
# Politecnico di Torino

Master's Degree in Electronic Engineering - Sistemi Elettronici (Electronic Systems)
Academic Year 2022/2023
October 2023

# Development of a versatile on board computer for small satellites

Supervisors:
Prof. Fabrizio Stesina (DIMEAS)
Prof. Sabrina Corpino (DIMEAS)

Candidate:
Simone Bollattino

# Abstract

The present thesis is about the design of an on board computer system for the Spei Satelles space mission. Turin Polytechnic has a rich history of CubeSat launches and on 2023 it launched a 3 units (3U) CubeSat mission under the name of Spei Satelles, promoted by the Dicastery for Communication of Vatican City in collaboration with the Italian Space Agency (ASI) and Consiglio Nazionale delle Ricerche (CNR).

The system in question has the goal to collect telemetry and scientific data about the spacecraft and space environment, store it inside non volatile memory and prepare it to be sent back to earth, where it will be used to check the spacecraft health and verify thermal and attitude models theorized during the design phase; a secondary achievement is to demonstrate the survivability in the space environment of a low-cost system completely built from Components Off The Shelf (COTS).

The thesis starts with a review of the state-of-the-art of CubeSat applications, focusing on aspects about space electronics; it gives a brief overview of the spacecraft mission and architecture; then covering all the subsystem design phases: from requirements and interfaces definitions to system design and manufacturing, testing, integration and post-launch operations; finally it presents the mission results and achievements.

# Table of Contents

**List of Abbreviations** **IX**

**List of Figures** **XIII**

**List of Tables** **XXI**

**1 Introduction to CubeSat** **23**
1.1 The CubeSat standard . . . 23
1.2 CubeSats evolution . . . 24
1.2.1 Some words on secondary payloads . . . 25
1.2.2 2000s: CubeSats introduction . . . 25
1.2.3 2010s: Growing interest on CubeSats . . . 26
1.2.4 2014: CubeSat constellations and dedicated rideshares . . . 27
1.2.5 2018: Deep space missions . . . 30
1.3 Polytechnic of Turin CubeSat heritage . . . 31

**2 CubeSat platforms** **33**
2.1 Spacecraft BUS . . . 33
2.2 CubeSat Interfaces . . . 34
2.2.1 PC/104 form factor . . . 34
2.2.2 Backplane board . . . 35
2.2.3 Daughter boards . . . 35
2.3 Electronics in space . . . 36
2.3.1 Radiation effects on electronics . . . 37
2.3.1.1 Single Event Latchup (SEL) . . . 38

**3 Spei Satelles** **41**
3.1 Motivations and mission . . . 41
3.2 Spacecraft architecture . . . 42
3.2.1 Spacecraft subsystems . . . 43
3.2.2 Spacecraft block diagram . . . 46
3.3 Spei Satelles team . . . 47

**4 Sensing Suite (Singer) subsystem** **49**

4.1 SPEISAT Secondary mission . . . 49
4.2 System requirements and specifications . . . 50
4.2.1 Functional requirements . . . 51
4.2.2 Performance requirements . . . 53
4.2.3 Interface requirements . . . 54
4.3 Design process . . . 55
4.3.1 Challenges . . . 55
4.3.1.1 Low development time . . . 55
4.3.1.2 Poor system reliability . . . 56
4.3.1.3 Reduced impact on primary mission . . . 56
4.3.1.4 Cost . . . 56
4.3.2 Design timeline . . . 57

**5 Hardware design** **59**
5.1 System architecture . . . 59
5.2 Electrical design . . . 60
5.2.1 ADC block . . . 60
5.2.1.1 Measurement chain characterization . . . 61
5.2.2 IMU block . . . 68
5.2.2.1 Gyroscope specifications . . . 69
5.2.2.2 Magnetometer specifications . . . 70
5.2.2.3 IMU reference plane . . . 70
5.2.3 Memory block . . . 71
5.2.4 RS422 block . . . 73
5.2.5 Processing unit . . . 73
5.2.5.1 Clock and peripherals assignment . . . 75
5.2.6 Power block . . . 76
5.2.6.1 Voltage regulator . . . 77
5.2.6.2 Domain protection circuits . . . 77
5.2.6.3 Domain interfaces . . . 84
5.2.7 Board electrical interfaces and connectors . . . 85
5.2.8 System power estimation . . . 88
5.3 PCB design and production . . . 89
5.3.1 Schematic . . . 90
5.3.2 Layout . . . 91
5.3.2.1 Floorplan . . . 93
5.3.2.2 Routing . . . 93
5.3.3 Production . . . 95

**6 Software design** **97**
6.1 Software overview . . . 97
6.1.1 development environment . . . 98
6.1.2 FreeRTOS . . . 98
6.2 Low level drivers . . . 100
6.2.1 Interfaces review . . . 101

6.2.1.1 SPI interface . . . 101
6.2.1.2 UART interface . . . 103
6.2.1.3 SWD interface . . . 104
6.2.2 HAL drivers . . . 104
6.2.3 UART drivers . . . 104
6.2.3.1 Peripheral drivers . . . 104
6.2.4 Utility libraries . . . 106
6.2.4.1 Buffer utilities . . . 106
6.2.4.2 Packet utilities . . . 107
6.2.5 IMU driver . . . 111
6.2.5.1 Xbus protocol and IMU op-modes . . . 111
6.2.5.2 Driver library . . . 113
6.2.6 Parrot communication library . . . 114
6.2.6.1 Parrot message frame . . . 114
6.2.6.2 Parrot communication library . . . 114
6.2.7 printf() implementation . . . 115
6.2.8 ADC driver . . . 115
6.2.8.1 AD7788 interface . . . 115
6.2.8.2 ADC logical wiring . . . 116
6.2.8.3 Driver library . . . 117
6.2.9 MRAM driver and log system . . . 117
6.2.9.1 AS301604 interface . . . 118
6.2.9.2 Low level driver . . . 120
6.2.9.3 Memory organization and log system . . . 121
6.3 High level software . . . 124
6.3.1 Inter-task communication . . . 125
6.3.1.1 Telemetry exchange . . . 125
6.3.1.2 Monodirectional requests . . . 126
6.3.1.3 Mutual exclusive access . . . 126
6.3.2 Sensors task . . . 127
6.3.3 Memory task . . . 128
6.3.4 Parrot task . . . 129
6.3.4.1 Available Parrot messages . . . 129
6.3.4.2 Task code structure . . . 131
6.3.5 Watchdog task . . . 132
6.3.6 Tasks timeout management . . . 133
6.3.7 Firmware compile modes . . . 134
6.3.7.1 Access Port debug console . . . 134
6.3.7.2 Time measurement mode . . . 134
6.3.8 Final firmware configuration . . . 135

**7 Test campaign, integration and launch 137**
7.1 Ground Support Equipment . . . 139
7.2 Development tests . . . 140

7.2.1 Breadboard model . . . 140
7.2.2 Debug console and GDB . . . 141
7.2.3 Parrot mock-up . . . 141
7.2.4 IMU bridge . . . 142
7.3 Acceptance tests . . . 142
7.3.1 Power test . . . 143
7.3.2 MRAM test . . . 144
7.3.3 RS422 lines test . . . 145
7.3.4 ADC test . . . 146
7.3.5 IMU test . . . 147
7.4 Flight preparation and integration . . . 148
7.4.1 Hardware corrections . . . 148
7.4.2 Nucleo board modifications . . . 149
7.4.3 Flight thermistors preparation . . . 150
7.4.4 Integration . . . 150
7.5 Functional tests . . . 152
7.5.1 Full functional and day-in-the-life tests. . . . 152
7.5.2 Mechanical fit and vibrational tests . . . 154
7.5.3 Thermal cycling test . . . 155
7.6 Laboratory failures . . . 155
7.6.1 Qualification model failure . . . 155
7.6.2 Flight model failure . . . 156
7.7 Spacecraft shipping . . . 159

**8 Mission results** **161**
8.1 Launch . . . 161
8.2 Mission operation . . . 162
8.2.1 Ground station . . . 162
8.2.2 Analysis of data . . . 164

**9 Conclusions** **171**

**Bibliography** **175**

**Technical documents** **183**

**Acknowledgements (Italian)** **187**

# List of Abbreviations

**a.u.** Arbitrary Unit
**ACK** Acknowledge
**ACS** Attitude Control System
**ADC** Analog to Digital Converter
**ADCS** Attitude Determination and Contol System
**ADS** Attitude Determination System
**AIV** Assembly, Integration and Verification
**AMSAT** Radio Amateur Satellite Corporation
**AODCS** Attitude and Orbit Determination and Control System
**AP** Access Port
**ARI** Associazione Radioamatori Italiani
**ARM** Advanced RISC Machine
**AS** Augmented Storage
**ASCII** American Standard Code for Information Interchange
**ASI** Agenzia Spaziale Italiana
**BB** BaseBand
**BGA** Ball Grid Array
**BID** Bus Identifier
**BJT** Bipolar Junction Transistor
**BOM** Bill Of Material
**BPB** Backplane Board
**CAD** Computer Aided Design
**CDH** Command and Data Handling
**CDS** Cubesat Design Specification
**CMOS** Complementary Metal-Oxide Semiconductor
**CNR** Centro Nazionale di Ricerche
**COMMSYS** Communication System
**COTS** Components Off The Shelf
**CPHA** Clock Phase
**CPOL** Clock Polarity
**CPU** Central Processing Unit
**CS** Chip Select
**DB** Database

**DDR** Double Data Rate
**DET** Direct Energy Transfer
**dev-board** development board
**DFN** Dual-Flat No-Leads
**DIMEAS** Department of Mechanical and Aerospace Engineering
**DMA** Direct Memory Access
**DNL** Differential Non Linearity
**DSP** Digital Signal Processor
**DSPI** Double Serial Peripheral Interface
**EDA** Electronic Design Automation
**EMI** Electromagnetic Interference
**EPS** Electric Power System
**ESA** European Space Agency
**ESD** Electro Static Discharge
**FIFO** First-In-First-Out
**fig.** Figure
**FM** Frequency Modulation
**FS** FlatSat
**FSW** Flight Software
**GCC** GNU Compiler Collection
**GDB** GNU DeBugger
**GEO** Geostationary Earth Orbit
**GMSK** Gaussian Minimum Shift Keying
**GND** Ground
**GPIO** General Purpose Input Output
**GS** Ground Station
**GSE** Ground Support Equipment
**GSFC** Goddard Space Flight Center
**GTO** Geostationary Transfer Orbit
**HAL** Hardware Abstraction Layer
**HSI** High Speed Internal
**HW** Hardware
**I/Q** In-phase and Quadrature
**I2C** Inter Integrated Circuit
**IAC** International Astronautical Congress
**IC** Integrated Circuit
**ID** Identifier
**IDE** Integrated Development Environment
**IEEE** Institute of Electrical and Electronics Engineers
**IF** Intermediate Frequency
**IMU** Inertial Measurement Unit
**INL** Integral Non Linearity
**IoT** Internet of Things
**ISR** Interrupt Service Routine

**IUSVE** Istituto Universitario Salesiano Venezia
**IWDG** Independent Watchdog
**JEDEC** Joint Electron Device Engineering Council
**LEO** Low Earth Orbit
**LET** Linear Energy Transfer
**LL** Low Level
**LSB** Least Significant Bit
**LSE** Low Speed External
**LSI** Low Speed Internal
**max** maximum
**MBU** Multiple-Bit Upset
**MID** Message IDentifier
**min** minimum
**MISO** Master In Slave Out
**MOSFET** Metal Oxide Semiconductor Field Effect Transistor
**MOSI** Master Out Slave In
**MPPT** Maximum Power Point Tracking
**MRAM** Magnetoresistive Random Access Memory
**MSB** Most Significant Bit
**MSI** Multi Speed Internal
**MUX** Multiplexer
**N/A** Not Applicable
**N/C** Not Connected
**NASA** National Aeronautics and Space Administration
**NATO** North Atlantic Treaty Organization
**NRO** National Reconnaissance Office
**NTC** Negative Temperature Coefficient
**OBC** On Board Computer
**OS** Operating System
**P-POD** Poly Picosatellite Orbital Deployer
**PC** Personal Computer
**PCB** Printed Circuit Board
**PLCC** Plastic Leaded Chip Carrier
**PLL** Phase Locked Loop
**PSLV** Polar Satellite Launch Vehicle
**PTC** Positive Temperature Coefficient
**QFN** Quad-Flat No-Leads
**QSPI** Quadruple Serial Peripheral Interface
**R&D** Research and Development
**RAM** Random Access Memory
**req.** Requirement
**RF** Radio Frequency
**RMS** Root Mean Square
**RTC** Real Time Clock

**RTOS** Real Time Operating System
**RX** Receive (or Reception)
**SCR** Silicon Controlled Rectifier
**SDR** Software Defined Radio
**SEB** Single Event Burnout
**SEE** Single Event Effect
**SEFI** Single Event Functional Interrupt
**SEGR** Single Event Gate Rupture
**SEL** Single Event Latchup
**SEU** Single Event Upset
**SIP** System In Package
**SLS** Space Launch System
**SMT** Surface Mount Technology
**SO** Small Outline
**SOIC** Small Outline Integrated Circuit
**SOM** System On Module
**SPEISAT** Spei Satelles
**SPI** Serial Peripheral Interface
**SRAM** Static Random Access Memory
**STO** Science and Technology Organization
**SWD** Serial Wire Debug
**tab.** Table
**TCB** Task Control Block
**TCS** Thermal Control System
**TID** Total Ionizing Dose
**TLE** Two Line Element set
**TRP** Task Repetition Period
**TTL** Transistor Transistor Logic
**TX** Transmit (or Transmission)
**typ** typical
**UART** Universal Asynchronous Receiver Transmitter
**uC** Microcontroller
**UHF** Ultra High Frequency
**USART** Universal Synchronous/Asynchronous Receiver/Transmitter
**USB** Universal Serial Bus
**UTC** Coordinated Universal Time

# List of Figures


1.1 Weight class of some standard and non standard CubeSats, picture from [91]. . . . 23
1.2 3U CubeSat (CSSWE) with P-POD dispenser, picture from [3]. . . . 24
1.3 Cubesats evolutionary tree, picture from [117]. . . . 24
1.4 Cubesat launches by type (1U,1.5U,2U,3U,6U,12U) from 2003 to 2023, data from [120]. . . . 25
1.5 CubeSats (1U,1.5U,2U,3U,6U,12U) launched by each launcher or family of launchers (denoted with "f.") in the period from 2003 to 2012 (left axis), the green trend line is the number of different launchers employed (right axis), data from [120]. . . . 26
1.6 Percentages of CubeSats (1U,1.5U,2U,3U,6U,12U) launched by type of institution, data from [120]. . . . 26
1.7 Percentages of launched CubeSats types (1U,1.5U,2U,3U,6U,12U) over the period from 2003 to 2023 (left axis), the blue trend line is the average size in Units (U) of launched CubeSats (right axis), data from [120]. . . . 27
1.8 Number of founded companies active in nanosats over the time period from 2003 to 2023, picture from [120]. . . . 28
1.9 Percentage of CubeSats mission type, picture from [90], pag. 12. . . . 28
1.10 CubeSats (1U,1.5U,2U,3U,6U,12U) launched by each launcher or family of launchers (denoted with "f.") in the period from 2013 to 2023, the green trend line is the number of different launchers employed (right axis), data from [120]. . . . 29
1.11 Cake graphs of mission status of all the CubeSats launched by each builder category, excluding constellations, at the end of 2015, data from [104]. . . . 30
1.12 Pictures from NASA's MarCO mission. 1.12a One of the twin CubeSats' solar array being tested. 1.12b Image of Mars captured by MarCO-B 6U CubeSat after the landing of InSight. Pictures from [60]. . . . 31
1.13 1.13a PicPoT satellite. 1.13b e-st@r-II CubeSat. 1.13c Spei Satelles CubeSat. 32

2.1 Family of PC/104 standards, picture from [80]. . . . 34
2.2 Example of PC/104 stack, picture from [80]. . . . 34
2.3 Example of PC/104 ADCS board developed at Polytechnic of Turin with mounted daughter boards (OBC and IMU), picture from [59]. . . . 35

2.4 Challenges of spacecraft electronics, data from [53]. . . . . 36
2.5 2.5a Cross section of CMOS inverter showing the parasitic SCR. 2.5b The equivalent circuit of the parasitic SCR. Both pictures from [49]. . . . . 38

3.1 Spei Satelles logo, designed by students of the IUSVE of Venice [87]. . . . . 41
3.2 3.2a Spei Satelles spacecraft. The access port can be seen in the middle of the left solar panel. 3.2b The nanobook glued to the satellite structure. . . 42
3.3 Main spacecraft building blocks and reference plane, solar panels not shown. 42
3.4 Spacecraft mechanical structure. . . . . 43
3.5 3.5a Solar panels after assembling. 3.5b DET board render. . . . . 44
3.6 Backplane board during integration . . . . . 45
3.7 Project render of the Ground Station at Polytechnic. . . . . 45
3.8 Spacecraft block diagram, green arrows are RF data links, blue arrows are wired data links, red arrows are wired power links. . . . . 46
3.9 SPEISAT Access Port, here the solar panels are not yet mounted. We can see the AP cable of Singer (down, gray) and one CDH (up, white). . . . . 47
3.10 Spei Satelles team composition. . . . . 47

4.1 Singer board renders. . . . . 49
4.2 Singer maximum mechanical dimensions (unit of measure for all dimensions: mm), all dimensions except the mounting holes diameter and spacing are the maximum allowable, mounting holes diameter and spacing have a tolerance of $\pm$ 0.1 mm. . . . . 55
4.3 4.3a Singer breadboard model. 4.3b Singer qualification model during testing. 4.3c Singer flight model, modifications on the Nucleo board for flight preparation can be spotted (components removal). . . . . 57
4.4 Spacecraft Flat Sat. . . . . 58
4.5 Singer development milestones. . . . . 58

5.1 Singer board block diagram. . . . . 59
5.2 Singer single ADC chain block diagram. . . . . 60
5.3 5.3a AD7788 $\Sigma$-$\Delta$ ADC block diagram, picture from [55]. 5.3b analog MUX pinout, picture from [4]. . . . . 61
5.4 5.4a NTCLE203E3103FB0 thermistor characteristic curve generated by MATLAB. 5.4b Thermistor resistance relative error. . . . . 62
5.5 Thermistor conditioning circuit. . . . . 62
5.6 Thevenin equivalent values with respect to temperature with nominal values of thermistor and pull-up. . . . . 63
5.7 Simulation pipeline for thermistor chain. . . . . 63
5.8 Conditioning circuit Thevenin equivalent and parasitic resistances of MUX and ADC. . . . . 64
5.9 Model of thermistor circuit in LTspice. . . . . 65
5.10 Comparison between exact temperature and the one reconstructed neglecting ADC input resistance (left axis). Absolute error introduced (right axis). 65

5.11 5.11a Temperature reconstructed considering tolerances(left axis). Maximum absolute error introduced (right axis). 5.11b Absolute error for each worst-case simulation. . . . . 66
5.12 Temperature reconstruction equation (left axis) and it's sensitivity with respect to measured voltage (right axis); vertical lines represent the measurement range of -40°C to 85°C, horizontal lines the corresponding sensitivities. 67
5.13 5.13a MTi-3 IMU compared to human finger. 5.13b MTi-3 pinout. . . . . 69
5.14 5.14a IMU plane origin and axis, modified picture from [64]. 5.14b IMU footprint on PCB. . . . . 70
5.15 AS3016204 MRAM pinout, picture from [42]. . . . . 71
5.16 LTC2852 internal architecture, picture from [5]. . . . . 73
5.17 Example of RS422 waveforms, picture from [5]. . . . . 73
5.18 5.18a NUCLEO-L452RE pinout, picture from [96]. 5.18b Nucleo board mounted as daughter board by the Morpho connectors. . . . . 74
5.19 System power delivery. . . . . 76
5.20 Power regulation circuit. . . . . 77
5.21 Simple type protection circuits. . . . . 78
5.22 Complex protection circuit block diagram. . . . . 79
5.23 Latch-up protection circuit. . . . . 79
5.24 5.24a INA138 high-side current measurement circuit, picture from [45]. 5.24b TL431 2.495 V reference generation. . . . . 80
5.25 Slow turn-on, fast turn-off high side switch circuit. . . . . 81
5.26 Latch-up protection circuit simulation model. . . . . 82
5.27 Voltage at load during simulated latch-up event in SPICE. . . . . 83
5.28 Simulated latch-up event, red: load voltage, blue: INA138 output, green: current through switch. . . . . 83
5.29 IC powering from I/O through ESD protection diode. . . . . 84
5.30 SN74AXC1T45 and SN74AVC4T245 used as voltage domain interface with the MRAM. . . . . 85
5.31 Singer board connectors names and function. . . . . 86
5.32 Samtec ISDF-07-D-M cable housing (left) and TFM-107-02-S-D-WT-P connector (right). . . . . 86
5.33 Nucleo header connectors pinout. . . . . 88
5.34 5.34a Voltage regulator symbol. 5.34b Voltage regulator footprint. 5.34c Voltage regulator 3D model. . . . . 89
5.35 Main Singer schematic sheet on KiCad. . . . . 90
5.36 PCB interconnections layout. . . . . 91
5.37 PCB layers. 5.37a Top layer. 5.37b Top inner layer. 5.37c Bottom inner layer. 5.37d Bottom layer. . . . . 92
5.38 PCB floorplan. Top view (left) and bottom view (right), areas of the same color are part of the same block. . . . . 93
5.39 5.39a Interconnection network with single segment $\pi$ model. 5.39b Simulation result. . . . . 94
5.40 The four produced Singer boards. . . . . 95

6.1 Singer firmware hierarchical structure. . . . . 97
6.2 FreeRTOS with static RAM allocation. . . . . 99
6.3 Example of FreeRTOS tasks scheduling. . . . . 100
6.4 Singer logical architecture. . . . . 100
6.5 Microcontroller pin assignments. . . . . 101
6.6 SPI basic concept with shift registers data swap. (Up) Shift registers content before the transaction. (Down) Shift registers content after the transaction. . . . . 101
6.7 Typical SPI transaction, the four possible combinations of clock polarity an phase are shown. . . . . 102
6.8 6.8a SPI in multi-drop configuration. 6.8b SPI in daisy-chain configuration. 102
6.9 UART communication between two devices. . . . . 103
6.10 UART frame with 8 bits of data and no parity bit. . . . . 103
6.11 6.11a UART driver outline. 6.11b UART driver data flow pipeline. . . . . 105
6.12 UART driver data structure. . . . . 106
6.13 Circular buffer and plain buffer structures and functions. . . . . 107
6.14 searchPacket() and searchPacketAdvance() functions headers. . . . . 107
6.15 Packet search rules. . . . . 108
6.16 Example of possible packet formats. . . . . 108
6.17 6.17a searchPacket() function flow graph. 6.17b Example of byte properties determination, heads are green, tails are red, partial head/tails are orange, other packet bytes are yellow. . . . . 109
6.18 Search packet state machine and mode 2 check. . . . . 109
6.19 Example of buffer alignment. (Up) Rotation needed. (Down) Rotation not needed. . . . . 110
6.20 Buffer advance flags and relative logic. Head bytes are in green, tail bytes in red. . . . . 110
6.21 Real-case example of searchPacketAdvance() applied to an incoming stream of data. Head bytes are in green, tail bytes are in red. Function calls are represented as red squares around the circular buffer, function flags that cause the buffer to advance on each step are shown on the left. . . . . 111
6.22 Xbus message frame, picture from [62]. . . . . 111
6.23 MTData2 message format, picture from [62]. . . . . 112
6.24 IMU operative modes, picture from [62]. . . . . 113
6.25 IMU driver outline. . . . . 113
6.26 Parrot communication protocol message format. . . . . 114
6.27 Binding of printf() to the AP UART through the definition of _io_putchar() 115
6.28 AD7788 single conversion mode timing diagram, picture from [55]. . . . . 116
6.29 Logical wiring of the two ADCs. . . . . 117
6.30 ADC driver outline. . . . . 117
6.31 MRAM internal architecture, picture from [42]. . . . . 118
6.32 MRAM arrays and corresponding address space. . . . . 119
6.33 MRAM interface registers, picture from [42]. . . . . 119
6.34 Timing of 1-1-1 instruction type, picture from [42]. . . . . 120

6.35 6.35a MRAM driver outline. 6.35b MRAM instruction set definition inside the driver. . . . 120
6.36 MRAM log system organization. . . . 121
6.37 SingerTableEntry structure. . . . 122
6.38 SingerTelemetry structure. . . . 122
6.39 6.39a SingerDataPacket structure. 6.39b ParrotSensorsMemory structure. . . 123
6.40 Log system library outline. . . . 123
6.41 6.41a Firmware task general form. 6.41b Firmware tasks and data flow paths. 124
6.42 TelemetryProdCons structure. . . . 125
6.43 Sequence diagram of telemetry exchanges. . . . 126
6.44 Sensors task state machine states. . . . 127
6.45 Sensors task flow chart and sampling state machine. . . . 128
6.46 Memory task flow chart. . . . 129
6.47 Parrot protocol messages exchange. . . . 130
6.48 SingerRequest structure. . . . 130
6.49 ParrotSensorsSerial structure. . . . 131
6.50 Parrot task state machine. . . . 131
6.51 Parrot task flow chart. . . . 132
6.52 Hierarchical timeout example. . . . 133
6.53 Access Port debug console. . . . 134
6.54 Time measurement mode output. . . . 135
6.55 Firmware memory usage. . . . 136

7.1 SPEISAT FlatSat AIV plan. . . . 137
7.2 SPEISAT Flight Model AIV plan. . . . 138
7.3 ST-Link programmer (still attached to the Nucleo board). . . . 139
7.4 Breakout boards. 7.4a Access Port interface board. 7.4b Loop-back board. 7.4c Singer interface cable. . . . 140
7.5 7.5a Singer breadboard model, first row from left: IMU, Nucleo, level shifters; second row from left: RS422 transceiver, PC104 board with ADCs, MRAM ICs. 7.5b SMT devices on breadboard socket. . . . 140
7.6 Singer downlink (left) requested by the Parrot mock-up (right), which then shows the response messages in hexadecimal format. . . . 141
7.7 MT Manager window, showing the data graphs and the IMU orientation view. . . . 142
7.8 Microcontroller used as IMU bridge to connect the IMU to MT Manager. . 142
7.9 Acceptance tests reference configuration. . . . 143
7.10 7.10a Latchup test setup. 7.10b Voltage at load during simulated latchup. . 144
7.11 Memory test AP output. . . . 145
7.12 7.12a RS422 test data flow. 7.12b RS422 test AP output. Each number represents an RS422 transceiver, the order of arrival between channels is not important since the messages were sent and received at the same time. 146
7.13 ADC test AP output. The thermistors cable was connected to J7 at this point, the last thermistor was being heated up by the operator (the code decreased). . . . 146

7.14 IMU test AP output. . . . . 147
7.15 PCB design error corrections on the qualification model. (Right) Correction on the MRAM. (Left) Added pull-up resistor. . . . . 148
7.16 Nucleo board modified/removed components. . . . . 149
7.17 7.17a Pre-crimped cables for thermistors. 7.17b Soldered thermistor. 7.17c Thermistors integrated in the spacecraft, before fixing them to the measurement points. . . . . 150
7.18 7.18a Nucleo board glued on Singer PCB, we can also notice the glue on PCB mounting screws. 7.18b Backplane and thermistor array connectors glued during integration. . . . . 151
7.19 The integrated flight model without the last two solar panels. . . . . 151
7.20 Functional tests reference configuration (only the point of view of Singer is shown). . . . . 152
7.21 fig. 7.21a The flexible antennas after deployment. fig. 7.21b Ground Station front-end. . . . . 153
7.22 7.22a SPEISAT inside the deployer dummy. 7.22b Vibrational test setup, the CubeSat was placed inside the dummy deployer. . . . . 154
7.23 Spacecraft on the dummy deployer inside the thermal chamber. . . . . 155
7.24 Solder paste residues from improper manufacturing, the distance between two legs is 0.5 mm. . . . . 156
7.25 Flight model accident scenario. . . . . 157
7.26 Garbage output after the incident. . . . . 157
7.27 Operators testing the spacecraft shipping container with the exposition model, the flight model can be seen on the foreground. . . . . 159

8.1 8.1a View inside the Falcon 9 second stage fairing of Transporter-8 launch, the arrow points to the position of SPEISAT, the approximate dimension of the CubeSat is drawn as a red rectangle, Credit: SpaceX. 8.1b Falcon 9 launch with SPEISAT onboard, picture shot by the author. Credit: SpaceX 161
8.2 ARI-Bra amateur radio station. The UHF antenna can be seen in the center. 162
8.3 Main Gpredict window showing SPEISAT position, trajectory and line of sight. . . . . 163
8.4 Ground Station setup at ARI-Bra. . . . . 163
8.5 Singer downlink waterfall shown on Gqrx (right), it's possible to spot the Doppler shift on the received frequency; Gpredict tabs are also visible (left) which were commanding the antenna rotator and correcting the reception frequency on Gqrx. . . . . 164
8.6 Singer temperature measurements versus time, y axis is in [°C]. . . . . 165
8.7 Parrot sensors temperature measurements of battery (up), and the two CDHs (down), y axis is in [°C] for both. . . . . 165
8.8 Parrot sensors battery voltage (up, [V]), discharge current (middle, [A]) and charge current (down, [A]) versus time. . . . . 166
8.9 Singer IMU data versus time: gyroscope (up, [rad/s]) and magnetometer (down, normalized no unit). . . . . 167
8.10 Singer reboots over the period from 25 June 2023 - 31 July 2023. . . . . 168

8.11 CDH 1 operative mode stored by Singer over the period from 25 June 2023 - 31 July 2023. . . . . 168
8.12 CDH 2 operative mode stored by Singer over the period from 25 June 2023 - 31 July 2023. . . . . 169

# List of Tables


4.1 Singer functional requirements. . . . . 51
4.4 Singer performance requirements. . . . . 53
4.6 Singer interface requirements. . . . . 54

5.1 MRAM current estimation. . . . . 72
5.2 MRAM average power estimation. . . . . 72
5.3 Nucleo L452RE clock sources characteristics, data from [113] (pag. 135) and [68], maximum error includes the contribution of tolerance on the frequency value and drift due to $V_{DD}$ and temperature (and aging for the LSE). . . . . 75
5.4 Used microcontroller blocks, associated function and clock. . . . . 76
5.5 Singer Access Port (J9) pinout. . . . . 87
5.6 Singer communication port (J8) pinout. . . . . 87
5.7 System power consumption summary . . . . . 88
5.8 PCB estimated tracks resistance and capacitance per unit length. . . . . 94

6.1 List of Parrot/Singer messages. . . . . 130
6.2 Firmware configuration. . . . . 135

7.1 Power test. . . . . 143
7.3 MRAM test. . . . . 144
7.5 RS422 test. . . . . 145
7.7 ADC test. . . . . 146
7.9 IMU test. . . . . 147
7.11 Full functional and day-in-the-life tests. . . . . 152
7.13 Mechanical fit and vibrational tests. . . . . 154
7.15 Thermal cycling test. . . . . 155

# Chapter 1
# Introduction to CubeSat

This chapter will introduce the reader to the CubeSat standard and give an overview of its evolution over time.

## 1.1 The CubeSat standard

The CubeSat project was born in 1999 from the minds of Prof. Jordi Puig-Suari (Cal Poly) and Prof. Bob Twiggs (Stanford) as a mean to provide students with an affordable way to develop space missions, the idea behind CubeSat is the standardization of small satellites' form factor which in turn lowers the costs of spacecraft design and simplifies launcher interfacing ([91], chapter 1,2).

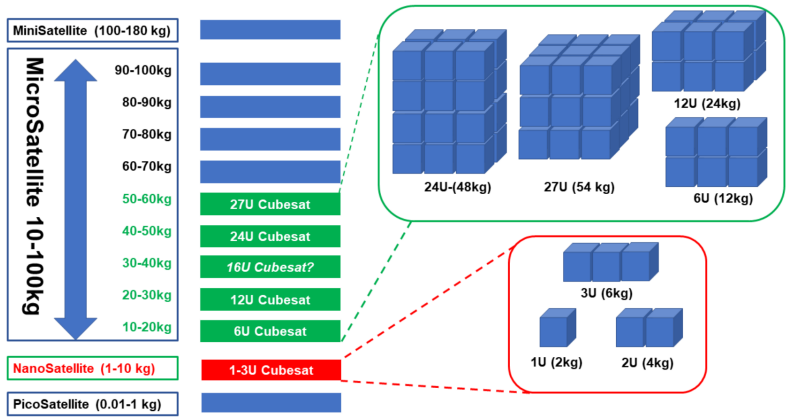



**Figure 1.1:** Weight class of some standard and non standard CubeSats, picture from [91].

"The intent of the CubeSat Project was to reduce cost and development time, increase accessibility to space, and sustain frequent launches." [21]

The CubeSat standard is released as the "CubeSat Design Specification" or CDS ([21]) available in the official website (https://www.cubesat.org/); CubeSat is a family of small satellites, each one has a cuboid volume that's a combination of (10x10x10) cm units (U), hence the type of CubeSat derives from the number of units: today the CDS officially defines 1U, 1.5U, 2U, 3U, 6U and 12U (see fig. 1.1), but several others have been launched (according to [120]) like 0.25U, 4U, 8U, 16U and also different units dispositions of the official classes (for example some 6U with the form 1Ux1Ux6U).

CubeSats are deployed into orbit by a standardized dispenser, the first dispenser developed was the Poly Picosatellite Orbital Deployer, or P-POD (fig. 1.2), that can house 3U in total and deploys them by means of a spring system, but today many others exist; the small and standardized form factor allows building low-cost spacecrafts and guarantees interoperability of all CubeSats with dispensers; another advantage of this approach is containment: the dispenser acts as a firewall between spacecraft and launchers so the flight qualification of CubeSats is easier ([100]).

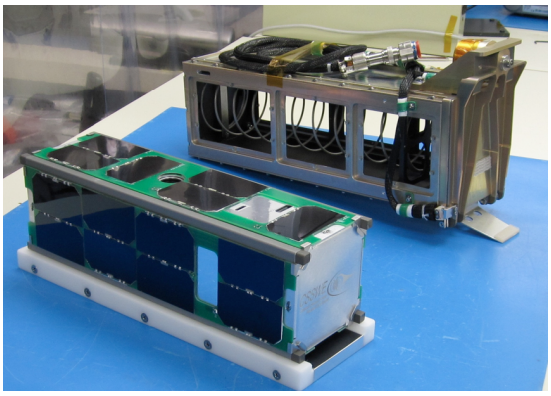


**Figure 1.2:** 3U CubeSat (CSSWE) with P-POD dispenser, picture from [3].

According to the CDS, each U can weight up to 2 kg, so they can be classified as belonging to NanoSatellites (1-10 kg, for 1U-3U range) and MicroSatellites (10-100 kg, for 6U+) classes (see fig. 1.1), with the NanoSatellite definition that is often used interchangeably with "CubeSat" since originally only the 1-3U range platforms were defined ([91]); the CDS also clearly states that the document should only be considered as a guidance and that the CubeSat developers should comply with the specific requirements of each launch provider, which usually are quite similar to the CDS if not less restrictive.

## 1.2 CubeSats evolution

This section has the purpose of drawing an evolution timeline for CubeSats (as seen on fig. 1.3); the analysis came from the last two NASA's state-of-the-art reports on small satellites ([90], [91]) and by direct observations on data from https://www.nanosats.eu, an online database that tracks small satellites missions. Other fonts include the paper from Burkhard and Weston ([106]) for the NATO Science and Technology Organization (STO) and the series of conference papers from Prof. Swartwout ([100]–[105]) about small satellites evolution.

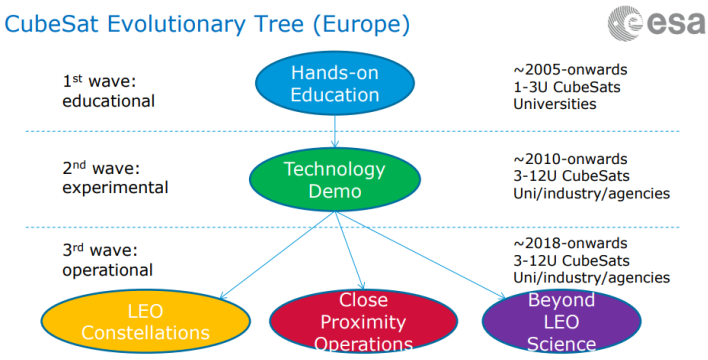



**Figure 1.3:** Cubesats evolutionary tree, picture from [117].

For simplicity, on the graphs made by the author (the ones that say "data from [120]"), only the numbers of the official CubeSats types (the mostly launched) have been used (1U,1.5U,2U,3U,6U,12U, where for 6U only the official 1Ux2Ux3U unit disposition has been considered), it's also worth mentioning that the database only tracks publicly available satellites, nevertheless this should give an idea on the evolutionary trends of this technology which is the main goal of this section. Also consider that the numbers for

the year 2023 include planned CubeSats besides the already launched, for this reason the numbers should be considered only as indicative.

