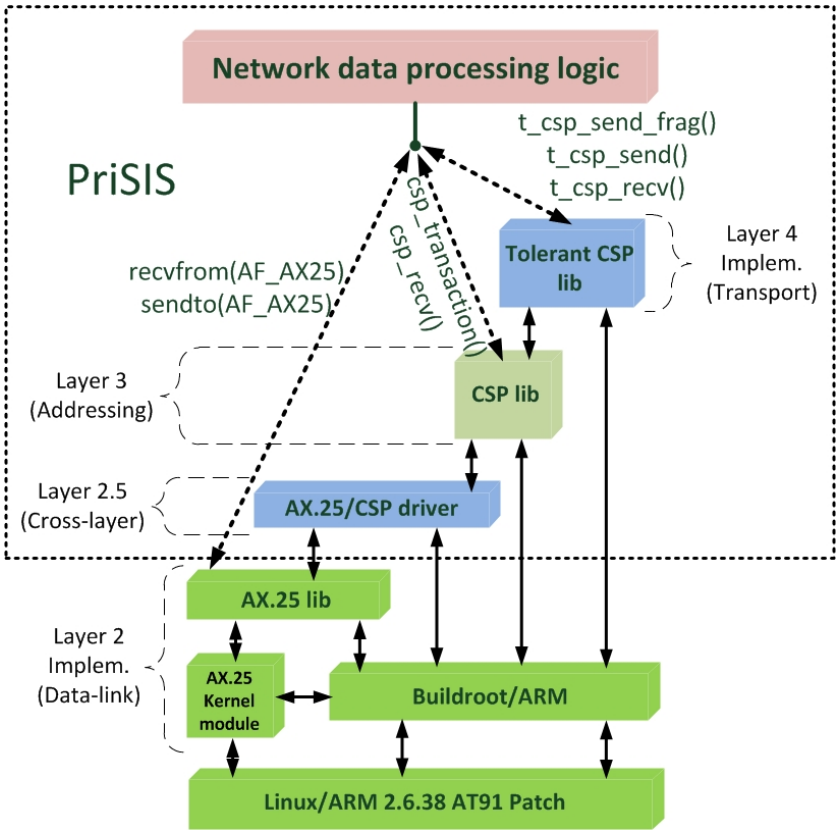


Figure 4.7: Complete network stack overview

[7] http://www.denx.de/wiki/U-Boot/ - accessed on 08-09-2012

Beacon and inter-system connections

The onboard beacon software (designated as *pulsar*) is a daemon process that injects into the downlink, thanks to the AX.25-library, the available information about the current satellite health. The pulsar daemon uses the berkeley socket standard primitives to inject the collected telemetry into the AX.25 kernel module.

1. `s = socket(AF_AX25, SOCK_DGRAM, 0);`
2. `bind(s, (struct sockaddr *)&src, slen);`
3. `message = get_telemetry_data();`
4. `sendto(s, message, strlen(message), 0, (struct sockaddr *)&dest, dlen);`

The first statement asks the kernel to create a socket file descriptor using the AX.25 protocol for a datagram. Using a (`DGRAM`) connection the kernel will send and receive the intended UI frames over this socket. The `bind()` primitive in the second step will assign the local AX.25 address (callsign) to the socket. Note that this `src` structure was previously encoded by one omitted AX.25-library primitive `ax25_aton()` for endianness compatibility. An infinite loop encloses the last two steps which are triggered every 5 seconds. This loop will finally send the telemetry data through the spacelink. The `pulsar` grabs the required telemetry information from the `/proc/` directory using the `sysinfo()` primitive declared on `sys/sysinfo.h`.

The code for receiving AX.25 UI frames follows the same structure as the sending one (using berkeley socket API). The `socket()` parameters `PF_PACKET/SOCK_PACKET` allow a raw based packet reading at the device level. This is used in the reception code because, apparently with the `AF_AX25/SOCK_DGRAM` socket configuration, the kernel do not pass to the program the received frames, maybe due to a kernel module or AX.25-library bug. After the correct payload reception from `recvfrom()` primitive, the GS application can now parse the frame payload, extracting the telemetry contents and display it to the GS operator. Fig. 4.8 shows the GS computer decoding the telemetry.

```
########## Decoded data: ##########
System date: 21/8/2012 - 14:45:30
System uptime: 0 days, 0 hours, 16 minutes
System load: last minute: 192 - last 5 minutes: 960 - last 15 minutes 704
Free memory (byte) = 16973824
################## EOD ####################
```

Figure 4.8: GS software decoding telemetry beacon

The COM also interacts with C&DH using the I^2C protocol over the PC/104 bus. On the COM side, this communication is made using the Linux kernel interfaces. This device was correctly detected by the kernel because the AT91 Linux kernel patch was previously used. The I^2C device is handled by the `linux/i2c.h` library as a generic file descriptor, relying on the kernel all the underlying

protocol specificities. There are other I^2C driver implementations. The linux native driver was used, mainly because it works as expected (meaning it is compatible with C&DH software) and is accessible by a clean and standard system interface. The COM interconnection with C&DH is done in a straightforward way, through a system file descriptor, as follows:

1. `device_file = open(filename,O_RDWR)) < 0);`
2. `ioctl(device_file,I2C_SLAVE,slave_address) ;`
3. `write(device_file,buffer,2) != 2);`
4. `read(device_file, buffer, 80) != 80);`

The first instruction opens the system device (pointed by `filename`) with read and write permissions, returning the intended file descriptor. Then the file descriptor is configured to talk with a slave node identified by `slave_address` (which is the C&DH on this case). Finally, the third instruction writes the output buffer to that descriptor and waits for the incoming expected C&DH response in instruction four.

Remote Command Service

The PriSIS implements the command reception/response and imagery gathering functionalities, using the CSP and T-CSP protocol. Since the PriSIS command/response functionalities are more critical than the telemetry, a higher scheduler priority is assigned to these functions taking advantage of the Linux soft real-time capabilities[8]. This is the best solution to ensure some scheduler process prioritization here. Other options are available, such as using the rt-preempt patch[9]. This patch changes the kernel in order to become a full Real-Time Operating System (RTOS), but unfortunately there are no support for the used kernel version. Some experiments with different patch versions were made but they prove to be very inadequate because they highly impact the overall system performance. Therefore, the use of rt-preempt patch was abandoned. Yet another solution is available, which is Xenomai[10] but it seems to be not very portable as it requires both library and system recompilation (with its own API) to work properly. Therefore, as a final solution, the PriSIS software runs with a high scheduler priority over the remaining running process without the certainty of access the processor as soon as it needs.

According to the protocol architecture, the PriSIS must implement the command reception/response mechanism over the CSP / AX.25 network stack as shown in Fig. 3.3. The CSP implementation is widely available[11] as an user-space library (libcsp) under the LGPL licence. This implementation currently does not support the AX.25 protocol. To overcome this lack of interoperability one

[8] Meaning that a running process cannot be preempted by another high priority process

[9] `https://rt.wiki.kernel.org` - accessed on 13-08-2012

[10] `http://www.xenomai.org/` - accessed on 13-08-2012

[11] `https://github.com/GomSpace/libcsp` - accessed on 19-08-2012

CSP-AX.25/UI driver was added to the csplib implementation. Fig. 4.9 depicts the details on this cross-layer driver implementation. This figure (Fig. 4.9) shows the details associated with layer 2.5 illustrated in Fig. 4.7.

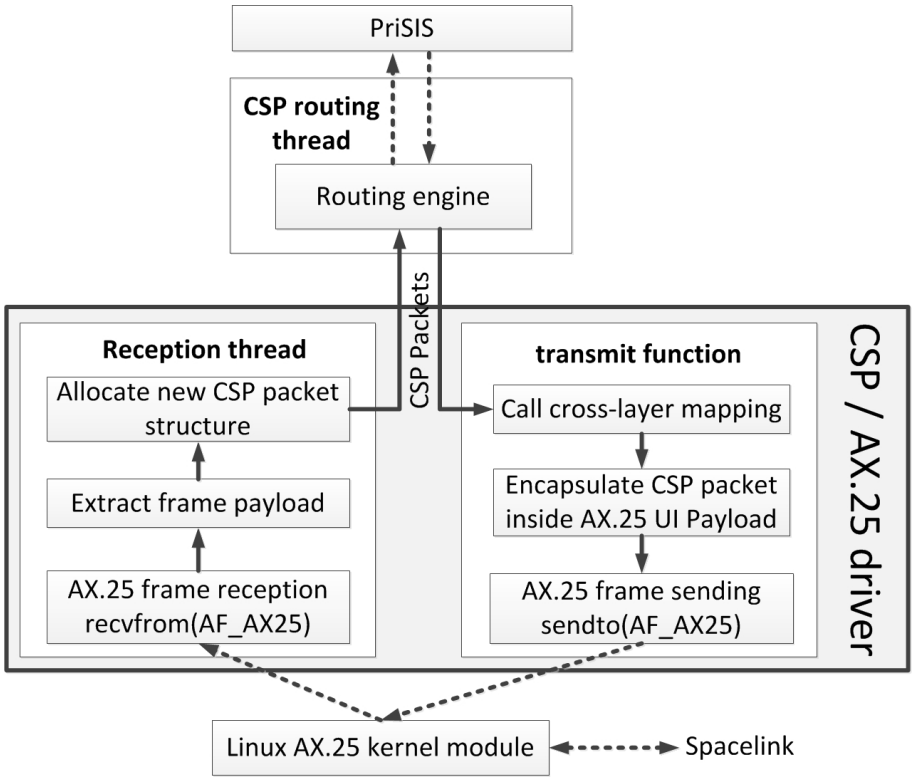


Figure 4.9: AX.25 / CSP driver implementation

When a AX.25 frame is received from the spacelink, the kernel module delivers it to the AX.25 socket (which was previously armed inside the reception thread). This reception thread reads the frame contents and extracts the CSP packet from its payload. After this, the reception thread inserts the CSP packet into a proper csplib packet structure (`csp_packet_t`) to be later processed by the router engine (implemented by csplib). When a packet is sent in the opposite way (e.g. a command response from PriSIS) the router engine delivers it to the driver transmit function. This function calls a mapper routine which is responsible for translating the destination CSP node identification into the AX.25 address (which is represented in a callsign plus Secondary Station Identifier (SSID) format). This process has the same objective as Address Resolution Protocol (ARP) found on typical IP networks. The mapping information is available in a static way (provided in compile time like the CSP node ID) due to the high communication restrictions imposed by the spacelink. On a such constrained link it is not to practical to implement a dynamic resolution protocol because it requires an undesirable extra message exchange. After this cross-layer mapping

the transmit function encapsulates the CSP packet into the AX.25 payload to be latter sent to the kernel module. On both ways (incoming and outgoing), the network endianness compatibility is assured by the respective `htons() / ntohs()` primitives.

The native CSP implementation smooths the code compilation/configuration process through a waf script[12]. In addition to the new AX.25/CSP code that was integrated into the native csplib, the waf script was also extended to include this new AX.25 support. Therefore, no extra step is required when the csplib needs to be compiled with a AX.25 support.