### 1.2.1 Some words on secondary payloads

The history of CubeSats is closely linked to the history of secondary payloads (rideshares or "piggyback" missions) and rocket launchers in general: launches are very costly and usually the primary mission only exploits a part of the launcher capabilities in terms of upmass, it was to be expected then that the idea of secondary payloads made it's way on the aerospace industry from the very beginning, with the first spacecraft launched as secondary payload being the 20-kg SOLRAD-1 in 1960 ([100]).

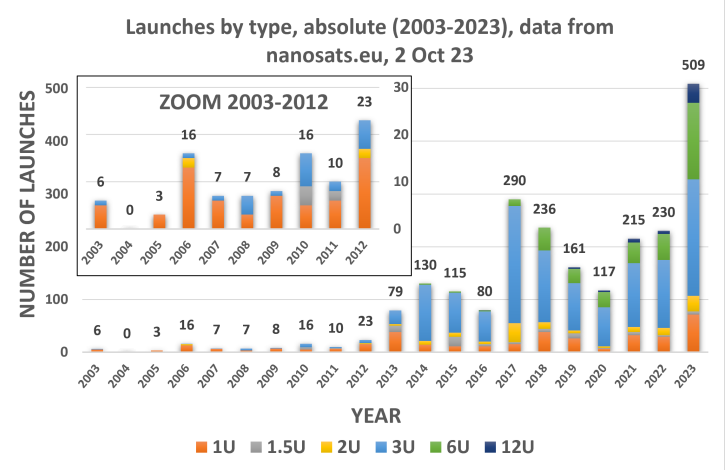



**Figure 1.4:** Cubesat launches by type (1U,1.5U,2U,3U,6U,12U) from 2003 to 2023, data from [120].

In the period from 1960 to 2012, only 7.5% of the spacecraft launched were secondary, then in 2012 the percentage was 30% and in 2013 53% ([105]), this trend continued growing until today (with CubeSats becoming the major contributors in their mass category as explained in [104]), but the original concept of secondary payload has almost lost its meaning: today it's common to see launches without a primary mission but with hundreds of "equally secondary" spacecraft ([105]) and the introduction of big constellations (hundreds if not thousands of identical satellites) has greatly skewed the numbers not only for CubeSat class ([106]).

### 1.2.2 2000s: CubeSats introduction

Initially, CubeSat platforms where mainly used by universities as a precious way for students to experience the design of a complex space project and traditionally CubeSats were built from Components Off The Shelf (COTS). This trend continued up until ∼2010/2011, with a relatively low number of launched CubeSats (in LEO) and the most popular mission objectives being technology demonstrations (ADS/ADCS, radio communications, radiation detection, basic earth observation and avionics validation in general), as explained in [106]. In fig. 1.4, we can see that back then the favorite CubeSat type was 1U, with 3U starting to take the lead towards the end of the decade. The first 6 CubeSats were launched in 2003.

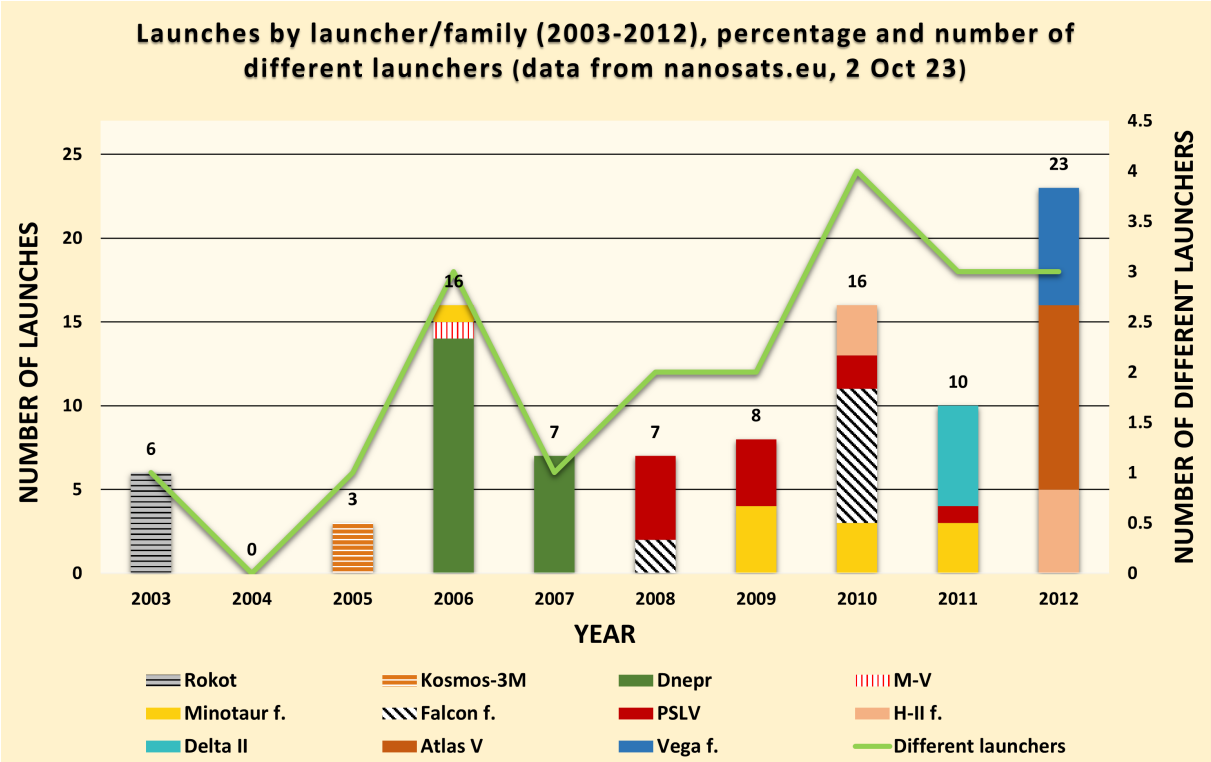



**Figure 1.5:** CubeSats (1U,1.5U,2U,3U,6U,12U) launched by each launcher or family of launchers (denoted with "f.") in the period from 2003 to 2012 (left axis), the green trend line is the number of different launchers employed (right axis), data from [120].

### 1.2.3 2010s: Growing interest on CubeSats

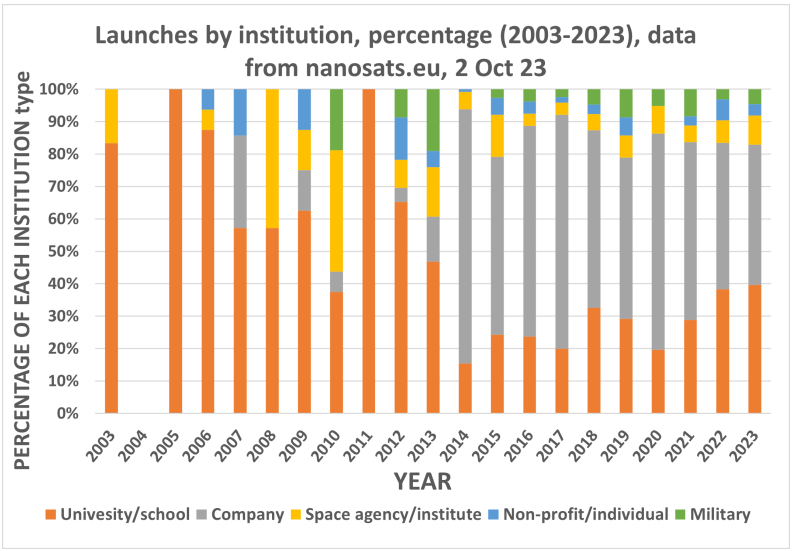



**Figure 1.6:** Percentages of CubeSats (1U,1.5U,2U,3U,6U,12U) launched by type of institution, data from [120].

Around 2010, the potential of CubeSats as a low-cost platform had been seen by more and more institutions outside of the educational context:

> "it was only 7 years ago that the Director of Advanced Systems and Technology at NRO told the audience of the Smallsat conference that his agency had no interest in CubeSats. And, now, in 2011 the NRO Colony program is one of the largest funding sources for CubeSat technology development" ([101]).

This trend can be seen in fig. 1.6, where the launches by type of institution are plotted. As stated in [106], CubeSats became more capable and therefore bigger platforms started being preferred since the 1U is quite limited in volume. It wasn't only the size though, because all the ecosystem of electronics and technology in general has drastically grown and today a CubeSat can be equipped with greater data processing, communication and navigation capabilities within the same unit of volume, so that also the smaller platforms are more useful.
Over this period many mission were launched with a scientific payload and a secondary technology demonstration purpose. This trend on choosing larger and larger platforms can be clearly seen in fig. 1.7: while initially almost all satellites were 1U, on the last decade the 3U platform has become the preferred choice, from its introduction around 2015 the 6U platform has grown steadily to around one third of the launched satellites, while today we can see that also the 12U started rising.

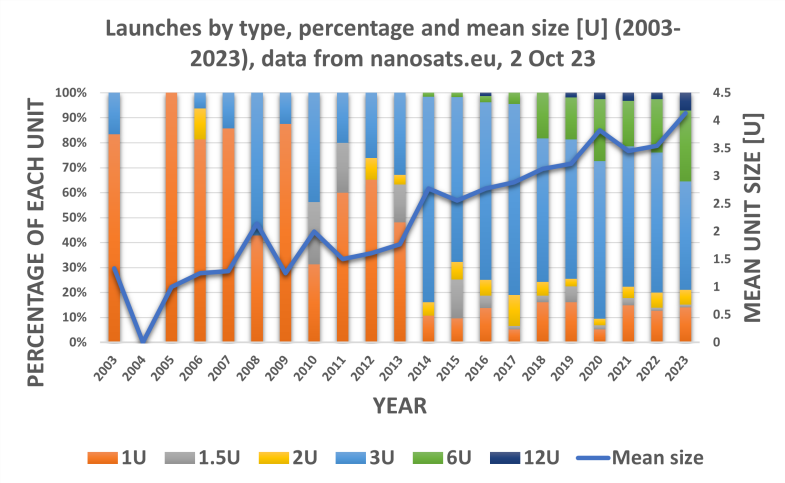



**Figure 1.7:** Percentages of launched CubeSats types (1U,1.5U,2U,3U,6U,12U) over the period from 2003 to 2023 (left axis), the blue trend line is the average size in Units (U) of launched CubeSats (right axis), data from [120].

### 1.2.4 2014: CubeSat constellations and dedicated rideshares

As highlighted by data, it's clear that the period around 2014 was a turning point on the CubeSat history: the percentage of CubeSats launched by companies passed from around 15% on 2013 to around 80% on 2014 (fig. 1.6), the total number of launched CubeSats was already growing on 2013 and in 2014 passed from tens of satellites of the 2000s to hundreds of them (fig. 1.4), with the number of 3U CubeSats growing from around 30% of 2013 to an 80% of total. Furthermore, the number of founded companies abruptly doubled with respect to the already growing number of the previous years, as in fig. 1.8. This is representative of a fundamental step: in 2014 the first CubeSat constellation (PlanetLab's earth observation "Dove"s) was launched, with almost a hundred 3U CubeSats deployed along the year, paving the way for CubeSat constellations, as stated in [106]; in the paper is estimated that in 2017 about half of all the CubeSats/nanosats launched were 3U constellations.

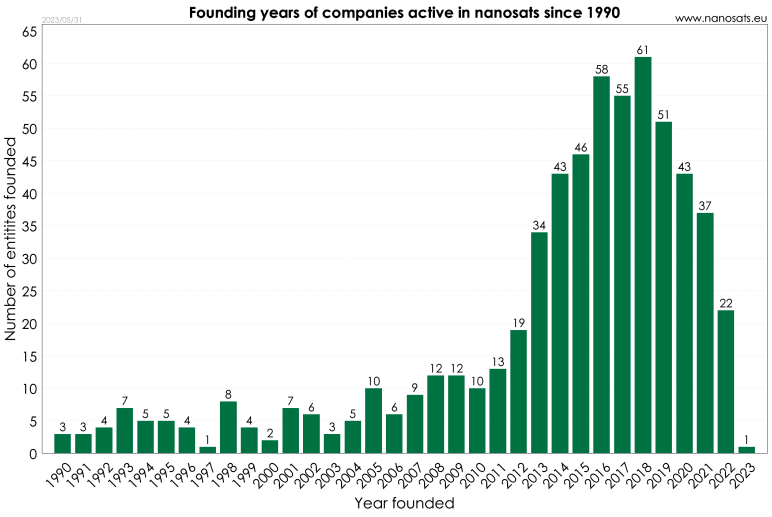



**Figure 1.8:** Number of founded companies active in nanosats over the time period from 2003 to 2023, picture from [120].

CubeSat constellations greatly exploit the intrinsic advantages of this type of satellites: the possibility to deploy a large array of identical low-cost payloads on a large area to capture simultaneous multi-point measurements, allowing for remote sensing, weather observation and IoT infrastructures, just to name some; another possibility with constellations is to deploy different payloads that perform together an integrated and versatile service, this is a more difficult path but is gaining attention due to its potential for applications like in-space infrastructure services, debris monitoring and other close proximity operations ([106]).

The capability offered by multiple smaller satellites can somehow match the one of a bigger and more expensive one ([91] pag. 207).

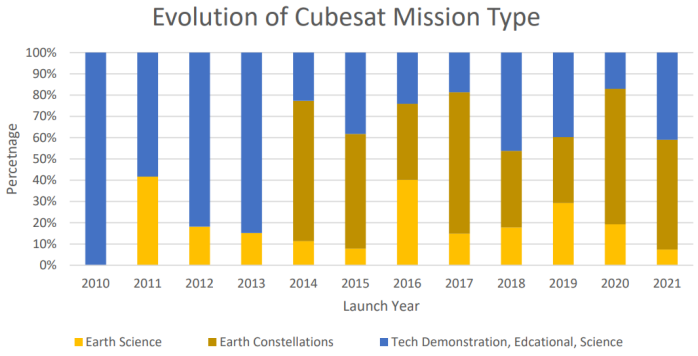



**Figure 1.9:** Percentage of CubeSats mission type, picture from [90], pag. 12.

As already said, the growth of CubeSats and secondary payloads was coupled with a contemporary growth of secondaries launched: the total number of space launches (and space launches that carry secondaries) had grown steadily but limitedly so the great increase on launched CubeSats is due to the growth on the average and total number of secondaries carried on each launch ([105]). Another view of this trend can be seen in fig. 1.10, where the growing number of different launchers used for CubeSats is plotted.

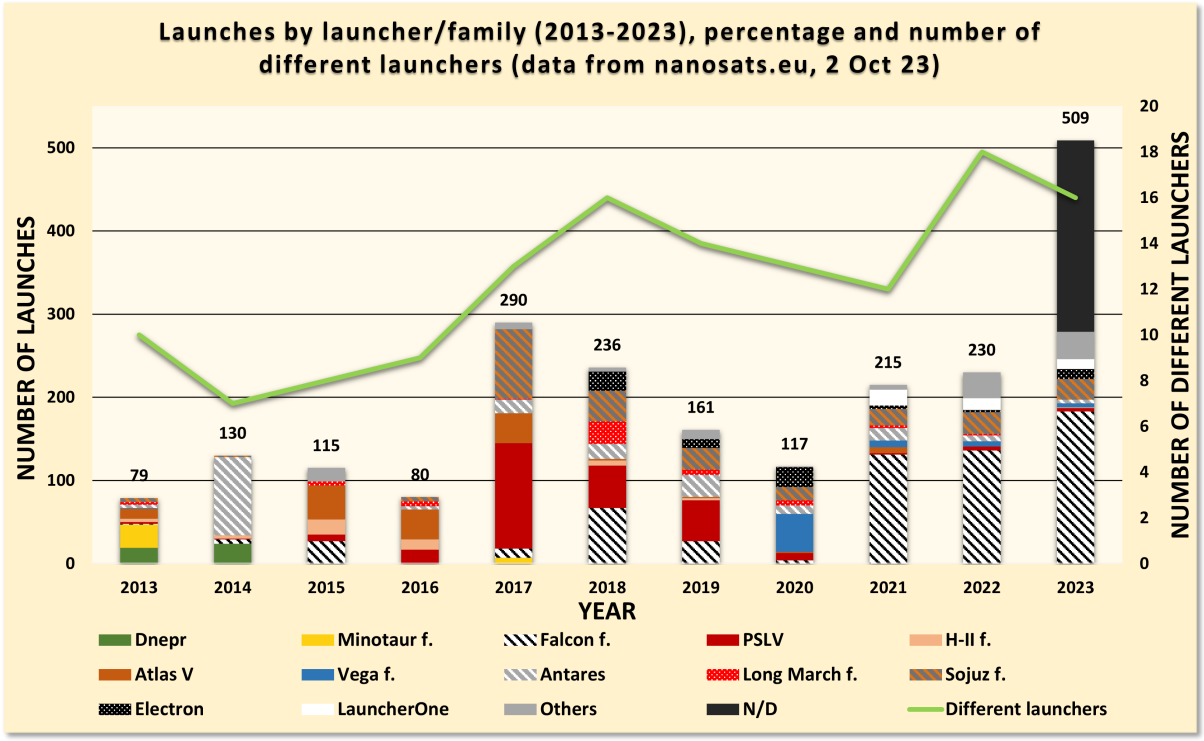



**Figure 1.10:** CubeSats (1U,1.5U,2U,3U,6U,12U) launched by each launcher or family of launchers (denoted with "f.") in the period from 2013 to 2023, the green trend line is the number of different launchers employed (right axis), data from [120].

Some records had been established meanwhile: in 2017, a PSLV launch set the record for the highest number of launched CubeSats as 101 spacecrafts, it also temporary set the record for the highest number of satellites launched (104) that was passed in 2021 by a SpaceX Falcon-9 launch with 149 small satellites (of wich 91 CubeSats) for the SpaceX Transporter program (data from [32], [120]).
The large availability of launch opportunities has been a major enhancer of CubeSats and will continue to drive their evolution ([106]), this is also thanks to the standardization of deployment systems (PODs) that allows launching CubeSats and other secondaries from many different vehicles ([103]).
In this context is important to distinguish CubeSat missions not only in their mission objectives but also in their acceptable risk and failure figures: until now we taught about CubeSat missions in general but it's important to notice that very different types of CubeSats manufacturers exist, each one with very different performance objectives, risk tolerances and so, cost. In [103], [104] a categorization is done on the 2000-2015 data between four types of builders:

- Hobbyst: Groups that approach CubeSat building with low cost, fast turnaround and high risk tolerance, with lack of standard practices on integration and tests. New universities are an example of hobbysts.

- Traditional Contractor: These professional groups build CubeSats the same way they

would build any other spacecraft, with high performance objectives and low risk tolerance, so the cost of their spacecraft is high. The examples given on the paper are Boeing or Lockheed Martin.

- SmallSat developers: This group is between the first two, they have experience in building satellites and have developed practices and risk profiles depending on the mission. Examples of this group are the AMSAT, experienced universities and some government agencies.

- Constellations builders: Technically they are in SmallSat category, but since they build a big number of exact copies of a single spacecraft they are excluded from the graphs because they will skew the statistics. The major example of this category is PlanetLab with its 143 launched CubeSats in the end of 2015 according to the paper (555 to date, from [120]).

In fig. 1.11, the data from each category until 2015 is presented as given on the paper, from the graphs we can immediately notice the significant differences between the various builder categories, with hobbysts that more than half of the times experienced at least an early loss of the spacecraft.

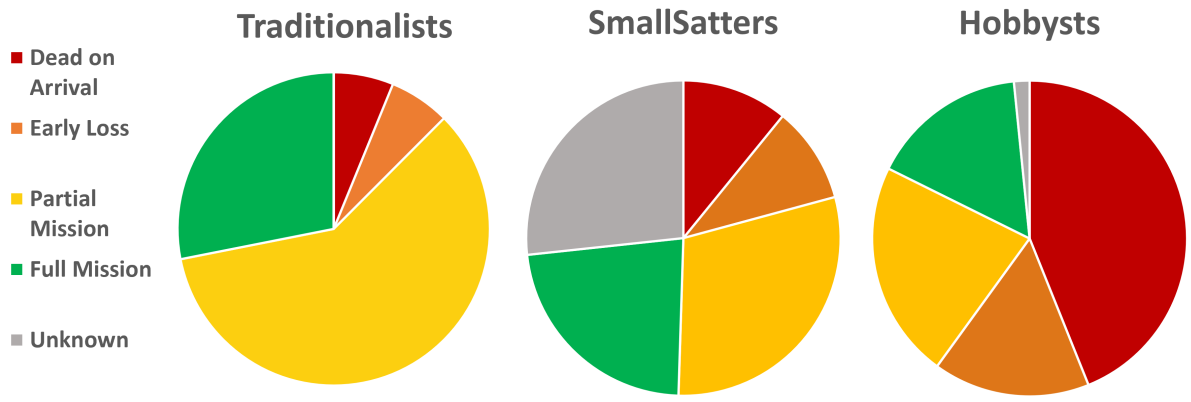


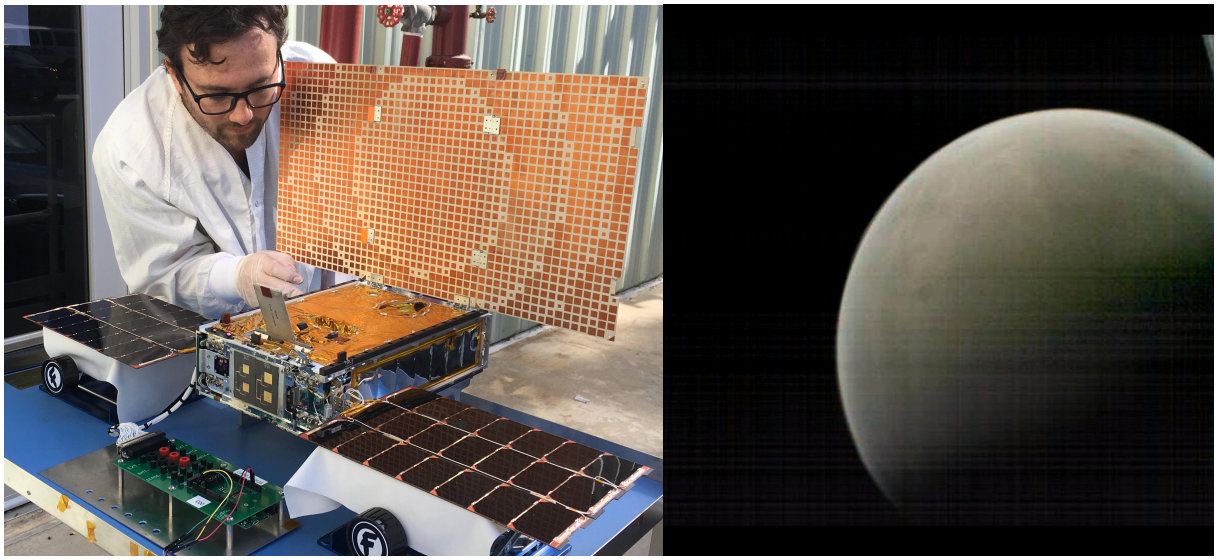

**Figure 1.11:** Cake graphs of mission status of all the CubeSats launched by each builder category, excluding constellations, at the end of 2015, data from [104].

### 1.2.5 2018: Deep space missions

In 2018, the first deep space mission (Mars Cube One or MarCO) was launched, as a couple of 6U CubeSats headed to Mars with the InSight lander ([60], [91], pag. 74).
The employment of CubeSats for deep space missions is expected to grow on the near future, in 2021 NASA's DART mission carried an Italian 6U CubeSat (LICIACube) to record the impact of the primary mission on the asteroid moon Dimorphos ([54]), other two CubeSats (Juventas and the italian Milani) are expected to visit the same asteroid in 2027, released by the european component (Hera) of this NASA-ESA collaboration ([110]).
In 2022 the Artemis-1 maiden flight of the new NASA's SLS (Space Launch System) launcher sent ten 6U CubeSats towards lunar or heliocentric orbits to collect science data

and demonstrate innovative technologies ([106], [120]). Deep space missions have different requirements than LEO CubeSats, for example in terms of radiation hardening (without the protection of earth's magnetosphere, as explained in [90], pag. ii) and transmission power (due to their limited size and power, CubeSats typically have low data transmission capabilities), as explained in [106].

> "The exposure to GTO, GEO, lunar, and interplanetary space will greatly broaden CubeSat overall capability as more technology is able to be characterized." [106]



**(a)** **(b)**

**Figure 1.12:** Pictures from NASA's MarCO mission. 1.12a One of the twin CubeSats' solar array being tested. 1.12b Image of Mars captured by MarCO-B 6U CubeSat after the landing of InSight. Pictures from [60].

## 1.3 Polytechnic of Turin CubeSat heritage

The Polytechnic of Turin has a rich history of launched CubeSats and the mission of which this thesis is about could draw from the valuable know-how and lessons learned from the past missions; follows a list of the Polytechnic's CubeSats fly heritage:

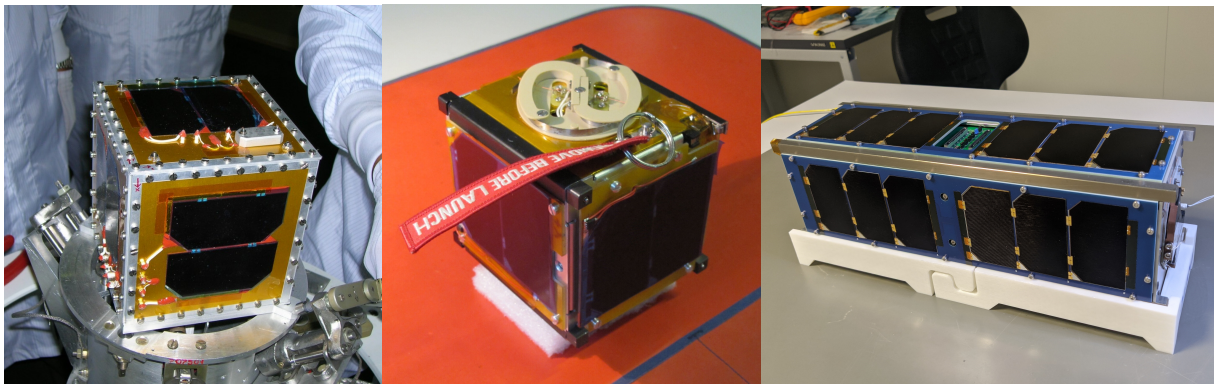
- PicPoT (2007): technically not a CubeSat but still having the cubic form factor with side length of 15 cm (fig. 1.13a), this spacecraft carried various technology demonstrations and aimed at gathering images from space, the mission failed when the Dnepr launcher exploded 86 seconds after launch ([74]).

- e-st@r (2012): this 1U CubeSat carried an ADCS demonstration experiment and was launched on board the Vega launcher on its maiden flight ([19], [28]).

- e-st@r-II (2016): this 1U CubeSat (fig. 1.13b) is an improved version of its predecessor, it was launched onboard a Sojuz launcher as part of ESA's "Fly Your Satellite!" program and is still operational to date ([28], [29], [94]).

- Spei Satelles (2023): the first 3U CubeSat (fig. 1.13c) of the list and last launched one;

built in less than 5 months, this is the spacecraft that contains the subsystem analyzed on this thesis; it hosts a telecommunication main mission and a secondary scientific one, with the goal of gathering temperature and attitude measurements in orbit; the mission also hosts some technology demonstration experiments, like a passive ACS system, radiation hardening techniques and testing of a redundant architecture; it was launched onboard a Falcon 9 launcher and is still operational to date ([47], [86]).




**(a)** **(b)** **(c)**

**Figure 1.13:** 1.13a PicPoT satellite. 1.13b e-st@r-II CubeSat. 1.13c Spei Satelles CubeSat.

All the Polytechnic missions were built totally or partially from in-house developed COTS hardware and were characterized by the participation of students on all phases of the development effort, this is the context where the student team "CubeSat team" (of which the author is part of) was in fact founded.
The CubeSat team is developing a new 3U mission with earth observation goals for the near future ([46]), meanwhile students of this and other teams (like "team Diana") and researchers of the Department of Mechanical and Aerospace Engineering (DIMEAS) are constantly working on high profile projects related to the CubeSat technology, like the ESA $\mu$Prop project ([12]), the SROC mission ([35]) or the SINAV project ([17]).

# Chapter 2
# CubeSat platforms

In this chapter we will discuss about the actual Cubesat building philosophy, by particularly focusing on the components more related to this thesis (electronic components).

## 2.1 Spacecraft BUS

The spacecraft BUS is the set of mechanical, electronic and software components that provide the necessary support functions to the satellite (should not be confused with the definition of BUS as communication channel typical of electronics), providing a working environment for the payload ([91]). Follows a list of the main elements of a CubeSat BUS, which are analyzed in details in the NASA state-of-art reports ([90], [91]), elements are accompanied by research contributions from the Polytechnic of Turin:

- Structural subsystem [13];
- Electrical Power System (EPS) [11];
- Communication System (COMMSYS) [14];
- On Board Computer (OBC);
- Attitude determination and Control System (ADCS) / Attitude Control System (ACS) / Attitude and Orbit Determination and Control System (AODCS) [93], [95];
- Thermal Control Subsystem (TCS) [18];
- Propulsion subsystem [92].

This general structure can be arranged in various ways and a large number of different solutions are described in the reports. The OBC in particular is a subsystem that has seen diverse implementations and is continuously mutating following the evolution of electronics, passing from a simple and centralized system to distributed architectures with multiple (different) processors that perform the diverse tasks. The main task of the OBC is to perform Command and Data Handling (CDH) operations, being the brain of the spacecraft and running the Flight Software (FSW); this definition is somehow limited since the moving towards distributed architectures leads to the co-existence of different OBCs inside the spacecraft that can share the CDH task or other processing duties; every subsystem can possess its own, more or less powerful, OBC and as stated in the reports

the FSW is in general all the software running on every spacecraft subsystem ([91], pag. 220).
As stated in [90] (pag. 6), in the last decade the manufacturing of CubeSats (and small satellites in general) has mainly bifurcated into two major options:

- Turnkey BUS solutions: a vendor offers a choice of pre-designed, fully integrated and verified spacecraft BUS solutions from which the customer can choose depending on its requirements.

- COTS BUS solutions: the customer can buy different components for their own integration, testing and operation of a custom BUS.

## 2.2 CubeSat Interfaces

Traditionally, universities rely on the COTS option to design CubeSats because it's cheaper and offers more learning opportunities, this is also made possible by the de-facto standardization of some CubeSat interfaces ([90]).

### 2.2.1 PC/104 form factor

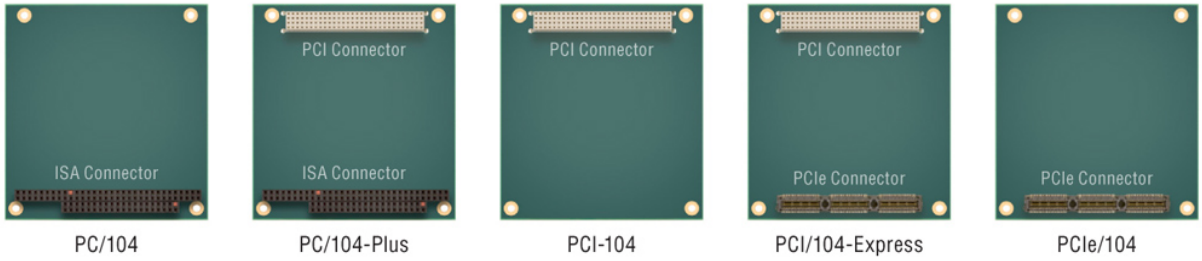



**Figure 2.1:** Family of PC/104 standards, picture from [80].

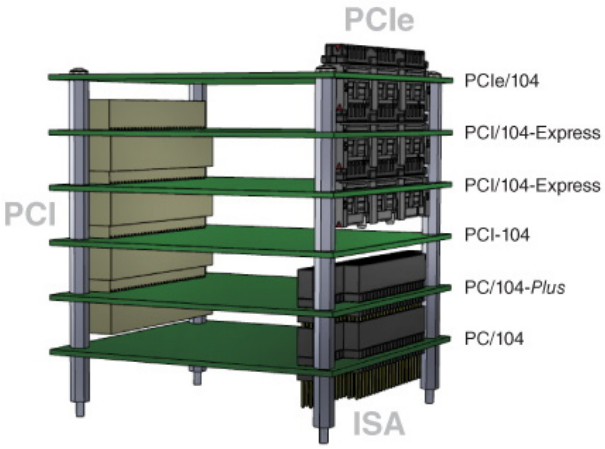



**Figure 2.2:** Example of PC/104 stack, picture from [80].

PC/104, together with CompactPIC, is a de-facto industry standard for CubeSats ([90]), it defines mechanical and electrical specifications for "stackthrough" connection modules (fig. 2.2), the original standard is described in the "PC/104 Specification" document ([71]) but various other versions exist (fig. 2.1) like the PCI/104 and PCIe/104 ([72],

[73]).
There are various levels of compliance with the standard, as explained in the specification document:

> "PC/104 "Compliant" [...] refers to "PC/104 form-factor" devices that conform to all non-optional aspects of the PC/104 Specification, including both mechanical and electrical specifications". [71]
> "PC/104 "Bus-compatible" [...] refers to devices which are not "PC/104 form-factor" (i.e., do not comply with the module dimensions of the PC/104 Specification), but provide a male or female PC/104 bus connector that meets both the mechanical and electrical specifications" [71]

The pin assignments of PC/104 boards for CubeSats can vary between vendors ([23]) and for this reason compatibility issues can arise during integration.

### 2.2.2 Backplane board

Another option is the "plug into a backplane" approach which is described in [6], it consists of a common Backplane Board (BPB) where all the subsystems are perpendicularly connected; this allows routing inside the backplane and eliminates the problems related to the PC/104 interface, like the big mechanical dimensions and the need to disconnect all the top boards to access a specific one. In [81], [112] an implementation of software reconfigurable BPB is built for the "BIRDS" satellite program, in this case a 50 pin connector is used and a programmable logic device allows software configuration of backplane routing, this way the satellite can be reconfigured to house different modules (or module iterations) without the need to fabricate a new backplane PCB.

### 2.2.3 Daughter boards

As explained in [90] (pag. 206), to address problems related to connection complexity and speed, retro-compatibility with technology advancements but also to employ COTS that are not developed specifically for CubeSats, many vendors and designers adopted the idea of daughter (or mezzanine) boards, of which an example is shown in fig. 2.3: the subsystem is composed of a carrier board with connectors to house a daughter board (i.e., a general purpose compute module, an RF transceiver, ...); the advantages of this approach are multiple: daughter boards can be designed/bought as general-purpose modules with a standardized interface, at the same time carrier boards will implement the application-specific aspects and connect the whole module to the rest of the spacecraft through classic connectors (i.e., PC/104). This solution is conceptually similar to how RAM banks are connected inside

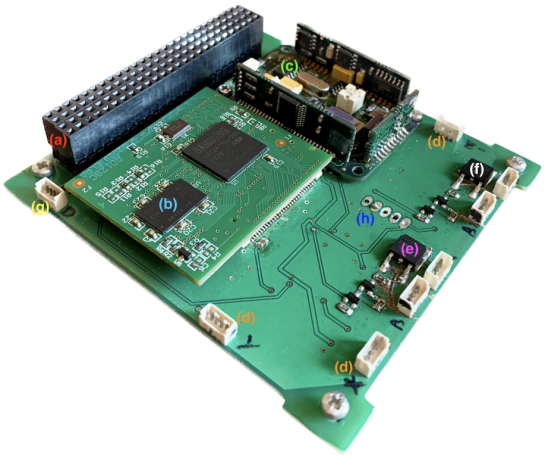



**Figure 2.3:** Example of PC/104 ADCS board developed at Polytechnic of Turin with mounted daughter boards (OBC and IMU), picture from [59].

laptops and indeed some carrier boards have been designed to house SO-DIMM form factor (the typical laptop RAM form factor) compute modules, like for example in [79] where an OBC was developed resorting to two Raspberry Pi CM3 as daughter boards. Another example is given in [122] where a Software Defined Radio (SDR) module is mounted as daughter board for the PRETTY satellite.
Another example given in [90] is the Qseven Computer-on-module standard:

> "The Qseven® concept is an off-the-shelf, multi vendor, Computer-On-Module that integrates all the core components of a common PC and is mounted onto an application specific carrier board. Qseven® modules have a standardized form factor of 70mm x 70mm or 40mm x 70mm and have specified pinouts based on the high speed MXM system connector" [78]

Notice that Qseven is an embedded computer-on-module standard that is not specifically conceived for CubeSats.