The PriSIS module relies on this new CSP/AX.25 driver to perform the GS-Satellite command/response mechanism. The PriSIS main program flow is presented in Fig. 4.10

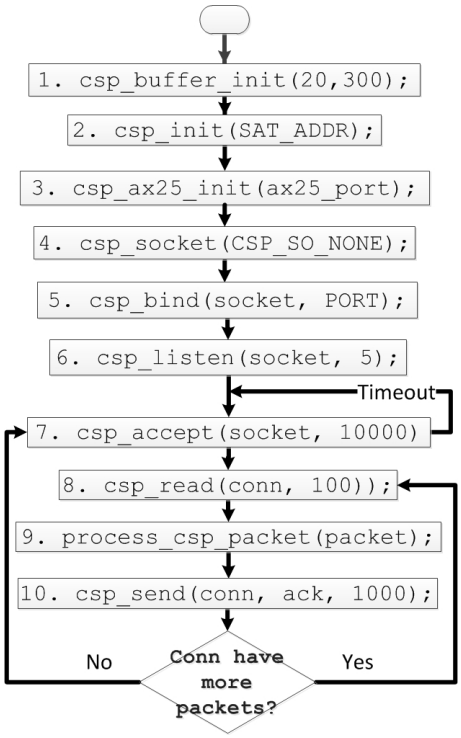


Figure 4.10: PriSIS boot and operation sequence.

On the first step, the PriSIS allocates memory for 20 packets with 300 bytes each. This memory (6 Mbytes) will be later used by AX.25 driver and by the router engine to handle the incoming/outgoing packets. Then, on step 2, the software will initiate the CSP stack, passing the local CSP node address. On step 3, the AX.25 driver is started, giving the location for AX.25 port as argument. After this, a CSP socket is created. Next, on step 5 the socket is associated with an application port (port 10 was assigned to PriSIS). On step 6, five possible simultaneous connections are allocated and the socket state is moved to accept state (step 7). Since the socket must be continuously

[12] `http://code.google.com/p/waf/`

active, when the configured timeout is reached (lacking of connections) the step 7 is re-executed. The blocking primitive on the step 8 returns the received CSP packet from the incoming connection. Finally the CSP packet carrying the GS command is ready to be processed on step 9, where the CSP command structure (described in Fig. 3.5) is parsed and the command executed. After this command execution the PriSIS sends back to the GS an acknowledgement carrying the success/error code together with the GS counter.

One GS application was also developed to allow the interaction with PriSIS by a remote GS operator. This solution also implements the same CSP/AX.25 stack as PriSIS do. After the GS software being initialized it asks the operator for a command which will be properly encoded and sent to PriSIS. Fig. 4.11 illustrates the GS software asking PriSIS about the on-board beacon status, issuing the function number 3. After that a CSP command is sent through the space-link with the clock id 3 in this case. The PriSIS response back with the message *beacon is not running* together with the same clock (3). Next the GS operator issues the *Beacon on* function repeating the CSP process but now with clock id 4, and PriSIS replies back with a successful code.

```
######### ISTNanosat-1 Monitoring software
# General information:
#   1 -> Beacon on
#   2 -> Beacon off
#   3 -> Check beacon status
# Other functions:
#   10 -> Fetch all captured images
##########################################
->3
Received ACK for clock: 3
Satellite message: Beacon is not running!
->1
Received ACK for clock: 4
Satellite message: Command successful!
```

Figure 4.11: GS software interacting with PriSIS software on-board satellite

The GS software also allows the operator to ask for imagery transmission (function 10).

Imagery gathering

The *Fetch All Images* function will be encapsulated as a normal CSP command. After this command is correctly received on PriSIS, it will trigger a Tolerant-CSP (T-CSP) transmission, sending all the onboard saved images to the GS. The T-CSP protocol is an extension to the already implemented CSP/AX.25 stack functions. It relies on the already developed primitives adding the new fragmentation and recovery functionalities. The developed T-CSP services are available through the T-CSP API functions presented in Fig. 4.12.

Fig. 4.13 shows how either `t_csp_send_frag()` and `t_csp_send()` interacts with the CSP library.

```
int t_csp_send_frag(char *path, int part, int dst_id, int dst_port);
int t_csp_send(char *path, int dst_id, int dst_port);
char * t_csp_recv(csp_socket_t *socket, int local_port, int *ret_s, int *f_nr,
        int delay);
```

Figure 4.12: T-CSP available functions

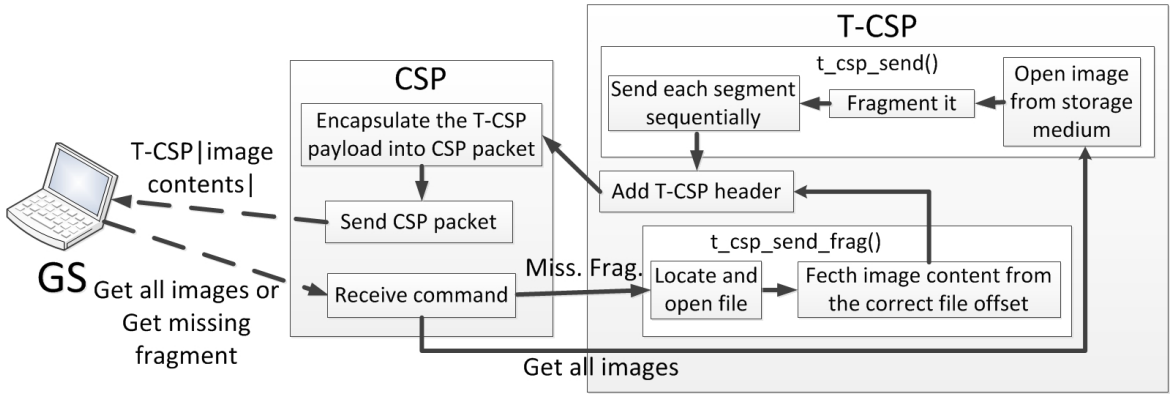


Figure 4.13: CSP to T-CSP interactions

When a *Get all images* function is received by PriSIS from the GS, it calls the T-CSP primitive `t_csp_send()`. This primitive abstracts the file operations, (e.g. open) and fragment each image file. After this, each fragment is passed to the underlying CSP `send()` primitive together with its T-CSP header. When the GS software requests for segment retransmission, the T-CSP `t_csp_send_frag()` is executed. This function locates the required file and extracts from it the piece of lost information. Before this, a T-CSP header is added and the complete T-CSP segment is passed to the CSP `send()` instruction, which sends back to the GS the missing segment. On the GS software this lost segment will be received by the `t_csp_recv()` function. This function abstracts the recovery process (using the SNACK approach described on chapter 3) such as detecting a segment lost, asking PriSIS for a new segment retransmission or link disruption detection. The `t_csp_recv()` locking function only returns to the application when a new file is completely received, or when the GS operator decide to abort due to e.g. hard link disruption occur or the communication contact opportunity is lost. When such heavy disruption occurs, the GS operator is informed about the number of segments received and missing, which allows him to decide if it is reasonable to keep the retransmission process.

Fig. 4.14 displays the GS software receiving one image from the satellite using the T-CSP protocol:

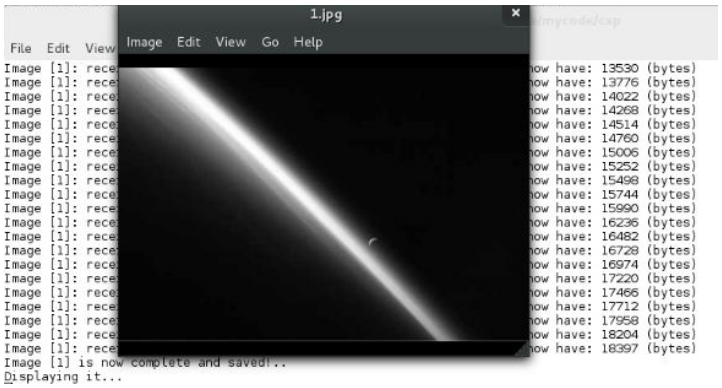


Figure 4.14: GS software receiving a satellite sample image using T-CSP protocol.

4.1.3.C C&DH Operating system and applications

According to the software design for the C&DH subsystem it must run the FreeRTOS Operating System. The FreeRTOS for MSPGCC toolchain is being ported to the MSP430F5438A MCU by the FreeRTOS project. This common available port[13] was tailored to support the moteist++s5 platform. Since FreeRTOS code structure uses a Hardware Abstraction Layer (HAL) to implement the platform specific code, which is MCU independent, a new moteist++s5 HAL for FreeRTOS was created. This new FreeRTOS moteists5++ HAL contains the major platform configuration and the external peripherals functions implementations. Fig. 4.15 illustrates the most important prototypes found on the new HAL (located at `MSP-430F5438_MOTEIST_HAL/hal_board.c`).

```
1. void halBoardInit(void);
2. void hal_i2c_init_slave(unsigned char my_addr);
3. void hal_setup_uart_9600_8N1(void);
4. void hal_setup_leds(void);
```

Figure 4.15: FreeRTOS/moteist++s5 HAL major functions

The first function configures the default I/O port states according to the MCU manufacturer directives. The second function configures the platform to work with the I^2C protocol in slave mode. Here, the I^2C RX and TX pins on the moteist++s5 are mapped on the MCU port 3 bit 7 and port 5 bit 4, and the required interrupts are enabled. The third function configures the MCU UART for 9600 baud rate taking into account the moteist platform port layout. The fourth, follows the same logic from the last three functions as it instructs the FreeRTOS about the correct LED port/bit locations. With this platform

[13] `https://github.com/pabigot/freertos-mspgcc` accessed on 19-08-2012

supported, the FreeRTOS can be used in a standard way using this platform abstraction code.

With the required moteIST I/O peripherals supported in FreeRTOS, the real-time application can be deployed. The developed application implements the redundant beacon function as well as the incoming I^2C data processing (e.g. from the COM subsystem). Beyond this functions, the application also receives simple serial commands from the Analog COM subsystem, namely the Data Framer component, replying with an ACK message if the command is correctly received. Fig. 4.16 depicts the C&DH application structure.

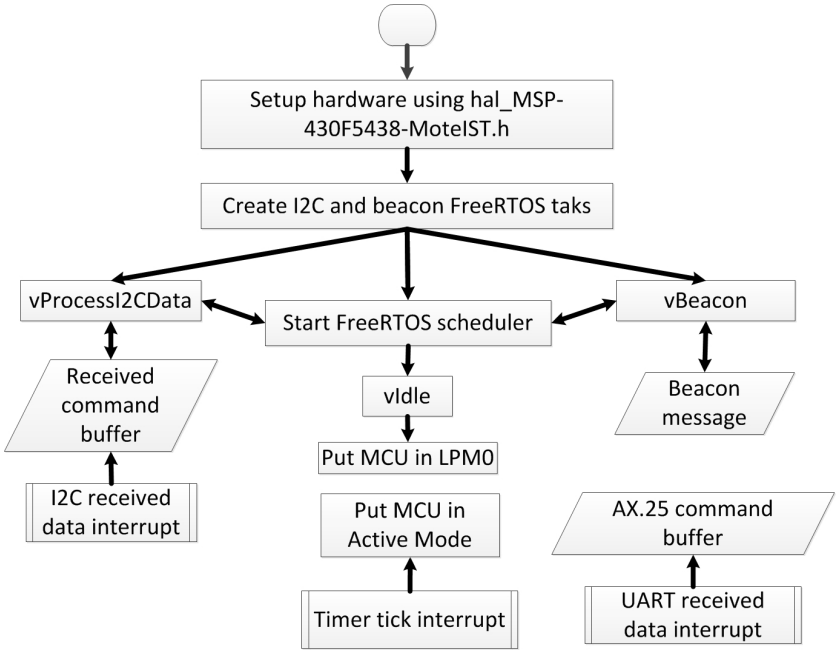


Figure 4.16: FreeRTOS C&DH software structure.

Firstly, the C&DH application configures the hardware peripherals using the developed HAL for moteist. After this, it creates three FreeRTOS tasks, `vProcessI2CData`, `vBeacon` and `vIdle`. Real-time tasks are well suited for this scenario, because they are a simple way to implement the previously described functions in a full real-time way. As alternative to real-time tasks, the FreeRTOS also provides Co-Routines, which are processed by the FreeRTOS scheduler according to a cooperative algorithm. This is not a desirable behaviour, because a co-routine cannot be preempted by another co-routine and the full real-time characteristics are lost. The `vProcessI2CData` task processes the received command buffer that is filled by an Interrupt Service Routine (ISR) assigned to the I^2C data reception interrupt. This ISR is triggered when the MCU receives serial data correctly from its USART. The `vBeacon` task injects into the UART transmission buffer the beacon message string every 4 seconds when the Heart unit is running on the safe-mode profile. The UART data reception interrupt is assigned to an ISR which stores the received commands from the space-link into the AX.25 command buffer. After the correct

command reception, it replies with an ACK message through the space-link, confirming the correctness of the received command.

Fig. 4.16 also illustrates how the real-time application handles the Low-Power Modes (LPMs) usage. Since the `vIdle` task is configured with the lowest scheduler priority it will run when there are no other tasks waiting to take processor time. Therefore, the `vIdle` task is programmed to put the MCU in the most energy conservative state - LPM-0. The MCU is awake back to active mode when some system interrupt occur. An MCU timer is configured to trigger an interrupt periodically. This interrupt will execute another ISR which will process a new scheduler decision. This will force the MCU to awake back to active mode periodically, allowing the stopped tasks to run again if it is necessary. With this strategy (put the MCU in LPM-0 when there are no real-time tasks to be processed) the overall C&DH power consumption decreases if the MCU is not under over-utilization.

All three real-time tasks are different in terms of importance within the application. The following task priority strategy was configured (most important first):

1. `vProcessI2CData` - This is the most important task, therefore it will always win processor time when it needs. This is considered the most important one, because it processes the inter-subsystem communications which can be for example an ADCS command to change the satellite attitude.
2. `vBeacon` - Intermediate priority, because it is not a critical instruction.
3. `vIdle` - Lowest priority, responsible to put the MCU in LPM-0. Executed when there are no other tasks waiting to run.

5

Experimental evaluation

In order to validate the reliability, functionality and quality of the developed solution, one set of tests were performed for the Hear Unit. Since the Analog COM subsystem is currently under research and the intended radio link characteristics are difficult to emulate in the laboratory, the space link was abstracted using a serial connection from the GS computer to the COM subsystem. The same method was also applied to interconnect the C&DH subsystem with the GS computer. The lack of realistic physical conditions on this important link, makes the test accuracy hard to achieve. The test design phase had this fact into account together with the requirement specifications, specifically the safety-critical points, and produce two main test groups. On one hand, the first test group aims at the solution's performance evaluation, where both base systems (C&DH and COM Operating System (OS)) and intended network-based functionalities were evaluated. On the other hand, the second test group targets the solution quality validation for safety-critical environments. This second test group was formalized taking into account some directives found on *NASA Software Safety Guidebook* (GB-8719.13) [63].

The main test scenario, illustrated in Fig. 5.1, is capable of testing both redundant operation profiles (normal mode and safe-mode). When normal mode is being tested, the serial connection with the COM will be used. Otherwise, for the safe-mode tests, the C&DH serial connection is used.

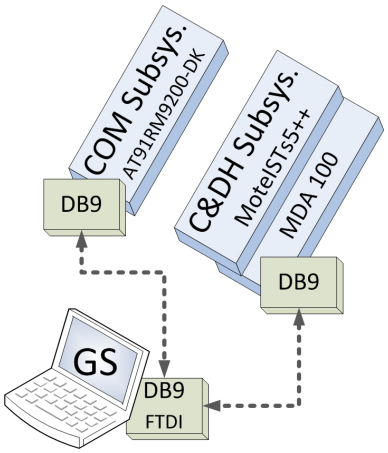


Figure 5.1: Test scenario

5.1 Performance evaluation

Taking into account the intended requirements for both COM and C&DH subsystems, this section will focus on presenting an evaluation about the solution's operational behaviour. This performance tests were mostly formulated taking the grey-box approach.

5.1.1 COM Subsystem

Besides the fulfilled energetic requirements, this subsystem must meet the remaining tight requirements described in section 3.1. Some system statistics were collected from the intrinsic COM diagnostic mechanism - telemetry beacon - which serves as the behaviour information source. Firstly, Fig. 5.2 shows how much flash storage is required for the entire COM software solution.

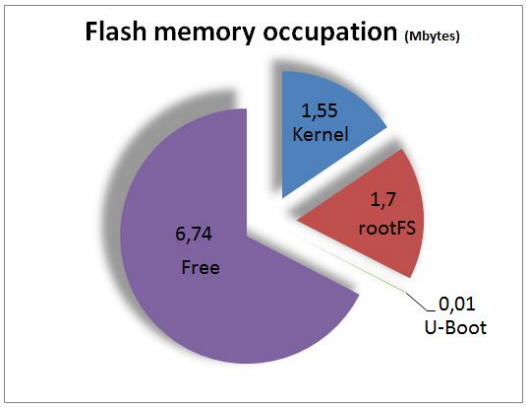


Figure 5.2: Flash memory utilization

As the graphic shows, the total required flash storage space is 3.6 Mbytes. This information was

collected from the boot-loader memory organization. The COM software systems roughly takes 37 seconds to be launched when electrical energy becomes available. This time (31 seconds) is mainly taken by the Linux kernel boot operations and rootFS init.d scripts. The remaining time is used by the boot-loader itself in copying the software from Flash to SRAM and performing the CRC operation over the kernel image file. The shutdown time takes about 3,7 seconds using the system ordered shutdown halt method.

Furthermore, the SRAM utilization was also evaluated. Fig. 5.3 shows the SRAM occupation after the system is ready for operation. This information was collected from the telemetry beacons gathered by the GS computer which has the developed telemetry decoding software installed. All the telemetry sent though the space-link immediately after the system is powered up was decoded and stored within a period of 3 minutes (recall that telemetry is sent every 5 seconds). Within this period the COM had all software functionalities available, but there was no extra processing request issued by the GS (e.g. command request). The GS only passively monitors the telemetry data.

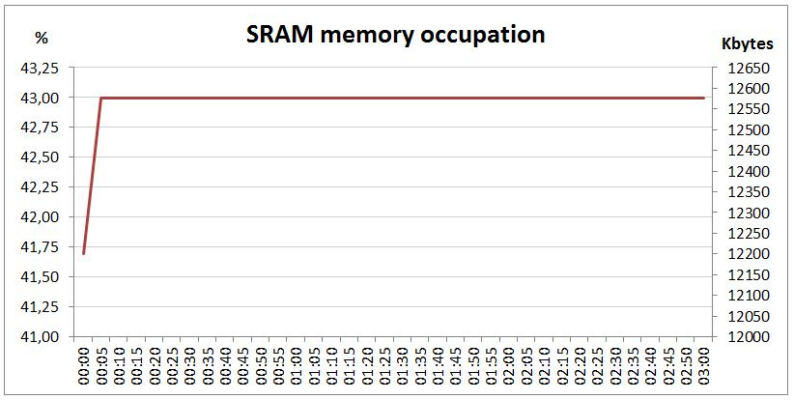


Figure 5.3: SRAM utilization after system boot.

From all the SRAM available (roughly 32 Mbytes) only 43% (12.6 Mbyte) is occupied when the subsystem is idling.

Apart from the storage (Flash) and working memory (SRAM) utilization the CPU state was also monitored within this period. Since the telemetry beacon carries also the average CPU load information provided by the Linux kernel. This parameter was also analysed. The result is summarized in Fig. 5.4.

Fig. 5.4 shows two different loads. One average load was evaluated based on the last minute CPU load and another based on the last 5 minutes. Every 5 seconds the telemetry beacon reflects these two values. Actually, the beacon carries three different average load values, measured over one minute, five minutes and fifteen minutes intervals. However, for the 3 minute test, the fifteen minutes interval was discarded. The Linux kernel measures this system load based on the correlation between the number of waiting process in the processor's queue and the number of CPUs available. The first telemetry

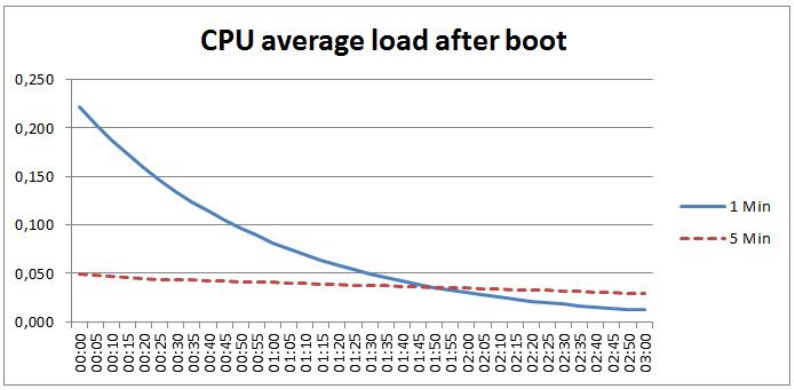


Figure 5.4: CPU load average after system boot.

information collected suggests that the CPU is under-utilized by ≈77.5%, on average, over the last minute (which encompass the remaining boot operations after the kernel is ready). After this stressed period, the CPU will drop to a further lower level of utilization. On idle, the entire COM subsystem only demand about 3,5% of CPU utilization. We saw in section 4.1.1 that this subsystem carries a 180MHz CPU. In this graphic the 5 minute average load shows a slight trend (but somehow inaccurate at the beginning due to lack of all 5 minute information) about this low CPU usage that will eventually converge to the values associated with the one minute average load. These good performance benchmarks shows that both the utilized SRAM and CPU power meet the system boot process requirements.

The described SRAM and CPU load tests were repeated on other expected scenarios, but with a larger time scale - 5 minutes. The next test tries to evaluate how much the PriSIS software will impact the overall system performance. To achieve this, the PriSIS process was killed and the telemetry monitored. Fig. 5.5 depicts that situation where, comparatively with the graphic in Fig. 5.3, there is no substantial RAM occupation reduction. This means that PriSIS only occupies a few bytes in RAM, and almost all of the available RAM is used by the kernel and rootFS operation.

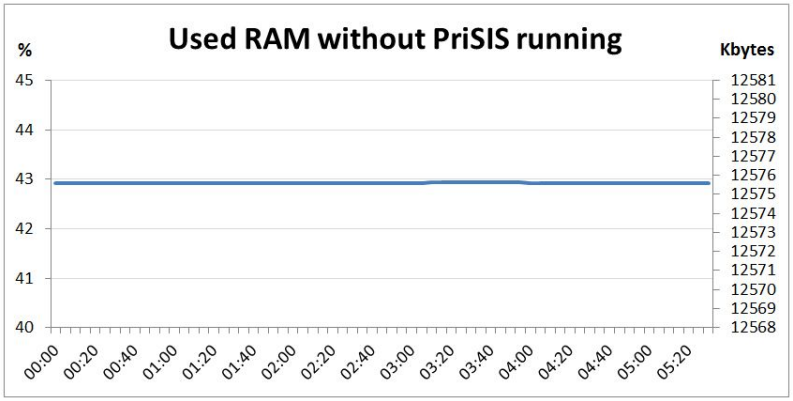


Figure 5.5: SRAM usage without PriSIS

The same behaviour is observed in the CPU load. The comparison between the one minute average from Fig. 5.4 and the one in Fig. 5.6 leads to the conclusion, that the load decays about 90% from the initial value. The analysed load values are very low and therefore it is possible to conclude that this software solution is able to run on a even a low-end class of CPU.