## 2.3 Electronics in space

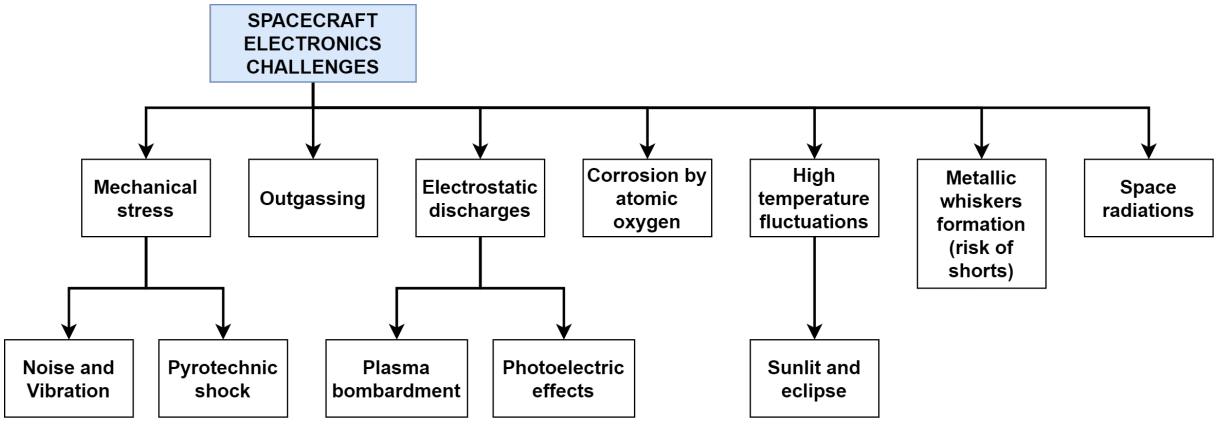



**Figure 2.4:** Challenges of spacecraft electronics, data from [53].

There' s a number of challenges that a spacecraft and its electronic components must face in order to fulfill their mission, in ([53]) some of them are listed (fig. 2.4):

- Mechanical stress: due to the intense vibration of launch vehicles and pyrotechnic shocks (for example during stages separations);

- Outgassing: exposure to void causes plastic, glues and adhesive materials to emit vapors that will orbit with the spacecraft, they can degrade its surfaces but also deposit on optical elements. One solution is use ceramic materials instead of plastic where possible;

- Electrostatic discharges: plasma bombardment and photoelectric effects can lead to violent electrical discharge events that can damage the spacecraft. One solution is to cover the spacecraft outside surfaces with conductive materials;

- Corrosion by atomic oxygen: the main atmospheric component in LEO (96%) is atomic oxygen (O) that can react with organic materials if they are not coated properly;

- High temperature fluctuations: of up to 300°C due to the alternation of sunlit (space-

craft illuminated by sun) and eclipse (spacecraft on earth's shadow) phases;

- Metallic whiskers: pure tin, zinc and cadmium are subject to spontaneous growth of these thin filaments that can cause electrical shorts, this process is enhanced in vacuum so those materials are prohibited for space applications by IEEE. Tin is common as solder metal so the addition of lead is one way to reduce whiskers formation. The notorious NASA's Cassini mission in 2011-12 had to shut down its plasma spectrometer for several months due to voltage fluctuations likely due to whiskers formation ([31]);

- Space radiation: this is one of the main concerns for space level electronics, the type and level of radiation strongly depends on the type of orbit and solar activity.

### 2.3.1 Radiation effects on electronics

Radiation effects are of particular relevance for electronic components because they are not only cause of physical degradation but also of information loss. There are various types of particles in space (protons, neutrons, electrons, alpha, heavy ions), these particles can be trapped inside Van Allen belts, ejected from the Sun or come from outer space (cosmic rays) ([53]). The energy of an ionizing particle that passes through a device will be absorbed with the generation of carrier pairs (electrons and holes), those pairs can recombine or being moved by the local electric fields and depending on the type and energy of the particle and the place where it hits there can be different outcomes ([9], [27]).
In [53], radiation effects are divided into two subcategories:

- Total Ionizing Dose (TID) is a long-term failure mechanism due to charges accumulated over time inside insulators: positive charges are trapped in oxides, changing the threshold voltage of transistors and inducing leakage currents through parasitic channels. Eventually the device may exit the nominal operating range or no longer work ([9]). TID is expressed with the unit of measure [rad]=[0.01 J/kg], which represents the total energy absorbed per unit of mass.

- Single Event Effects (SEEs) are random events that manifest whenever a highly energetic particle strikes microelectronic circuits; there are various types of SEEs that can have a wide variety of outcomes, from no observable effect to information corruption to chip degradation/destruction ([27]).

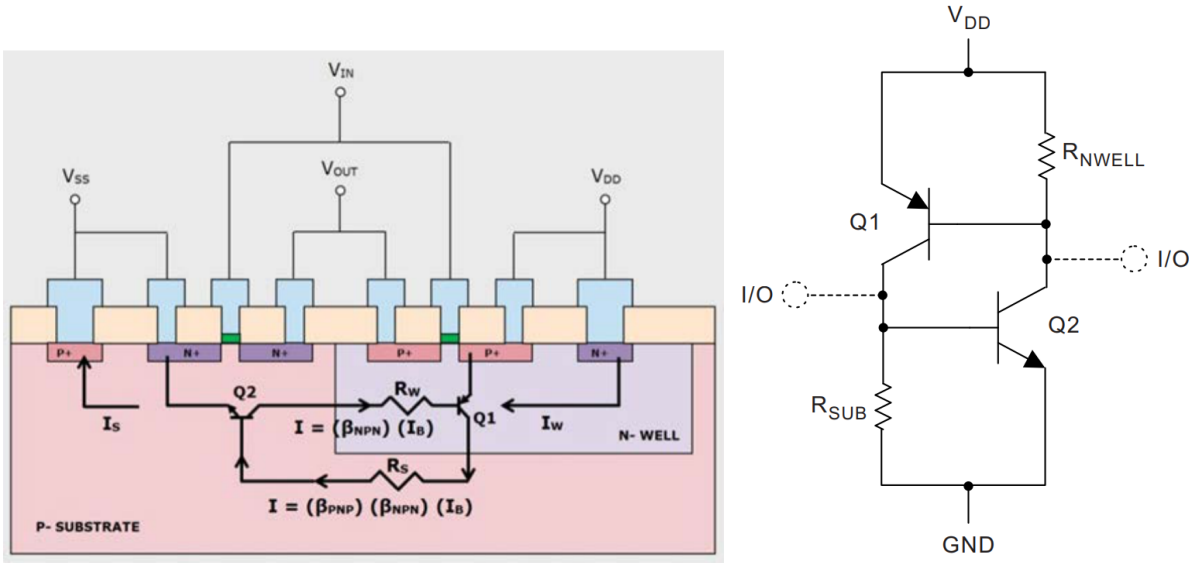
In [53] the classification done by the Joint Electron Device Engineering Council (JEDEC) on SEEs is reported:

- Soft errors: non destructive errors that are typically associated with software/data errors: Single Event Upset (SEU) happen when the energetic particle changes one bit of information in a memory device, SEU in multiple bits are called Multiple-Bit Upsets (MBUs) and SEU that happen in bits that cause the device functionality to be somehow compromised (bits associated with controls/instructions) are called Single Event Functional Interrupts (SEFI);

● Hard errors: destructive errors that cause the device to be irreversibly compromised that are typically associated with hardware damages: Single Event Latchup (SEL) are an example of this type of errors that will be further investigated in this thesis. Other examples are Single Event Gate Rupture (SEGR) and Single Event Burnout (SEB).

#### 2.3.1.1 Single Event Latchup (SEL)

Latch-up is a phenomenon in which a parasitic Silicon Controlled Rectifier (SCR) that exists in CMOS cells (fig. 2.5) can be activated by an injected current or an overvoltage condition, creating a low impedance path between the supply rails; once activated, the latch-up condition is self-sustaining and the only way to stop it is to perform a power cycle of the device. An uncontrolled Latch-up can have severe effects and led to the partial or complete destruction of the IC due to overheating([49]).




**(a)** **(b)**

**Figure 2.5:** 2.5a Cross section of CMOS inverter showing the parasitic SCR. 2.5b The equivalent circuit of the parasitic SCR. Both pictures from [49].

Normally the risk of latch-up is restricted to the application of currents or voltages to the IC I/O pins outside the limits specified in the datasheet [49], this is not true in high radiation environment, where the energy liberated by ionizing particles can trigger latch-up conditions in any point of the IC. SELs are characterized by a series of dimensions which can be used to model the event probability from the energy liberated by ionizing particles. The Linear Energy Transfer (LET) represents the particle energy loss per unit of path length, normalized with respect to material density, its unit of measure is MeV*cm$^2$/mg ([27]). The correlation between particle LET and number of SEL events is expressed by the cross section $\sigma$, which represents the ratio between the number of LET events and the particle fluence in [particles/cm$^2$] ([75], pag. 26).
In [50] it's suggested that while CMOS devices are particularly susceptible to SEL, linear circuits build in bipolar technology are almost immune to this effects, while still suffering

from effects related to TID, in [16], [118], [121] is also suggested that technology scaling reduces the effects of TID but enhances SEEs.
Finding documentation about the radiation behavior of specific COTS devices is quite difficult and mainly depends on testing performed by researchers on similar devices using ion beams or lasers; some examples are [51], [115], [121], where various devices were tested resulting in the majority of cases on latch-up conditions with currents ranging from 40 mA to 1A; it's clear that the range of possible latch-up current values is quite broad and depends on the specific target devices.
A good source of data about radiation effects on electronic devices is the IEEE Workshop on Radiation Effects Data ([44]), an yearly conference which gathers radiation test reports on various devices, but various other databases exist, like for example the ESA's and NASA's Radiation Test Databases ([33], [39]), in some cases the devices chosen for this thesis were found in these databases or standalone papers, in other cases similar devices (same technology node, same functionality) where chosen as representative.
It must be said that since the nature of the system was experimental and the development time was really tight, there wasn't a formal analysis on the expected figures in orbit for this system, the author chose devices with higher scores (when this data was found) considering also their price, both for TID and threshold SEE LET, he also implemented techniques for mitigate the effects of SEE since these were considered somehow inevitable with COTS devices.

# Chapter 3
# Spei Satelles

This thesis was conceived in the context of Spei Satelles mission, of which this chapter will give an overview.

## 3.1 Motivations and mission

> "In the midst of the pandemic, on March 27, 2020, Pope Francis, alone, in the rain, in the dark of that evening, went up to St. Peter's Square to pray with and for all humanity afflicted by Covid. [...] That moment remains engraved in our minds and hearts, forever and in history, and has become an icon of hope. [...] Starting from that day various initiatives were promoted by the Dicastery for Communication of the Holy See in unity with the Pope [...] so that this event would not be forgotten, but would maintain its driving force." [87]

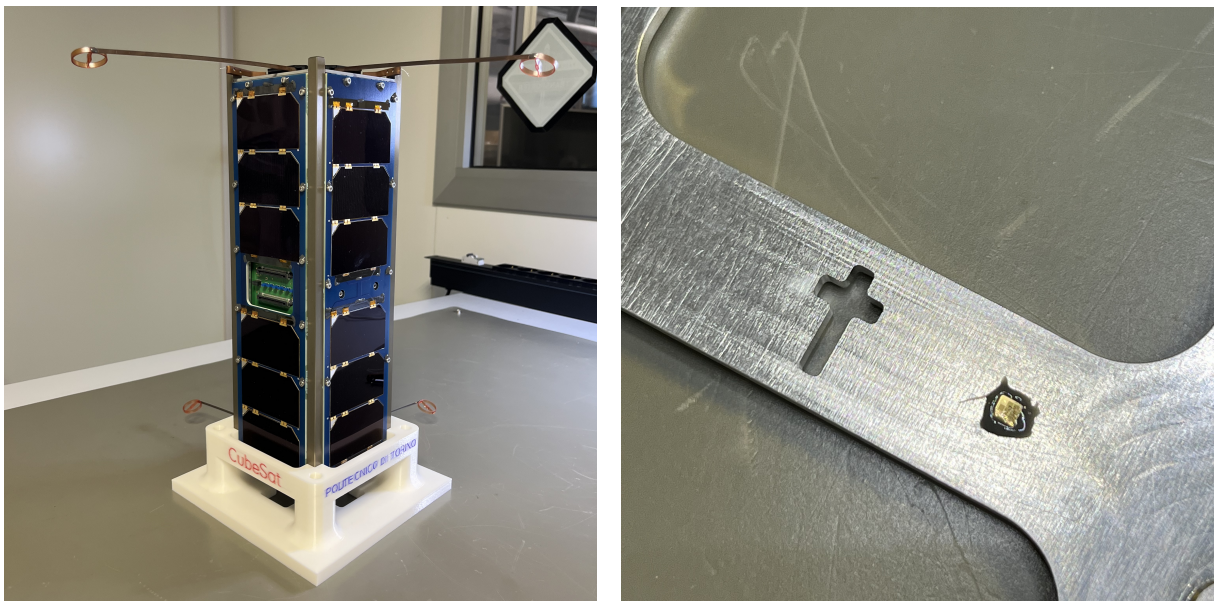
Spei Satelles is a space mission that consists of a 3U CubeSat, built by students and researchers of Polytechnic of Turin with some support from external companies, involved students were members of student teams "CubeSat Team" and "Team Diana", the former being the team the author is part of. Students from the IUSVE of Venice were also involved in the realization of the mission logo (fig. 3.1) and other multimedia material. The spacecraft can be seen in fig. 3.2a.


**Figure 3.1:** Spei Satelles logo, designed by students of the IUSVE of Venice [87].

As described in its official website ([87]), the mission was promoted by the Dicastery for Communication of Vatican City with coordination from the Italian Space Agency (ASI), its main goal is to carry a silicon nanobook (a little piece of silicon engraved with the book of Pope Francis "Why are you afraid? Have you no faith?") made by the Institute for photonics and nanotechnologies of the National Research Council (CNR) and fixed to the satellite structure (fig. 3.2a); the satellite will also periodically send phrases of the book in different languages, to be captured by radio amateurs all around the globe.

The Polytechnic of Turin decided to combine the main spiritual mission with a more scientific purpose, by equipping the spacecraft with a suite of sensors to collect data about the space environment and spacecraft behavior; developing this secondary mission was

the task assigned to the author of this thesis.




(a) (b)

**Figure 3.2:** 3.2a Spei Satelles spacecraft. The access port can be seen in the middle of the left solar panel. 3.2b The nanobook glued to the satellite structure.

## 3.2 Spacecraft architecture

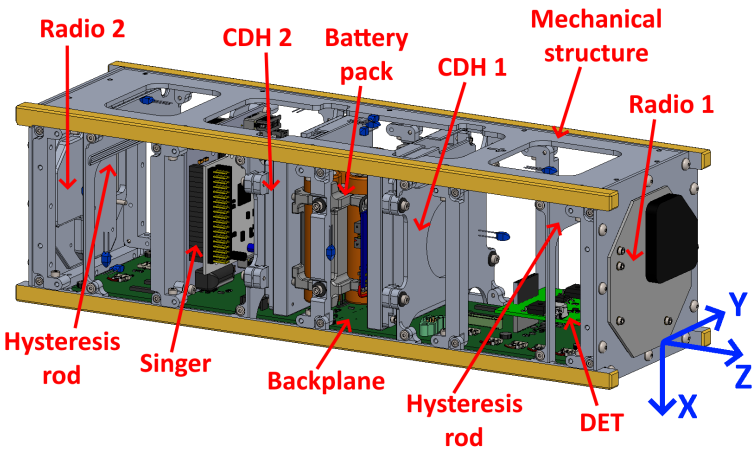



**Figure 3.3:** Main spacecraft building blocks and reference plane, solar panels not shown.

This wants to be a brief overview about the spacecraft architecture and subsystems, necessary to give an insight on the context on which the subsystem treated on this thesis (Singer subsystem) is collocated and to serve as a reference for the various spacecraft building blocks and relative tasks, since they will often be recalled on the rest of the thesis.

The spacecraft is a 3U CubeSat (code-named SPEISAT from now on) designed and built by Polytechnic students and researchers with support from external companies for some subsystems, due to the strict deadlines imposed by the project. The main building blocks

are shown in fig. 3.3, some of them are grouped with others to form an unique spacecraft subsystem, as subsequently described.

### 3.2.1 Spacecraft subsystems

The spacecraft is composed of these main subsystems:

- Mechanical structure: The mechanical structure (fig 3.4) was fully developed by members of our team: at first a dummy mass model was produced in order to test the mechanical interfacing with the satellite deployer and perform some preliminary vibrational tests; then the final structure was designed, manufactured by an external company and integrated with the satellite avionics and deployer to perform some final vibrational tests and verify that launcher requirements were met.

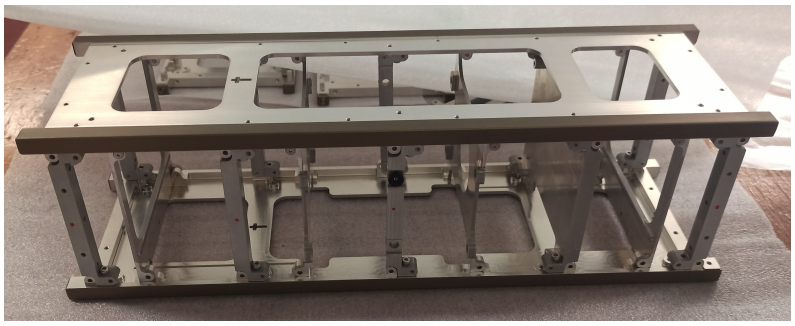
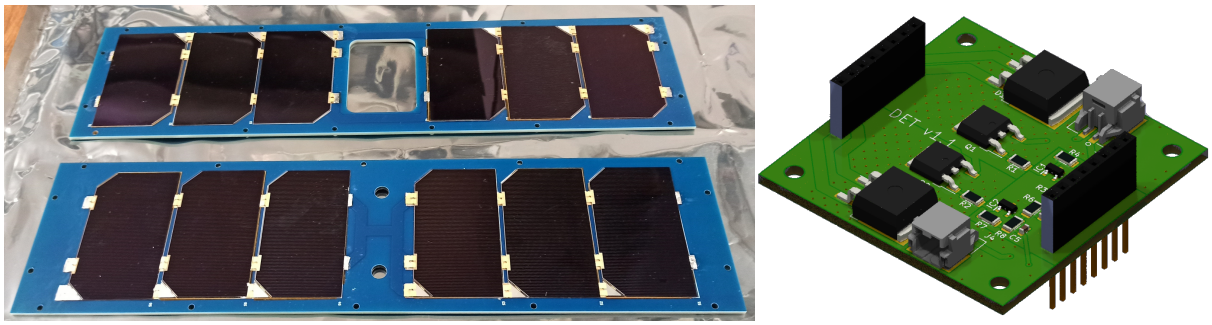


**Figure 3.4:** Spacecraft mechanical structure.

- Electric Power System (EPS) and Thermal Control System (TCS): Power generation/dissipation and thermal behavior are strictly tangled and of critical importance in space applications, where heat exchange between the spacecraft and the external environment only happens radiatively. The spacecraft power is generated by 24 GaAs solar cells, sponsored by CESI ([15]), mounted on four panels of six cells each (fig 3.5a). The panels were realized with PCBs that offer both mechanical support and interconnection to the cells and that were realized by members of our team; each panel has a theoretical maximum power generation of about 6W, but this figure is strongly influenced by orbital parameters and orientation with respect to the sun. We opted for the Direct Energy Transfer (DET) method due to the relaxed power requirements of the avionics and to keep the subsystem as simple as possible: in this method the solar panels are directly connected to the battery (through a reverse current blocking diode) instead of having a Maximum Power Point Tracking (MPPT) transformer, so they only require a limiting circuit to clamp the power bus voltage to the maximum battery voltage (as described in [30]). The limiter circuit was another system developed by the author (fig 3.5b). but was not treated in details in this thesis, it gets the name from the power transmission method (DET) and is basically a parallel shunt regulator based on TL431 with a redundant BJT power stage that dissipates the excess power to the satellite structure through shunt resistors or to the board ground plane through the BJTs (depending on the actual current to dissipate and so the transistor working region).
The battery pack was supplied by an external company and is a redundant 3S Li-Ion pack with self heating capability, needed to keep the batteries warm and being the only active Thermal Control System (TCS).

The spacecraft power bus only supplies the unregulated battery voltage to all subsystems: each one generates the needed voltage by itself on the board. The spacecraft has an idle power consumption of 3W, with peaks up to 8W during radio transmission; a recharge mode has also been provided with a 1.8W consumption, automatically activated if the battery voltage goes below a threshold, during which the spacecraft functionalities are reduced and some subsystems are turned off to let the battery level go up again. Our team performed simulations by integrating orbital and attitude predictions with electrical (SPICE, numerical) and thermal models to estimate the actual power budget and thermal profiles of the spacecraft, verification of these models are one of the purposes of Singer.



**(a)** **(b)**

**Figure 3.5:** 3.5a Solar panels after assembling. 3.5b DET board render.

● Attitude Control System (ACS): The attitude control system is very simple and implemented in passive way due to the fast development time of the mission: permanent magnets are placed along the mechanical structure to align the satellite to the earth magnetic field lines, while hysteresis rods are used to damp oscillations in the low friction space environment. Models about the attitude behavior of the spacecraft were developed and one of the purposes of Singer is collect data to verify this models.

● Command and Data Handling (Parrot subsystem): Command and Data Handling is performed by two processing boards in redundant configuration, the processing boards were sponsored by an external company and run an embedded Linux kernel on ARM processors. The boards manage communication with earth through the COMMSYS, receiving and dispatching commands to the software services and other subsystems and sending back telemetry and scientific data, as well as performing part of the satellite primary mission of periodically transmitting phrases of the Book, which is why they have been grouped together in the so called Parrot subsystem.

● Communication system (COMMSYS): Uplink and downlink capability is given by two transceivers, one for each CDH board and supplied by the same company of the latter. Communication with earth is performed in amateur UHF band, at a frequency of 437.500 MHz, with GMSK modulation at 9600 baud and AX.25 protocol for data link layer. The antennas, that are closed during launch and are subsequently deployed using heating elements, are two L-dipoles oriented in opposite directions with respect to each other on the same plane, orthogonal to the spacecraft z axis as can be seen in fig. 3.2a.

● Spacecraft interconnection (Backplane board): The spacecraft adopted the backplane board philosophy to interconnect the various subsystems, basically due to the CDH boards being designed for this approach. The backplane board was made by an external collaborator under specifications from our team, it provides interconnection and power delivery to the spacecraft building blocks and houses the spacecraft access ports (electrical interfaces with the spacecraft subsystems to be used when the spacecraft has been partially or fully integrated for testing and diagnostics). Fig. 3.6 shows the backplane board during spacecraft integration, with the DET board, the battery pack and one CDH already mounted on it. The spacecraft access port can be seen in fig. 3.2a and fig. 3.9.

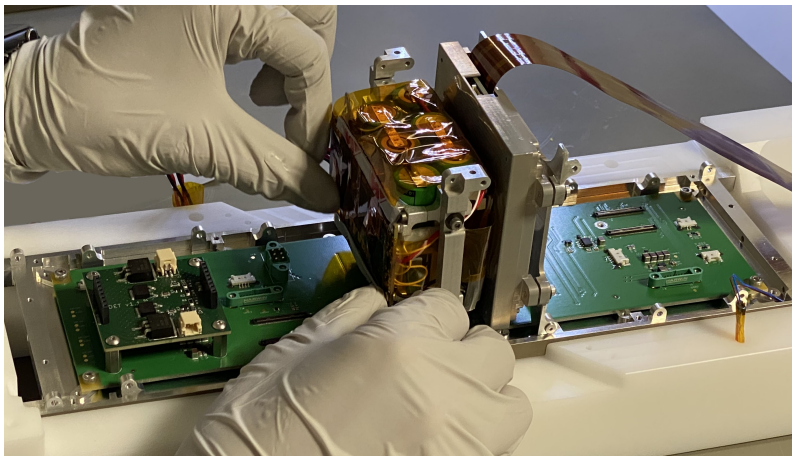


**Figure 3.6:** Backplane board during integration

● Sensing suite (Singer): This is the system covered on this thesis, and so will be treated exhaustively on following sections; its main goal is to gather telemetry data of scientific interest, specifically thermal and attitude data, being at the same time a platform for technology demonstrations and helping with the spacecraft health monitoring.

● Ground Station (GS): Even if not exactly part of the spacecraft, the ground station is obviously a fundamental part for mission operations and no less complex of the spacecraft itself. It consists of all the infrastructure needed to track an object moving in orbit, receive/transmit modulated RF signals, code/decode them and implement all the network stack layers and databases to store received data. Our team is working to build a ground station at the Polytechnic (fig. 3.7) but for SPEISAT we needed support from external partners, specifically the radio amateur community; more details about this aspect are covered in section 8.2.

**Figure 3.7:** Project render of the Ground Station at Polytechnic.

### 3.2.2 Spacecraft block diagram

Spacecraft subsystems are organized as shown in fig 3.8, the backplane is not shown but is responsible of all wired power and data connections between subsystems. Mechanical and thermal interfaces are also not shown.

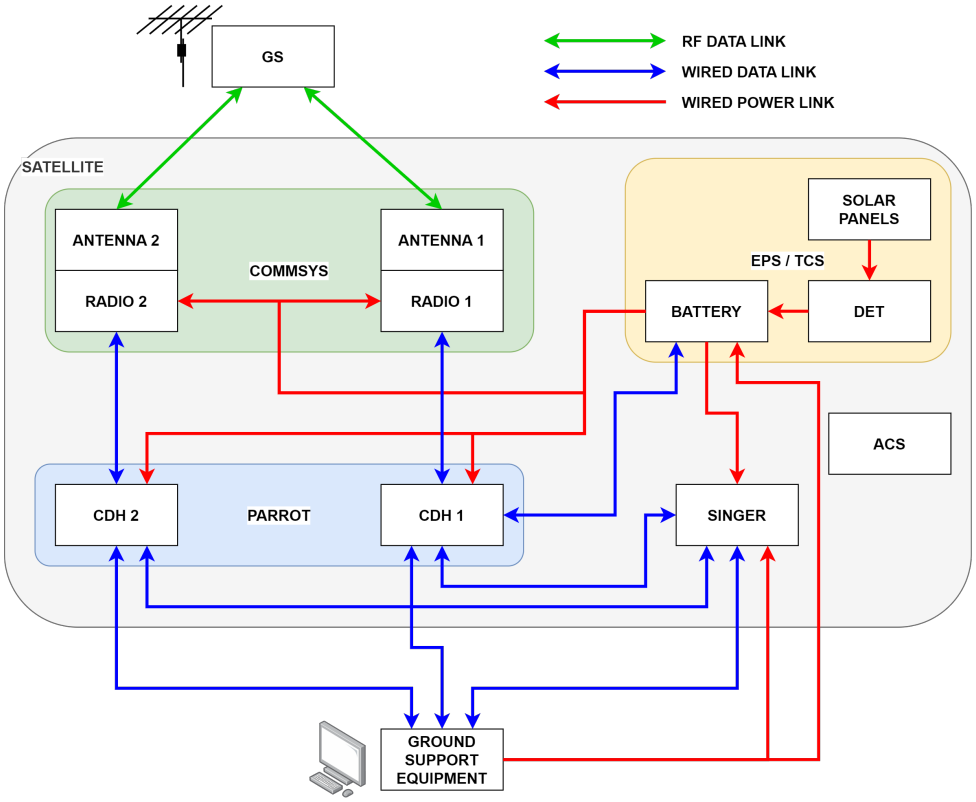



**Figure 3.8:** Spacecraft block diagram, green arrows are RF data links, blue arrows are wired data links, red arrows are wired power links.

Wired connection of the laboratory equipment (Ground Support Equipment or GSE) to the spacecraft (as shown in fig. 3.8) happens through the so-called Access Port (AP), this consists of a series of connectors (fig. 3.9) from which it's possible to power, program and debug the spacecraft subsystems.

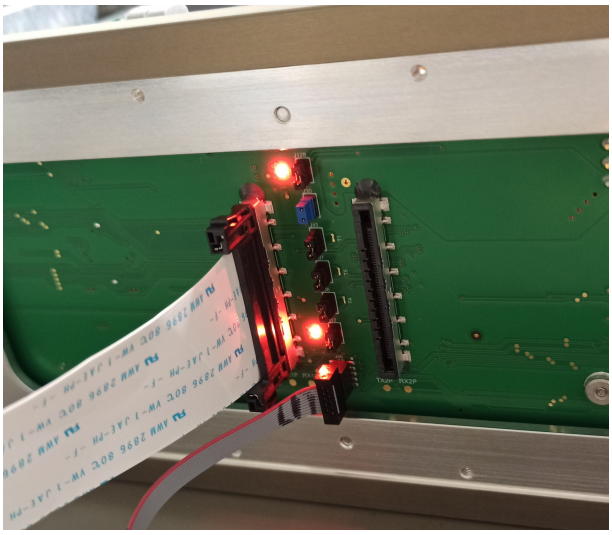


**Figure 3.9:** SPEISAT Access Port, here the solar panels are not yet mounted. We can see the AP cable of Singer (down, gray) and one CDH (up, white).

## 3.3 Spei Satelles team

The SPEISAT team was composed of students and researchers coming from different courses: most of them were from Aerospace engineering but there were also members from Electronics (like the author), Informatics, and Telecommunications. Fig. 3.10 summarizes the Spei Satelles team composition.

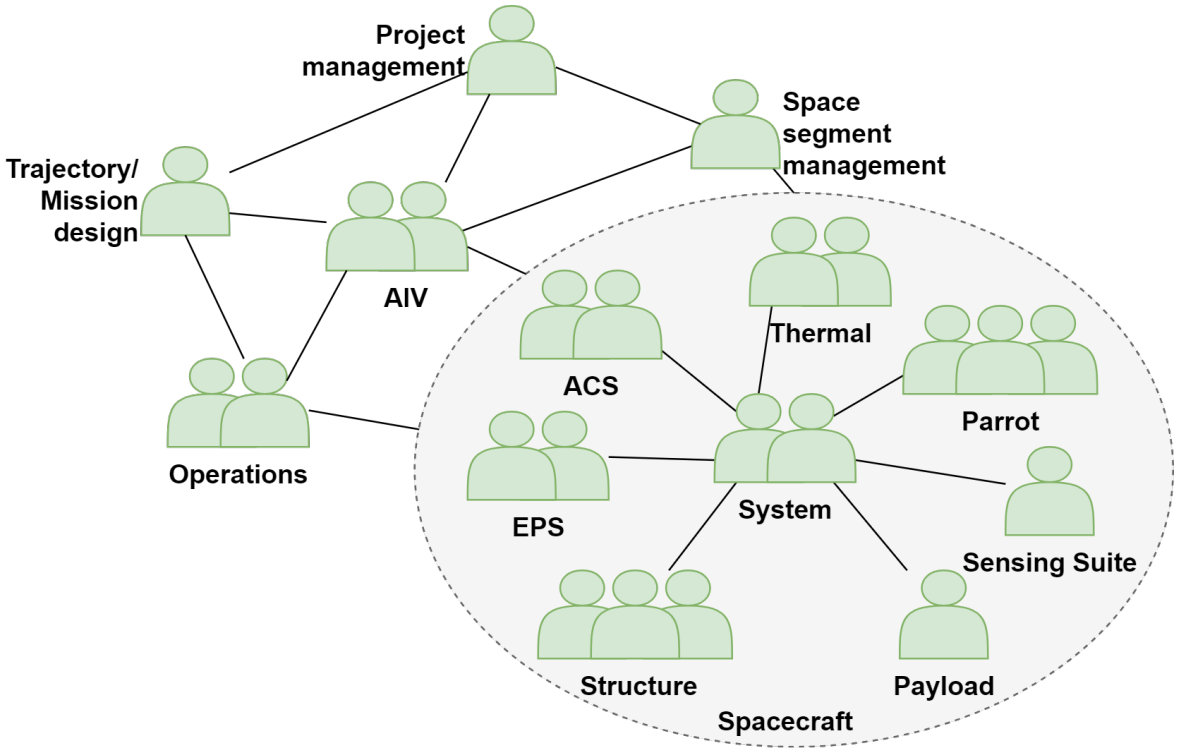



**Figure 3.10:** Spei Satelles team composition.

As can be seen, some of the personnel shown in fig. 3.10 worked on the SPEISAT system and various subsystems, but there were others crucial roles necessary for the success of the mission:

- Project management: responsible of managing the whole project and also all aspects non strictly related to engineering, like budget, personnel, public relationship and promotion;
- Space segment management: specifically responsible of managing the spacecraft design process;
- Trajectory/Mission design: responsible of mission design aspects, including orbital and environmental analysis and spacecraft operative modes;
- Assembly, Integration and Verification (AIV): responsible of following the building process by tracking the development status of the various subsystems, planning the spacecraft integration and testing activities;
- Operations: this group incorporates the designer of the Ground Station and the additional operators (some of them coming from other groups) necessary to follow the mission after launch.

# Chapter 4
# Sensing Suite (Singer) subsystem

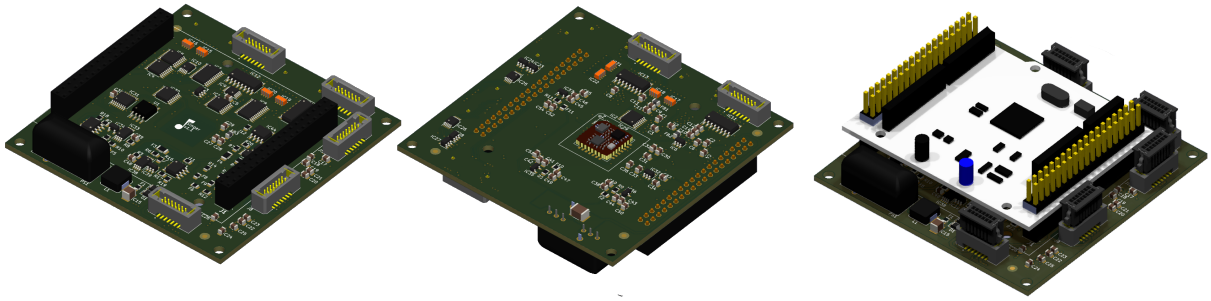


**Figure 4.1:** Singer board renders.

The Sensing Suite subsystem (fig. 4.1) implements the SPEISAT secondary mission and was developed by the author of this thesis, following all the design phases from system requirements definition to mission operation. The system formal name is Sensing Suite, later nicknamed as Singer to more easily identify and refer to it, in contrast with the Parrot subsystem which implements the primary mission.

## 4.1 SPEISAT Secondary mission

The main secondary mission goals are:

- Acquire temperature measurements on various points of the spacecraft to characterize the spacecraft thermal behavior and interaction with the space environment;
- Acquire attitude measurements to characterize the spacecraft passive ACS system.

Secondary achievements of the system are:

- Help checking the spacecraft health status by collecting an history of telemetry data alongside the scientific one;
- Verify the behavior of a low-cost electronic system built from COTS orbiting in LEO;
- Verify the application of system-level techniques for the mitigation of radiation effects in LEO;
- Verify the behavior of a commercial MRAM in LEO.

## 4.2 System requirements and specifications

Here a summary of the main Singer system requirements will be presented, divided on functional, performance and interface requirements; the low level details of each requirement will then be discussed on the following chapters. Nice-to-have requirements will be listed by labeling them accordingly.
Requirements are coded as three field strings with the format:
$< type > - < scope > - < number >$
requirement types are F (functional), P (performance), I (interface), the possible requirement scopes are the following:

- T: timing;
- S: sensing;
- M: memory;
- C: communication;
- A: Access Port;
- P: power;
- TH: thermal;
- MI: mechanical interfacing.