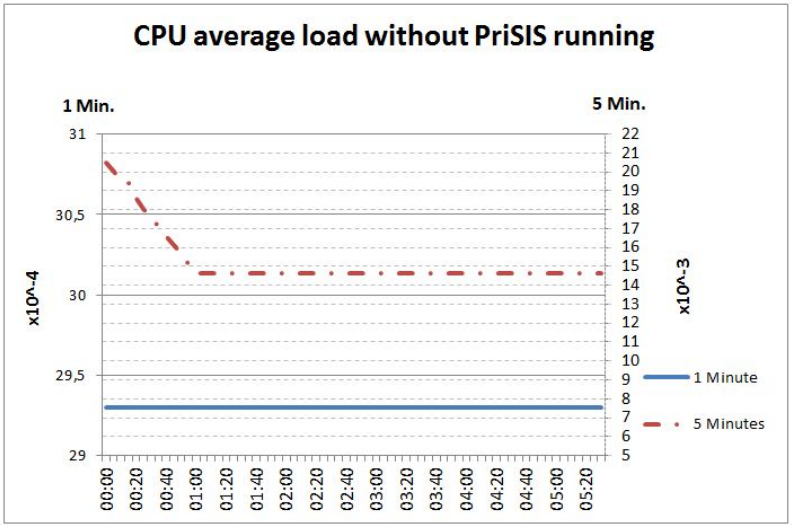


Figure 5.6: PriSIS software impact on CPU load

The next scenario tested consist in consecutive commands issued by the GS operator. Fig. 5.7 shows that SRAM utilization starts increasing but at some point in time seems to stabilize around the 55%. This stabilization gives a clue that probably there are no memory leaks on the PriSIS code (this aspect will be discussed later).

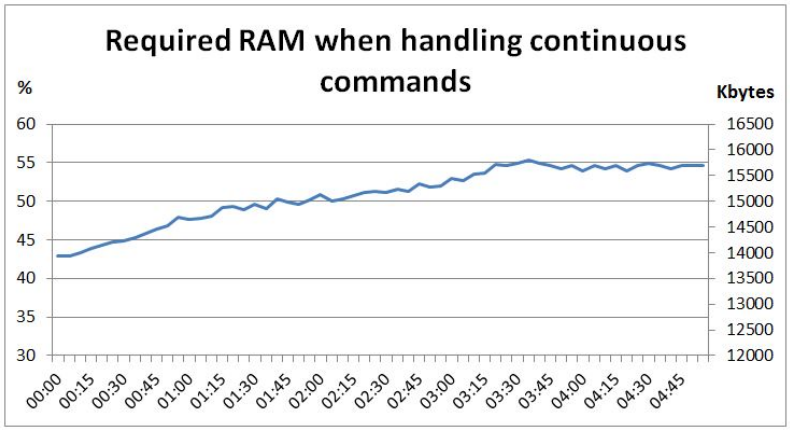


Figure 5.7: RAM utilization over continuous command handling

The same behaviour is observed in the load values in Fig. 5.8. As long as more and more GS commands arrive at PriSIS, it requires more processing power to deal with them, thus raising the load

values to around 65% of CPU utilization.

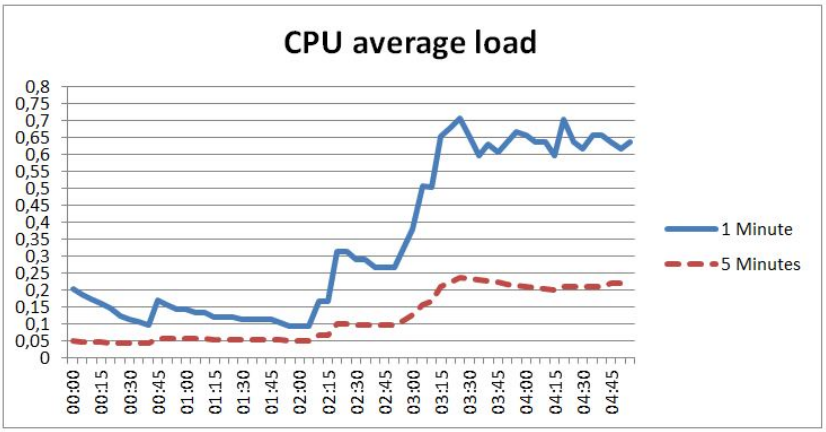


Figure 5.8: CPU loads over continuous command handling

This high stressful scenario, which maybe is unlikely to happen in a real-world setting, it is not yet the *killer application* in terms of COM processing power and memory characteristics.

A likely stressful scenario is the transmission of the on-board stored pictures. Fig. 5.9 highlights the normal behaviour on RAM utilization when the T-CSP functionalities are being used. This is another good sign about no memory leakage on the PriSIS software when performing fragmentation.

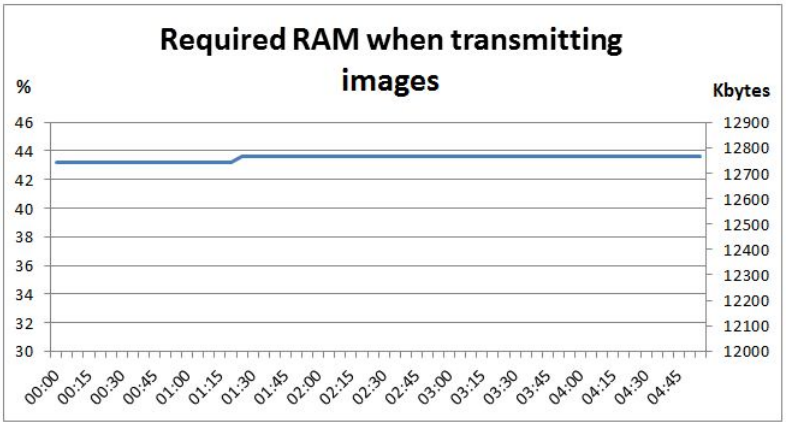


Figure 5.9: RAM utilization when transmitting an image

Fig. 5.10 shows that, like the RAM occupation, there is no excessive system load increase.

Another aspect that was taken into account was the developed T-CSP time efficiency. Fig. 5.11 shows how much time T-CSP takes to transmit a set of segments when some disruption occurs in the space-link. This test was performed, using a 1200 bit/s serial connection for two possible different window verification sizes ($x = 1$ and $x = 5$[1]). These two sizes were chosen as a way to produce an

[1] x is the missing segment window size that reflects the maximum number of unreceived segments in a burst segment transac-

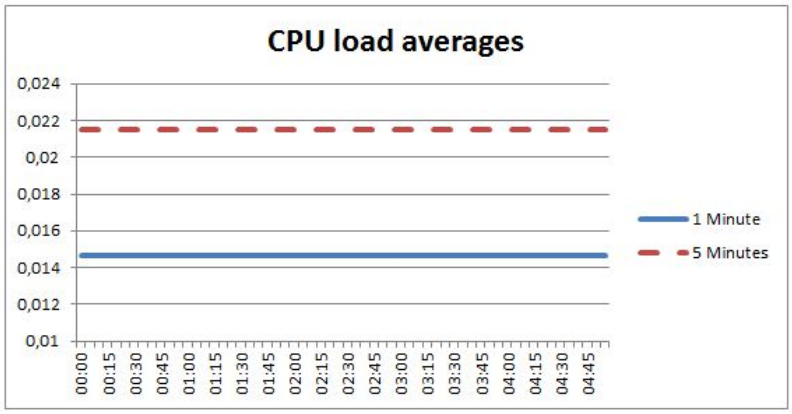


Figure 5.10: CPU loads over image transmission

evident contrast between these opposite (minimum and large) window sizes. The T-CSP receiver (GS on this case) timeout was configured to be 2,1 seconds since the frame time (with maximum payload) is about 1,8 seconds. Also, the `t_csp_recv()` was slightly modified for this test in order to auto-accept the retransmission process when the timeout was reached, bypassing the possible user-defined delay to trigger the retransmission mechanism. The abscissa axis in Fig.5.11 represents the number of lost segments during multiple file transmissions. These lost segments were originated by unplugging the serial cable during multiple file transmissions, each transmission with different number of lost segments. The ordinate axis represents the time spent by T-CSP to transmit a sample image with ≈18Kbytes which require 75 segments.

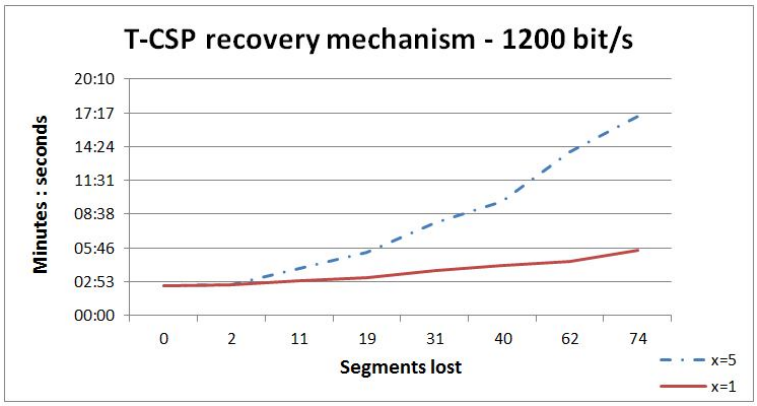


Figure 5.11: File transmission at 1200 bit/s using T-CSP facing link degradation.

As Fig.5.11 illustrates, the lowest window verification size ($x = 1$) is more time efficient with these low bitrates. This confirms the initial thoughts from the design phase. It is also possible to conclude, that the minimum time required to receive this 75 segment sequence at this bit-rate is a little less than

tion and relative to the last correctly received segment

3 minutes. This naturally occurs when there are no segment retransmissions. With $x = 5$ worse timings are obtained. This as to do with the foreseen unnecessary link over-utilization, since multiple useless retransmission are issued on the uplink. During this test ($x = 5$) was also obvious that PriSIS buffers become easily full and then some retransmission messages were discarded due to RAM overflow. Note that for $x = 5$ values, after 25 segments lost, it is more quick to ask for a full file retransmission (which takes about 3 minutes) than trigger the retransmission process (which takes roughly 6 minutes and 30 seconds). Asking for a full file retransmission saves about 30 seconds in this case. It is also clear that for $x = 1$, the worst case scenario (when almost all segments are lost - 74 in this case) is still advantageous the use of the T-CSP retransmission, rather than ask for the entire image retransmission. This efficiency gain regarding the $x = 5$ window as to do with less uplink utilization and fastest missing fragment verification on the GS software. With $x = 1$ the PriSIS buffer not overflow, which also makes the entire retransmission fastest. Finally, if an expected satellite orbit is such that allows 20 minutes of communications opportunity, about 500 segments ($\approx$ 120 Kbytes of effective data) can be transmitted under optimistic conditions (without retransmissions).

Fig. 5.12 represents the same test but using a 9600 bit/s bitrate. In this test the T-CSP `t_csp_recv()` timeout was reduced to 500 milliseconds since the frame time (with maximum payload) is about 0,23 seconds.

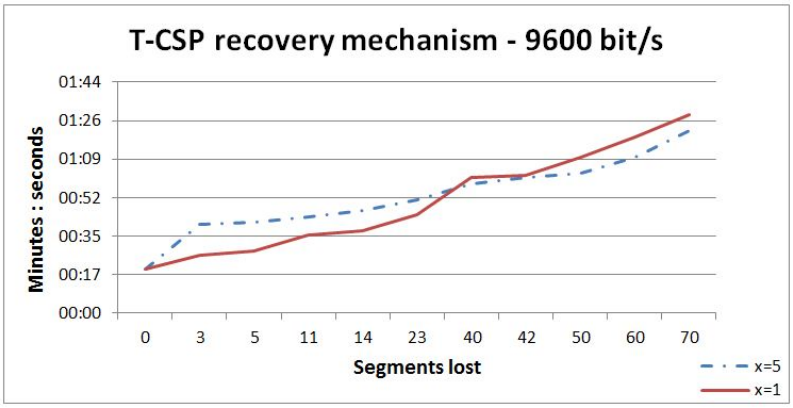


Figure 5.12: File transmission at 9600 bit/s using T-CSP facing link degradation.

The figure shows that the minimum time spent to transmit the 75 segments is $\approx$ 19 seconds when there are no segments lost. The graphic shows also that the $x = 1$ window have a loss of performance when compared to the $x = 5$ case above the 40 segments mark. At this higher bit-rate the PriSIS hardly becomes overflowed (actually during the 9600 bit/s tests it never did) and also the problem associated with large window sizes is attenuated in terms of time impact. On one hand, above the 40 segments lost, it seems the $x = 5$ option allows a quicker response from PriSIS which is compensatory even with some duplicated messages transmitted. On the other hand, for $x = 1$, every lost segment will trigger a

timeout, while for $x = 5$ it does not happen, speeding up the process. At this bitrate, on the worst case scenario (when about 74 segments are lost, using $x = 1$), about 1 minute and 30 seconds are required to transmit the entire 75 segment sample image. Under an optimistic scenario (no segments lost) during a 20 minute communication opportunity orbit $\approx$ 5000 segments can be transmitted allowing up to 1.2 Mbyte of effective data transmission.

5.1.2 C&DH Subsystem

The C&DH subsystem is difficult to test since it is a highly integrated platform. Unlike the COM, the C&DH does not have an intrinsic performance report mechanism (as the telemetry beacon on COM) and the installed RTOS debugging over the Joint Test Action Group (JTAG) programmer is very difficult, due to the preemption mechanism. The entire C&DH software solution (FreeRTOS OS, HAL functions and C&DH application code) consumes 7Kbytes (2.7%) from the 256Kbyte available flash on the MSP430F5438A MCU. This utilization is acceptable and does not limit future C&DH software enhancements. This value is gathered from the "text" field in the output of size command, described in Fig. 5.13. The size command is available on the MSP-GCC toolchain.

```
$ msp430-size cdh.elf
   text            data             bss             dec             hex         filename
   7150               4           10530           17684            4514         cdh.elf
```

Figure 5.13: C&DH binary size information

Besides the Flash memory, the C&DH software RAM utilization is hard to evaluate on compile time. The output illustrated in Fig. 5.13 shows that at least 10534 bytes (64%) of RAM (bss segment + data segment sizes) from the 16384 bytes available is required. Note that the bss segment stores statically-allocated uninitialized variables where the data stores the global and static initialized variables. This does not include the runtime/dynamically allocated memory. One evident missing case here is the RAM required by the tasks. Since the C&DH software uses three different FreeRTOS tasks (idle, I^2C and UART tasks), and they are configured to use the `configMINIMAL_STACK_SIZE` directive which implies 120bytes per task stack, they will require more 360 bytes. Therefore, it is more accurate to predict the minimum RAM requirement for this solution with $10530 + 4 + 360bytes \approx 10.6Kbytes$. This represents 66.25% of all the MCU available RAM (16Kbytes). This RAM calculations give a clue about the C&DH software solution requirements. The "dec" and "hex" fields on the output described on Fig. 5.13 are the total file size in decimal and hexadecimal formats, respectively.

5.2 PriSIS and beacon software safety/quality

This section describes the required procedures taken to assess the safety and quality of the Heart unit developed software solutions. The code inspections and tests were mainly engineered taking into account the directives found on *NASA Software Safety Guidebook (GB-8719.13)* [63].

The most important software applications developed for the Heart unit was the PriSIS and the beacon software aboard the COM subsystem. With this in mind, a larger test effort was employed on these software components to ensure its quality and reliability. The first test was the code visual inspection, taking into account the guidelines, precautions and standard verification processes found on [63, p. 214-218]. These guides state the limitations and problems with C language, programming C standards and the ten commandments for C programmers. This document also provides a ”Good Programming Practices Checklist” [63, p. 384-388]. All of this information is summarized and described in the next points.

- **Verify read medium contents -** The U-boot bootloader is instructed to perform CRC calculations over the Linux kernel image (which encompass the vital subsystem software function) before executing it. This ensures that all kernel image contents are righteous which low the hypotheses of execute wrong code that can triggers potentially wrong/unexpected behaviours.

- **Use watchdog timers -** This function may be useful to recover from unexpected system crash, but it is not implemented because it would require creating the AT91RM9200 hardware watchdog interface.

- **Use Stack Checks -** A special care about the stack boundaries was taken. First, both pulsar and PriSIS were compiled with the GCC stack guard functionalities `-fstack-protector-all` which implies a little code overhead to protect against buffer overflows such as *stack smashing* problems. This protection was correctly installed on PriSIS but only with the glib version. Unfortunately the μClibc does not provide such kind of feature. So the buffer overflow problems were avoiding by using the more secure memory C primitives such as `snprintf()` or `memmove()`. This primitives allow a more strict control over the number of bytes copied which helps avoiding such kind of problems.

- **Initialize memory -** All the dynamic allocated buffers were previously initialized with empty pattern ’0’ to enforce the correctness of later readings.

- **Do not implement ”one big loop program” -** The biggest loop present on the code is the PriSIS CSP command id match function. The remain developed code implements every functionality on a separated function.

- **Avoid indiscriminate use of interrupts -** The Linux kernel abstracts the interrupt handling to the application. There are no explicit interrupt-based functions on the developed code.
- **Provide a orderly system shutdown -** Since the PriSIS and pulsar are called on the boot, it won't require special shutdown functions.
- **Always initialize the software into a known safe state -** The rootFS image recorded into the flash memory is immutable. This practice will allow the system to revert to a secure state after a reboot if something unexpected occurs.
- **Special care with thread-based programs -** The PriSIS relies on a thread to wait for incoming AX.25 frames carrying CSP packets. This thread is created by the developed AX.25/CSP driver which processes and injects into the CSP routing engine the arrived CSP packets. The inter-thread communication (between the driver and the router engine) is done by the CSP primitive `csp_new_packet()`, which later calls `pthread_queue_enqueue()` primitive that will enqueue the new packet using the required mutual exclusion mechanisms. This inter-thread communication seems to be well formulated and brings more confidence about the resilience against e.g. possible packet queue race-conditions.
- **Do not implement delays as empty loops -** The rationale behind this question as to do with incoherent effective delay time among different architectures, compilers, etc. There is no such practice on the developed code, where the used delays are abstracted by the CSP primitives which relies on `pthread_*` intrinsic time mechanisms.
- **Protect against out-of-sequence transmitted messages -** The only inter-system message exchange is done within the space-link, from GS to PriSIS. The CSP-based space-link command transmission carries a logical clock into the message command which allows the GS software track back which received ACK belongs to each issued command.

Beyond the above described visual and conceptual verification test, the document [63] also suggests a set of tests that can be automated by some widely available test tools. These tests are grouped into statical analysis, which are done against the source code without executing the produced binaries, and dynamic analysis which are performed while executing the compiled binaries.

5.2.1 Static analysis

The first static analysis tool used was the intrinsic GCC compiler warning enforcement by activating the flags `-Wstrict-prototypes -Wmissing-prototypes -Wmissing-declarations -Wall` on the respective makefiles. This flags enforce a more strictly code production which avoids later problems. The final PriSIS and pulsar are compiled without any relevant GCC errors/warnings.

Both PriSIS and pulsar software were also inspected using a lint program called splint[2]. This static analysis tool reported a lot of possible errors on the produced code. The Fig. 5.14 gives an example of one of the bugs found by the splint tool:

```
pulsar.c:58:12: Fresh storage portcall not released before return
  A memory leak has been detected. Storage allocated locally is not released
  before the last reference to it is lost.
```

Figure 5.14: Splint tool example of error detection

This particular bug, as the description suggests, is a memory leak which was corrected by freeing the `portcall` pointer with `free(portcall)` primitive. All that major problems that were not detected by the GCC compiler enhancement flags, and that were highlighted by the splint tool, were corrected.

5.2.2 Dynamic analysis

This group of tests are composed in this case by two different tests: code profiling and runtime memory check analysis. On one hand, code profiling produces a dependency map in terms of function execution flow proving also the time spent on each executed function. This is useful to realise which are the functions that can be enhanced in terms of execution performance. On the other hand, the memory check analysis allows a detailed inspection over the runtime used memory, reporting the inefficiencies/problems associated with it. Both tests were made using the valgrind[3] tool.

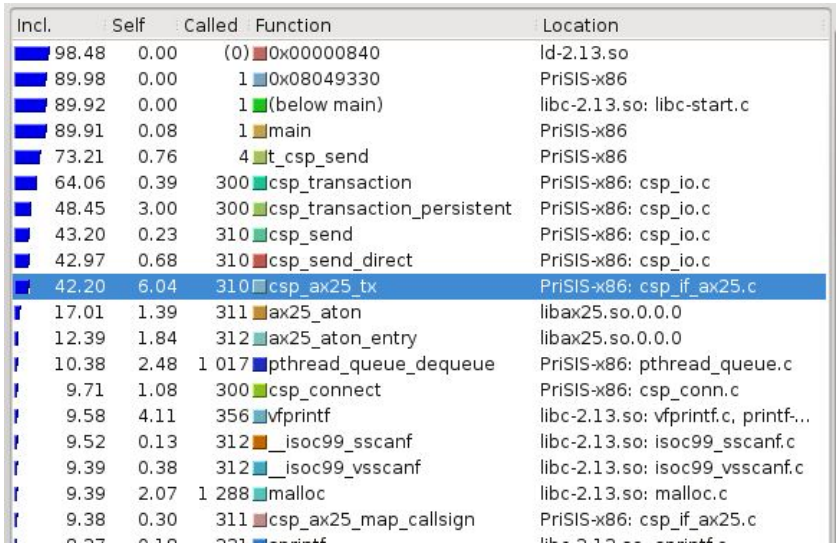
Since it is hard to put the valgrind tool running over the COM system due to the lack of storage memory, the PriSIS and beacon software were compiled also for the x86 architecture (host architecture). With this x86-based PriSIS and pulsar binaries, it was possible to run these tests on a separated the x86 machine that will emulate the COM hardware. Running the tests over a different CPU architecture is the least worst solution available that allow both these tests. Fig. 5.15 shows how valgrind tool was used to start the profiling test on PriSIS on the x86 machine:

```
$ valgrind --tool=callgrind --dump-instr=yes \
--simulate-cache=yes --collect-jumps=yes ./PriSIS-x86
```

Figure 5.15: Valgrind as profiler tool.

[2] http://splint.org/
[3] http://valgrind.org/

Here the `callgrind` is the profiling tool available from valgrind suite and the `--dump-instr=yes` allows output of assembly annotations. When the valgrind/callgrind is running it invokes the PriSIS that will wait for CSP commands. When PriSIS is waiting, the GS operator issues one photo request command followed by four generic commands (such as beacon toggle) plus another two photo request commands in a row. After this sequence of commands the valgrind produced one output file which is later analysed by a graphical tool called `kcachegrind`[4]. Fig. 5.16 shows the relevant information captured from the `kcachegrind` report:



Incl.	Self	Called	Function	Location
98.48	0.00	(0)	0x00000840	ld-2.13.so
89.98	0.00	1	0x08049330	PriSIS-x86
89.92	0.00	1	(below main)	libc-2.13.so: libc-start.c
89.91	0.08	1	main	PriSIS-x86
73.21	0.76	4	t_csp_send	PriSIS-x86
64.06	0.39	300	csp_transaction	PriSIS-x86: csp_io.c
48.45	3.00	300	csp_transaction_persistent	PriSIS-x86: csp_io.c
43.20	0.23	310	csp_send	PriSIS-x86: csp_io.c
42.97	0.68	310	csp_send_direct	PriSIS-x86: csp_io.c
42.20	6.04	310	csp_ax25_tx	PriSIS-x86: csp_if_ax25.c
17.01	1.39	311	ax25_aton	libax25.so.0.0.0
12.39	1.84	312	ax25_aton_entry	libax25.so.0.0.0
10.38	2.48	1 017	pthread_queue_dequeue	PriSIS-x86: pthread_queue.c
9.71	1.08	300	csp_connect	PriSIS-x86: csp_conn.c
9.58	4.11	356	vfprintf	libc-2.13.so: vfprintf.c, printf-...
9.52	0.13	312	__isoc99_sscanf	libc-2.13.so: isoc99_sscanf.c
9.39	0.38	312	__isoc99_vsscanf	libc-2.13.so: isoc99_vsscanf.c
9.39	2.07	1 288	malloc	libc-2.13.so: malloc.c
9.38	0.30	311	csp_ax25_map_callsign	PriSIS-x86: csp_if_ax25.c

Figure 5.16: Kcachegrind output.