The verification method for each requirement is also listed, with the following options:

- Design: the requirement is satisfied by design, which means that the author did an analysis starting from the data available (for example in components' datasheets) to demonstrate that the requirement is satisfied, this verification is true for any requirement but some of them were verified only this way (see chapter 5);
- Test: the requirement is satisfied by rigorous testing (see chapter 7);
- Inspection: the requirement is satisfied by inspection, which means that a simple verification of the requirement was performed (typically by measurement or observation) without a rigorous testing plan, usually during the design phase (see section 7.2);
- Demo: the requirement was satisfied by a simplified demonstration of a limited use case which could not be considered a complete and rigorous test;
- N/A: Not Applicable, this is used for nice-to-have requirements that were not implemented;
- Accident: besides the other verification methods, some unwanted failures verified the requirement fulfillment (see section 7.6).

As can be seen, it was not always possible to test each requirement with rigorous testing,

due to lack of available time but most importantly of equipment.

### 4.2.1 Functional requirements

**Table 4.1:** Singer functional requirements.

| Code | Description | Verification method |
|---|---|---|
| F-T-0 | The system should be able to keep an epoch timestamp. | Test |
| F-T-1 | The system should be able to update the timestamp of req. F-T-0 from the interfaces of req. I-C-0 *note_1*. | Test |
| F-S-0 | The system should be able to measure up to 32 different temperatures. | Test |
| F-S-1 | The system should be able to measure 3-axis gyroscope and magnetometer data from an Xsens MTi-3 IMU *note_2*. | Test |
| F-S-2 | The system should be able to receive telemetry data from the interfaces of req. I-C-0 *note_1*. | Test |
| F-M-0 | The system should be able to store and retrieve in non-volatile mode the data from req. F-S-0, F-S-1, grouped into packets containing the last received telemetry of req. F-S-2, the current timestamp of req. F-T-0 and the current values from req. F-M-1 and F-M-2, labeled with an incremental packet counter. | Test |
| F-M-1 | (NICE-TO-HAVE) The system should be able to keep a counter with the number of system resets in the memory from req. F-M-0. | Test |
| F-M-2 | (NICE-TO-HAVE) The system should be able to keep a counter with the number of RAM or flash single and double bit flips in the non-volatile memory from req. F-M-0. | N/A *note_3* |
| F-C-0 | The system, under request from the interfaces of req. I-C-0, should be able to retrieve and send the last stored packet from the memory of req. F-M-0 (single telemetry mode) *note_1*. | Test |

| | | |
|---|---|---|
| F-C-1 | The system, under request from the interfaces of req. I-C-0, should be able to retrieve and send multiple requested packets from the memory of req. F-M-0, starting from the last stored and going backwards until the number of requested packets has been sent, no packets remain or the communication is stopped from the interface that sent the request (downlink mode) $^{note\_1}$. | Test |
| F-C-2 | (NICE-TO-HAVE) The system, under request from the interfaces of req. I-C-0 should be able to reset the non-volatile memory from req. F-M-0 $^{note\_1}$. | Test |
| F-A-0 | The system should allow reprogrammability and debug from the interface of req. I-A-0. | Test |
| F-P-0 | The system should be able to receive power from two different supply lines (specified on req. I-P-0), providing isolation between the two lines in order to avoid back-propagation of current from one to the other. | Test |
| F-P-1 | The system should isolate electrical faults without propagating them to the interfaces of req. I-C-0 and I-P-0. | Demo (simulated short), Accident (shorted components) |
| F-P-2 | (NICE-TO-HAVE) The system should implement techniques to reduce the effects of latch-up events and improve the system resilience by isolating the failure of non vital sub-elements without compromising the rest of the system. | Demo (simulated latchup) |

$^{note\_1}$ The communication protocol is described in details in sections 6.2.6.1 and 6.3.4.1.
$^{note\_2}$ The IMU model was selected by the ACS team and was given to the author as a requirement, more details are given in section 5.2.2.
$^{note\_3}$ This was a nice to have requirement that was implemented at non-volatile memory level (a field in the non-volatile memory storage was inserted) but for time issues was never implemented at functional level (the actual error detection).

### 4.2.2 Performance requirements

**Table 4.4:** Singer performance requirements.

| Code | Description | Verification method |
|---|---|---|
| P-T-0 | The timestamp of req. F-T-0 should have precision of 1 second and accuracy of at least ± 10%. | Inspection |
| P-S-0 | The measurement accuracy of req. F-S-0 should be ± 2 °C. | Design |
| P-S-1 | The measurement range of req. F-S-0 should be -40 °C to 85 °C. | Demo |
| P-S-2 | The sampling of req. F-S-0, F-S-1 and consequent storage (as on req. F-M-0) should happen every minute with time accuracy of ± 3%. | Inspection (timing), Test (software) |
| P-M-0 | The storage capacity of the memory from req. F-M-0 should be enough for at least 12 hours of data. | Test |
| P-M-1 | The packet counter from req. F-M-0 should not overflow in less than 10 years. | Design |
| P-M-2 | The packet dimension from req. F-M-0 should not exceed 150 bytes. | Design |
| P-C-0 | The system should respond to the requests of req. F-C-0, F-C-1 and F-C-2 in less than 350 ms. | Inspection (timing), Test (software) |
| P-C-1 | The system should send the packets of req. F-C-1 with a period of 300 ms (± 5 ms). | inspection (timing), Test (software) |
| P-P-0 | The system should work with a power supply range of 9.5V to 13V and a nominal power consumption of less than 300mW. | Test |
| P-TH-0 | The system operating temperature range should be -40°C to +85°C. | Demo (reduced temperature range) |
| P-MI-0 | The system should resist the mechanical stresses of the launch [note_1]. | Test |
| P-MI-1 | The system should correctly operate in vacuum. | Design |

[note_1] Expected mechanical stresses during launch were given by the launch providers and the structure team guided the author on fulfilling this requirement.

### 4.2.3 Interface requirements

**Table 4.6:** Singer interface requirements.

| Code | Description | Verification method |
|---|---|---|
| I-MI-0 | The system should provide mechanical fixing points and comply with the maximum mechanical dimensions specification of fig. 4.2 [note_1]. | Inspection |
| I-MI-1 | The system should have an overall weight (cables included) of less than 250g. | Inspection |
| I-A-0 | The system should provide an Access Port (AP) interface that satisfies req. F-A-0. | Test |
| I-C-0 | The system should provide a communication interface to fulfill req. F-T-1, F-S-2, F-C-0, F-C-1 and F-C-2, this interface should provide two independent, unterminated RS422 differential serial lines with 3.3V nominal voltage for each line, implementing an UART communication protocol with speed of 115200 baud, 8 bit data and no parity bit. | Inspection (RS422), Test |
| I-P-0 | The system should provide two independent supply lines as specified on req. F-P-0, one line should be on the same connector of the interfaces of req. I-C-0, the other should be on the same connector of the interfaces of req. I-A-0. | Test |

[note_1] Mechanical specifications were discussed with the mechanical structure team and also included the maximum extension and cable dimensions for all temperature sensors and backplane connection cables, this will not be presented here.

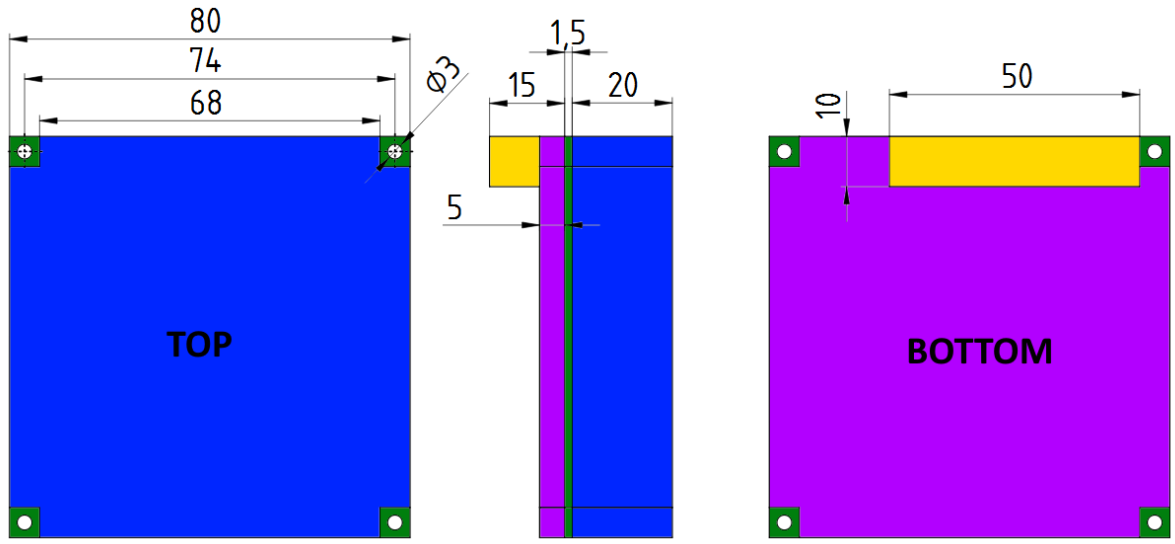



**Figure 4.2:** Singer maximum mechanical dimensions (unit of measure for all dimensions: mm), all dimensions except the mounting holes diameter and spacing are the maximum allowable, mounting holes diameter and spacing have a tolerance of ± 0.1 mm.

## 4.3 Design process

This section wants to be a review of the Singer subsystem design process, here the main challenges that were faced will be presented, followed by an overview of the applied methodology and a timeline of the various design phases.

### 4.3.1 Challenges

#### 4.3.1.1 Low development time

The design effort for SPEISAT was heavily influenced by the very short design time that our team had available: the whole spacecraft was built from scratch in a period of around 5 months; this short amount of time had the biggest impact on many design choices that were made for the whole project:

- The possibility of carrying out only one manufacturing cycle led to choices that would reduce the risk of improper design and fine-tuning and allow the execution of simple hardware debugs and fixes. Electronic devices were selected by favoring bigger packages with exposed leads (no DFN, QFN or BGA packages, preferring SOIC packages with 1.27 mm pitch if available) and passive components were mainly selected with 0603 form factor (0.6 in x 0.3 in) or higher. Already integrated building blocks were chosen whenever possible: the processing unit is a microcontroller dev-board in daughter board configuration (see last image on fig. 4.1), allowing replacement of the unit if needed, an easy access to the microcontroller GPIOs and also improving the testability; for the power supply regulation, a fully integrated buck-converter System In Package (SIP) was chosen; for the attitude measurements a System On Module (SOM) Inertial Measurement Unit (IMU) was selected; this choices also helped reducing the development time while slightly

increasing the cost figure.

- The procurement bottlenecks, due to the relatively long shipment and manufacturing delays with respect to the available design time, led to the need of careful planning and arrangement of design phases in order to exploit any available man-hour at its best. The system requirements definition was performed in such a way as to give priority to those that resulted in hardware selection and purchase, delaying those relating to purely software aspects, following the timeline described in section 4.3.2.

#### 4.3.1.2 Poor system reliability

Another important driver was the lack of precise figures about the expected system reliability, both due to the relative scarcity of existing documentation about the behavior in high radiation environment of commercial COTS not designed for this purpose and of a solid failure analysis which would have needed these figures to be performed:

- The system was designed with the assumption that any component could fail at any time and that this should not influence the correct operation of the system but most importantly should not propagate to the rest of the spacecraft. This led to the implementation of multiple, isolated voltage domains on the Singer board and of various power supply protection techniques, as described more in details on section 5.2.6.2.

- Latch-up protection techniques were implemented to try reducing the effects of this events on the electronics.

- Whenever possible, the most rugged components available were chosen: active devices were selected with automotive grade temperature range of -40°C/-55°C to +125°C if available and passive capacitors were all selected with flexible terminations to enhance the board resistance to mechanical stresses.

#### 4.3.1.3 Reduced impact on primary mission

Of particular importance was the decision to impact as little as possible the primary mission functionality (carried out by Parrot) with the secondary one (carried by Singer), to reduce the risk of adding complexity and so possible points of failure on the former, for this reason the functions related to the secondary mission were relegated as much as possible to the Singer board, an example can be the storage of measurements that could have been done by the Parrot subsystem, already equipped with non volatile memory, but instead was completely assigned to Singer. The only thing that Parrot does is to forward packets between Singer and the COMMSYS. This came with the advantage of being able to exploit the Singer storage capability to also periodically record Parrot telemetry, enriching the scientific measurements sent back to earth with an history of the spacecraft house-keeping data.

#### 4.3.1.4 Cost

Last but no less important is the cost figure. In this thesis the actual costs will not be declared but they obviously played a decisive role on components choice. The use of COTS is mainly dictated by the very high cost of space-grade electronics.

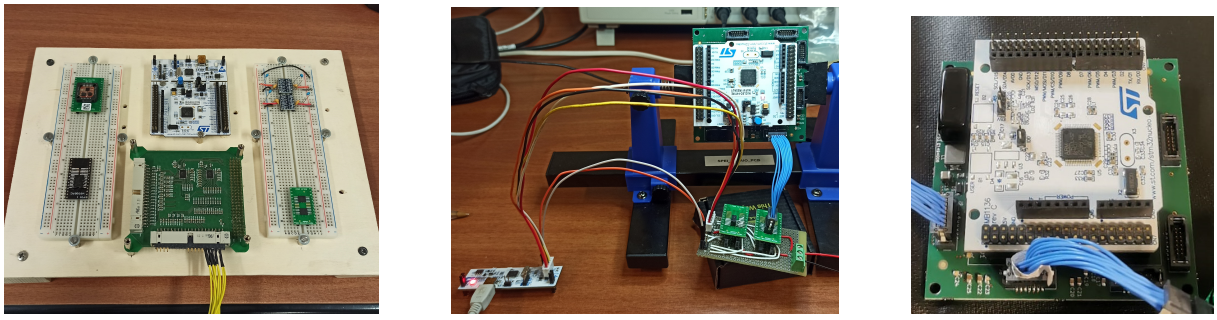
### 4.3.2 Design timeline

The Singer design was characterized by three major phases, each corresponding to a different model of the system:

- Breadboard model: after a preliminary analysis and the definition of the main hardware requirements a first procurement cycle was made to build this first concept model of the system and start working on low level software (device drivers), the breadboard model (that can be seen in fig. 4.3a) was built both from hardware specifically bought from the project and some already available in the laboratory, it helped refining the hardware requirements and begin to outline the high level software architecture;

- Qualification model: a second procurement was made, including the Singer Printed Circuit Board (PCB) manufacturing to realize the so-called qualification model (fig. 4.3b); this was the longest phase of the design effort because it led to the finalization of the hardware (after acceptance tests and error corrections on the board, see chapter 7) and of the high level firmware; the qualification model development was performed in parallel with the realization of a Flat Sat (FS) model (fig.4.4) and an Exposition Model of the spacecraft and was eventually integrated with them. The FS was the bench model of the spacecraft and the main testing platform, it was composed of all the avionics and could be reconfigured as needed. The integration of Singer with the rest of the FS gave the possibility of verifying the electrical interfaces with the backplane and test the communication channels with the CDHs, ultimately leading to the firmware completion. The exposition model was a non-functional copy of the spacecraft that was useful to test its configuration and integration procedures, finally leading to its use for promotional purposes.

- Flight model: this was the final model of the system and the one that would be launched, so it was treated with the greatest possible care; it faced hardware modifications aimed at preparing the system for flight (better explained in section 7.4), was later washed and from that point only handled inside the clean room. It faced acceptance tests before and after flight preparation to certificate its operability and was subsequently integrated inside the spacecraft; a final test campaign was then performed with the whole integrated system that allowed the identification and correction of final minor bugs on the software.



**(a)** **(b)** **(c)**

**Figure 4.3:** 4.3a Singer breadboard model. 4.3b Singer qualification model during testing. 4.3c Singer flight model, modifications on the Nucleo board for flight preparation can be spotted (components removal).

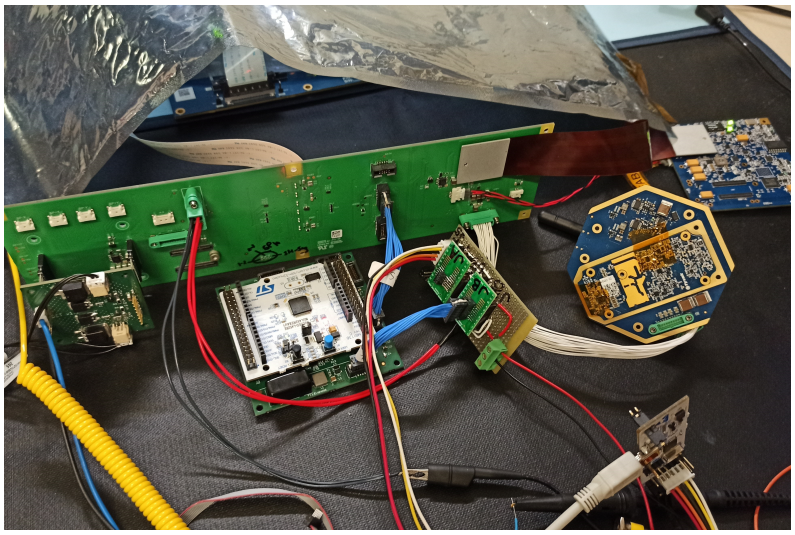


**Figure 4.4:** Spacecraft Flat Sat.

Fig. 4.5 contains the main Singer development milestones: an approximate timeline of the subsystem evolution was made and compared with the various spacecraft models, time is approximated and not to scale.

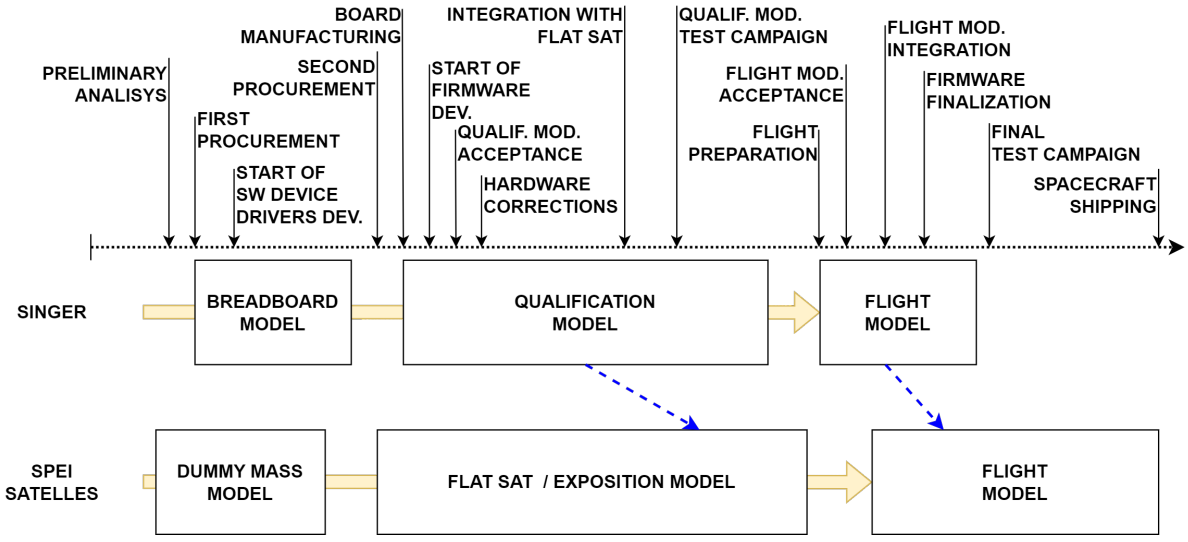



**Figure 4.5:** Singer development milestones.

# Chapter 5
# Hardware design

The goal of this chapter is to address the aspects related to the system hardware design; the system architecture will be presented, followed by an analysis of the various board building blocks and the rationales that guided the author in component selection and configuration; finally, some details about the PCB design and production will be presented.

## 5.1 System architecture

In fig. 5.1 the system block diagram is shown.

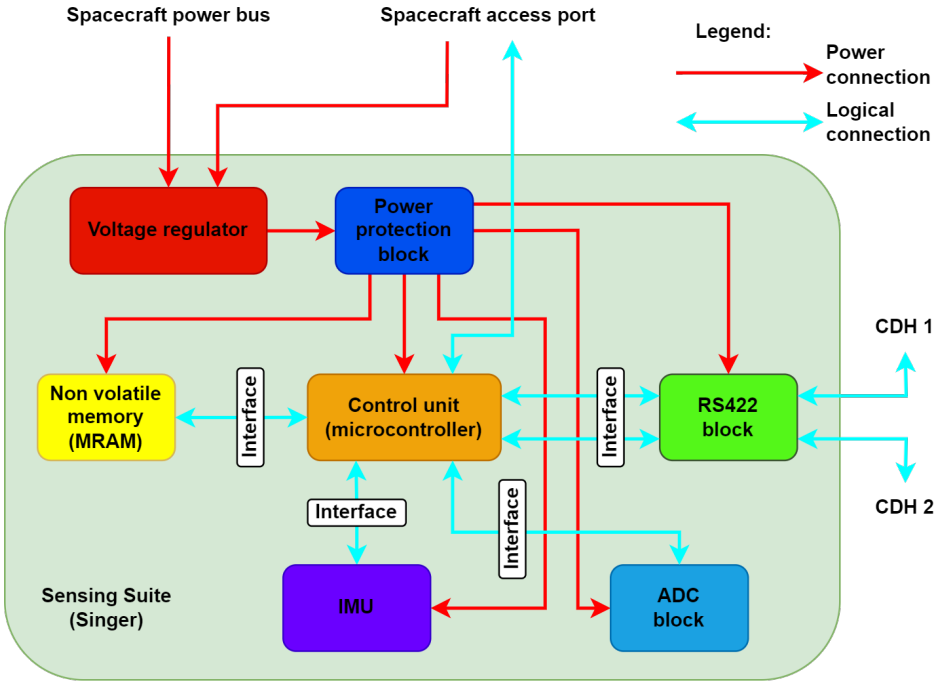



**Figure 5.1:** Singer board block diagram.

The system is composed of:

- An ADC block that can measure up to 32 temperatures, consisting of 32 thermistors and relative conditioning, two 16-bits analog multiplexers and two Analog to Digital Converters (ADC);

- An Inertial Measurement Unit (IMU) System On Board;
- A control unit consisting of a microcontroller System On Board, able to be programmed and debugged from outside the spacecraft through the Access Port;
- A non volatile memory consisting of an Magnetoresistive Random Access Memory (MRAM) chip;
- An RS422 block, consisting of two different RS422 transceivers, each one implementing a communication channel versus one Command and Data Handling (CDH) board;
- A buck voltage regulator that provides 3.3V supply to the whole system from the spacecraft power bus or the AP;
- A power protection block, consisting of different types of circuits to individually protect each system element from latch-up and isolate a faulty element from the rest;
- Interface blocks to isolate faulty elements on the logical interconnections side;

In section 5.2 all the system building blocks will be individually analyzed.

## 5.2 Electrical design

This section will analyze each system building block at the electrical level, providing information about the devices electrical characteristics and the design choices that were made in their selection.

### 5.2.1 ADC block

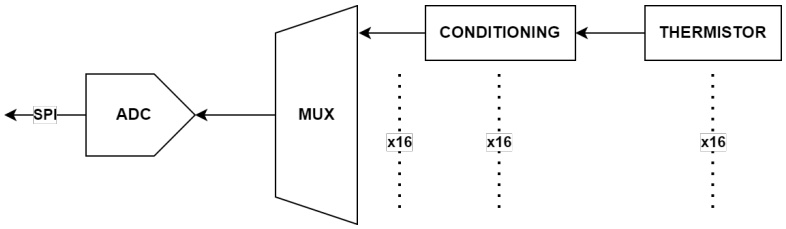
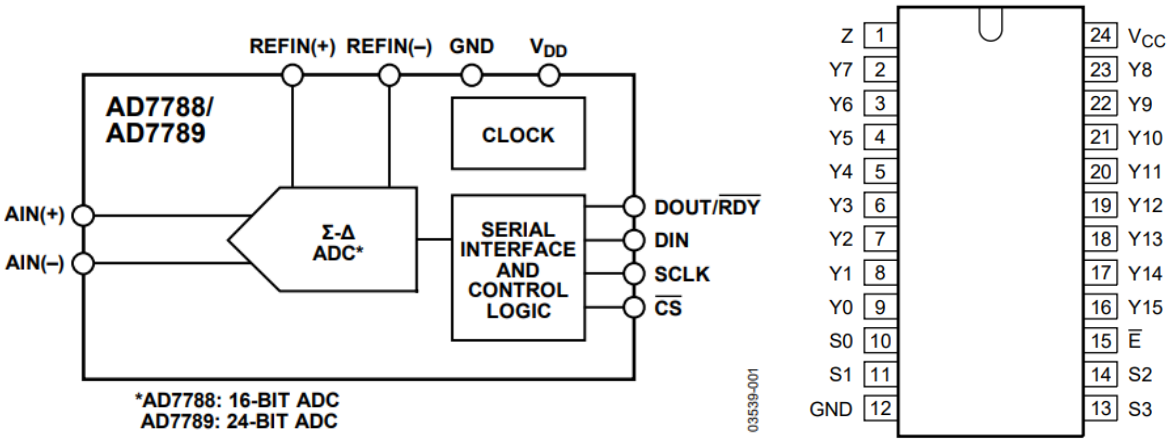



**Figure 5.2:** Singer single ADC chain block diagram.

Temperature measurement is carried out on up to 32 different channels and doesn't have strict requirements on measurement accuracy. After trading-off between conversion speed and PCB area usage, the choice was to use two single-channel AD7788 $\Sigma$-$\Delta$ ADCs from Analog Devices (datasheet: [55], fig. 5.3a), each coming with a 74HC4067 16-channel analog MUX (datasheet: [4], fig. 5.3b).
This ADC has a very low power consumption (75 $\mu$A max. with 3.3V supply, so 248 $\mu$W) and is particularly conceived for low frequency, high dynamic range measurements, as stated in the product datasheet. A single ADC chain is shown in fig. 5.2.

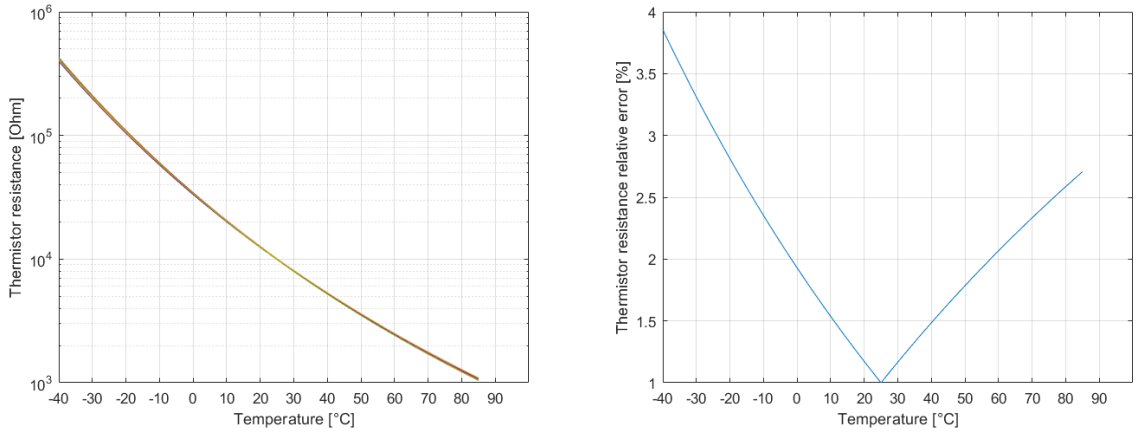



**(a)** **(b)**

**Figure 5.3:** 5.3a AD7788 Σ-Δ ADC block diagram, picture from [55]. 5.3b analog MUX pinout, picture from [4].

#### 5.2.1.1 Measurement chain characterization

The chosen temperature sensor model is NTCLE203E3103FB0 thermistor from Vishay (datasheet: [67]), this thermistor has a nominal resistance @ 25°C of $R_{25}$ = 10 kΩ ±1% and a $B_{25/85}$ = 3977 K ±0.75%. The $B_{25/85}$ (from now only called B) allows computing the thermistor characteristic curve,as explained in Vishay application note [66], the equation that defines this value is $B = \ln \frac{R_1}{R_2} * \frac{1}{\frac{1}{T_1} - \frac{1}{T_2}}$, where $T_x$ and $R_x$ represent a chosen absolute temperature [K] $x$ and the corresponding thermistor resistance value [Ω]. From this equation and by knowing the thermistor resistance value @ 25°C from the datasheet, we can compute the thermistor resistance variation with temperature using the following equation: $R_1 = R_2 * e^{(\frac{1}{T_1} - \frac{1}{T_2}) * B}$. The B factor should be constant in a first approximation, in reality it slightly varies with temperature, as stated in the application note, but to get more precise results one should get a number of coefficients representing the factors of a polynomial fitting curve that are not given for this particular component.
In fig. 5.4a we can see the computed characteristic curve, with logarithmic y-axis, it's difficult to see because they are very close but this graph includes various curves representing the combinations of tolerances on $R_{25}$ and B, these tolerances result in the relative error shown in fig. 5.4b; the thermistor resistance varies from a value of 412 kΩ @-40°C to a value of 1.07 kΩ @85°C.




**(a)** **(b)**

**Figure 5.4:** 5.4a NTCLE203E3103FB0 thermistor characteristic curve generated by MATLAB. 5.4b Thermistor resistance relative error.

The thermistor's conditioning circuit is very simple due to the great number of channels and only consists of a 5 kΩ pull-up resistor in series with the thermistor and a 10 nF filtering capacitor, as shown in fig. 5.5.

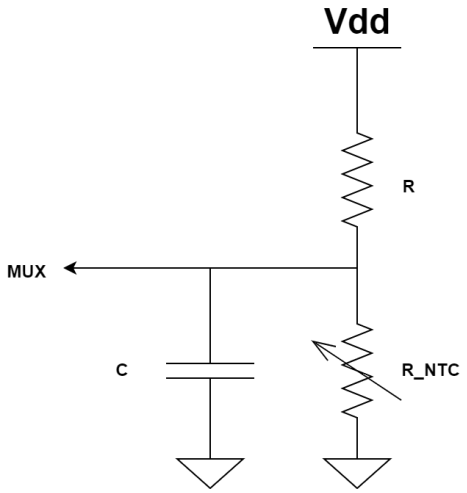



**Figure 5.5:** Thermistor conditioning circuit.

The capacitor helps filtering out high frequency noise, for now let's ignore it and compute the Thevenin equivalent circuit (as can be seen in fig. 5.8), resulting in an equivalent voltage of:

$$V_{eq} = V_{dd} * \frac{R_{NTC}}{R_{NTC} + R} = V_{dd} * \frac{1}{1 + \frac{R}{R_{25}} * e^{(\frac{1}{(298.15K)} - \frac{1}{T}) * B}} \tag{5.1}$$

and equivalent series resistance of:

$$R_{eq} = R_{NTC} || R = \frac{R_{NTC} * R}{R_{NTC} + R} = \frac{R}{1 + \frac{R}{R_{25}} * e^{(\frac{1}{(298.15K)} - \frac{1}{T}) * B}} \tag{5.2}$$

We can invert eq. 5.1 to obtain the temperature corresponding to a given $V_{eq}$:

$$T = \frac{(298.15K) * B}{B - (298.15K) * ln[(\frac{V_{dd}}{V_{eq}} - 1) * \frac{R_{25}}{R}]} \tag{5.3}$$

The chosen component for R is a resistor with a nominal value of 5 kΩ with tolerance of 0.1% and a temperature derating of ±25 ppm/°C, yielding an additional worst case deviation in the specified working temperature range of the system of 8.1 Ω (0.17%) for a total rounded tolerance of 0.3%. The power supply will be considered later as part of the ADC errors.
In fig. 5.6 we can see the values from eq. 5.1 and eq. 5.2, as expected we have a maximum of $V_{eq}$ and $R_{eq}$ at low temperatures, when the thermistor resistance is higher, the value of $V_{eq}$ varies from 3.26 V @-40°C to 0.58 V @85°C, while $R_{eq}$ goes from 4.94 kΩ @-40°C to 881 Ω @85°C.

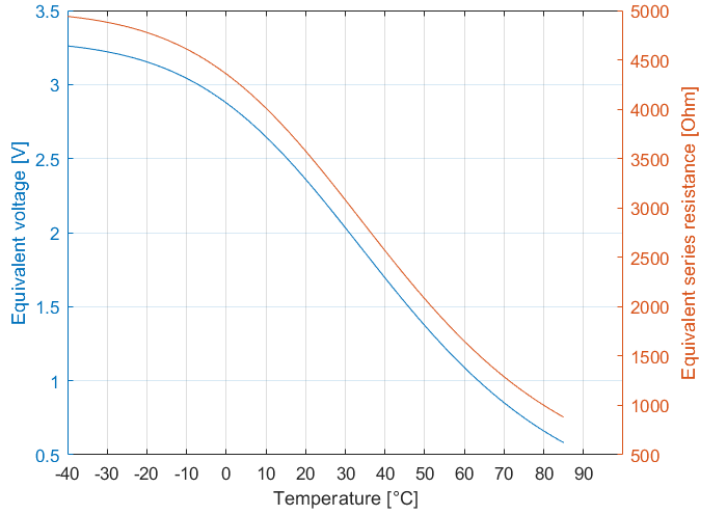



**Figure 5.6:** Thevenin equivalent values with respect to temperature with nominal values of thermistor and pull-up.

The numerical simulation pipeline of fig. 5.7 was set up to perform a worst-case analysis of the circuit, including all components tolerances.

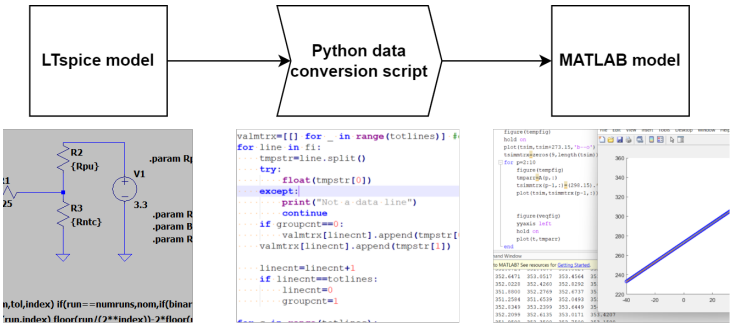



**Figure 5.7:** Simulation pipeline for thermistor chain.

The simulation pipeline includes an LTspice (website: [56]) electrical model of the circuit from which raw simulation data coming from multiple simulations can be exported in text format; this data is rearranged by a Python (website: [77]) script that removes the human readable text and reorganizes data coming from different simulation into a single matrix-like output file; finally this data is fed to a MATLAB (website: [61]) script that

performs heavy calculation and generates graphs.
The SPICE simulation was set up to perform a worst-case analysis, testing all the possible combinations of components tolerances and a final nominal simulation, as explained in [57].
Now we can compute the error on $V_{eq}$ measured by the ADC ($V_{ADC}$), by considering the ADC input resistance $R_{ADC}$ and its effect due to the MUX parasitic resistance $R_{ON}$ and $R_{eq}$, in order to do that we need an estimation of $R_{ON}$ and $R_{ADC}$.The equivalent model can be seen in fig. 5.8.

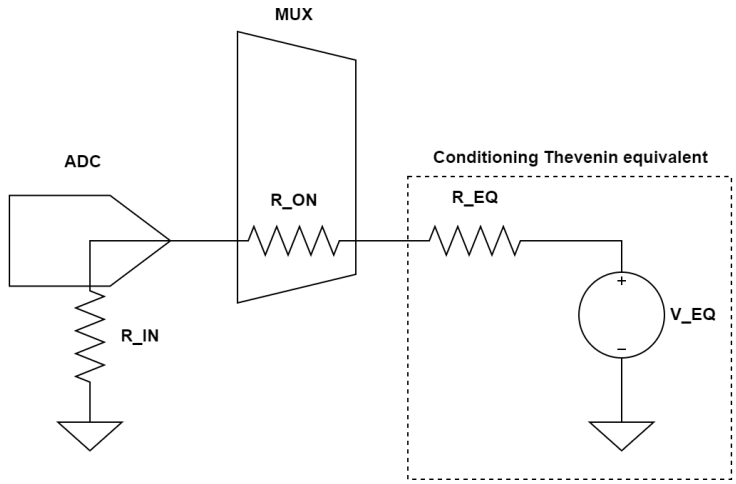



**Figure 5.8:** Conditioning circuit Thevenin equivalent and parasitic resistances of MUX and ADC.

The ADC datasheet only states an average input conductance $G_{ADC} = 400$ nA/V with a temperature derating of ±50 pA/V, stating that this value is an assumption based on design and not proven and giving no tolerance for it; since no more information is given we will consider it a good approximation of the nominal value, resulting in an $R_{ADC}$=2.5MΩ and making a reasonable assumption for its tolerance to be ± 10%; the temperature derating adds a variation of ± 3.25 nA/V over the full working range of the system (-40°C to +85°C) and we' ll assume that this is included in the $R_{ADC}$ tolerance figure.
The MUX $R_{ON}$ is instead specified as a typical value of 110 Ω @25°C and maximum value of 180 Ω @25°C or 225 Ω @ 85°C; this is in reality specified with a 4 V supply voltage, but since its value is quite lower than the minimum $R_{eq}$ of the conditioning circuit, it will have a smaller impact with respect to the latter, so considering a nominal $R_{MUX} = 112.5$ Ω ± 100% (so a value ranging from 0 to the maximum @85°C) is reasonable and probably even exaggerated. The final SPICE model of the circuit can be seen in fig. 5.9.

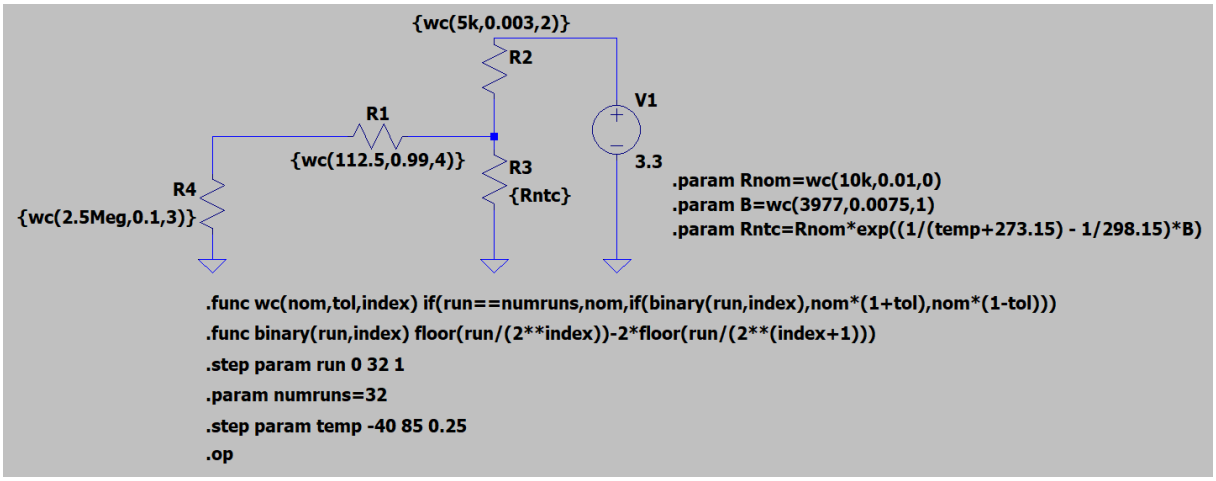



**Figure 5.9:** Model of thermistor circuit in LTspice.

A first simulation with nominal values gives the results of fig. 5.10: in this figure eq. 5.3 is used to reconstruct the measure (neglecting the effect of the ADC input resistance) and the absolute error introduced with respect to the true (simulated) temperature is evaluated.

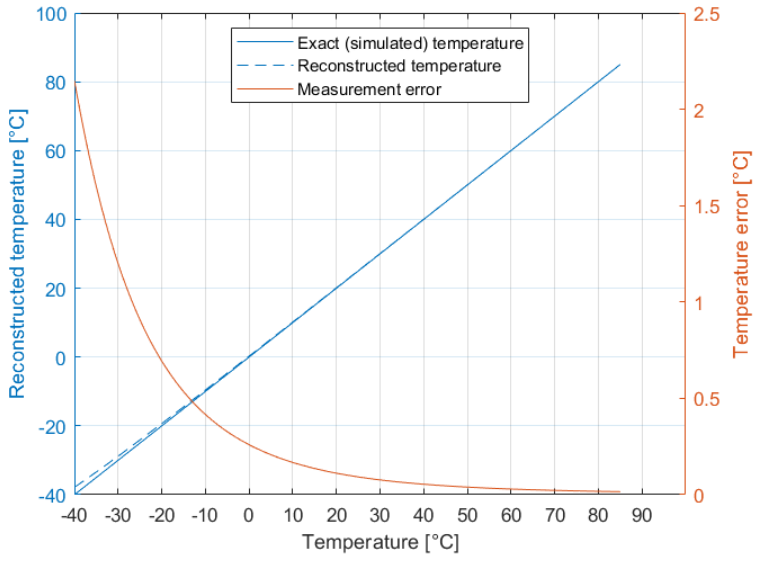
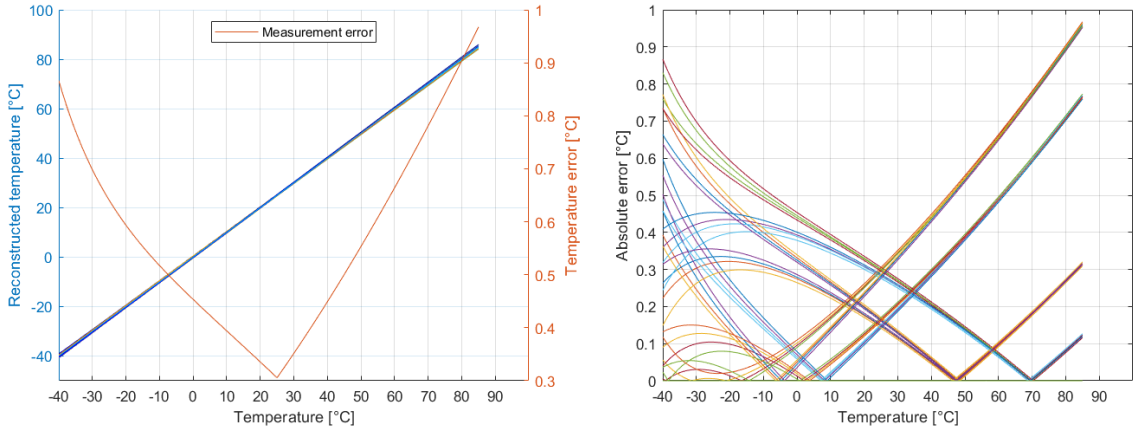



**Figure 5.10:** Comparison between exact temperature and the one reconstructed neglecting ADC input resistance (left axis). Absolute error introduced (right axis).

We can see that the effect of the ADC input current is particularly relevant with low temperatures, giving an error of 2.2°C @-40°C; this makes sense since it's when the equivalent resistance has it's maximum value (fig. 5.6). This effect can be compensated during measurement reconstruction; the most precise compensation method is to analytically resolve the network of fig. 5.8 to obtain the temperature reconstruction formula from the voltage $V_{ADC}$ measured by the ADC:

$$T = \frac{(298.15K) * B}{B - (298.15K) * ln[(\frac{\frac{V_{dd}}{V_{ADC}} - \frac{R}{R_{ADC}}}{1 + \frac{R_{MUX}}{R_{ADC}}} - 1) * \frac{R_{25}}{R}]} \tag{5.4}$$

Applying this equation results in a perfect cancellation of the effect of ADC input impedance in nominal conditions.
The next step is to apply the reconstruction eq. 5.4 to all the curves resulting from the worst-case analysis and evaluate the total absolute error due to component tolerances; a total of 32 simulations were automatically performed by spice to test all the possible combinations of tolerances on $R_{25}$, B, the R pull-up resistor, $R_{MUX}$ and $R_{ADC}$. The result can be seen in fig. 5.11a, in this graph the temperature reconstructed from each simulation, using equation 5.4, is plotted on the left axis; the lines are very close and it's difficult to distinguish them but on the right axis the maximum absolute error is also reported, from which we can conclude that the maximum absolute error introduced by component tolerances is $Err_{tol}$=0.97°C.




**(a)** **(b)**

**Figure 5.11:** 5.11a Temperature reconstructed considering tolerances(left axis). Maximum absolute error introduced (right axis). 5.11b Absolute error for each worst-case simulation.

In fig. 5.11b we can have a better view of the different impact of tolerances on the simulation results: here all the 32 error lines are plotted, we can notice how at high temperatures they are mainly bundled in four groups, corresponding to the thermistor tolerances, that's because in this condition the $R_{eq}$ is very low and mainly determined by the thermistor resistance, the effect of ADC impedance is reduced; instead at low temperatures the effect of the ADC input current starts to become significant and the other components tolerances have a more noticeable effect. The highest values among all the curves of fig. 5.11b are the one reported in the right axis of fig. 5.11a.
We can now evaluate the ADC errors from the values stated in the datasheet: 16 bits of precision corresponding to a quantization error of ± 26 $\mu$V, INL of ± 15 ppm of full scale corresponding to a nonlinearity error of ± 50 $\mu$V, full scale linearity error of ± 10 $\mu$V (since the ADC gain and offset errors are not compensated we will use the full scale linearity error that includes both). From these values we obtain a total ADC error of around 86 $\mu$V, to estimate the impact of this error on temperature measurement we can linearize eq. 5.4 with respect to $V_{ADC}$, obtaining the following derivative:

$$\frac{\partial T}{\partial V_{ADC}} = -(298.15K) * \frac{1}{(1 - \frac{(298.15K)}{B} * ln[(\frac{\frac{V_{dd}}{V_{ADC}} - \frac{R}{R_{ADC}}}{1+\frac{R_{MUX}}{R_{ADC}}} - 1) * \frac{R_{25}}{R}])^2} * \\ *(\frac{298.15K}{B} * \frac{1}{\frac{\frac{V_{dd}}{V_{ADC}} - \frac{R}{R_{ADC}}}{1+\frac{R_{MUX}}{R_{ADC}}} - 1} * \frac{V_{dd}}{1 + \frac{R_{MUX}}{R_{ADC}}} * \frac{1}{(V_{ADC})^2}) \tag{5.5}$$

In fig. 5.12 the equation 5.4 and its derivative (eq. 5.5) have been plotted in nominal conditions; vertical lines represent the measured voltage range in the -40°C to +85°C temperature range, these correspond to a range of $V_{ADC}$ of 3.26 V to 0.58 V. The maximum computed sensitivity corresponds to the -40°C extreme and was evaluated to be -384 °C/V, we can then compute the measurement error introduced by ADC non-idealities $Err_{ADC}$ = 384 °C/V * 86 $\mu$V = 0.033 °C, practically negligible with respect to the error introduced by component tolerances.

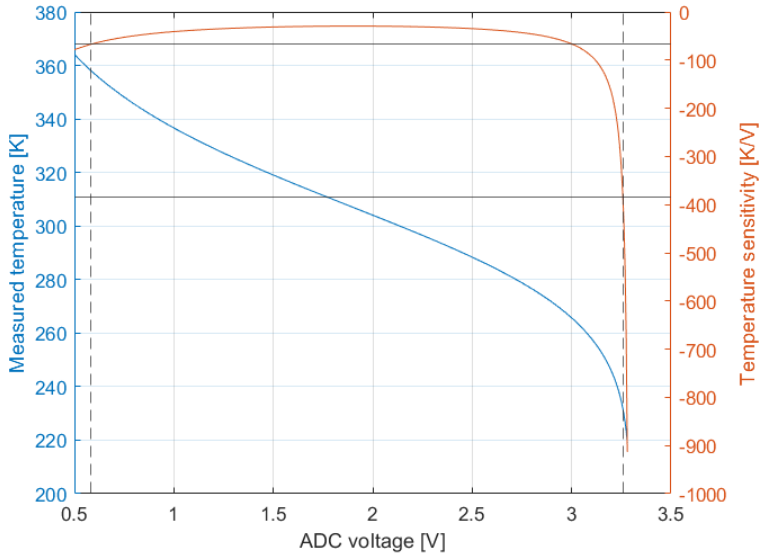



**Figure 5.12:** Temperature reconstruction equation (left axis) and it's sensitivity with respect to measured voltage (right axis); vertical lines represent the measurement range of -40°C to 85°C, horizontal lines the corresponding sensitivities.

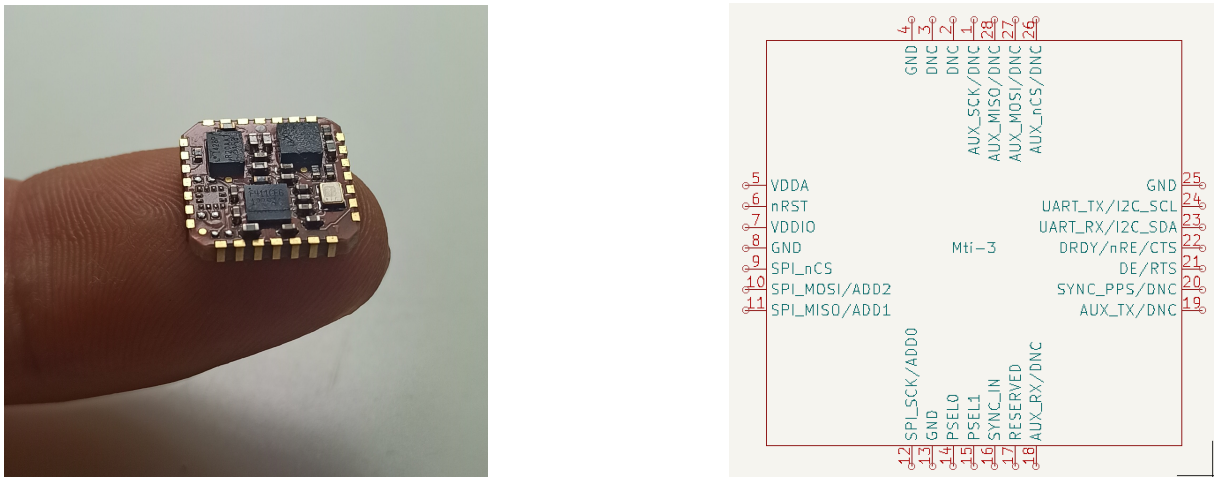
As explained in section 5.2.6.2, the ADC has an additional cause of error (a quite big one) coming from the fact that the supply voltage of ADC and thermistors is different. The ADC is powered by the same supply of the thermistors, but it has a PTC fuse in the middle with a maximum resistance of 50 Ω, so the error introduced by this element must be considered on the measurement error. The actual value of PTC resistance can go from 6 Ω to 50 Ω while the ADC+MUX current (the latter evaluated interpolating the currents stated on the datasheet @10V and @6V of supply) can reach a maximum of 80 $\mu$A; this introduces a drop on the ADC supply of 50 Ω * 80 $\mu$A = 4 mV, part of this value can be compensated on eq. 5.4 by considering the drop as a difference between the supplies of 2 mV ± 2 mV, so eq. 5.4 becomes:

$$T = \frac{(298.15K) * B}{B - (298.15K) * ln\{[(\frac{V_{dd,therm} * 2^{16}}{(V_{dd,therm} - 2mV) * D} - \frac{R}{R_{ADC}})/(1 + \frac{R_{MUX}}{R_{ADC}}) - 1] * \frac{R_{25}}{R}\}} \tag{5.6}$$

where D is the digital value on the ADC output. The error introduced on the measurement by the tolerance on this difference is ± 2 mV, corresponding to a measurement error of 2 mV * 384 °C/V = 0.77 °C, this quite large number is the price for ADC protection.
Finally we can estimate the noise figure; the ADC datasheet states a typical noise of $V_{n,ADC} = 1.5\ \mu V_{RMS}$, the thermal noise introduced by the conditioning circuit is only determined by the filtering capacitor C of fig. 5.5 and is equal to $V_{n,therm} = \sqrt{\frac{k_b * T}{C}} = 0.7\ \mu V_{RMS}$ @ 85°C; the major source of noise in this case becomes the MUX $R_{ON}$, since it has no filtering capacitor at its output, it's bandwidth mainly depends on the ADC input capacitance; the ADC input capacitance is not stated in the datasheet so we could take the declared input capacitance of digital pins (10 pF), this only gives a reasonable approximation but as we will see the noise is not a major issue in this system.; The MUX noise effective voltage would become $V_{n,MUX} = 22\ \mu V_{RMS}$ @ 85°C, with a total combined noise assuming $R_{ADC} \to \infty$ of $V_{n,tot} = \sqrt{V_{n,ADC}^2 + V_{n,therm}^2 + V_{n,MUX}^2} = 22.3\ \mu V_{RMS}$ @ 85°C; this corresponds to a noise in temperature measurement with variance $\sigma_{temp} = V_{n,tot} * 384$°C/V = 0.009 °C, or an added error (with 3 sigma confidence) of 0.027 °C, this is the combination of the maximum noise @ 85°C and the maximum sensitivity @ -40°C, so the real number would be smaller than that. The total measurement error if we add all the contributions of tolerances, ADC errors and noise is $T_{err,tot} = \pm 1.80$ °C.
From this result we can conclude that the temperature measurement chain should be inside the specified error requirement of ± 2°C, this is mainly dictated by the components tolerances, the presence of an ADC protection circuit and the wide measurement range, in fact the higher error is produced near the extremes of the measurement range and is significantly less if we reduce the latter. The error figure was also derived by making a certain amount of assumptions on components nominal values and tolerances and surely can be refined with a better characterization of the sampling chain and by choosing more precise components.
To estimate the power consumption for each ADC block (16 thermistors) we can sum the maximum ADC supply current of 75 $\mu$A, the maximum MUX supply current of 5 $\mu$A and the worst case current through the conditioning circuit (with thermistor resistance at 0Ω) of 10.6mA for 16 thermistors, this results in a power consumption of 35mW.

### 5.2.2 IMU block

As previously stated in section 4.2, the MTi-3 Inertial Measurement Unit was chosen by discussing the possible solutions with the ACS team and finally given as a Singer payload requirement, analysis of three axis inertial data is a complex task which is out of the scope of this thesis, for information about the metrological characteristics of the unit the datasheet can be referred to ([64]). This is a very small System On Board (fig. 5.13) with outline compatible with the 28-pins PLCC package.




**(a)** **(b)**

**Figure 5.13:** 5.13a MTi-3 IMU compared to human finger. 5.13b MTi-3 pinout.

Various interfaces are available to communicate with the IMU (I2C, SPI or UART), of these options the UART interface was chosen because it allows interfacing easily to the IMU from the Xsens software suite running on a PC during testing, as explained in section 7.3.5; the communication interface can be selected by pins 14 and 15 (refer to fig. 5.13b). The power consumption declared in the datasheet is 100mW, this figure is in reality stated to be at 3V analog supply and 1.8V I/O supply so it's unclear what the real consumption will be in our case (3.3V for both supplies); since no more information is provided this number will be used for power estimation.

#### 5.2.2.1 Gyroscope specifications

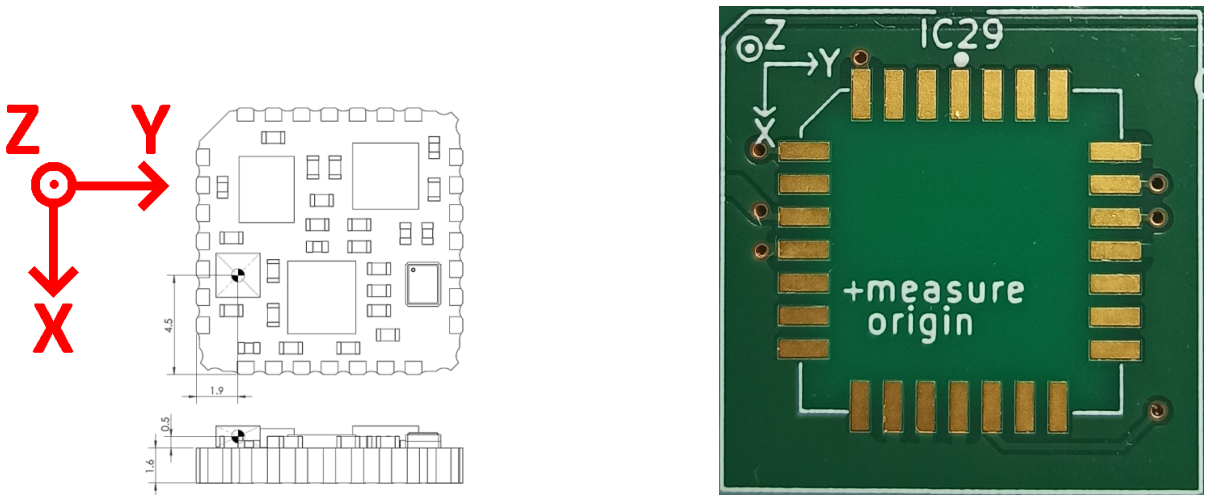
The gyroscope outputs the absolute angular velocity on each axis (°/s or rad/s) and has a full scale of ± 2000°/s.
The IMU datasheet states an in-run bias stability for the gyroscope of 10°/h ($2.8\text{x}10^{-3}$°/s) and a worst case scale factor variation of 1.5%, these numbers represent an absolute and relative error terms, respectively; the bias instability could in theory be compensated by calibration but the datasheet doesn't state anything on if this is performed by the IMU DSP. The non-linearity error is also expressed as 0.1% and so is another cause of relative error.
Regarding stochastic causes of error, the datasheet declares a noise amplitude density of $0.007°/s/\sqrt{Hz}$, from which we can compute the expected RMS noise at the maximum sensor bandwidth of 255 Hz, being $n_{RMS,gyro} = 0.11°/s$.
The gyroscope g-sensitivity is stated as calibrated (since the IMU is also equipped with an accelerometer) and for this reason should be already compensated by the DSP.
Finally, the datasheet states the non orthogonality error of the gyroscope axis of 0.05° (0.09%) and an alignment error with respect to the module board of 0.25°, computed by summing the specified alignment error of the accelerometer plane with respect to the board (0.2°), which is the only one specified, with the alignment error of the gyroscope with respect to the accelerometer (0.05°); these are other causes of relative error which require an analysis of the actual vector values measured on each axis to be evaluated.

#### 5.2.2.2 Magnetometer specifications

The magnetometer output is normalized with respect to the factory calibration values and so has no unit of measurement (arbitrary unit or a.u.), so it acts as a 3-dimensional compass; its full scale is stated to be 8 G.
The datasheet states a non-linearity error of 0.2% (relative error) and a resolution of 0.25 mG; the resolution introduces an absolute error which is difficult to evaluate without knowing the magnetic field intensity the IMU is calibrated on, by assuming a calibration magnetic field of 0.40 G (as stated in the datasheet of another line of IMUs from Xsens: [65]) we end up with a resolution of $6.25\text{x}10^{-4}$ a.u. .
The RMS noise is also stated as 0.5 mG, by applying the same logic of the resolution we came up with an RMS noise of $1.25\text{x}10^{-3}$ a.u. .
Finally, the non orthogonality error of the magnetometer axis is stated to be 0.05° (0.09%), like the gyroscope, and the alignment error with respect to the module board can be evaluated from the datasheet to be 0.3° by summing the contribution of misalignment of magnetometer plane with respect to accelerometer plane and the misalignment of the latter with the board.

#### 5.2.2.3 IMU reference plane

The IMU has the measurement origin and reference axis shown in fig. 5.14a, the mechanical structure team provided precise information about the Singer positioning inside the spacecraft and from these the IMU origin (fig. 5.14b) was placed to be aligned with the spacecraft X-Y plane origin.




(a) (b)

**Figure 5.14:** 5.14a IMU plane origin and axis, modified picture from [64]. 5.14b IMU footprint on PCB.

The IMU reference plane is rotated with respect to the Singer (spacecraft) reference plane

(that can be seen in fig. 3.3) as follows:

$$\begin{bmatrix} \hat{x}_{Singer} \\ \hat{y}_{Singer} \\ \hat{z}_{Singer} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} * \begin{bmatrix} \hat{x}_{IMU} \\ \hat{y}_{IMU} \\ \hat{z}_{IMU} \end{bmatrix} \Rightarrow \begin{cases} \hat{x}_{Singer} = \hat{x}_{IMU} \\ \hat{y}_{Singer} = -\hat{y}_{IMU} \\ \hat{z}_{Singer} = -\hat{z}_{IMU} \end{cases} \tag{5.7}$$

This corresponds to a 180° rotation around the X-axis.

### 5.2.3 Memory block

The choice for non volatile storage of measured samples was oriented towards Magnetoresistive Random Access Memory (MRAM) technology, this technology is particularly adapt to space applications because it's practically immune to SEE, as explained in NASA's MRAM technology status report ([48]), it must be said that this applies to the memory bank itself and that the surrounding CMOS circuitry is still susceptible to these events.

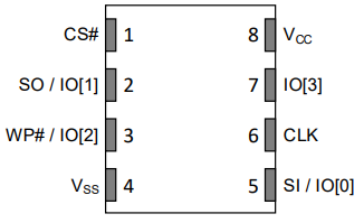



**Figure 5.15:** AS3016204 MRAM pinout, picture from [42].

While the first integrated MRAMs exploited the same Magnetoresistance mechanism that is commonly used in hard disks, today MRAMs resort to the more sophisticated Tunnelling Magnetoresistance, by which the tunnel current through an insulator separating tho ferromagnetic elements has different intensities depending on the polarization of these elements, it's then possible to program this basic cell by changing the polarization of one side with an external magnetic field and reading the cell basically becomes a current measurement.
This type of memory also has practically infinite endurance, simplifying the usage of the memory bank than doesn't require block-level erasing and wear-leveling techniques like the common NAND Flash. The main disadvantage of this technology is the reduced density and high cost per bit, with commercial components available with densities of some MB, in any case more than enough for our specific application.
The chosen device is the AS3016204 from Avalanche Technology (datasheet: [42]); this 40nm Tunnel Magnetoresistive MRAM is provided with an high pitch SOIC-8 package, SPI (single, dual or quadruple) interface with frequency up to 108MHz and 2MB of storage. The device pinout can be seen in fig. 5.15. The power supply voltage is 3.3V and the maximum current consumption declared (during write operation in single data rate, single SPI mode) @108MHz is 28mA, in our case we will use the device with a much lower frequency (4MHz). We can extrapolate an estimating formula for power consumption with respect to frequency, since the datasheet states current consumptions @54MHz and @108MHz. The dynamic current consumption (due to gates switching) is linearly proportional to switching frequency, and so even the current $I_{dyn} \propto f_{sw}$, we can estimate current consumption given a switching frequency with the formula:

$$I_{dd}(f_{sw}) = (I_{54MHz} - \frac{I_{108MHz} - I_{54MHz}}{108MHz - 54MHz} * 54MHz) + \frac{I_{108MHz} - I_{54MHz}}{108MHz - 54MHz} * f_{sw}$$
$$\Rightarrow I_{dd}(f_{sw}) = (2 * I_{54MHz} - I_{108MHz}) + \frac{I_{108MHz} - I_{54MHz}}{54MHz} * f_{sw} \quad (5.8)$$

Since the memory will be in write/read mode only for a brief time, a better estimation of the power consumption must consider the relative time (duty cycle, $\tau_{DC}$) of each mode, tab. 5.1 summarizes the values declared on the datasheet and applies equation 5.8 to estimate the current consumption on each mode @4MHz, tab. 5.2 summarizes the estimated average memory power consumption $P_{av}$.

**Table 5.1:** MRAM current estimation.

| Mode | $I_{108MHz}$ | $I_{54MHz}$ | $I_{dd}(f_{sw})$ [mA] | $I_{4MHz}$ |
|---|---|---|---|---|
| Read | 15mA | 9mA | $3mA + 0.111 * f_{sw}[MHz]$ | 3.44 mA |
| Write | 28mA | 16mA | $4mA + 0.222 * f_{sw}[MHz]$ | 4.89 mA |
| Standby | - | - | - | 400 $\mu$A $^{note1}$ |

$^{note1}$ Standby current is written on $I_{4MHz}$ field but obviously doesn't depend on frequency, this is the value from datasheet

**Table 5.2:** MRAM average power estimation.

| Mode | $I_{4MHz}$ | $\tau_{DC}$ | $I_{av}$ | $P_{av}$ |
|---|---|---|---|---|
| Read | 3.44 mA | $8.53x10^{-4}$ $^{note1}$ | 2.9 $\mu$A | 10 $\mu$W |
| Write | 4.89 mA | $4.26x10^{-6}$ $^{note2}$ | 21 nA | 69 nW |
| Standby | 400 $\mu$A | $\sim 1$ | 400 $\mu$A | 1.32 mW |
| Total | | | 403 $\mu$A | 1.33 mW |

$^{note1}$ Read will happen twice every 300ms in the worst case to read a maximum of 128 bytes (256 $\mu$s @ 4MHz)
$^{note2}$ Write will happen once a minute to write a maximum of 128 bytes (256 $\mu$s @ 4MHz)

From tab. 5.2 is clear that, as could be foreseen, the extremely rare memory accesses result in a very low power consumption of around 1.33mW, and even using higher frequencies the number would be nearly the same due to the low duty cycle of read and write modes.

### 5.2.4 RS422 block

For the RS422 interfaces, two LTC2852 transceivers (datasheet: [5]) were chosen, of which the internal architecture and pinout can be seen in fig 5.16; those have a maximum supply current of 1mA in transmission mode without load on the line, and provide a receiver high impedance of at least 96kΩ even when the device is unpowered, ideal for our application in which the device voltage domain can be shut off (see section 5.2.6).

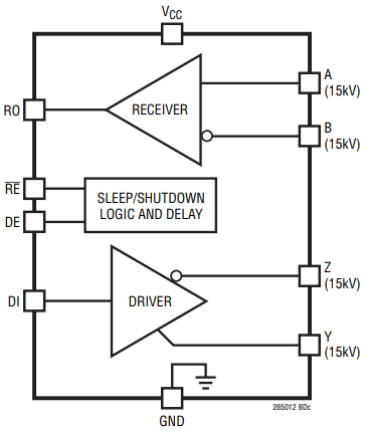



**Figure 5.16:** LTC2852 internal architecture, picture from [5].

RS422 is a physical layer standard, meaning that it only defines the electrical characteristics of the interconnection, in our system it will be the medium for UART interfaces; it implements balanced (differential) signals and so is immune to common mode noise injected on the line. It any case it was given as requirement since the CDH boards already came with this interface option due to its high immunity to noise, ideal for space applications. In fig. 5.17 an example of RS422 waveforms is given, in this example DI is the transmitter input and A/B is the differential output, the line is terminated with a 54 Ω resistance and that's why the $V_A - V_B$ difference is only around 2 V, in our case the line will not be loaded and so the difference will be near 3.3 V.

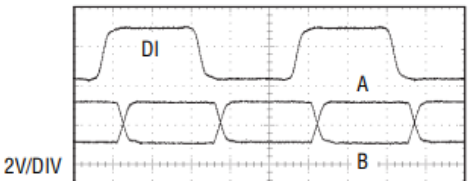


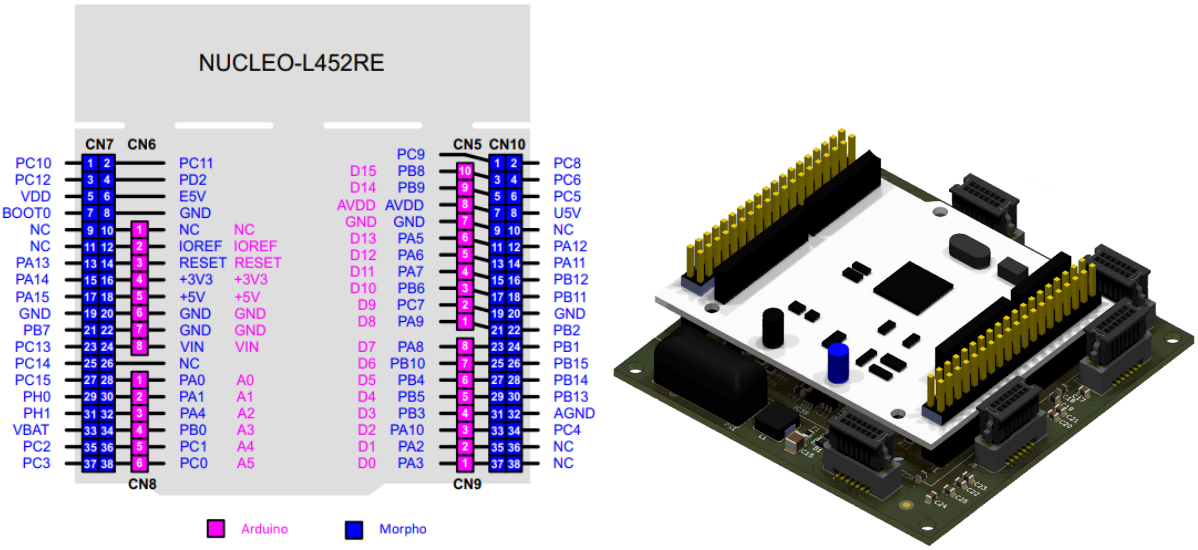

**Figure 5.17:** Example of RS422 waveforms, picture from [5].

The datasheet declares a maximum supported line length of around 1.2km at 115200 bps, this number is declared with a cable compliant with the RS422 standard, in our case the maximum interconnection distance is around 30cm; for long distances a termination resistor is needed to avoid reflection on the transmission line, but for our short signal path we don't need it and can avoid this additional source of power consumption.
Since the declared static current in receive mode is similar to the one in transmission (0.9mA) we can consider the latter as a good estimation of the device supply current, corresponding to a power consumption of 3.3 mW.

### 5.2.5 Processing unit

The processing unit is the logical brain of the system, it's responsible of gathering data coming from different sources, organize this data for storage and retrieval and interface with the other spacecraft subsystems. Among its duties there is also the timekeeping task and in general the management of the entire secondary mission. The choice for this important job was an STM32 microcontroller, specifically the L452RE model (datasheet: [113]); this 32-bit ARM microcontroller is part of the low power line (L) of ST, it can work with a clock frequency of up to 80MHz, has 512KB of code memory (flash) and 160KB of SRAM. This microcontroller was also chosen because the code memory (and part of

the RAM) of the L line embeds an error correction algorithm that can correct single error bit-flips and detect (but not correct) double bit-flips, this happens while reading each memory region and for this reason the software includes a routine reset of the system to increase the number of reads of each memory region in order to correct eventual SEUs on the flash.
This microcontroller also has the needed number of four UART peripherals (a number that is not quite common among STM32 microcontrollers) and three SPI interfaces (of which two were needed). As a last point there's the fact that ST microcontrollers are the usual choice for electronics courses at Turin Polytechnic and for this reason the author was familiar with them, saving precious design time.
This microcontroller comes with an integrated 12 bit ADC peripheral, so an option could have been to use this peripheral instead of the two separate ADCs for temperature measurements; this was avoided since the internal ADC can reach a total unadjusted error of 5.4 LSB in our configuration, corresponding to a measurement error of 4.35mV, meaning that at the maximum sensitivity of the temperature measurement chain of 384 °C/V this would have introduced an additional absolute error of 1.67°C, too high to fulfill the specified error requirement by also counting the error from component tolerances of the ADC chain.
As previously mentioned, the design philosophy was towards the adoption of already integrated units in order to reduce the design effort and the replacement difficulty; for this reason the choice has been to adopt the ST Nucleo-64 dev-board (user manual: [96]) carrying the microcontroller, mounted on top of the Singer PCB as daughter-board, as in fig. 5.18b; differently from the image, the board was prepared to flight by removing the unnecessary components (like the buttons), as explained in section 7.4.2.




**(a)** **(b)**

**Figure 5.18:** 5.18a NUCLEO-L452RE pinout, picture from [96]. 5.18b Nucleo board mounted as daughter board by the Morpho connectors.

The nucleo was connected to the PCB by the "Morpho" connectors (fig. 5.18a).

It's difficult to evaluate the power consumption of a microcontroller this complex, because it depends on a large number of factors, like for example the clock frequency and the number of used peripherals; for this task the STM32CubeMX software suite (website: [98], used to generate the microcontroller configuration) comes to help by providing an estimation of the board power consumption based on the applied configuration. The power estimation given by the tool is 8.6mA (28.4 mW).

#### 5.2.5.1 Clock and peripherals assignment

STM32 microcontrollers have a number of internal oscillators and the possibility to connect external ones to generate the various clocks for the core and peripherals. Four oscillators are available inside the microcontroller (see [113], pag. 135): a 32 kHz Low Speed Internal (LSI) clock, two High Speed Internal (HSI) clocks of 48 MHz (HSI48) and 16 MHz (HSI16) and a Multi Speed Internal (MSI) clock with selectable frequency from 100 kHz to 48 MHz. The Nucleo L452RE board (schematic: [68]) instead comes with an already soldered Low Speed External (LSE) of 32.768 kHz. Tab. 5.3 summarizes the possible clock sources and their characteristics.