The *Incl* - Inclusive - and *Self* columns represents each function cost in terms of time and CPU usage. The *Inclusive Cost* represents the cost of all functions that are called from the current function, while the *Self Cost* represents the cost for that particular function. Observing the values in the Fig. 5.16, it is clear that the most heavy (with 6.04% as Self Cost) function is the `csp_ax25_tx`, which belongs to the basic primitive implemented by AX.25/CSP driver to send CSP packets encapsulated in AX.25 frames over the radio link. After some inspection on the `csp_ax25_tx()` implementation it was concluded that this high Self cost was originated with high probability by these four functions:

The highlighted four routines above are responsible for CSP packet encapsulation over a temporary buffer that will later be transmitted using the AX.25-lib `sendto()` primitive. These functions are processor and I/O exigent since they are intended to allocate new memory space from the heap (step 1), cleaning the new space (step 2) and copy the CSP packet contents into the new memory space (step 3 and 4). Note that the CSP packet header (`packet->id.ext`) is required to be previously converted into a network endian format using the proper CSP primitive for that effect - `csp_hton32(packet->id.ext);`,

[4] `http://kcachegrind.sourceforge.net`

```
1. txbuf = (char *) malloc(packet->length+CSP_HEADER_LENGTH);
2. memset(txbuf, 0, CSP_HEADER_LENGTH+packet->length);
3. memcpy(txbuf, &packet->id.ext, CSP_HEADER_LENGTH);
4. memcpy(&txbuf[CSP_HEADER_LENGTH], &packet->data, packet->length);
```

Figure 5.17: CSP packet encapsulation.

before the packet was sent. All these functions ensures that each CSP packet structure information is sent in the right order. It was not found a more efficient workaround for these 4 packet processing instructions. It is possible to bypass the second step, but by doing that the good programming directive *"Initialize memory"* will not be met.

The next test is the run-time memory inspection, which mainly tries to find obfuscated memory leakage in the developed code. This test was also performed using the valgrind tool as Fig. 5.18 depicts:

```
$ valgrind --tool=memcheck --leak-check=yes --show-reachable=yes \
--track-fd=yes ./PriSIS-x86
```

Figure 5.18: Valgrind as memory inspection tool.

After the PriSIS enters in the wait incoming packet's state, the same CSP commands from the previous test was sent to it. First, the image request command, which will trigger the T-CSP functionalities followed by four CSP commands, e.g. beacon on/off and another two image request commands. With this sequence of events all the PriSIS functionalities were called allowing a complete memory utilization behaviour report. The final inspection summary is described in Fig. 5.19.

The heap summary shows that 12.386 bytes are under used when the test is over (when PriSIS process was explicitly killed). The leak summary shows that at the moment that PriSIS was killed 12.114 bytes are still reachable. This reachable bytes are the required working memory, such as CSP buffers allocated from `csp_buffer_init(20,300);` , etc. This 12.114 bytes are not critical, taking into account that PriSIS was hard killed with `sigkill` signal, which avoids the software to release its working memory at that particular time. There are no definitely and indirectly lost memory and that is a good sign, which will approve the intuition on critical memory leakage inexistence previously discussed on Fig. 5.7 and Fig. 5.9 commentaries.

Finally, the last test made was the space-link fuzz test. This test consists in injecting random data through the space-link targeting the PriSIS software. This random data is encapsulated into AX.25 UI

```
==21233== HEAP SUMMARY:
==21233==     in use at exit: 12,386 bytes in 33 blocks
==21233==   total heap usage: 980 allocs, 947 frees, 133,613 bytes allocated
==21233== LEAK SUMMARY:
==21233==    definitely lost: 0 bytes in 0 blocks
==21233==    indirectly lost: 0 bytes in 0 blocks
==21233==      possibly lost: 272 bytes in 2 blocks
==21233==    still reachable: 12,114 bytes in 31 blocks
==21233==         suppressed: 0 bytes in 0 blocks
```

Figure 5.19: Valgrind/memcheck PriSIS report.

frames that will later be decoded by PriSIS. To automate this test one simple bash script, described in Fig. 5.20, was made. This script was issued from the GS together with the telemetry decoder running, allowing the satellite beacon gathering while the fuzz test is running.

```
#!/bin/bash
while [ 1 ]; do
        /usr/sbin/beacon -s -d "CS5CEP-11" spacelink \
        "‘cat /dev/urandom | head -c 256‘" 2>/dev/null;
done
```

Figure 5.20: Automated fuzz test.

This bash script collects 256 bytes from the `urandom`, which is a Linux random data generator device. This 256 bytes are passed to the AX.25 beacon tool, which is a simple way of sending UI frames with custom messages through the spacelink. The infinite loop was executed multiple times within different periods: 60, 300, 600 and 1200 seconds. After each period, the PriSIS operational status was checked using the GS software, sending one CSP command and expecting one response/ACK from it. This stress test that also mimics high space-link noise, shows that PriSIS demonstrates resilience against this stressful scenario acting as expected, discarding all the malformed CSP packets, and continuing with an healthy operation status. The collected telemetry also confirms the proper operation in terms of RAM usage and CPU loads.

5.3 Discussion

Clearly this chapter gives more attention to the developed COM software, mainly the CSP/T-CSP protocol implementations (PriSIS) and the on-board beacon software (pulsar). The COM subsystem deserved more attention because it is where most of the software developments were made and also because it is very difficult to test the C&DH subsystem due to the lack of debugging options and test tools available.

Another difficult task was the test formulations taking into account some prior expertise found on the specific documentation for safety/critical software development. Among all the documentation found, it was necessary to choose and tailor some information subsets in order to meet the Heart unit project task deadlines.

Beyond these developed tests, the Heart unit lacks integration tests. These integration tests must inspect how well the C&DH and COM interacts with the remaining subsystems, such as the future EPS and ADCS.

There were some extra worries on the entire solution compression/miniaturization which can, among other benefits, keep good power energy consumption efficiency. Also there are some external physical phenomena that were not considered since it was assumed that the physical cubesat structure can serve as a proper shield against all possible hazardous events, such as high cosmic radiation.

6

Conclusion and future work

The developed Heart unit presents a solution capable of handling the space-link communications between the ISTNanosat-1 and potential Ground Stations on Earth (Communications subsystem). It also presents an ultra low power solution capable of acting as the satellite central decision unit (C&DH subsystem). Both solutions were engineered taking into account the low space and power budget available on-board. Actually, these Cubesat intrinsic restrictions shaped the entire hardware and software developments. On one hand, to meet the Heart unit requirements for the Communications subsystem it was necessary to fully parametrize a GNU/Linux Operating System trying to minimize as much as possible the scarce on-board resources utilization. On top of this GNU/Linux Operating System it was developed the communications software (Primary Satellite Interface Software) which implements the space-link network protocol details. The developed protocols allow a GS compatible telemetry beacon transmission, command reception from GS providing the required acknowledgements and imagery transmission from the CubeSat. This PriSIS software implements the developed Amateur X.25/CubeSat Space Protocol driver as well as the new transport protocol called Tolerant-CSP. On the other hand, to meet the C&DH subsystem requirements it was necessary to port the FreeRTOS version for TI MSP430F5 MCU with the MSPGCC toolchain for the moteists5++ platform. After the moteists5++ FreeRTOS port was

done, it was developed an energetic-awake solution which is capable of communicate with the remaining subsystems using the system bus and implement, on future projects, the intended decision making process, using the deployed real time scheduler.

Extra time was spent analysing the best COM solution. The first intention was to use the rt-preempt patch[1] over the COM Linux kernel to transform it into a full Real-Time Operating System (RTOS). Unfortunately, this patch is not available for the last AT91 supported kernel version and the tests made with cross versions have shown a heavy system performance penalty.

It was thought that a Delay Tolerant Network solution over the COM subsystem may be useful for a future Inter-Satellite Links usage. Three different implementations were tried, DTN2, IBRDTN and ION. The DTN2 was infeasible to utilize on top of the designed system because it requires the OASYS library which depends on TCL development version library. This TCL library is not ported to the buildroot packages, so the attempt to port it as buildroot failed due to the lack of available storage. The IBRDTN was also not suitable since it requires the C++ support aboard that also require too much storage availability. The last attempt was the ION implementation. This last implementation is not ported for μClibc. With all these issues the DTN implementation solution was abandoned. Clearly the lacking of available storage on-board leads to a more requirement-oriented protocol design leaving out this extra ISL functionality.

The COM performance tests suggests that maybe more energy consumption can be saved if the processor was less powerful, since it is underused by the entire deployed software solution. But, if the DTN idea for ISL remains a open question for future ISTNanosat-1 developments, more CPU power and storage memory is required to host the actual DTN implementations demands. These implementations are not well supported by real embedded system infrastructures yet. To use for example, IBRDTN the COM operating system should be the OpenWRT where this implementation is well ported. This forces the COM OS to run a more common OS in order to run these actual DTN implementations.

The test chapter addressed the safety and quality aspects of the developed work. It states that the final solution fit on the proposed tight requirements. The most problematic requirement was the energy consumption. The energy available influenced almost every decision taken on this work. The final Heart solution roughly consumes 180mW if both subsystems are operating at same time. This power consumption characteristic was mainly achieved by overall low resource usage.

One of the Heart major contributions was the development of the AX.25/CSP driver. This driver can easily be integrated into another CSP based projects since it was developed as a CSP extension without any extra software dependencies, except the AX.25-library. The T-CSP protocol was another achievement. This transparent API, abstracts all the underlying transport functionalities to the application. The retransmission mechanism developed for this transport protocol proved that its use is time advantageous over disruptive scenarios in common space-link bit-rates. The T-CSP also allows an orbit-aware retrans-

[1] https://rt.wiki.kernel.org/index.php/Main_Page

mission mechanism. To enable all the AX.25 network stack inside the COM subsystem two buildroot packages were developed and are widely available. These packages automates the compilation, patching and installing of theAX.25-library and AX.25-tool inside the buildroot infrastructure. These can be useful for another radio embedded systems that needs AX.25 support, for example a portable Software-Defined Radio device. Another major contribution is the FreeRTOS port for the moteists5++ platform, which relies on a free and opensource toolchain (GCC). This port allows a complete usage of all moteist capabilities since this development chain does not restrict the amount of code produced (the commercial toolchains do).

The developed Heart unit relies on a external redundant profile switching. This complex task can be achieved on future developments to ensure a complete power-aware on-demand satellite functionalities. The entire solution also needs to be subjected to a more realistic test scenario to attest the observed performance and quality results. Another interesting future study might be how can a more common hardware/software platform be energetically efficient and at the same time support the actual ideas for ISL using some DTN implementation. Finally, the security issues are still an open question. Future developments can use the CSP intrinsic security mechanisms to employ e.g. remote command authentication, avoiding unauthorized satellite interaction.