**Table 5.3:** Nucleo L452RE clock sources characteristics, data from [113] (pag. 135) and [68], maximum error includes the contribution of tolerance on the frequency value and drift due to $V_{DD}$ and temperature (and aging for the LSE).

| Clock | Type | Frequency | Maximum error |
|---|---|---|---|
| LSI | RC | 32 kHz | 6.25 % |
| HSI16 | RC | 16 MHz | 2.85 % |
| HSI48 | RC | 48 MHz | 4.6 % |
| MSI | RC | 100 kHz to 48 MHz | 14.31 % |
| LSE | crystal | 32.768 kHz | 25.3 ppm |

From the table is clear that the MSI has bad characteristics and the best choice for time keeping is the crystal LSE, this oscillator can only be used for the RTC and some interfaces (UART, I2C). Nevertheless at the end the decision was to avoid using the external oscillator because it introduces an additional point of failure to the system, in favor of the LSI, also because the RTC gets continuously updated from Parrot and the sampling period is not computed by the RTC, as explained below.
Tab. 5.4 summarizes the used microcontroller peripherals and functional blocks, their associated function and clock source/frequency, the clock frequency is the final one and can be different from the oscillator frequency because it passes through a tree of Phase Locked Loops (PLLs) and prescalers that increase or decrease its value.

**Table 5.4:** Used microcontroller blocks, associated function and clock.

| Peripheral | Function | Clock source | Final clock frequecy |
|---|---|---|---|
| SPI1 | MRAM | HSI16 | 32 MHz |
| SPI2 | ADC | HSI16 | 32 MHz |
| UART4 | RS422 to CDH 1 | HSI16 | 32 MHz |
| USART1 | Access Port | HSI16 | 32 MHz |
| USART2 | IMU | HSI16 | 32 MHz |
| USART3 | RS422 to CDH 2 | HSI16 | 32 MHz |
| RTC | Keep timestamp | LSI | 32 kHz |
| GPIO | MRAM CS/ADC CS and MUX | HSI16 | 64 MHz |
| IWDG | Watchdog timer | LSI | 32 kHz |
| Core (CPU) | Processing | HSI16 | 64 MHz |

From the table we can compute the expected time precision of the system: the timestamp (from RTC) gets the tolerance from the LSI oscillator and thus has a precision of ± 6.25 % (the requirement was 10 %), while the sampling time (computed from the system tick and not the RTC since the timestamp is considered unreliable) gets the precision from the HSI16 and thus has a precision of ± 2.85 % (the requirement was 3%).

### 5.2.6 Power block

The system power delivery was implemented subdividing the architecture into different power domains, as can be seen in fig. 5.19. Each domain is powered at 3.3V, generated by a voltage regulator from the spacecraft power bus or from the access port power bus.

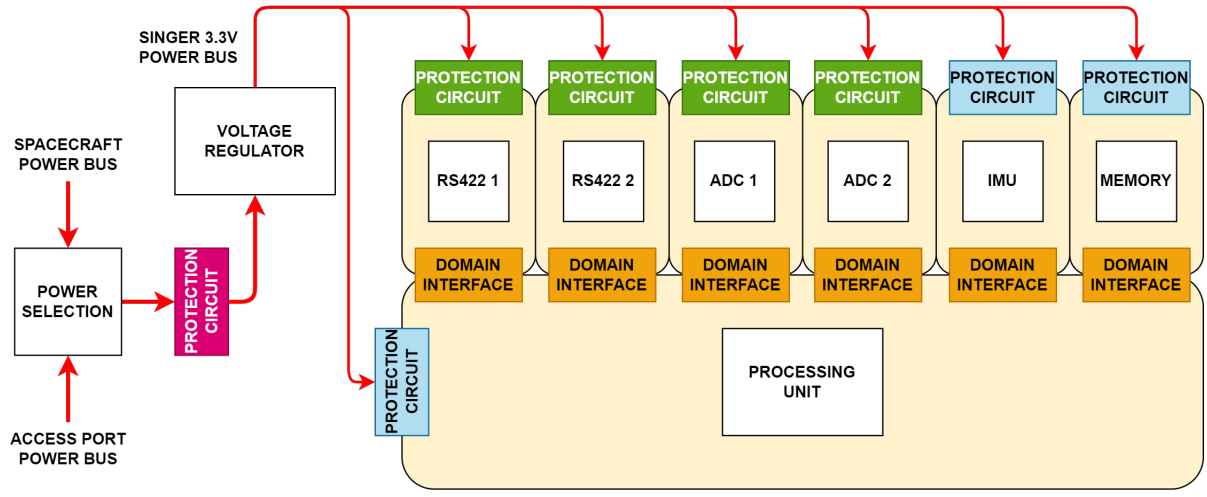



**Figure 5.19:** System power delivery.

Each component resides into a different individually protected domain, logical interfacing blocks are placed between each domain and the processing unit. This way in case of failure of one component inside a power domain, the protection circuit and the domain interface will insulate the failure from the power bus and the processing unit, the whole domain is lost but the system can continue to work (with limited functionality). The protection circuit will in any case try to reactivate the domain after a certain amount

of time, depending on the type of protection circuit, to restore operation in case of a temporary latch-up event. It's clear that if the voltage regulator or the processing unit is the one to fail permanently, the system cannot continue to operate.

### 5.2.6.1 Voltage regulator

Designing a switching voltage regulator takes time and effort and must take into consideration a substantial amount of factors; for this reason the choice was to select an already integrated System In Package regulator. The TMR3-2410WIR DC/DC from Traco Power (component web page: [111]) was chosen for this purpose; it has an input voltage range of 9V to 36V, a maximum output current of 700mA and a typical efficiency of 76%; the efficiency is quite low, this is due to the fact that the regulator is an insulated type, specifically chosen because the insulation avoids propagating the unregulated spacecraft voltage to the board in case the regulator stops working, with the risk of producing fumes inside the spacecraft or propagate the high unregulated voltage to Parrot through the RS422 transceivers. This DC/DC was also chosen because it has railway certification, meaning that it was designed to be particularly robust and withstand mechanical shocks and vibration, temperature excursions and electrical surges. The manufacturer also suggests EMI filtering networks to reduce the noise on the power bus and PCB layout guidelines; the final power regulation circuit can be seen in fig. 5.20, we can also see the power selection block implemented with diodes and the voltage regulator protection circuit, which is a self-resetting PTC fuse with nominal tripping current in the 30mA to 90mA range that will isolate a global fault on the Singer board from the spacecraft power bus.

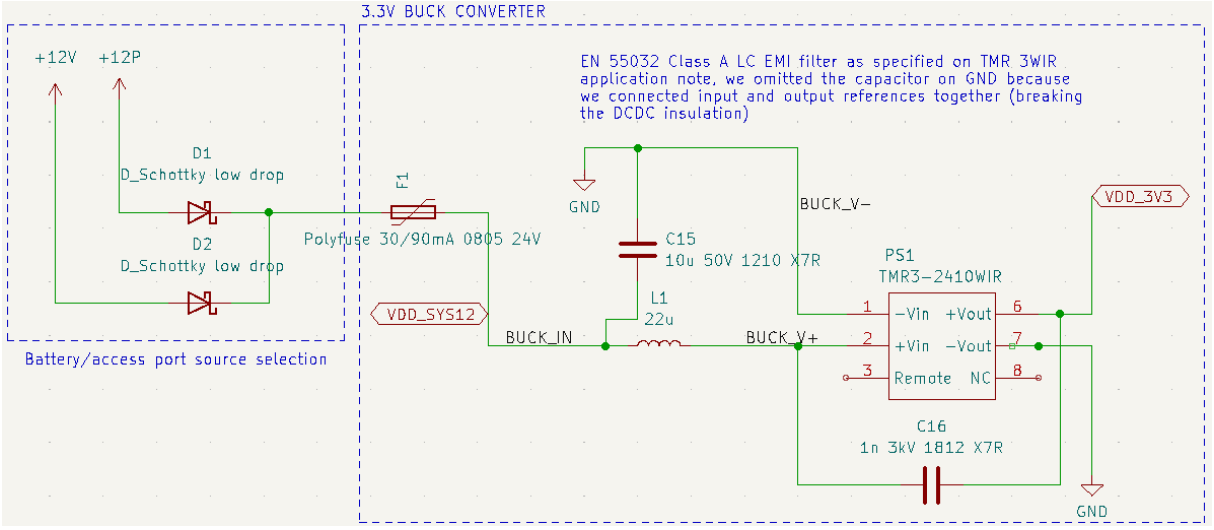



**Figure 5.20:** Power regulation circuit.

### 5.2.6.2 Domain protection circuits

There are mainly two types of domain protection circuits, depending on the expected device peak current consumption. The simplest type of protection circuit is applied to devices with low peak current (ADC and RS422 transceiver, provided that no termination resistance is present on the TX line) and can be seen in fig. 5.21.

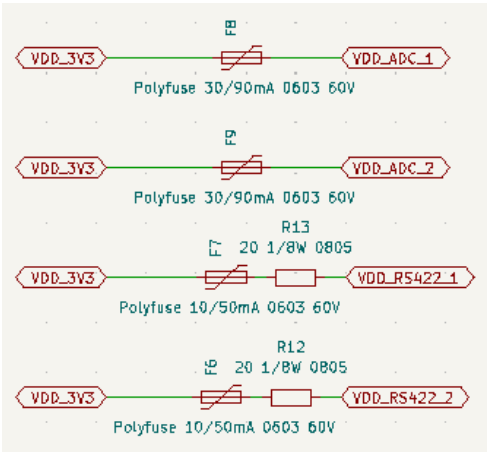



**Figure 5.21:** Simple type protection circuits.

It consists of a series of PTC fuse and/or latch-up protection resistor, the latch-up protection resistor, together with the PTC fuse intrinsic resistance will drop the supply voltage in case of current surge due to a latch-up event; this drop should stop the latch-up if the holding voltage is reached and corresponds to the full 3.3 V supply in a range of 66 mA to 550 mA for the ADC protection (15-50 Ω PTC resistance range) and 27 mA to 95 mA for the RS422 transceiver (15-100 Ω PTC resistance range + 20 Ωresistor), but the drop needed to reach the latch-up holding voltage should be less.
Even if the ADC draws very low current, an higher value PTC and no resistor was placed because they would have reduced too much the ADC supply with respect to the thermistor supply (directly coming from the regulator output), resulting in a measurement error outside specifications. In this configuration the ADC power domain can draw up to 80 $\mu$A (75 $\mu$A for the ADC and around 5 $\mu$A for the MUX) , corresponding to a worst case drop of 4 mV through this specific PTC maximum resistance of 50 Ω). The PTC will also trip in case of permanent short circuit failure of one component. It must also be noticed that since the thermistors are outside the ADC voltage domain, if the PTC has tripped the ADC and MUX can be powered anyway through the GPIO ESD diodes of the MUX (as explained in section 5.2.6.3) from the conditioning circuit, this doesn't represent a problem for the latchup protection since in the worst case the combined resistance to $V_{DD}$ (parallel of pull-up resistors) is 312.5 Ω but the domain working in this state would output completely wrong values until the PTC is restored.
Actually it's unclear if the PTC can restore while the power is still on, since this information is not given in the products datasheet, it probably should be able to restore if the current remains below the hold current (the lower one), in any case these devices are very slow to trip and should do it only in case of permanent short circuit on the domain supply line.
For devices with higher peak power consumption, like the MRAM, IMU and microcontroller, a more complex latch-up protection circuit was developed. The circuit block diagram can be seen in fig. 5.22

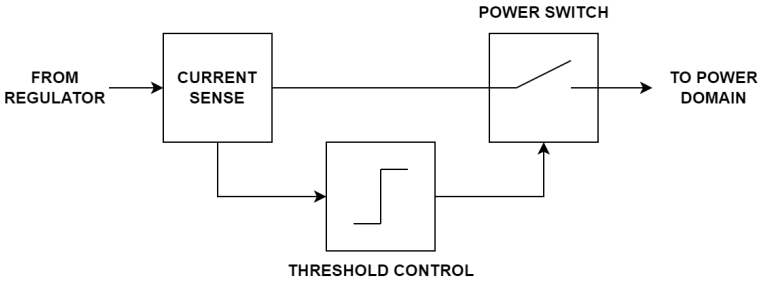



**Figure 5.22:** Complex protection circuit block diagram.

The circuit was partially inspired by solutions from [89], it uses the same INA138 high-side current measurement IC (datasheet: [45]) to measure the current going to the load, a TL431 (datasheet: [109]) precision programmable reference to generate a given threshold voltage and a TL331 comparator (datasheet: [108]) that compares the INA138 output with the reference generated by the TL431, whenever the threshold is reached (current surge on the load) the comparator will interrupt the supply of the power domain through an high-side switch; the switch is designed to provide a fast turn-off delay and a slow turn-on delay, this way after some time from the surge event the circuit will try to power again the domain, repeating the cycle if an high current is still measured. All these components are built with bipolar technology (or single power MOSFETs) and for this reason should not suffer from latch-up; in any case the power supply to the protection circuit, directly coming from the unregulated spacecraft bus, is protected with a PTC fuse. The overall circuit schematic can be seen in fig. 5.23, a jumper (J11) to bypass the whole circuit was provided in case of incorrect functioning of the latter but was not soldered on the board at the end, there was also a PTC fuse on the 3.3V line before the switch but its intrinsic resistance was too high and it was later removed (substituted with a 0 Ω jumper).

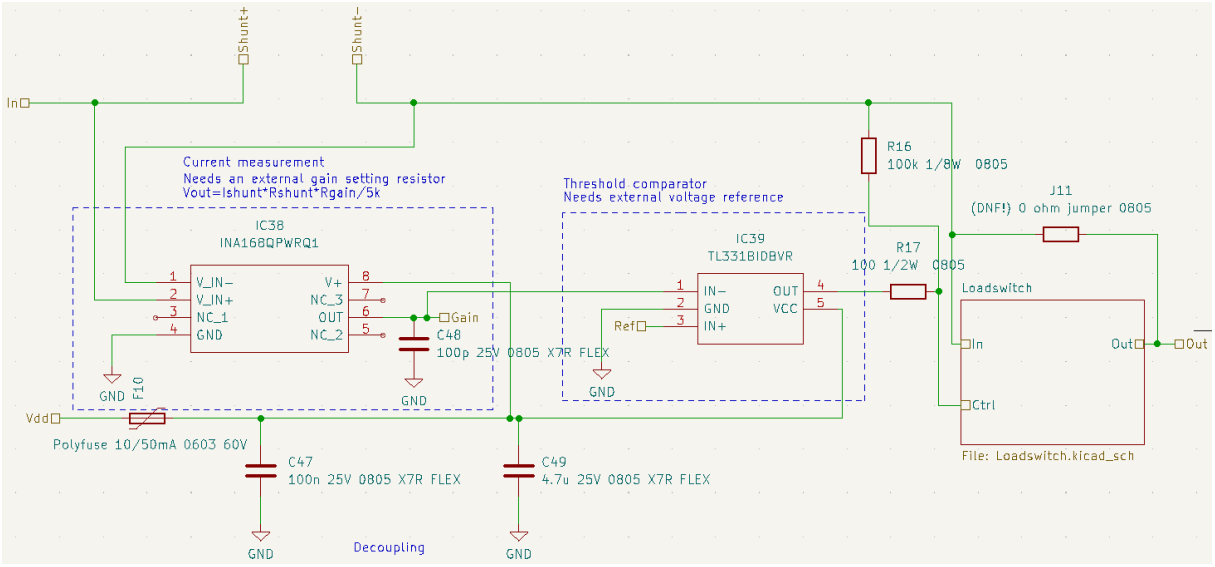
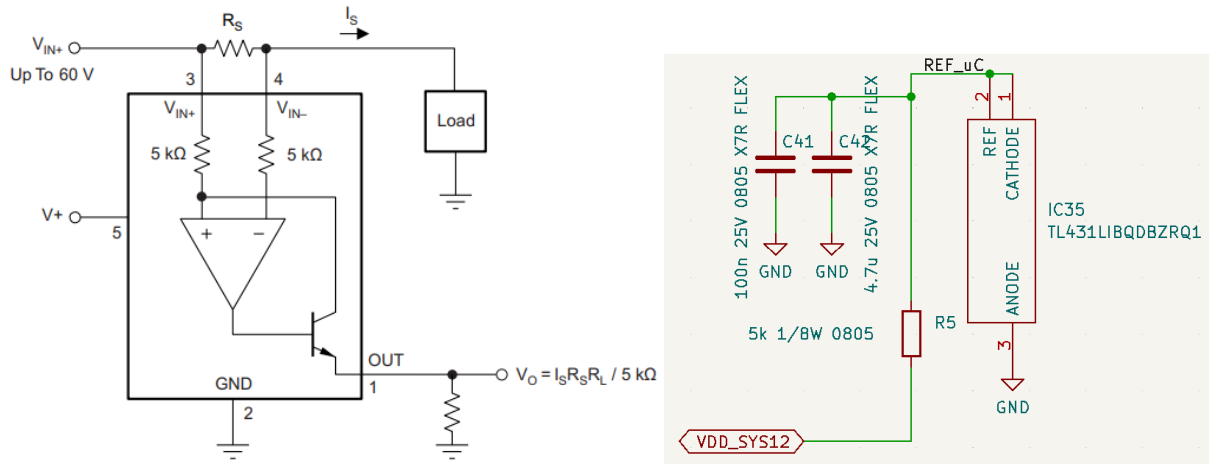



**Figure 5.23:** Latch-up protection circuit.

The INA138 measures current with a shunt resistor in series with the power line, measurement gain can be set by choosing the shunt resistor value and a second gain setting resistor at its output, as can be seen in fig. 5.24a, the output voltage follows the equa-

tion $V_{out} = \frac{I_{shunt} * R_{shunt} * R_{out}}{5k\Omega}$, for our application the shunt resistor value was chosen to be $R_{shunt} = 1\ \Omega$ while for the output resistor a value of $R_{out} = 120\ \text{k}\Omega$ was chosen, resulting in a transresistance gain of $G = 24V/A$, the gain tolerance is the combination of tolerance on external resistors (chosen to be 1%) and the INA138 maximum output error, declared to be ± 2.5%; the total error on the gain figure happens to be ± 4.5%.
The threshold voltage is generated by the TL431 circuit of fig. 5.24b, in this configuration the output reference is 2.495 V, the TL431LIB variant was chosen, having a tolerance on the voltage reference of ± 12 mV. By comparing the output of these two ICs, we get a nominal threshold current of $I_{th} = 104$ mA, the tolerance can be computed by adding the absolute error of the TL431 to the absolute error on the INA138 output at the threshold condition, resulting in a threshold voltage absolute error of ± 124 mV, or an absolute error on the threshold current of ± 5.2 mA (about 5%). From the value of the shunt resistor we can conclude that the maximum voltage drop on the power lines due to the latter is 104mV.




**(a)** **(b)**

**Figure 5.24:** 5.24a INA138 high-side current measurement circuit, picture from [45]. 5.24b TL431 2.495 V reference generation.

An additional capacitor in series with the INA138 output resistor can set the device bandwidth, in this case a 100 pF capacitor was chosen that gives a bandwidth of around 20 kHz, this capacitor will actually slightly increase the intervention time but it was placed to filter out the voltage regulator output ripple and avoid false triggering of the circuit.
A slow turn-on, fast turn-off high side switch circuit was then needed in order to interrupt a latch-up event as fast as possible and then slowly turn on again the power domain, the slow turn-on time will also reduce power consumption in case of irreversible loss of the power domain due to a device supply pin shorted to ground. The switch circuit was designed as shown in fig. 5.25.

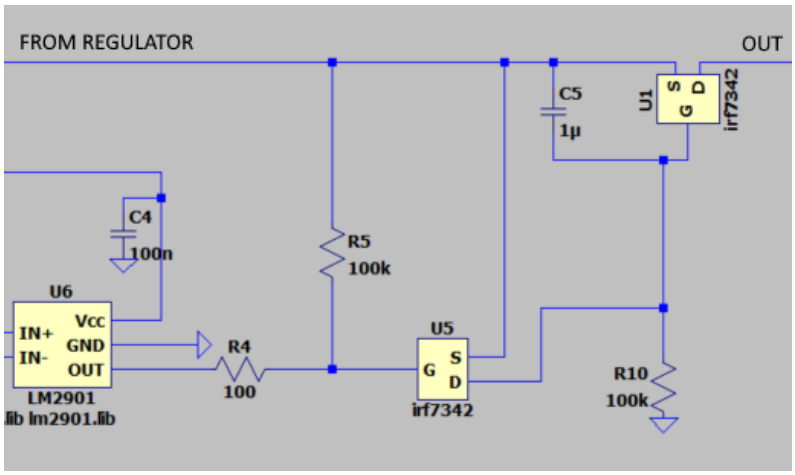



**Figure 5.25:** Slow turn-on, fast turn-off high side switch circuit.

Two p-type power MOSFETs are used, one will act as the actual switch (U1 in fig. 5.25) while the other (U5) is used to control it and provide the fast turn-off capability. The IRF7342 IC (datasheet: [41]) was chosen for the purpose, this chip provides two identical p-type MOSFETs in a single SO-8 package. The switch MOSFET has a 1 $\mu$F capacitor (C5) in parallel to its gate capacitance, this will provide the slow turn-on when the control MOSFET is turned off and C5 is charged through the R10 100 k$\Omega$ resistor. The time for the gate voltage to reach the declared MOSFET threshold of -1V can be computed as $t_{th} = -\tau * ln(\frac{2.2}{3.3}) = 36$ ms, this number is only indicative since the MOSFET will take some more time to reach the triode zone, also depending on the applied load, but it gives an idea of the rather slow gate charging time.
Whenever the measured current reaches the threshold condition, the INA138 output will go up with a time constant determined by the RC filter at its output $\tau_{INA138}$= 12 $\mu s$, the actual time to reach the threshold depends on the current surge value, if we assume a current step going from 0 to 110% of the threshold level, the threshold will be reached in 29 $\mu s$; at this point the comparator open collector output will go to GND; the control MOSFET will turn on, discharging the C5 capacitor and turning off the switch MOSFET. A LTSPICE numerical simulation was set up to predict the time behavior of the system and the switch delay, the circuit can be seen in fig. 5.26.

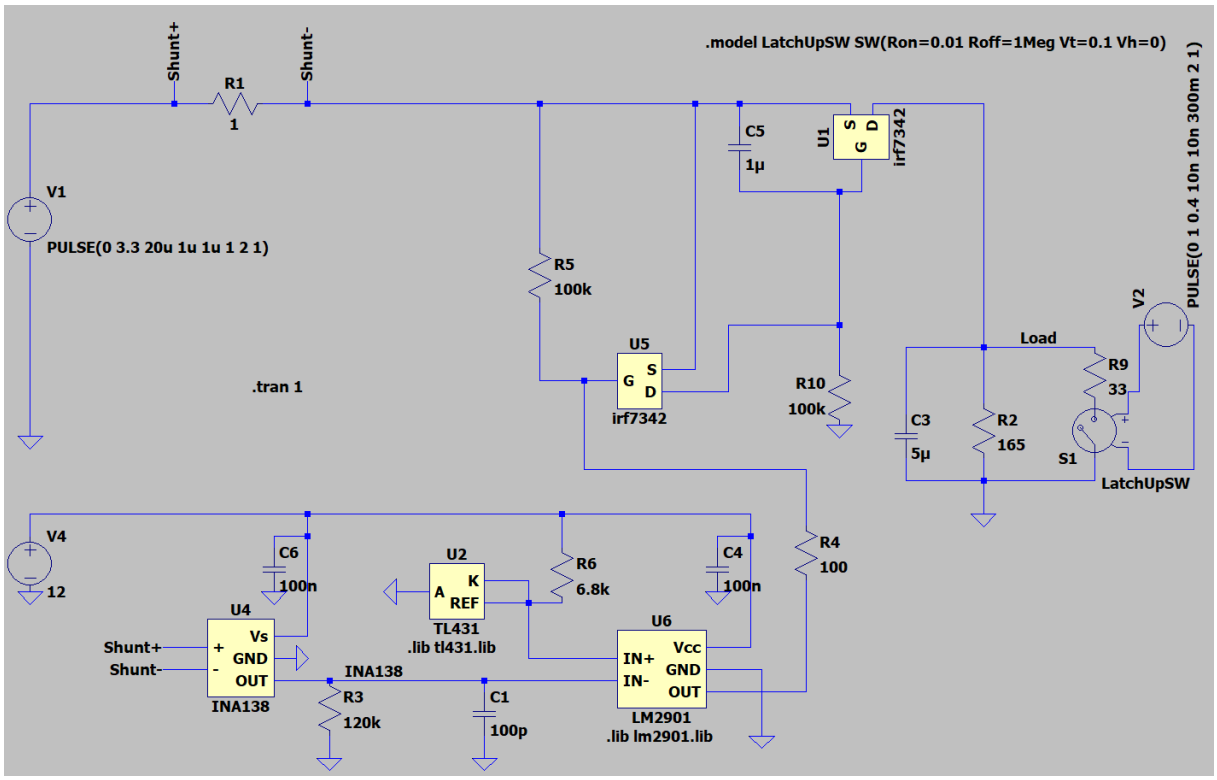



**Figure 5.26:** Latch-up protection circuit simulation model.

In this simulation, the load is modeled with a 165 Ω resistance (R2 in fig. 5.26), drawing a current of 20 mA, the load capacitance is also modeled with a 5 $\mu$F capacitor; the latch-up low resistance path is modeled with another low value resistor (R9 in fig. 5.26) with a switch in series, the switch gets closed to simulate the latch-up event; the resistance value in this case was chosen to a E12 normalized value of 33 Ω, in order to reach the previously mentioned value of around 110% of the threshold current (120 mA in this case). In fig. 5.27 we can see the simulated behavior of the load voltage when the resistor gets connected: at first the switch closes and the voltage goes down, then after more or less 120 ms the circuit tries to turn on again, since in this case the latch-up simulation resistance is still present it turns off again, repeating the cycle approximately every 150 ms.

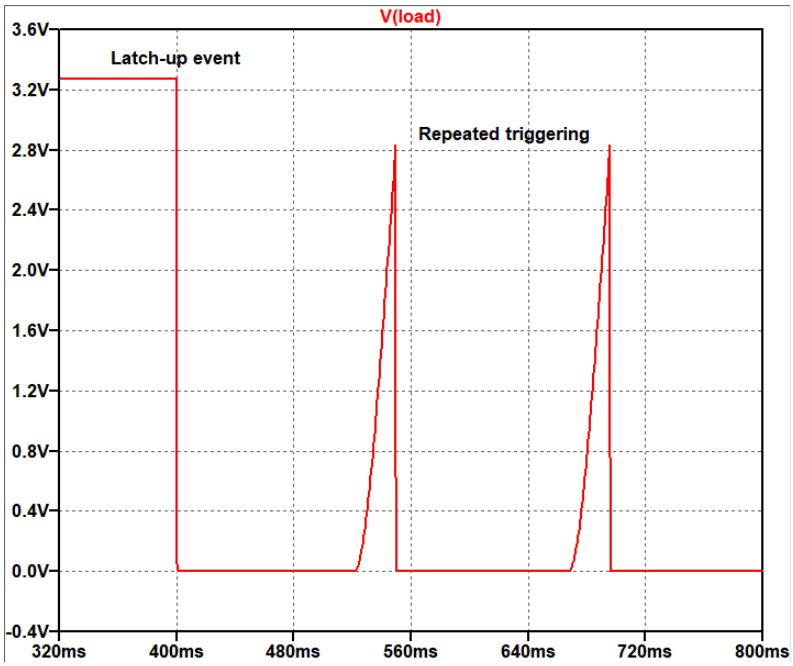



**Figure 5.27:** Voltage at load during simulated latch-up event in SPICE.

From the simulation we can also get an estimation of the circuit behavior during turn-off: from fig. 5.28 we can see that the intervention delay is approximately 55 us, meaning that this is the delay from the beginning of the event and the MOSFET switch turn-off.

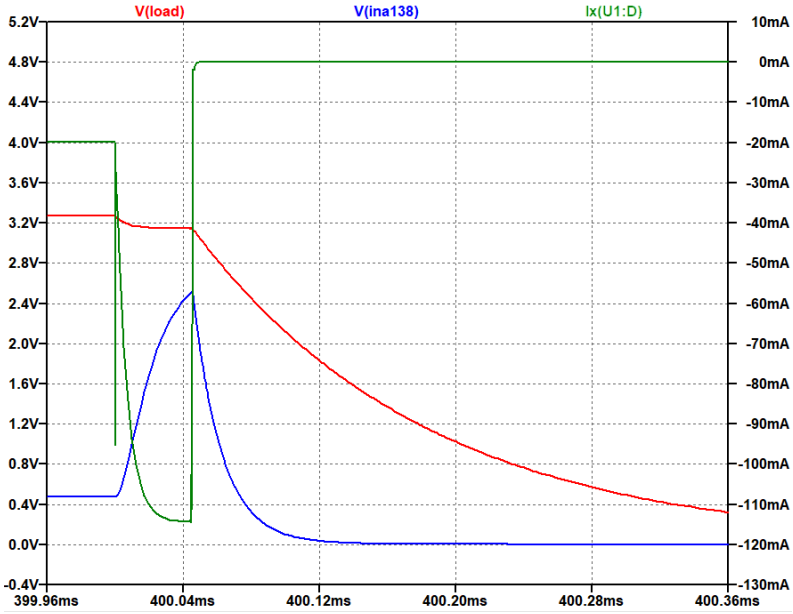



**Figure 5.28:** Simulated latch-up event, red: load voltage, blue: INA138 output, green: current through switch.

On the other end we can see that the load voltage will not be shorted to ground immediately, since the load capacitance needs to discharge through the load itself; this behavior can be problematic since the load capacitance continues to provide charges to power the latch-up for some time, it could be solved by adding a second n-type MOSFET that quickly discharges the load capacitance during turn-off but since the latch-up protection was a nice-to-have requirement this implementation was considered satisfactory enough. The INA138 has a declared supply current of 60 $\mu$A, the TL331 consumes a maximum of

0.43 mA and the TL431 reference circuit in that configuration drains 2.1 mA in the worst case (13 V $V_{dd}$); the estimated power consumption for each complex protection circuit is 34 mW; this number is mainly dictated by the reference voltage generation and is a big fraction of the overall system power budget. An estimation of the maximum power consumption in case of irreversible fault can be done from the waveforms of fig. 5.27: if we approximate the waveform with triangular shapes of base 50 ms and height 104 mA, we get an average current of 52 mA during the repeated startup ramp, if we multiply this for the startup duty cycle (50ms/125ms = 0.4) we get an average current of 21 mA, or a power of 69 mW.

#### 5.2.6.3 Domain interfaces

The domain interfaces are needed to interrupt logical interconnections between power domains in case one gets shut down by the corresponding protection circuit. This is needed because a powered down IC can be supplied anyway by its I/O pins through the ESD protection diodes, as seen in fig. 5.29: normally the ESD protection diodes are inversely polarized and only turn on in case of an ESD event, this is true if the IC is correctly supplied and the voltage levels on its I/O pins are inside the supply rails; if instead a voltage higher than the positive supply (or lower than GND) is applied, and this voltage is also high (or low) enough to overcome the diode threshold voltage, current will flow directly to the supply rails from the I/O pin. This represents a problem in systems like the one under analysis in which different power domains are present that can be individually powered off. The effect is even more problematic if the turned off domain has the positive supply shorted to ground due for example to a destructive latch-up: in this case any device that is connected to the broken IC gets its outputs shorted too, hence an interface block is needed to avoid this unwanted behavior.

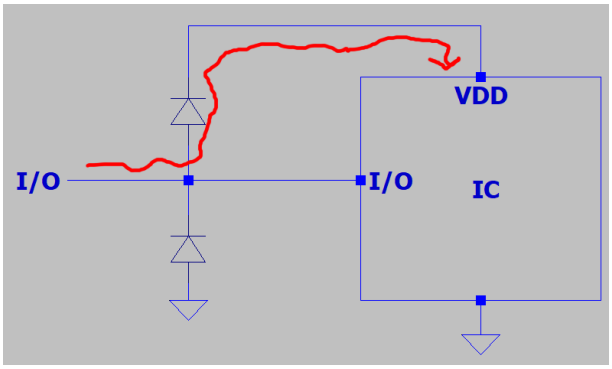



**Figure 5.29:** IC powering from I/O through ESD protection diode.

The solution was to use the SN74AXC1T45 (datasheet: [83]) and SN74AVC4T245 (datasheet: [82]) bus transceivers, these ICs are designed to interface circuits with different supply voltages and deal with this type of problem by implementing a Partial-Power-Down mode: whenever either of the two supply lines is turned off, all the IC pins go to an high impedance state, delivering a maximum current (from datasheet) $I_{off}$ of 5 $\mu A$. Two different models were chosen to optimize the circuit area usage, since one of them has a single logical line while the other has four of them, with direction configurable in groups of two. The variant with four lines also has Output Enable pins that allow using them in an SPI bus shared between multiple functional blocks, as is the case for the ADCs. Fig. 5.30 shows the usage of both ICs to interface the MRAM domain with the processing unit,

notice that in the image an error on the PCB can be spotted: the MRAM chip select pin CS# is left floating when the bus transceiver output goes to high impedance, this was later corrected by applying pull-up resistors to all the ICs that presented this problem.

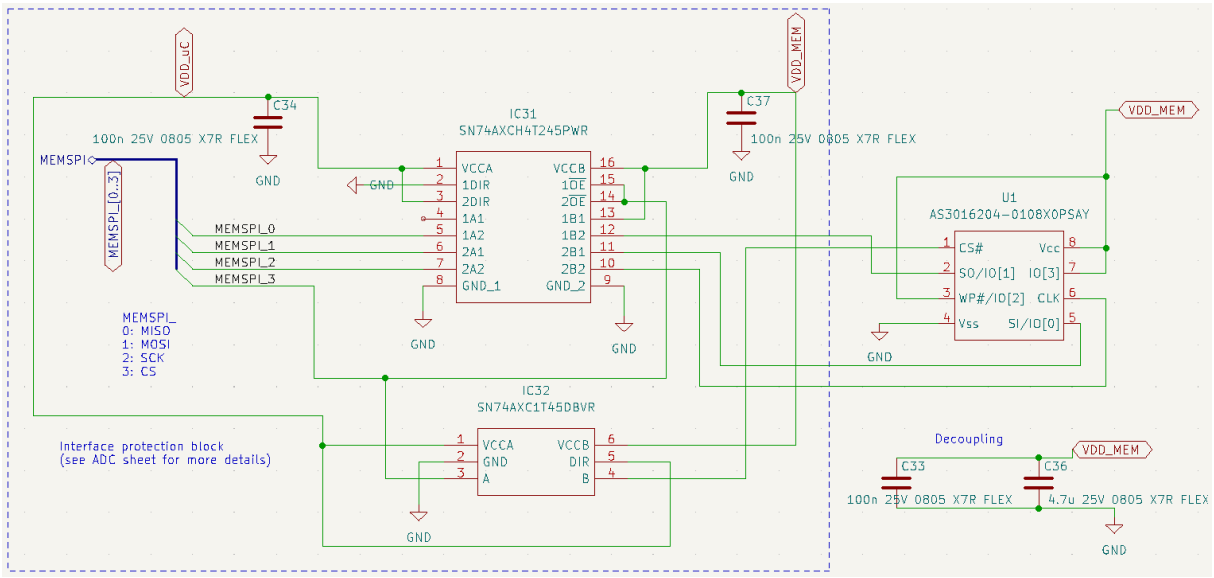



**Figure 5.30:** SN74AXC1T45 and SN74AVC4T245 used as voltage domain interface with the MRAM.

The ICs have a declared maximum supply current (both rails combined) of 16 $\mu$A (53 $\mu$W).

### 5.2.7 Board electrical interfaces and connectors

The system has two wired interfaces towards the outside: one is the communication interface through which the two RS422 lines and the spacecraft power bus are routed, the other is the Access Port interface through which the system can be programmed and externally powered.

Other connectors are present to interface elements of the system itself: five connectors for thermistors and two header connectors for the Nucleo. In fig. 5.31 the connectors names and relative functionalities are shown.

For all connectors except the Nucleo header, Samtec's TFM-107-02-S-D-WT-P was chosen (datasheet: [1]), to be coupled with the ISDF-07-D-M cable housing (datasheet: [2]). This polarized connector has 2 rows of 7 gold-plated pins with 1.27 mm pitch, it comes with the -WT (Weld Tab) ordering option, having metal tabs that couple with hooks on the cable housing (as can be seen in fig. 5.32) and on the other end have through hole soldering terminations (only for metal tabs, electric pins are surface-mount), this improves the connection robustness to the heavy stresses during launch (in any case the connections were glued during integration by the mechanical structure team).

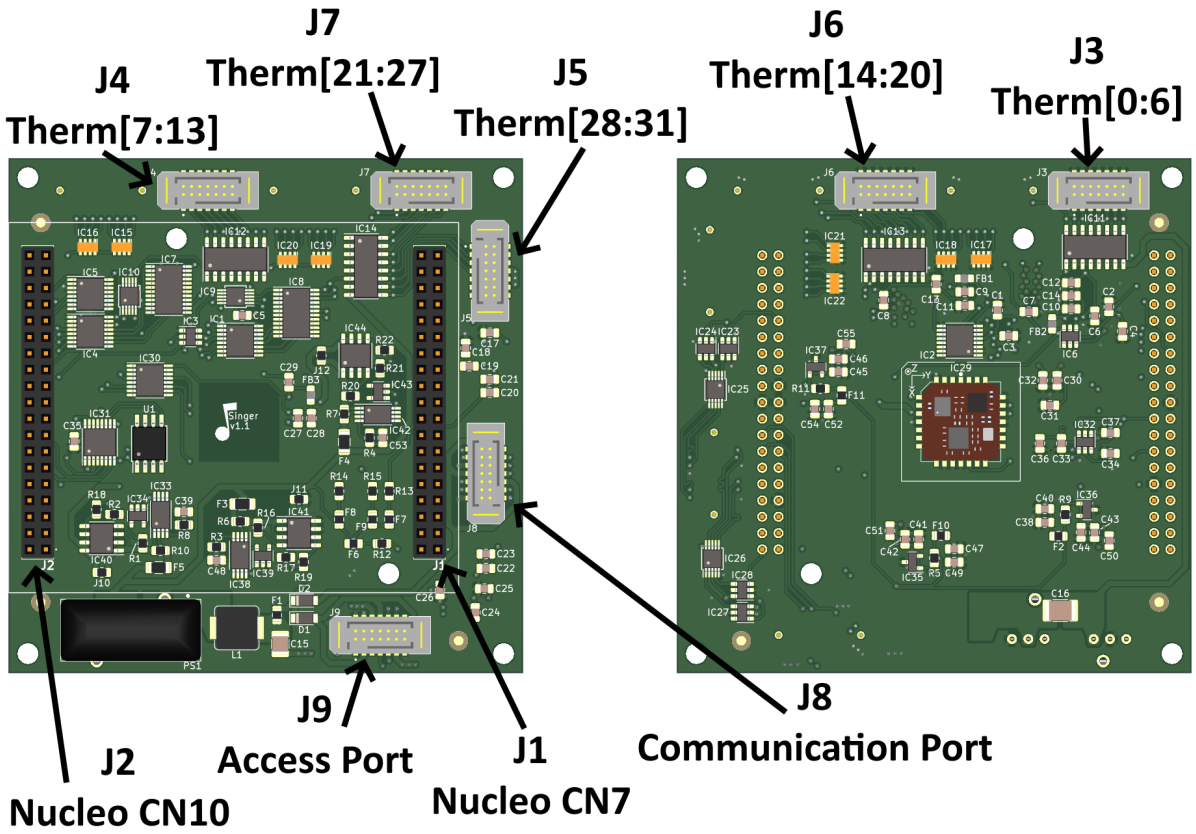



**Figure 5.31:** Singer board connectors names and function.

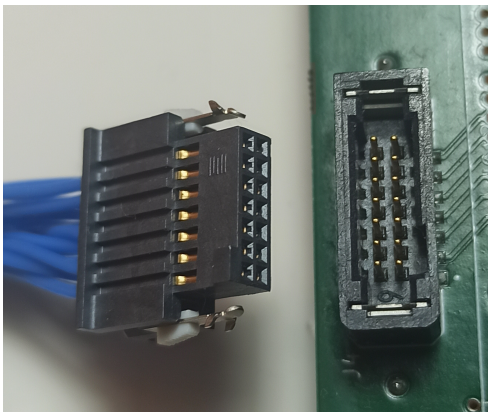


**Figure 5.32:** Samtec ISDF-07-D-M cable housing (left) and TFM-107-02-S-D-WT-P connector (right).

The Access Port connector (J9) pinout can be seen in tab. 5.5

**Table 5.5:** Singer Access Port (J9) pinout.

| | | | |
|---|---|---|---|
| 1 | GND | VDD_TARGET | 2 |
| 3 | VDD_AP | SWCLK | 4 |
| 5 | N/C | GND | 6 |
| 7 | UART_AP_TX | VDD_AP | 8 |
| 9 | UART_AP_RX | SWDIO | 10 |
| 11 | GND | nRST | 12 |
| 13 | VDD_AP | SWO | 14 |

**Legend:**
GND - Ground reference
VDD_AP - Access port positive supply line

nRST - SWD interface reset pin
VDD_TARGET - SWD interface target supply voltage
SWCLK - SWD interface clock pin
SWDIO - SWD interface data pin
SWO - SWD interface debug output interface (note: this is a placeholder and is not wired to the Nucleo)

UART_AP_TX - Access port service UART TX line
UART_AP_RX - Access port service UART RX line

N/C - Not connected pin

The communication port connector (J8) pinout can be seen in tab. 5.6

**Table 5.6:** Singer communication port (J8) pinout.

| | | | |
|---|---|---|---|
| 1 | RSS422_1_TX_P | GND | 2 |
| 3 | RSS422_1_TX_N | RSS422_2_TX_P | 4 |
| 5 | VDD_SYS | RSS422_2_TX_N | 6 |
| 7 | GND | VDD_SYS | 8 |
| 9 | RSS422_1_RX_P | GND | 10 |
| 11 | RSS422_1_RX_N | RSS422_2_RX_P | 12 |
| 13 | VDD_SYS | RSS422_2_RX_N | 14 |

**Legend:**
GND - Ground reference
VDD_SYS - Communication port positive supply line

RS422_$x$_TX_P - Positive TX polarity for RS422 port $n$
RS422_$x$_TX_N - Negative TX polarity for RS422 port $n$
RS422_$x$_RX_P - Positive RX polarity for RS422 port $n$
RS422_$x$_RX_N - Negative RX polarity for RS422 port $n$

For all the thermistors' connectors (J3, J4, J5, J6, J7) the following configuration holds:

all the even pins are connected to GND; any odd pin is wired to a different ADC channel, so that every thermistor is wired from an odd pin to the even pin in front of it; connector J5 only has 4 thermistors, all the other have 7.
The nucleo headers pinout can be seen in fig. 5.33.

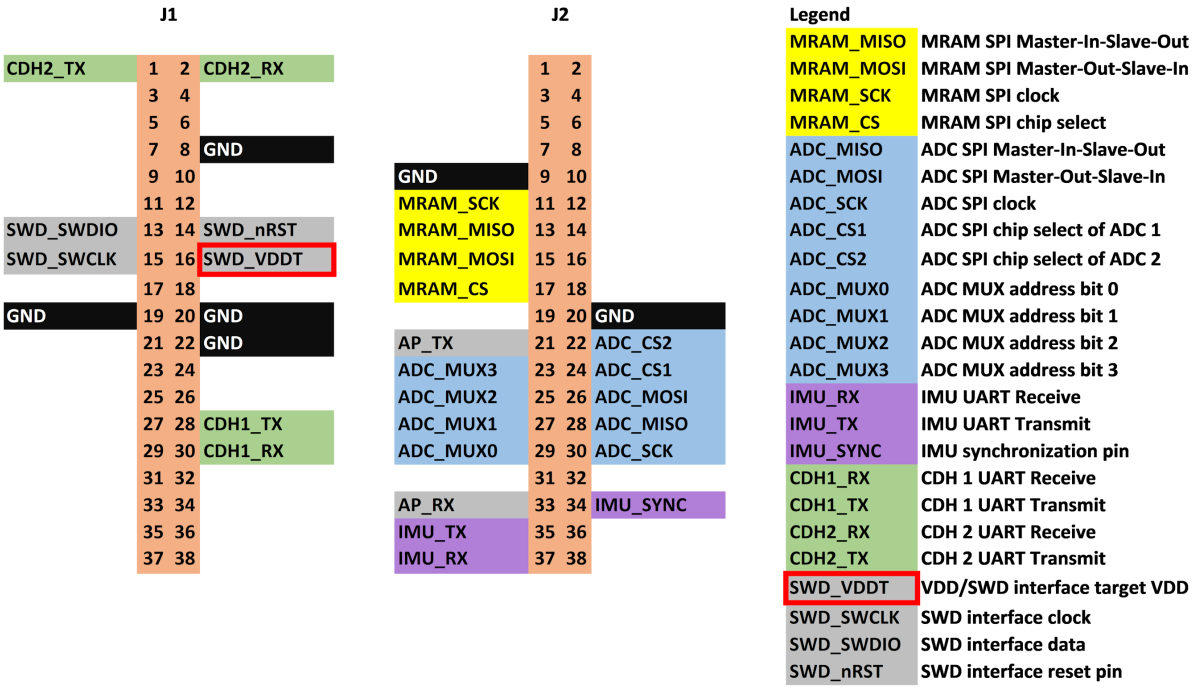



**Figure 5.33:** Nucleo header connectors pinout.

### 5.2.8 System power estimation

In tab. 5.7 a summary of the worst case power consumption for each system building block has been made.

**Table 5.7:** System power consumption summary

| Block | Power consumption | Number of blocks | Total |
|---|---|---|---|
| ADC | 35 mW | 2 | 70 mW |
| Memory | 1.33 mW | 1 | 1.33 mW |
| IMU | 100 mW | 1 | 100 mW |
| RS422 | 3.3 mW | 2 | 6.6 mW |
| Control unit | 28.4 mW | 1 | 28.4 mW |
| Domain interfaces | 53 $\mu$W | 9 | 477 $\mu$W |
| Regulator | 65 mW $^{note1}$ | 1 | 65 mW |
| Latch-up protection | 34 mW | 3 | 102 mW |
| Total | | | 374 mW |

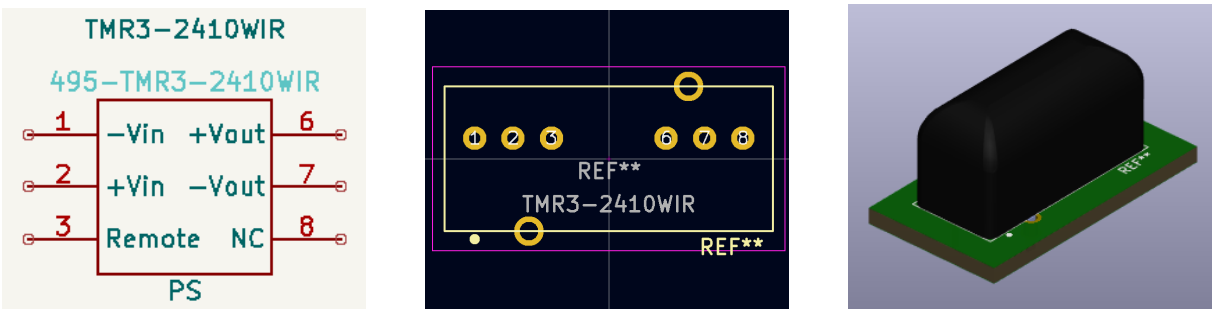
$^{note1}$ The regulator consumption was estimated from the 3.3V components consumption with an efficiency of 76%

As can be seen in tab. 5.7, the power consumption requirement of 300 mW was not reached. While some blocks' consumption cannot be further improved, three errors from

the author can be identified that could have been implemented in a more efficient way: the first is the thermistors block, where higher pullup resistors values would have reduced the worst case power consumption (for example using 10 kΩ instead of 5 kΩ would practically reduce the worst case power of this block to half its value, by increasing on the other end the equivalent resistance of the conditioning circuit and so the effect of the ADC input current), on the other end it must be recognized that this worst case power consumption of thermistors is reached if all 32 are at very high temperature (low thermistor resistance), while in normal operation the majority of them are at temperatures lower than 10°C and so the actual power is way lower than that. The second error is on the TL431 reference voltage generators of the protection circuits, these circuits consume 2.1 mA from the spacecraft bus, while the minimum current needed by the TL431 is 1 mA; the maximum resistance value that could have been applied instead of 5 kΩ is determined by the minimum bus voltage of 9.2V (the minimum 9.5V minus the drop on input diode of around 0.3V) and is 6.7 kΩ (so not much higher than the actual), another possibility would have been to place only one reference circuit shared by all protection circuit, lowering the system redundancy. The third error, that was more a conscious choice than a mistake, was to choose a voltage regulator that is not particularly efficient; a more efficient, less rugged regulator could have been selected instead.
In any case measurements done on the physical system, performed at room temperature, revealed a power consumption of the whole system of 286 mW.
Keep in mind that this figure is the nominal one and that if the system faults the maximum power that can be drained from the spacecraft supply is determined by the PTC fuse before the voltage regulator, this type of fuse has a very broad range of trip current that can go from 30mA to 90mA, corresponding to a power consumption range of 390mW to 1.17W, so the spacecraft should provide a way of shutting off the board in this case.

## 5.3 PCB design and production

Singer PCB design was carried out with KiCad EDA ([52]), an open-source yet powerful PCB CAD software that allows following the complete design of a board, from schematic drawing to layout and 3D rendering. The software comes with a default set of symbols (fig. 5.34a), footprints (fig. 5.34b) and relative 3D models (fig. 5.34c) of common packages.




(a) (b) (c)

**Figure 5.34:** 5.34a Voltage regulator symbol. 5.34b Voltage regulator footprint. 5.34c Voltage regulator 3D model.

It was often necessary to manually create some of them (especially the symbols, containing the pinout of specific devices) or all of them for specific components (like the voltage regulator shown in fig. 5.34). For 3D modeling the SOLIDWORKS 3D CAD software was used (website: [85]).

### 5.3.1 Schematic

The schematic is organized by using KiCad hierarchical sheet tool, the main sheet can be seen in fig. 5.35; it approximately follows the blocks subdivision of fig. 5.1 and the connections to the microcontroller headers of each block are defined as multi-wire buses. The complete schematic will not be treated to not overfill this document with images (and because connections are somehow trivial) but a glimpse on some of the hierarchical sheets' content has been given on section 5.2.

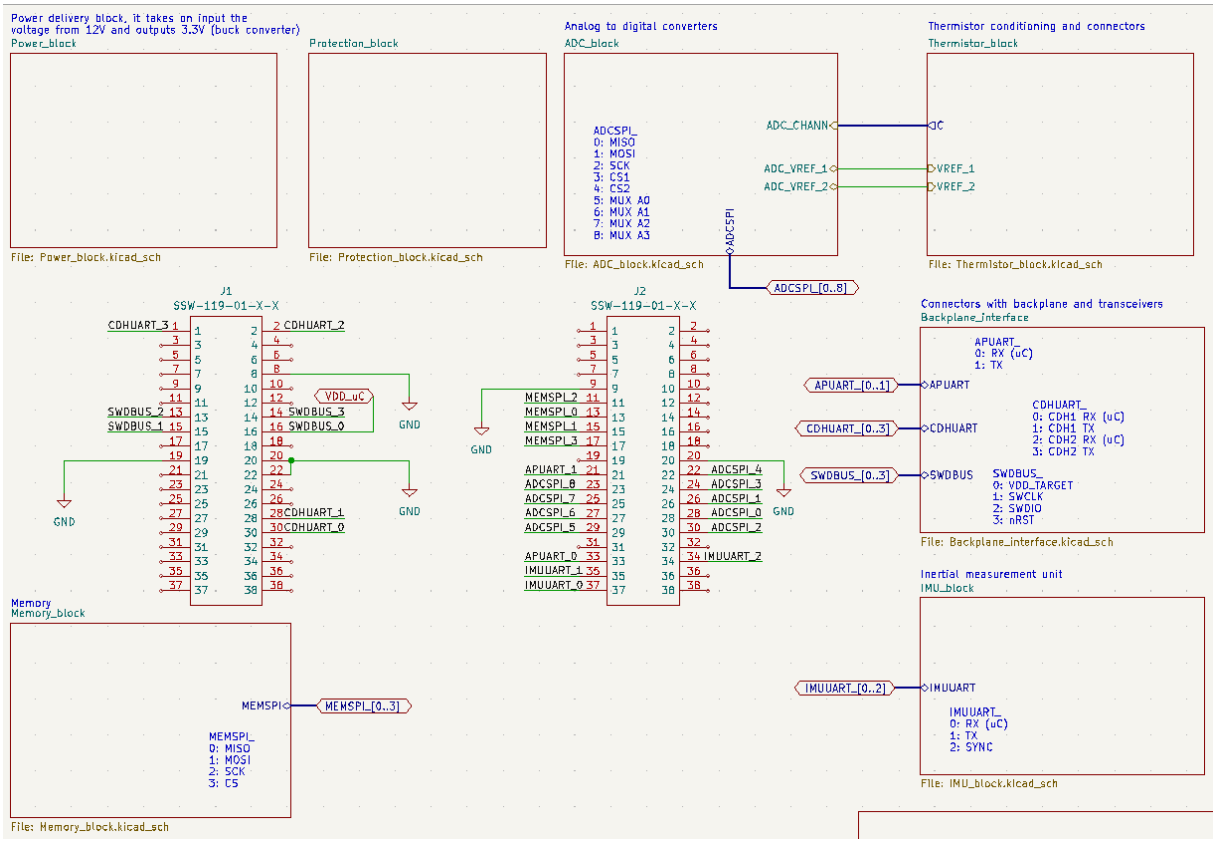



**Figure 5.35:** Main Singer schematic sheet on KiCad.

Each functional block that represents a power domain has the same basic structure (of which a good example could be seen in fig. 5.30), containing:

- The main devices implementing the block functionality, as discussed on section 5.2;
- Power supply decoupling capacitors, usually a 4.7 $\mu$F capacitor for each domain and 100 nF capacitors on each individual supply pin (unless otherwise specified on the datasheet), to be placed as close as possible to the package; for devices with analog supply (ADC and IMU) a ferrite bead has been placed in series to the power line to clean it from high

frequency noise;

- Domain interface devices for that specific block;
- A bus wire, grouping all the logical connections towards the microcontroller and connected to the hierarchical sheet pin (which can be seen exiting each sheet in fig. 5.35);

### 5.3.2 Layout

Since the circuit has a fair amount of interconnections a 4-layers PCB was adopted for the system, with a standard copper density per unit area of 1 oz/ft$^2$, corresponding to a thickness of 34 $\mu$m; the interconnections layout can be seen in fig. 5.36

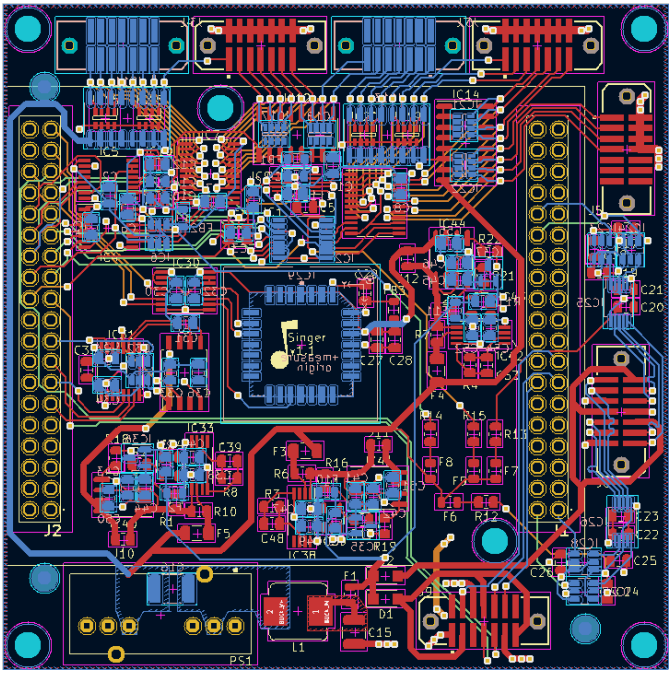
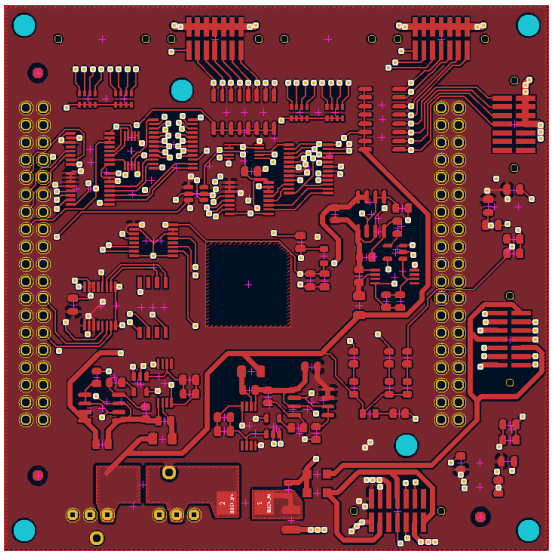


**Figure 5.36:** PCB interconnections layout.

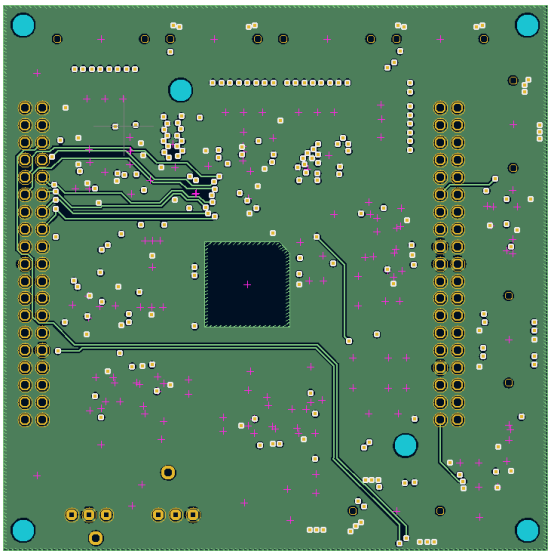
Each layer has a certain number of interconnections and the remaining area is filled with copper planes connected to either the microcontroller $V_{DD}$ or GND, as follows:

- Top layer: Microcontroller $V_{DD}$;
- Top inner layer: GND;
- Bottom inner layer: Microcontroller $V_{DD}$;
- Bottom layer: GND;

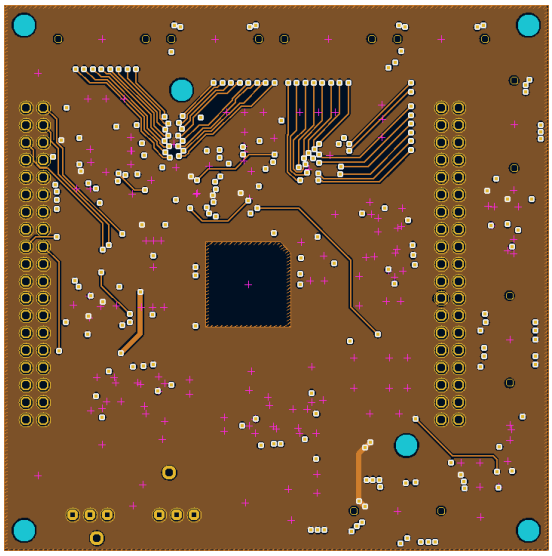
The presence of copper planes at a constant voltage helps shielding the interconnections and reducing the resistance of power lines, especially the GND. The choice of connecting copper planes to the microcontroller $V_{DD}$ was made because this supply needs to be distributed to an high number of devices (the voltage domain interfaces). In fig. 5.37 the layout of each PCB layer is shown; we can notice that no copper was placed at the center, beneath the IMU, that's a specific guideline given on the IMU hardware integration manual ([40]).



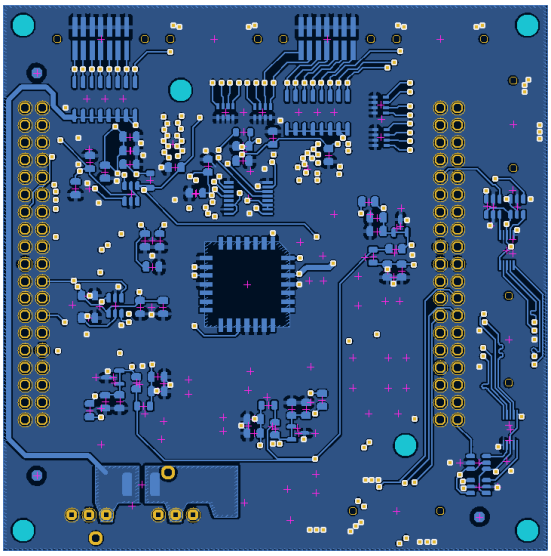
(a)



(b)



(c)



(d)

**Figure 5.37:** PCB layers. 5.37a Top layer. 5.37b Top inner layer. 5.37c Bottom inner layer. 5.37d Bottom layer.

#### 5.3.2.1 Floorplan

Fig. 5.38 shows the placement of system blocks inside the PCB.

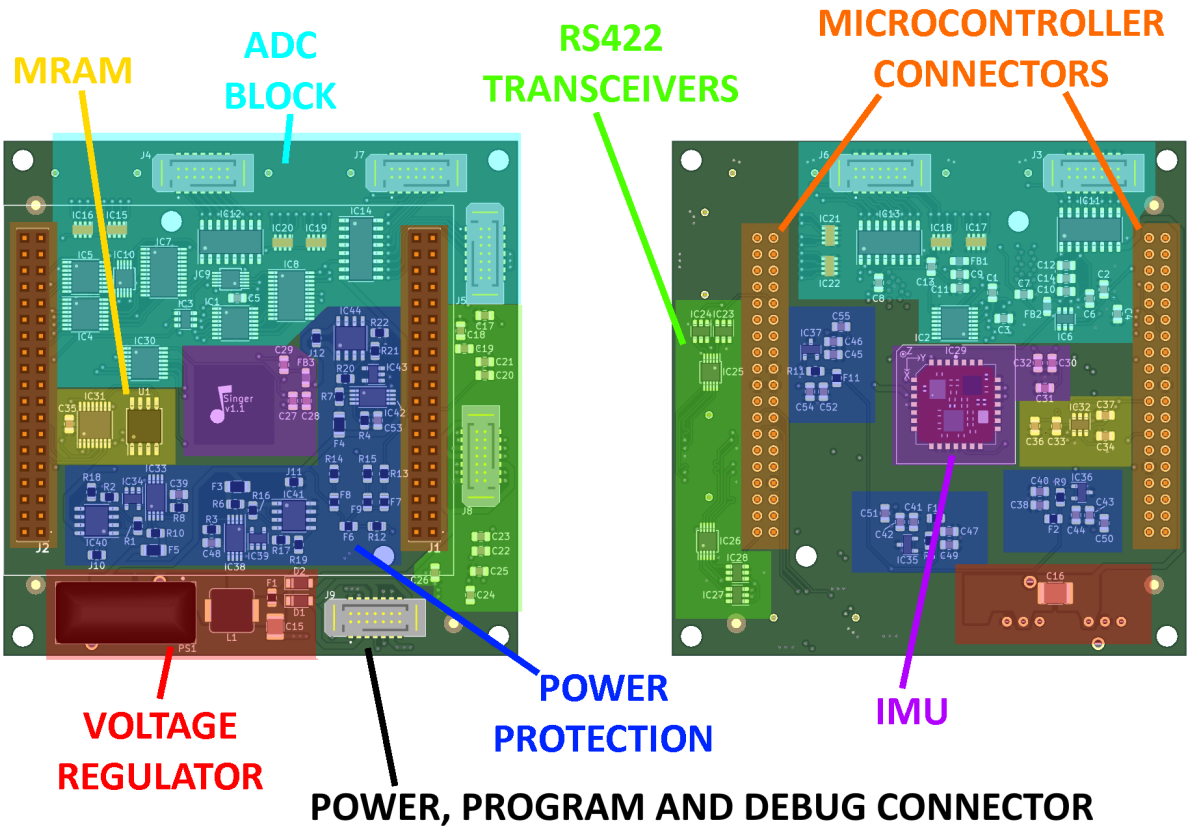



**Figure 5.38:** PCB floorplan. Top view (left) and bottom view (right), areas of the same color are part of the same block.

As can be seen, the block that consumes more PCB area is the ADC block, followed by the power protection block; these two occupy more than 50% of the total area. The blocks were placed by trying to minimize interconnections lengths, especially the ones towards the Nucleo header.

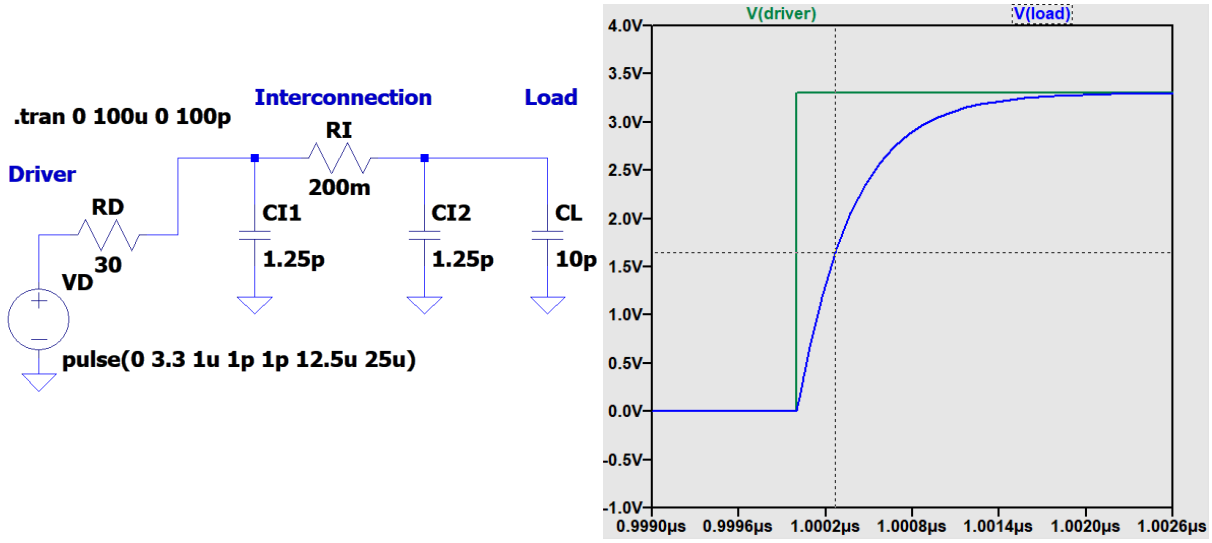
#### 5.3.2.2 Routing

Routing has been done utilizing two track widths: 0.25 mm for logical interconnections and 0.8mm for power lines. Tab. 5.8 summarizes the estimated track resistance and capacitance per unit length; the resistance has been evaluated with a copper resistivity $\sigma$ of 1.7x10$^{-8}$ $\Omega$m and the capacitance with an FR4 dielectric with $\epsilon_r$=4.3, neglecting the effect of neighboring tracks and considering the parallel plane capacitance of an inner layer track (having two copper planes on the two sides, neglecting fringing fields). As previously said the copper thickness is 34 $\mu$m, the PCB thickness is assumed to be 1.5 mm, with layers equally spaced ( 0.75 mm distance).

**Table 5.8:** PCB estimated tracks resistance and capacitance per unit length.

| Track width | Resistance p.u.l. | Capacitance p.u.l. |
|---|---|---|
| 0.25 mm | 20 mΩ/cm | 0.25 pF/cm |
| 0.8 mm | 6.3 mΩ/cm | 0.81 pF/cm |

From these values we can easily guess that there will be no problem for our <10MHz frequency circuit, but we can anyway try to estimate the interconnection delay and rise time introduced on logical interconnections. We'll assume a 10 cm track length (the longest track on the PCB), from which we obtain an interconnection resistance of 200 mΩ and stray capacitance of 2.5 pF; we also need to estimate the driver equivalent output resistance and the receiver capacitance of our integrated circuit, so we'll assume a 30 Ω output resistance and a 10 pF load capacitance, the equivalent circuit adopting a single segment $\pi$ model for the interconnection can be seen in fig. 5.39a. A SPICE model was set up to quickly simulate the network and evaluate the 50%-50% propagation delay and the 10%-90% rise time, whose result can be seen in fig. 5.39b.




(a) (b)

**Figure 5.39:** 5.39a Interconnection network with single segment $\pi$ model. 5.39b Simulation result.

The simulation gave a 50%-50% propagation delay $t_d = 0.27$ ns, it can be interesting to compare this value with the one given by the Elmore delay approximation $t_{pd,elmore} = R_D * C_{I1} + (R_D + R_I) * (C_{I2} + C_L) = 0.38$ ns. The 10%-90% rise time given by the simulation is $t_r = 0.83$ ns; it's clear than this doesn't pose a problem for our maximum clock period of $T_{4MHz} = 250$ ns.

### 5.3.3 Production

The PCB was manufactured and assembled by an external company, this required the author to provide a number of documents:

- Gerber files, containing masks for copper layers, PCB edge cuts, silkscreens (the writings on the PCB), solder paste and components placement (.gbr extension);
- Drill files, containing masks for PCB holes drilling: vias, through hole pads, mounting holes (.drl extension);
- Bill Of Material (BOM): a list of all components, the relative number and name on the circuit to be correlated with the placement gerber file (.csv extension);
- 3D models/Renders of the board, not strictly necessary but useful to give the manufacturer the expected PCB appearance and eventually help solving doubts;

All these files were easily generated by KiCad with the exception of the BOM, the latter is generated by python scripts and the default one outputs a file that is quite difficult to read, fortunately KiCad is completely open-source and allows deep customization including the possibility of adding custom BOM scripts. The entire fabrication process took less than 3 weeks to complete, a total of four boards (fig. 5.40) were produced: 2 boards without a soldered IMU (qualification models), 2 boards with the IMU soldered (flight model and spare); the choice of not soldering the IMU on the qualification models was made to reduce the cost, since the IMU had a cost higher than the overall system without it. In any case we had a IMU to be used for testing that was wired to the boards when needed.


**Figure 5.40:** The four produced Singer boards.

# Chapter 6
# Software design

This chapter addresses the development of the board firmware. At first, an overview of the software structure and development environment is made, follows a description of the software building blocks divided i two major sections: the low level software (drivers and utilities) and the high level part (tasks).

## 6.1 Software overview

The hierarchical organization of the firmware can be seen in fig. 6.1.

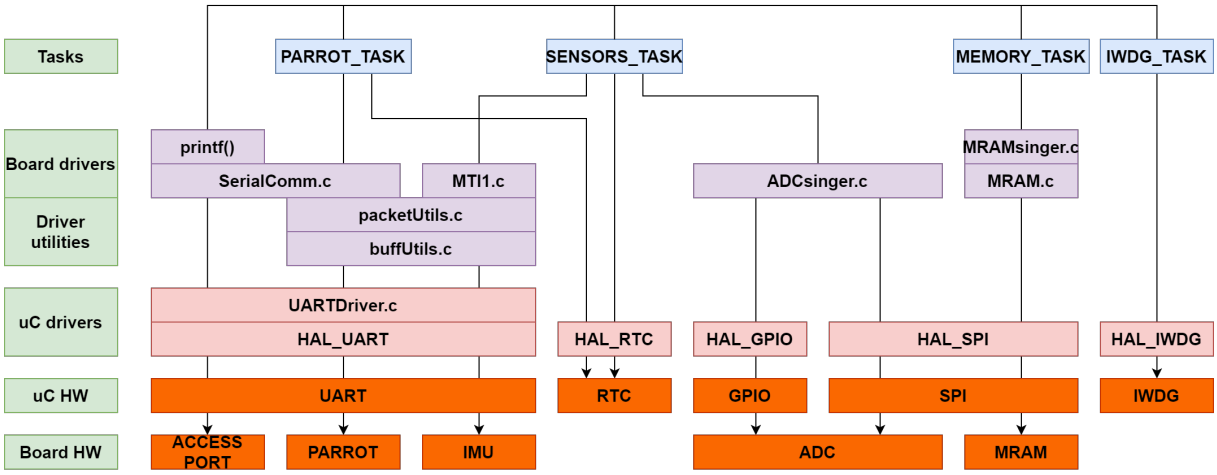



**Figure 6.1:** Singer firmware hierarchical structure.

As can be seen, the firmware is organized in hierarchical layers, starting from the bottom:

- board hardware: consisting on the PCB hardware that was already presented in chapter 5;

- uC hardware: the microcontroller peripherals used to interface with the board hardware or to implement other firmware functionalities, the image doesn't show the peripherals not directly involved in functionalities related with the mission but vital for the microcontroller operation (like the reset and clock controller and the interrupt controller);

- uC drivers: the drivers for the microcontroller peripherals, which include the ones provided by ST (Hardware Abstraction Layer, or HAL libraries) but also developed for

the purpose (like the second level UART driver);

- driver utilities and board drivers: consisting of drivers developed for the board hardware and utility libraries they rely on;

- tasks: the upper layer of the firmware, consisting of the actual tasks that implement the various system functionalities as state machines.