A

Software installation guide

A.1 COM Operating System installation and tailoring

A.1.1 Cross-compile enviroment

In order to cross-compile the Linux kernel on a x86 host machine is necessary to have an ARM-enable toolchain which provide all the needed tools (compilers, assemblers, linkers, etc.) to produce ARM-based binaries. This toolchain can become available on a Debian machine by installing the apt packages described on Fig. A.1.

```
$sudo apt-get install linux-libc-dev-arm-cross gcc-4.6-arm-linux-gnueabi-base
$sudo apt-get install libc6-arm-cross libc6-dev-arm-cross
$sudo apt-get install uboot-mkimage
```

Figure A.1: Installation of cross-compilation tools on a x86 Debian host

A.1.2 COM base system

With the toolchain installed on host system, the Linux/ARM kernel can be compiled in a quasi-standard way, like:

```
1. ~/linux-2.6.38$ patch -p1 < 2.6.38-at91.patch
2. ~/linux-2.6.38$ make ARCH=arm at91rm9200dk_defconfig
3. ~/linux-2.6.38$ make ARCH=arm menuconfig
4. ~/linux-2.6.38$ make ARCH=arm CROSS_COMPILE=arm-linux- uImage
```

Figure A.2: Linux/ARM kernel cross-compilation on a x86 Debian host

On the first step in the Fig. A.2 above, the AT91 patch is applied to the code. It makes the proper changes (step 2) to the Linux kernel in order to support the AT91RM9200 MCU and the intended peripherals. The configurations and the code modifications made by this patch are not enough to full fill the required COM architecture.

The last step on Fig. A.2 (step 4) finally builds the kernel targeting the ARM architecture, producing an uImage file. This uImage is a typical kernel image (bzImage) but converted using the uboot-mkimage tool. In this uImage format, the Linux kernel image is ready to be used on the U-boot loader.

In order to build a fully parametrized ARM-based rootFS, the buildroot[1] infrastructure was used. The Fig. A.3 shows the fundamental required steps in order to configure and compile the rootFS image:

```
1. ~/buildroot-2012.05$ make menuconfig
2. ~/buildroot-2012.05$ make
```

Figure A.3: COM rootFS cross-compilation targeting ARM architecture

On the first step in the Fig. A.3 the framework will pop-up an GNU ncurses/dialog based menu, where the different buildroot configurations can be done. Finally, the step 2 on the Fig. A.3 will produce a 1.5 Mbyte final image on buildroot-2012.05/output/images/ path.

The required u-boot configuration are the follows:

The first two configuration directives copy the kernel and rootFS images from flash memory to SRAM using the proper memory locations. Note that 0xC0 is the flash and 0x21 is the SRAM regions. The third directive is the boot arguments that are passed to the Linux kernel indicating from where it should load the rootFS image (0x21118f0b0 in this case). Another extra options are also passed such as the

[1] http://buildroot.uclibc.org/

```
1. kernelMove=cp.b 0xC0000000 0x21000000 0x18f0a0
2. userlandMove=cp.b 0xC018f0b0 0x2118f0b0 0x1c4a1b
3. bootargs=root=/dev/ram rw initrd=0x2118f0b0,10m ramdisk_size=15360 console=ttyS0
,115200 mem=32M
4. bootcmd=run kernelMove; run userlandMove; bootm 0x21000000
```

Figure A.4: Das U-boot configuration directives

debug console device (ttyS0) and its baudrate. Finally, on step 4, the bootcmd directive instructs the bootloader which are the required steps with its respective execution order. The last instruction on this step 4 will trigger the entire system boot by executing the loaded Linux kernel.

The original AX.25-library and AX.25-tools source code use the GNU autotools as standard compilation process. After that, the ax25-lib package was incorporated into the buildroot infrastructure by adding it to the Config.in meta-file in packages/ branch. The Fig. A.5 shows the created .mk file major contents, for AX.25 library.

```
LIBAX25_VERSION = 0.0.12-rc2
LIBAX25_SOURCE = libax25-$(LIBAX25_VERSION).tar.gz
LIBAX25_SITE = http://www.linux-ax25.org/pub/libax25/
LIBAX25_INSTALL_STAGING = YES
LIBAX25_INSTALL_TARGET = YES
$(eval $(call AUTOTARGETS,package,libax25))
```

Figure A.5: Buildroot AX.25-library mk package

The meta-information included in this mk file will be used by the buildroot infrastructure to automatically compile and install the AX.25-library into the rootFS output image. The mk file have the original library code location on the web together with its version. The buildroot process to install this library into the system will perform the following tasks:

1. Read the meta-information on the package mk file;
2. Fetch the original library code based;
3. Apply the code correction patch to make the original library code compatible with the buildroot infrastructure;
4. Compile the library, invoking the autotols buildroot macro (last line in Fig. A.5);
5. Install the library in the rootFS file.

With this AX.25 protocol full enabled on the rootFS, the embedded system can now configure its serial line to be used as AX.25 interface. To do this, the following line was added to the `/etc/ax25/axports` configuration file:

```
spacelink CS5CEP-11 9600 255 2 SAT to GS Link
```

Figure A.6: AX.25 axports configuration file contents

This configuration line contains the serial connection name, the AX.25 callsign associated with the embedded system, the serial line baudrate (9600 bit/s in this case), max frame size and frame transmission window (which both can be ignored when UI frames are used) and a connection description. After the configuration setup, the command described on Fig. A.7 apply it to the kernel logic network interface.

```
/usr/sbin/kissattach /dev/ttyS0 spacelink 10.10.10.10
```

Figure A.7: Apply axports configurations to logical AX.25 interface

The kissattach command tells the kernel which physical interface should be used to send/receive the encoded AX.25 frames (ttyS0 in this case) and the correspondent configuration line on axports file (spacelink configuration in this case). The IP address as last arguments, is only intended for IP over AX.25 networks thus can be ignored in this scenario. After this command, the kernel generates a logic interface (e.g. ax0) that can be directly used as a typical Berkeley socket API together with the installed AX.25-library. All the developed communication services rely on this kernel logical interface, to perform the AX.25 UI framing process.

A.1.3 PriSIS interface interaction

Fig. A.8 illustrates the highlighted code on the developed GS side software, which implements the same Berkeley socket, but for receiving the AX.25 UI frames. After the corrected frame reception, its payload is ready to be decoded/parsed.

```
1. s = socket(PF_PACKET, SOCK_PACKET, htons(proto))
2. size = recvfrom(s, buffer, sizeof(buffer), 0, &sa, &asize);
```

Figure A.8: Receive data using Berkeley AX.25-based socket

A.2 Additional notes on complexity

A.2.1 PriSIS and pulsar compilation

The buildroot infrastructure was configured to produce a parametrized rootFS image serving as underlying platform together with the Linux ARM kernel for the developed software - PriSIS and pulsar. After the rootFS compilation it was necessary to cross-compile and install this two software components (PriSIS and pulsar) inside the generated buildroot image. To automate this entire process, the `configureImage` shell script was created. The major steps performed by this tool can be summarized as follows:

1. Open the produced rootFS image, decompressing and mounting it;
2. Copy the AX.25 configuration files and startup scripts that will run both PriSIS and pulsar on system boot;
3. Cross-compile to ARM architecture and install the libcsp library with the new AX.25 driver inside the rootFS image. Note that this EABI-based cross-compilation was done using the buildroot toolchain using:
   `--toolchain=buildroot-2011.11/output/host/usr/bin/arm-linux` and `--enable-if-ax25` as new waf script parameter. Recall that this toolchain will use the μClibc library.
4. Compile the PriSIS-ARM together with the developed T-CSP library and linked with previously installed on rootFS libax25 and libcsp. Note that libax25 was previously complied and installed using the developed buildroot libax25 package
5. Copy the resultant binaries (PriSIS-ARM and pulsar-ARM) into the rootFS image;
6. Unmount and compress back (with bzip2 algorithm) the final rootFS solution, ready to be burned into on-board flash memory.

A.2.2 COM flash memory programming

In order to program the COM board with the developed software an Trivial File Transfer Protocol (TFTP) server was created on hos system (outside the embedded system). This TFTP server stored the intended COM flash embedded files (rootFS image and kernel image). Using the bootloader TFTP client capabilities each image is copied from the external TFTP server to the on-board RAM, which are later burned into the flash memory. This was the best way found to pass the developed software to the COM subsystem.

Bibliography

[1] J. Bouwmeester and J. Guo, “Survey of worldwide pico- and nanosatellite missions, distributions and subsystem technology,” Acta Astronautica, vol. 67, no. 7-8, pp. 854–862, Oct. 2010. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0094576510001955

[2] W. A. Beech, D. E. Nielsen, and J. Taylor, “AX. 25 Link Access Protocol for Amateur Packet Radio,” Tucson Amateur Packet Radio Corporation, Tucson, no. July, 1998. [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:AX+.+25+Link+Access+Protocol+for+Amateur+Packet+Radio#0

[3] L. Wood, W. Eddy, C. Smith, W. Ivancic, and C. Jackson, “Saratoga: A Scalable Data Transfer Protocol (Internet Draft),” 2011. [Online]. Available: http://tools.ietf.org/html/draft-wood-tsvwg-saratoga-10

[4] M. Davidoff, The Radio Amateur’s Satellite Handbook, 1st ed. Newington: The American Radio Relay League, 2003.

[5] M. Swartwout, “The promise of innovation from university space systems: are we meeting it,” in Proceedings of the 23rd AIAA/USA Small Satellites Conference, Logan, USA, 2009, pp. 1–6. [Online]. Available: http://www.usu.edu/ust/pdf/2009/october/itn10120930.pdf

[6] S. Lee, A. Hutputanasin, A. Toorian, W. Lan, and R. Munakata, “CubeSat Design Specification, Rev. 12,” 2009. [Online]. Available: http://www.cubesat.org/images/developers/cds_rev12.pdf

[7] A. Bonnema, “ISIS Missions, Services and Technology Trends,” in CubeSat Summer Workshop, Logan USA, 2011, p. 21.

[8] J. Bluck, “GeneSat-1 - Mission overview,” 2007. [Online]. Available: http://www.nasa.gov/centers/ames/missions/2007/genesat1.html

[9] K. Woellert, P. Ehrenfreund, A. J. Ricco, and H. Hertzfeld, “Cubesats: Cost-effective science and technology platforms for emerging and developing nations,” Advances in

Space Research, vol. 47, no. 4, pp. 663–684, Feb. 2011. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0273117710006836

[10] V. Cerf, Google/JPL, S. Burleigh, A. Hooke, L. Torgerson, NASA/JPL, R. Durst, K. Scott, The MITRE Corporation, K. Fall, Intel Corp, H. Weiss, and I. SPARTA, "rfc4838 - Delay-Tolerant Networking Architecture," 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4838.txt

[11] V. Cerf, "InterPlaNetary Internet," in DARPA Proposer's Day, vol. 26, no. 6, Nov. 2004, p. 17. [Online]. Available: http://symoon.free.fr/scs/dtn/biblio/Cerf-IPN-DARPA.pdf

[12] V. Cerf and I. Society, "rfc3271 - The Internet is for Everyone," 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3271.txt

[13] A. Huurdeman, The Worldwide History Of Telecommunications. New Jersey, Canada: Wiley Online Library, 2003. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/0471722243.fmatter/summary

[14] L. Summerer and L. Jacques, "Prospects For Space Solar Power In Europe," in International Astronautical Congress. Cape Town: IAF, 2011, p. 4.

[15] E. Gill, P. Sundaramoorthy, J. Bouwmeester, B. Sanders, and C. Science, "Formation flying to enhance the qb50 space network," in Small Satellite Systems and Services Symposium, no. June, Funchal, Portugal, 2010, p. 3.

[16] A. Toorian, K. Diaz, and S. Lee, "The cubesat approach to space access," in Aerospace Conference, 2008 IEEE, vol. 1, no. 1. IEEE, 2008, pp. 1–14. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4526293

[17] European Space Agency - Education Office, "Educational Payload on the Vega Maiden Flight," 2008. [Online]. Available: http://esamultimedia.esa.int/docs/LEX-EC/CubeSat_CFP_issue_1_rev_1.pdf

[18] C. S. Clark, "An Advanced Electical Power System For CubeSats," in European Small Satellite Services Symposium, no. June, Madeira, Portugal, 2010. [Online]. Available: http://www.clyde-space.com/documents/1807

[19] D. W. Miller and J. Keesee, "Spacecraft Power Systems - MIT/Satellite-Engineering course," 2005. [Online]. Available: http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-851-satellite-engineering-fall-2003/lecture-notes/l3_scpowersys_dm_done2.pdf

[20] S. Notani and S. Bhattacharya, "Flexible electrical power system controller design and battery integration for 1U to 12U CubeSats," in Energy Conversion Congress and Exposition (ECCE), 2011

IEEE. IEEE, 2011, pp. 3633–3640. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6064262

[21] M. Correia, A., Catela, J., Rocha, R. & Piedade, "Smart ultra low power energy harvesting system," Networked Embedded Systems: Special Issue International Journal of Adaptive, Resilient and Autonomic Systems, 2012.

[22] V. Francois-Lavet, "Study of passive and active attitude control systems for the OUFTI nanosatellites," Master Thesis, University of Liège, 2010. [Online]. Available: http://vincent.francois-l.be/OUFTI_ADCS_2010_05_31.pdf

[23] O. L. de Weck, "Attitude Determination and Control System (ADCS) - MIT/Satellite Engineering Course," Mar. 2001.

[24] R. Izadi-Zamanabadi and J. A. Larsen, "A fault Tolerant Control Supervisory System Development Procedure For Small Satellites - The AAUSAT-II Case." [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/cbdv.200490137/abstract

[25] G. Falbel, J. Puig-Suari, and A. Peczalski, "Sun oriented and powered, 3 axis and spin stabilized cubesats," in Aerospace Conference Proceedings, 2002. IEEE, vol. 1. IEEE, 2002, pp. 1–447. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1036864http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1036864

[26] S. Waydo, D. Henry, and M. Campbell, "CubeSat design for LEO-based Earth science missions," in IEEE Aerospace Conference, vol. 1. Ieee, 2002, pp. 1–435–1–445. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1036863

[27] B. R. Elbert, The Satellite Communication Applications Handbook, 2nd ed., Artech House, Ed., Norwood, 2004.

[28] J. Hales, M. Pedersen, and K. Krogsgaard, "Attitude Control and Determination System for DTUsat - a CubeSat contribution," TU of Denmark - Department of Automation, Tech. Rep. March 2002, 2002. [Online]. Available: http://microsat.sm.bmstu.ru/e-library/Algorithms/CommonDesign/dtu_torsten.pdf

[29] C. McNutt, R. Vick, H. Whiting, and J. Lyke, "Modular Nanosatellites–(Plug-and-Play) PnP CubeSat," in 7th Responsive Space Conference, Los Angeles, CA, 2009. [Online]. Available: http://www.urweb.tv/electronics/plugandplaySatellites/PlugandPlay3.pdf

[30] G. Manyak, "Fault Tolerant and Flexible CubeSat Software Architecture," Ph.D. dissertation, CalPoly - California Polytechnic State University, 2011. [Online]. Available: http://digitalcommons.calpoly.edu/theses/550/

[31] D. L. Bekker, T. a. Werne, T. O. Wilson, P. J. Pingree, K. Dontchev, M. Heywood, R. Ramos, B. Freyberg, F. Saca, B. Gilchrist, A. Gallimore, and J. Cutler, "A CubeSat design to validate the Virtex-5 FPGA for spaceborne image processing," in 2010 IEEE Aerospace Conference. Ieee, Mar. 2010, pp. 1–9. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5446700

[32] S. D. Jong and G. Aalbers, "Improved Command and Data Handling system for the Delfi-n3xt Nanosatellite," in 59th International Astronautical Congress, no. October, Glasgow, 2008. [Online]. Available: http://www.lr.tudelft.nl/fileadmin/Faculteit/LR/Organisatie/Afdelingen_en_Leerstoelen/Afdeling_SpE/Space_Systems_Eng./Publications/2008/doc/IAC-08.D1.4.11_Jong_IMPROVED_CDHS_n3Xt.pdf

[33] B. Klofas, J. Anderson, and K. Leveque, "A Survey of CubeSat Communication Systems," The AMSAT Journal, no. November/December, p. 25, 2009.

[34] B. K. Kfzeo and J. A. Kigiv, "A Survey of CubeSat Communication Systems," 2008.

[35] S. Heath, Embedded Systems Design, 2nd ed., Newnes, Ed., 2003.

[36] Indian Institute Of Technology, "Jugnu IIT Kanpur Nanosatellite."

[37] K. Avery, J. Finchel, J. Mee, W. Kemp, R. Netzer, D. Elkins, B. Zufelt, and D. Alexander, "Total Dose Test Results for CubeSat Electronics," in Radiation Effects Data Workshop (REDW), 2011 IEEE. IEEE, 2011, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6062504

[38] Diamond Systems Corporation, "Earthquake Research Satellite Demonstrates Ruggedness of Diamond's SBCs," 2011. [Online]. Available: http://www.diamondsystems.com/articles/13

[39] A. Kalman, A. Reif, D. Berkenstock, J. Mann, and J. Cutler, "MISC ™ – A Novel Approach to Low-Cost Imaging Satellites," in Conference on Small Satellites, 2008.

[40] Z. Jacobs, K. Morgan, M. Caffrey, J. Palmer, and L. Ho, "LANL CubeSat Reconfigurable Computer (CRC)," in CubeSat Summer Workshop, 2010, pp. 1–14. [Online]. Available: http://mstl.atl.calpoly.edu/~bklofas/Presentations/SummerWorkshop2010/Jacobs-reconfigurable_computer.pdf

[41] H. Heidt, J. Puig-Suari, A. Moore, S. Nakasuka, and R. Twiggs, "CubeSat: A new generation of picosatellite for education and industry low-cost space experimentation," in Proceedings of the Thirteenth Annual AIAA/USU Small Satellite Conference, Logan, UT, 2000. [Online]. Available: http://www.space.aau.dk/cubesat/documents/CubeSat_Paper.pdf

[42] G. Manyak and J. M. Bellardo, “PolySat’s Next Generation Avionics Design,” 2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology, pp. 69–76, Aug. 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6007777

[43] A. Silberschatz, G. Gagne, and P. Baer Galvin, Operating System Concepts, 7th ed., J. W. . S. Inc, Ed., USA, 2005.

[44] L. Beus-Dukic, “Criteria for Selection of a COTS Real-Time Operating System: a Survey,” European Space Agency-Publications-ESA SP, vol. 457, pp. 387–394, 2000. [Online]. Available: http://users.wmin.ac.uk/~beusdul/papers/dasia00.pdf

[45] IEEE-SA Standards Board, “IEEE Standard for Information Technology — Standardized Application Environment Profile — POSIX ® Realtime Application Support (AEP),” The Institute of Electrical and Electronics Engineers, Tech. Rep., 1998. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=798785

[46] R. P. Karn, H. E. Price, and R. J. Diersing, “Packet radio in the amateur service,” IEEE Journal on Selected Areas in Communications, vol. 3, no. 3, pp. 431–439, 1985. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1146214

[47] J. McGuire, I. Galysh, K. Doherty, H. Heidt, and D. Neimi, “FX.25 - Forward Error Correction Extension to AX.25 Link Protocol For Amateur Packet Radio,” pp. 1–10, 2006.

[48] K. Nakaya, K. Konoue, H. Sawada, and K. Ui, “Tokyo Tech CubeSat: CUTE-I-Design & Development of Flight Model and Future Plan,” AIAA 21st International, pp. 1–10, 2003. [Online]. Available: http://www.aric.or.kr/treatise/journal/content.asp?idx=48512

[49] T. I. O. T. Laboratory For Space Systems (LSS), “SRLL(Simple Radio Link Layer).” [Online]. Available: http://lss.mes.titech.ac.jp/ssp/cubesat/srll/srll.htm

[50] J. D. Nielsen and J. A. Larsen, “A decentralized design philosophy for satellites,” in Recent Advances in Space Technologies (RAST), 2011 5th International Conference on. IEEE, 2011, pp. 543–546. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5966896

[51] GomSpace, “CubeSat Space Protocol - Network-Layer delivery protocol for CubeSats and embedded systems,” 2011. [Online]. Available: http://gomspace.com/documents/GS-CSP-1.1.pdf

[52] K. Scott, The MITRE Corporation, S. Burleigh, and NASA Jet Propulsion Laboratory, “RFC5050 - Bundle Protocol Specification,” 2007. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5050.txt

[53] L. Wood, J. McKim, W. Eddy, W. Ivancic, and C. Jackson, “Using Saratoga with a Bundle Agent as a Convergence Layer for Delay-Tolerant Networking,” 2010. [Online]. Available: http://tools.ietf.org/html/draft-wood-dtnrg-saratoga-07

[54] I. Rutter, T. Vladimirova, and H. Tiggeler, “A CCSDS Software System for a Single-Chip On-Board Computer of a Small Satellite,” in 15th Annual/USU Conference on Small Satellites, Utah, 2001. [Online]. Available: http://personal.ee.surrey.ac.uk/Personal/T.Vladimirova/Publications/UTAH01_paper.pdf

[55] Consultative Committee for Space Data Systems, “CCSDS - Recomendation for Packet Telemetry,” 2000. [Online]. Available: http://public.ccsds.org/publications/archive/102x0b5s.pdf

[56] R. C. Durst, P. D. Feighery, and M. J. Zukoshi, “User ’ s Manual for the SCPS Reference Implementation Software,” 1997. [Online]. Available: http://yazzy.org/docs/SCPS/SCPS_RI_1_1_113/docs/ri.pdf

[57] R. Durst, “Space Communications Protocol Standards Overview,” 1998. [Online]. Available: http://www.scps.org/Documents/SCPSoverview.PDF

[58] T. Vladimirova, X. Wu, and C. Bridges, “Development of a Satellite Sensor Network for Future Space Missions,” in Aerospace Conference, 2008 IEEE. Ieee, 2008, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4526248

[59] K. Vega, D. Auslander, and D. Pankow, “Design and modeling of an active attitude control system for cubesat class satellites,” in AIAA Proceedings.[np]., no. August, 2009. [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Design+and+Modeling+of+an+Active+Attitude+Control+System+for+CubeSat+Class+Satellites#0

[60] V. Lappas, N. Adeli, L. Visagie, J. Fernandez, T. Theodorou, W. Steyn, and M. Perren, “CubeSail: A low cost CubeSat based solar sail demonstration mission,” Advances in Space Research, vol. 48, no. 11, pp. 1890–1901, Jun. 2011. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0273117711003991

[61] J. Fitzpatrick, “An interview with Steve Furber,” Communications of the ACM, vol. 54, no. 5, pp. 34–39, 2011.

[62] ATMEL, “AT91RM9200 Datasheet,” ATMEL, Tech. Rep., 2009. [Online]. Available: http://www.atmel.com/Images/doc1768.pdf

[63] NASA - National Aeronautics and Space Administration, NASA Software Safety Guidebook (GB-8719.13), nasa-gb-87 ed., 2004.