An important constraint during design of all the firmware was to never use dynamic memory allocation, dynamic allocation can be unpredictable and for this reason its usage is discouraged, NASA's list of programming best-practices explicitly says that on [43].

### 6.1.1 development environment

The microcontroller configuration was generated by STM32CubeMX graphical configuration tool (from now called CubeMX) that is provided by ST (site: [98]), with this powerful utility it's possible to configure all the microcontroller peripherals and even include and setup the microcontroller Real Time Operating System (RTOS). Various Integrated Development Environments (IDEs) were used for firmware programming and compiling, ST provides the STM32CubeIDE (site: [97]) IDE that is already integrated with CubeMX and provides all the needed tools for compiling (GCC compiler for ARM) and debugging (GDB), this was used not only for the complete firmware but also for the hardware test programs (see section 7.3). Visual Studio Code (site: [116]) was another IDE used for the most intensive coding phases because it offers a better user experience than CubeIDE, increasing the productivity, it comes with the disadvantage of not being natively integrated with the ST development pipeline and ARM utilities.

### 6.1.2 FreeRTOS

The firmware is based on FreeRTOS (website: [34], manuals: [10], [107]), an open-source Real Time Operating System (RTOS) distributed under MIT license that basically allows the generation and scheduling of threads (here called tasks) and provides inter-task communication primitives; this simple but complete set of capabilities is ideal for embedded projects and in fact CubeMX directly provides FreeRTOS as a configuration option and assists on its setup. Tasks are represented by a structure called Task Control Block (TCB), where all the variables to control the task an keep track of its state are located. Since the scheduler swaps the tasks context in any moment of their execution, each task owns a separate stack. FreeRTOS can allocate tasks in memory dynamically or statically, since in our application we prohibited the former the static mechanism is used, in this case the memory arrays to store the TCBs and stacks are given to the OS by the programmer and typically are declared as global variables.
FreeRTOS implements various inter-task communication mechanisms, the ones that are used in the firmware are queues and mutexes; FreeRTOS queues are First-In-First-Out (FIFO) primitives that allow buffering data in thread-safe manner between tasks and/or Interrupt Service Routines (ISR), while mutexes are the well known resource locking objects that are used to access data in mutual exclusive manner and in FreeRTOS their implementation is based on queues. Like tasks, also these objects can be allocated dynamically or statically, the latter being the adopted option. Fig. 6.2 shows the resulting

RAM allocation of FreeRTOS.

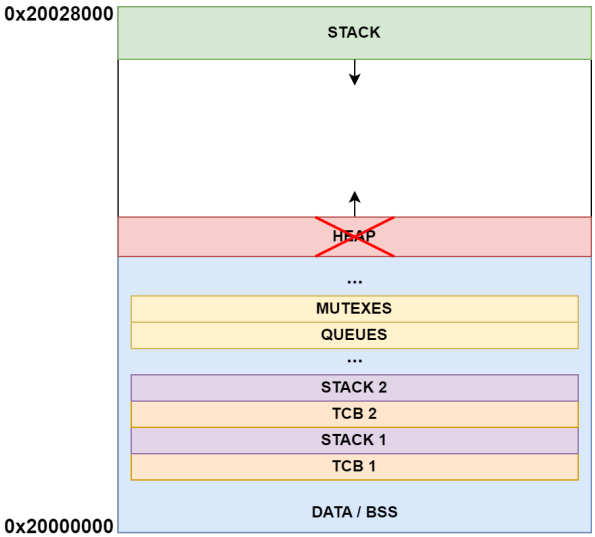



**Figure 6.2:** FreeRTOS with static RAM allocation.

FreeRTOS tasks are characterized by a priority and one of four states: ready, running, blocked, suspended; the suspended state can be set do temporarily disable a task from running and is never used in our application, tasks will always change state between ready (the task can be executed as soon as possible), running (the task is in execution) and blocked (the task is waiting to become ready due to some events). The FreeRTOS scheduler is called periodically by a timer interrupt with configurable period (in our case 1 ms) or when a task calls a blocking function that changes its state to blocked (for example waiting for a mutex to become available); when the scheduler is called it will change the running task with the following logic:

- if a ready task has higher priority than all the others, it will run;
- if there are multiple ready tasks with the same priority and higher than all the others, a round-robin logic executes the tasks in turn (time slicing).

From that, it's clear that higher priority tasks need to spontaneously pass in blocked state in order for the lower priority tasks to have some CPU time, this is usually done with the vTaskDelayUntil() function which blocks the task for a predefined amount of time, we will call this time the Task Repetition Period (TRP). The task with the lowest possible priority can never spontaneously block and it will be executed when no other task is ready, in fact FreeRTOS automatically starts an hidden idle task with lowest possible priority in order to guarantee that there's always a task running. An example of scheduling is shown in fig. 6.3

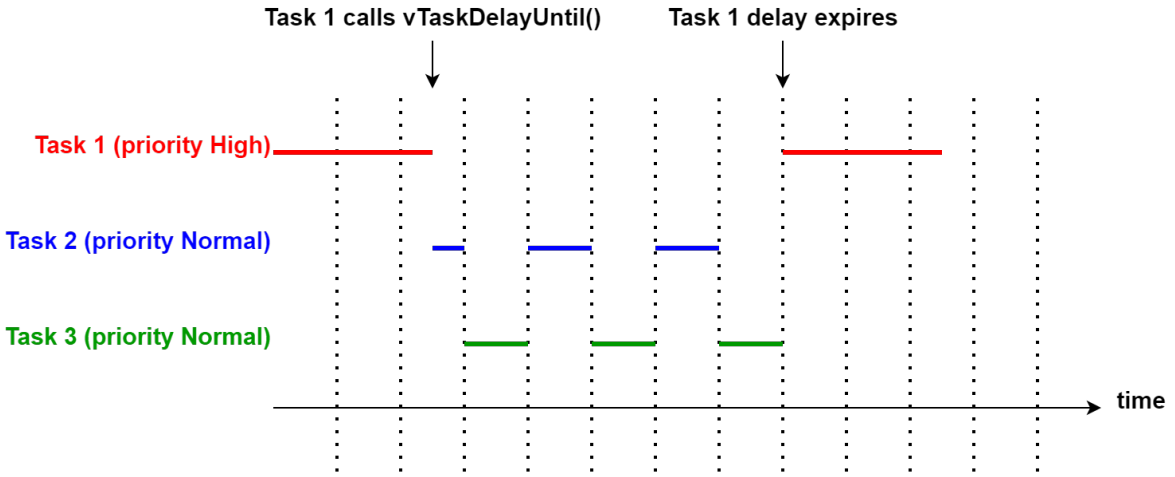



**Figure 6.3:** Example of FreeRTOS tasks scheduling.

## 6.2 Low level drivers

This section will overview all the firmware layers except the task level (refer to fig. 6.1). In fig. 6.4 the system logical interconnections are shown.

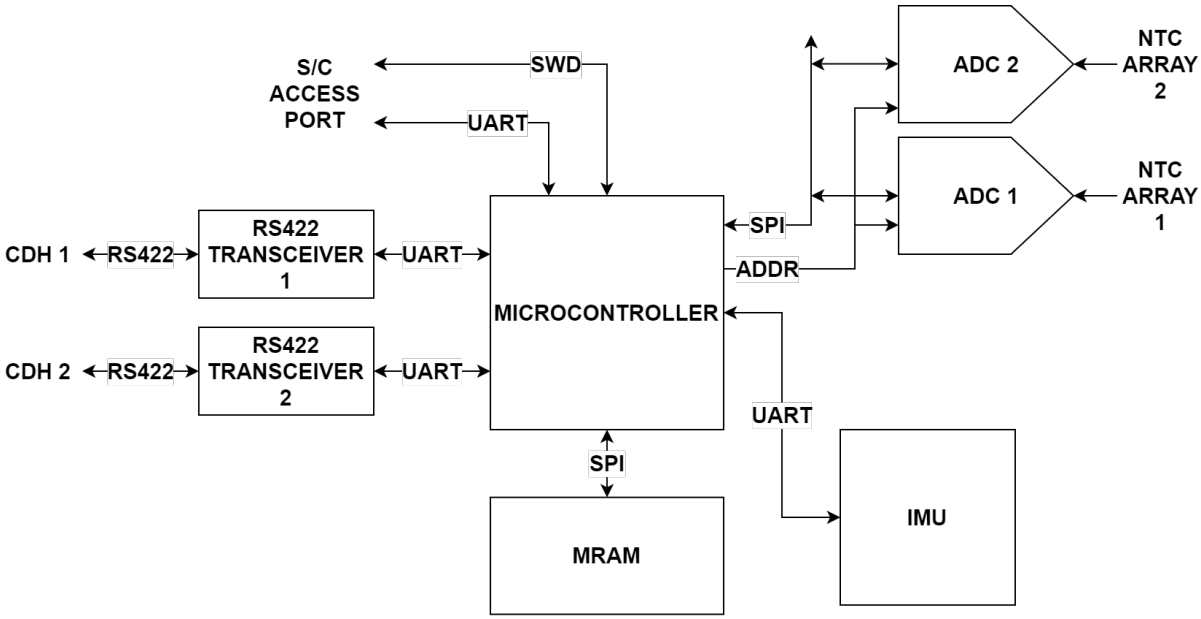



**Figure 6.4:** Singer logical architecture.

As can be seen, the system makes mainly use of two types of logical interconnections: UART and SPI; there is also the SWD interface that is the standard program and debug interface for ST microcontrollers. Some GPIOs were needed as chip select lines for the SPI interfaces and for setting the ADC MUX address. Fig 6.5 shows the microcontroller pin assignments in the CubeMX pinout configuration tab; for unused pins, the CubeMX function was used that automatically sets the pin mode to analog input, this solution reduces the power consumption of floating input pins.

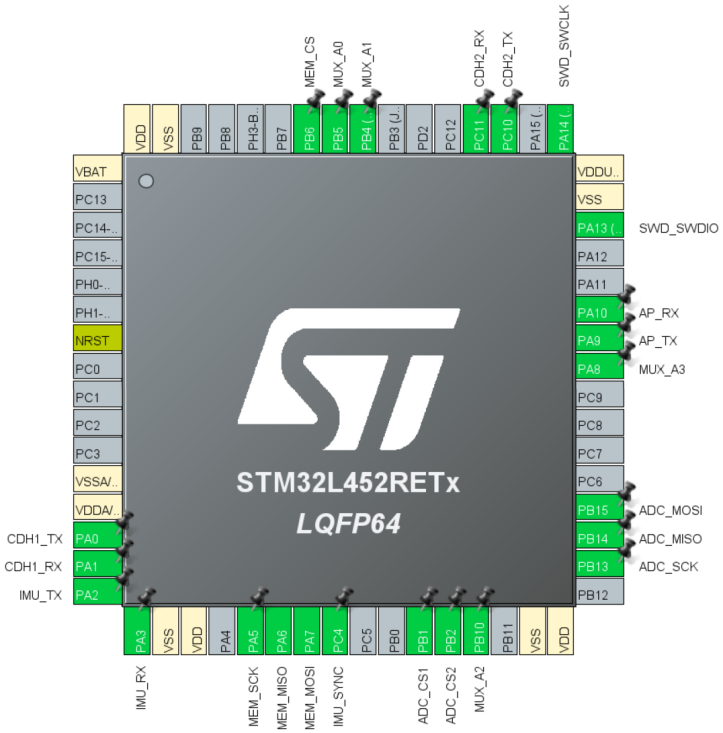



**Figure 6.5:** Microcontroller pin assignments.

## 6.2.1 Interfaces review

Before entering in the details of the driver implementations, it can be useful to have a brief review of the mentioned interfaces to serve as reference.

### 6.2.1.1 SPI interface

Introduced by Motorola/Freescale ([114]), the Serial Peripheral Interface (SPI) is a well known serial interface for board level transmission, it's a synchronous interface and so has both data and clock lines, it allows exchanging data between multiple slave devices and an unique master device (but also a multi-master configuration is possible); the basic idea behind SPI implementation (like for our microcontroller: [99] pag. 1306) is to swap the content of two or more shift registers during the exchange of data, as in fig. 6.6;

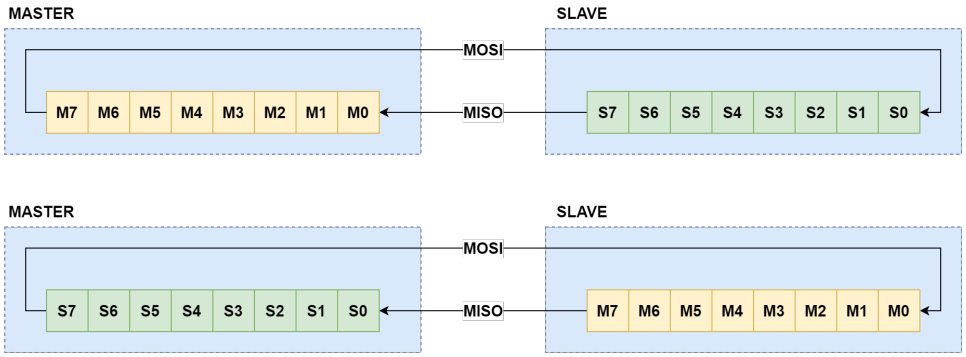



**Figure 6.6:** SPI basic concept with shift registers data swap. (Up) Shift registers content before the transaction. (Down) Shift registers content after the transaction.

Data exchange can happen by sending the data starting from the Most Significant Bit

(MSB) or the Least Significant Bit (LSB), typically data exchange happens with units of 8 bits but other variants exist (the microcontroller that we used can be set to exchange from 4 to 16 bits). The line that goes towards the master is called the Master In Slave Out (MISO) line, the other is called the Master Out Slave In (MISO).
The master device is the one that starts the SPI transaction with a specific slave by changing the state of a chip select (CS) line (typically but not always, the chip select line is active low), the remaining slaves will keep their output drivers to an high-impedance state; the master then sends the clock and the two devices exchange their data one bit per clock period. The clock can have low or high value at rest (usually called clock polarity or CPOL) and the data can be valid at the first or second clock edge (usually called clock phase or CPHA), fig. 6.7 represents the timing diagram of a typical SPI transaction.

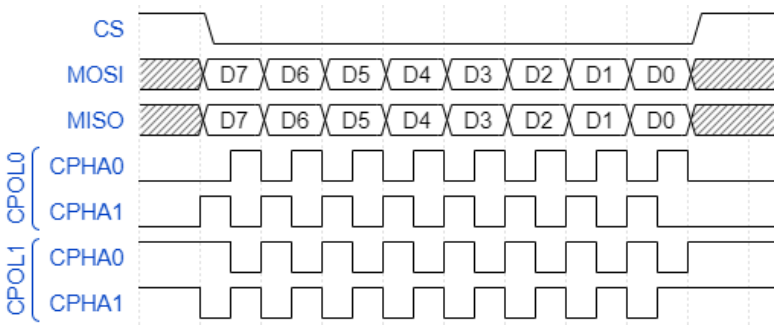
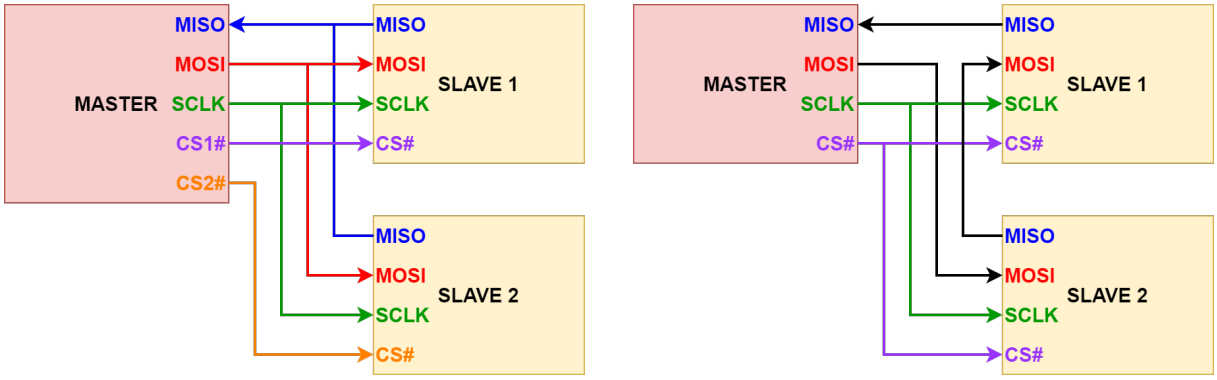



**Figure 6.7:** Typical SPI transaction, the four possible combinations of clock polarity an phase are shown.

SPI allows multiple configurations of slaves, the two most notorious configurations are shown in fig. 6.8.




**(a)** **(b)**

**Figure 6.8:** 6.8a SPI in multi-drop configuration. 6.8b SPI in daisy-chain configuration.

- Multi-drop: in this configuration all the slaves share the MISO and MOSI lines, the master selects one slave at a time and performs one or multiple transactions with the latter, this configuration is the one that was adopted for all SPI buses on Singer; of particular concern for this configuration is that one and only one slave must be active at any time, otherwise there's the risk of causing short circuits due to multiple strong drivers trying to set the line state.

- Daisy-chain ([25]): in this configuration the MISO and MOSI lines are arranged to form

a continuous ring, the master selects all the slaves at the same time and then sends the data for some or all the slaves, this data needs to be aligned accordingly to obtain at the end that each slave has the properly aligned data on its shift register. This configuration completely eliminates the problem of multiple drivers on the same line because there's no more a common bus and also reduces the number of chip select lines (that with multi-drop can become significant); the main problem is that if one of the slaves breaks, the whole bus is lost. Daisy chain also needs the slave to consider the data as valid only when the chip select is disabled at the end, ignoring everything that is received in the meantime; this could not be the case with complex devices and in fact it wasn't the case neither with the ADC nor the MRAM of Singer, which actively read the received data for commands and then immediately respond inside the same chip select window.

#### 6.2.1.2 UART interface

UART (Universal Asynchronous Receiver Transmitter) interface implements an asynchronous communication protocol between two devices (fig. 6.9) and as the name suggests doesn't provide a clock line together with the data, the communication speed (baud rate) needs to be agreed between sender and receiver for the communication to correctly take place and usually the UART peripheral allows choosing from a wide range of standard values; the typical UART frame is shown in fig. 6.10. An UART communication channel is bidirectional and full duplex but the two lines are identical and nothing prevents using only one direction for simplex communication.

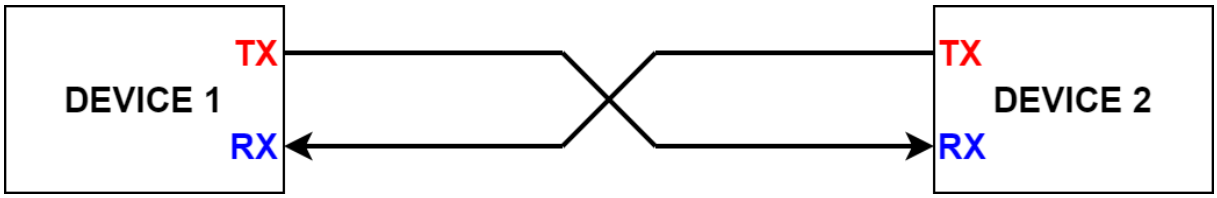



**Figure 6.9:** UART communication between two devices.

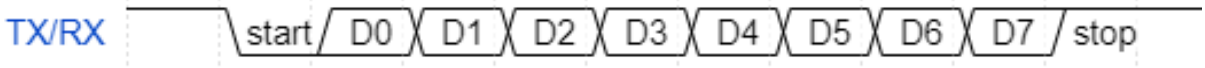



**Figure 6.10:** UART frame with 8 bits of data and no parity bit.

While in idle state, the UART line will have an high logic level, to start sending a frame the logic level of the line is set to low for one bit period (start bit), at this point the receiver will synchronize with the incoming frame and start sampling the line (a typical implementation, like in our microcontroller, is to sample the line at a rate much higher that the transmission baud rate and apply a voting policy to determine the bit values), synchronization is necessary because the transmitter and receiver clocks will always have some difference and drift with time one respect to the other; the transmitter sends a configurable number of data bits (8 in our application) and optionally a parity bit (not used in our application), then the line state is set to high (stop bit) for at least one bit period before a new start bit can be sent. UART is the name of the interface peripheral that implements the UART frame and outputs the data with TTL/CMOS logic levels through the device I/O pins, this levels are commonly translated to different physical layer

protocols like for example RS232 (like in [84]) or RS422 (like in Singer for communication with the CDHs); for short distances the bare CMOS output from integrated circuits can also be used (like in Singer for the IMU and Access Port).

#### 6.2.1.3 SWD interface

The Serial Wire Debug (SWD) interface is the program and debug interface for ARM based devices and consists of a bidirectional interface by which data is read or written on the microcontroller. The data TX/RX line is called SWDIO and the corresponding clock line is called SWCLK. The STM32 microcontrollers also need two more lines to be correctly programmed: the VDD_target line, which is used by the programmer to identify the target supply voltage and the microcontroller reset pin nRST that is used to reset the microcontroller and execute the boot sequence for programming. Entering in the details of this interface is out of the scope of this thesis, for more information see the specifications on [8].

### 6.2.2 HAL drivers

The Hardware Abstraction Layer (HAL) libraries, provided by ST (documentation: [26]) and generated by CubeMX, are the base on which the entire firmware is coded. These libraries, as the name suggests, provide an abstracted interface to interact with the microcontroller hardware and save the designer from building the basic drivers to manage the numerous peripherals; the downside of this approach is that these libraries are intended to be as portable as possible on different microcontrollers and for this reason are often over-complicated and somehow cryptic; especially while debugging it's often difficult to understand what's going on at the low level and surely in situations in which development time is not an issue it should be better to code the firmware resorting to the LL (Low Layer) libraries or develop the drivers from scratch.

### 6.2.3 UART drivers

The UART peripheral is the most used in the system with four instances being involved: one for each RS422 communication interface, one for the IMU and one for the Access Port debug console. In all these cases it's used with a speed of 115200 baud, 8 bit of data and no parity bit. The UART is also the interface with the highest number of library layers due to its asynchronous nature and the fact that is often used to frame more complex packet structures, as we will later see in this section.

#### 6.2.3.1 Peripheral drivers

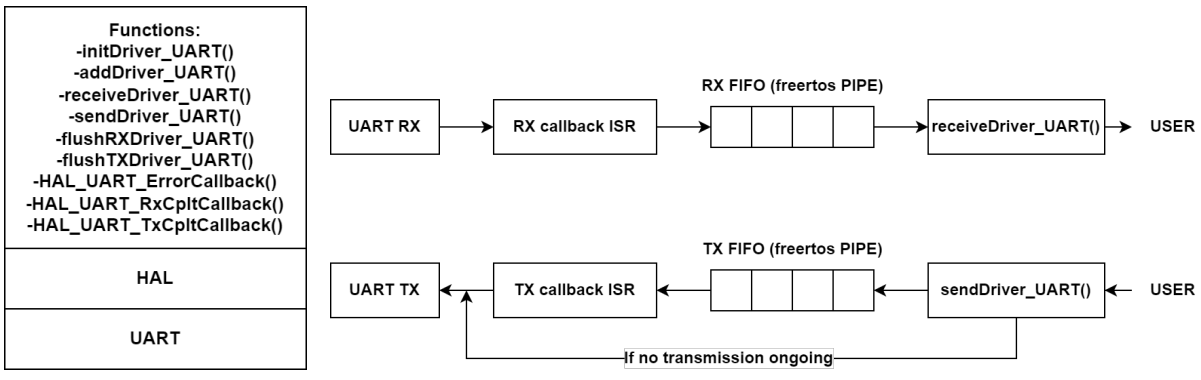
Using the UART peripheral in polling mode makes it practically sure that an high number of frames gets lost, especially when running a RTOS with multiple running tasks and a preemption period of 1 ms (even with 9600 baud we would receive a byte every 1.04 ms, for 115200 baud it becomes 86.8 $\mu$s) . For this reason the only feasible options were to use it in interrupt mode or Direct Memory Access (DMA), the first one was chosen because it's the most deterministic one and our relatively low speed and CPU usage allowed that: assuming a 100 instructions Interrupt Service Routine (ISR) and 50 clock cycles for context change we get an ISR execution time of 2.3 $\mu$s @ 64MHz of CPU clock frequency, if we multiply this number for the worst case scenario of all four UARTs transmitting and receiving at the same time we get 18.8 $\mu$s, that with a frame period of 86.6 $\mu$s yields 22 %

of CPU usage for UART interrupts; this number is surely not ideal but in our context is good enough for choosing this handling method, it also has to be noticed that the Access Port UART is not used in orbit and this reduces the percentage to around 16 %.
The UART driver is then based on the HAL interrupt functions but these are limited in functionality (especially in reception) for the following reasons:

- the philosophy behind HAL interrupt drivers is that the user should assign a buffer of chosen dimension and a callback function to be called when the buffer has been filled (or emptied in transmission) with the requested number of bytes, this is obviously impractical in a real life scenario because the number of expected bytes is rarely known a-priori and even in that case the loss of one RX frame due to channel errors would potentially lead to the callback never being called. The solution is to call the interrupt reception and transmission functions with a single byte buffer, leading to the next problem;

- the HAL libraries don't provide a proper method of executing mutually exclusive code (if we exclude interrupt disabling) nor some primitives for queuing data between ISR and loop code, it could be easily implemented in a first approximation with monodirectional flags (load and store operations are atomic) but this solution would have basically reduced the UART management back to polling mode. The simplest alternative would be to implement a monodirectional ring buffer exploiting the atomic load and store of 32 bit variables, an approach that is surely feasible if an RTOS is not used but in our case the simplest way was to exploit the FreeRTOS queue primitive.

The UARTDriver.c library implements just that: it provides an interface that resembles the read() and write() functions of an UNIX character driver, here called receiveDriver_UART() and sendDriver_UART(); these functions will mask the user from the underlying mechanisms of transmission and reception in interrupt mode, the driver outline and data flow pipeline can be seen in fig. 6.11.




**(a)** **(b)**

**Figure 6.11:** 6.11a UART driver outline. 6.11b UART driver data flow pipeline.

The receiveDriver_UART() and sendDriver_UART() functions will basically read/write a character from/into FreeRTOS queues, the HAL interrupt mode functions will be called by passing a single character buffer at a time, the RX callback logic is quite simple: it just inserts the newly received character into the queue and then call the HAL receive

function again; the TX logic is a bit more complicated because it must address two possible conditions: transmission ongoing (queue not empty), in this case the HAL transmit function will be automatically called in the next callback and the sendDriver_ just inserts the new character in the queue; transmission not ongoing (queue empty): in this case the sendDriver_ function must call the HAL TX function after inserting the new byte, it also must handle the problematic condition in which the byte was removed from the queue by the ISR just before the sendDriver_ was called (and so the next callback has not been yet executed).
The library provides two possible policies to apply if the reception queue is full: the keep_old policy will discard new bytes until there is again space in the queue; keep_new policy will shift out the older bytes to make space for the newly arrived ones. Multiple UART peripherals can be managed by the library, which provides a function to add new UART handlers and places them in a proper data structure (fig. 6.12). It also provides functions to flush the RX or TX queues.

```
typedef struct DriverHandle_UART
{
    volatile uint8_t _usageFlag;                        //to flag that the strcture is valid
                                                        //(0 if not used)
    uint8_t _rxByte;                                    //where the HAL ISR will store the
                                                        //received byte
    UART_HandleTypeDef* _huartHandle;                   //UART handle
    IRQn_Type _irq;                                     //irq number of the uart
    QueueHandle_t   _rxQueueHandle;                     //rx queue handle
    uint8_t _rxQueueStorageBuffer[SERIAL_RX_BUFF_LEN];  //rx queue data buffer
    StaticQueue_t _rxQueueBuffer;                       //rx queue buffer
    QueueHandle_t   _txQueueHandle;                     //tx queue handle
    uint8_t _txQueueStorageBuffer[SERIAL_TX_BUFF_LEN];  //tx queue data buffer
    StaticQueue_t _txQueueBuffer;                       //tx queue buffer
    fifo_policy _policyRX;                              //rx buffer policy
} DriverHandle_UART;
```

**Figure 6.12:** UART driver data structure.

### 6.2.4 Utility libraries

These utility libraries were developed to address the common problem of buffering and packet searching inside a serial stream of bytes, for this reason they are mainly used inside drivers that involve the UART but they could be potentially applied to any steam.

#### 6.2.4.1 Buffer utilities

The buffUtils.c library was developed to provide two types of buffer implementations and relative utility functions: plain and circular. Its goal is to offer a more comfortable way of handling buffers, especially considering the amount of metadata that usually needs to be carried with them, like for example the current number of contained elements; this metadata is packed together with the buffer pointer inside an handle structure and a set of functions have been developed to easily insert/remove elements from the begin or the end, shift or rotate the buffers, convert one type to the other.
The plain buffer is basically a normal buffer (consecutive portion of memory starting at index 0) that is packed with information about the buffer size and current number of elements; the circular buffer instead is a ring buffer implementation that needs an additional metadata about the index of the first element inside the memory region; the

two buffer structures and relative functions can be seen in fig. 6.13.

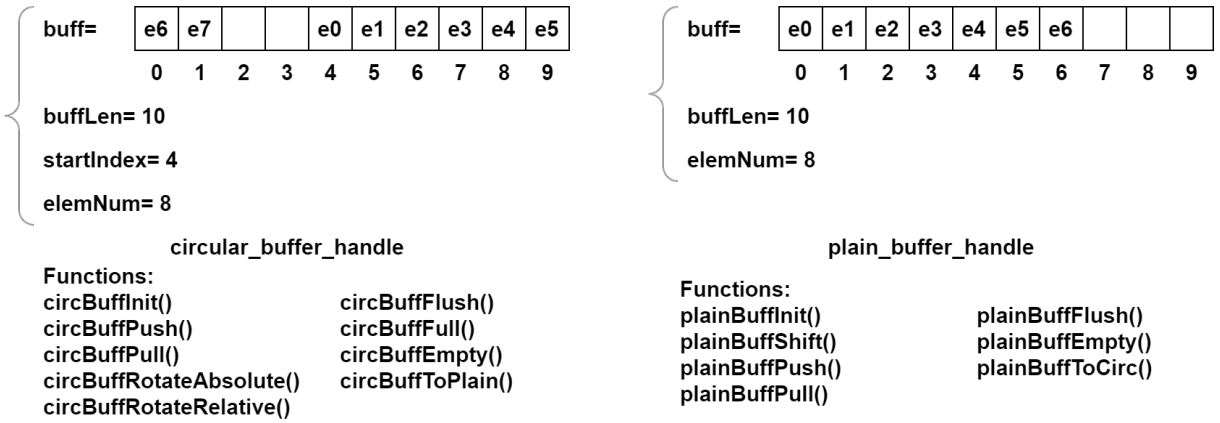



**Figure 6.13:** Circular buffer and plain buffer structures and functions.

Of particular interest for our application is the function that converts a circular buffer to a plain one, circBuffToPlain(), this operation involves a rotation of the circular buffer to align it back with the memory (make the starting index 0) and is heavily used because it links the context in which circular buffers are most suited (receiving and buffering a stream of characters from the UART driver) to the one in which plain buffers are the best option (packet searching), leading to the next library in the list: the packetUtils library.

### 6.2.4.2 Packet utilities

The packetUtils.c library is the core library of all packet-searching jobs of the firmware and is used both for the IMU driver and for Parrot communication, it's a quite complex but flexible library which provides two function, covering two main objectives:

- search packets inside a stream of data: by identifying them with a set of rules defined by the user; the rules include the definition of a packet head sequence and/or a tail sequence, a minimum packet length and a policy to apply when bytes that are part of the head/tail sequences are found inside the packet, this is implemented by the searchPacket() function of the library;

- ensure a continuous flow of data: by implementing rules to automatically shift out bytes from the search buffer whenever a packet is found or the buffer is full, so that the user doesn't need to care about this aspect and only needs to continuously call the function to search for new packets and fill the buffer whether some space is available, this is implemented by the searchPacketAdvance() function (that wraps searchPacket()).

```
uint32_t searchPacket(plain_buffer_handle* handle, plain_buffer_handle* pckt,
                      search_pckt_rule * rule);

uint8_t searchPacketAdvance(circular_buffer_handle* handle, plain_buffer_handle* pckt,
                            search_pckt_rule * rule, uint8_t shiftFlags);
```

**Figure 6.14:** searchPacket() and searchPacketAdvance() functions headers.

The searchPacket() function (header in fig. 6.14) is applied to a plain buffer from the buffUtils.c library, other function arguments are another plain buffer where the eventually

found packet is placed and the search rules structure, the rules can be set up to search various types of packets and are passed to the function by a rule structure (fig. 6.15), this struct contains the head and/or tail patterns as arrays, the minimum allowable packet length and a policy to apply if partial head/tail patterns are found inside the packet; fig. 6.16 shows some of the possible packet formats that the function is capable to find.

```
//policy for packet search in case not head or tail but parts of them are found inside the packet
typedef enum{
    hard,       //hard policy, even a part of head/tail will make
                //the function reject the packet
    medium,     //medium policy, a part of head/tail will not make the packet rejected
                //but a complete head or tail will
    soft,       //soft policy, the packet can contain bytes that are part of head/tail,
                //also complete heads/tails
} head_tail_policy;

//directives for packet search
typedef struct{
    uint8_t * head;     //buffer with packet head
    uint32_t headLen;   //length of head buffer
    uint8_t * tail;     //buffer with packet tail
    uint32_t tailLen;   //length of tail buffer
    uint32_t minLen;    //minimum packet length (head and tail excluded)
    head_tail_policy policy;    //policy for partial heads' or tails' bytes inside packet
} search_pckt_rule;
```

**Figure 6.15:** Packet search rules.

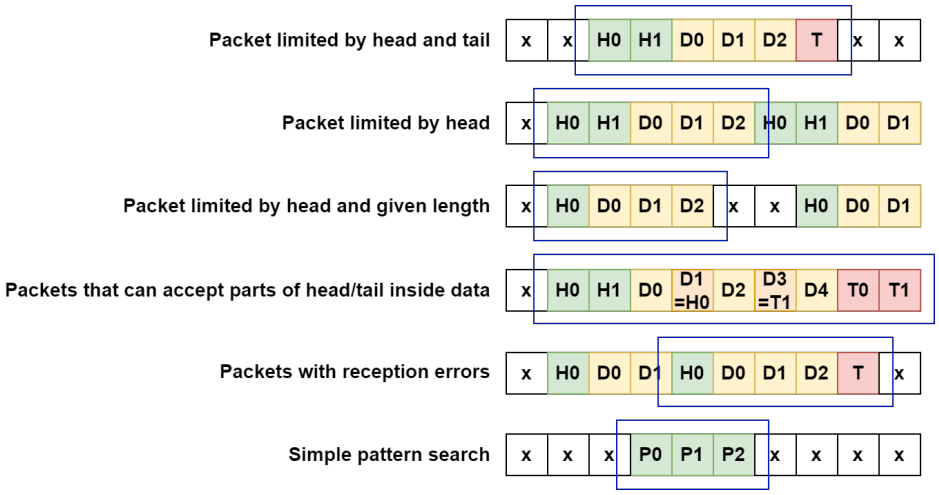



**Figure 6.16:** Example of possible packet formats.

To do so, the function implements the flow graph of fig. 6.17a: the plain buffer is analyzed byte by byte starting from the beginning, then a routine is executed on each byte that determines the byte properties; byte properties (fig. 6.17b) are flags and variables that will signal the subsequent code that a given byte is part of a complete or partial head/tail pattern and its positional index inside the pattern. This data is then fed to a state machine that implements the core logic of the function and that can be seen in fig. 6.18; the function is told to work in mode 1 if the search rule includes both an head and a tail, if only an head is given the function is told to work on mode 2 and it will search an occurrence of the given head followed by a number of bytes equal to the given minimum packet length, the mode 2 check logic is also shown in fig. 6.18.

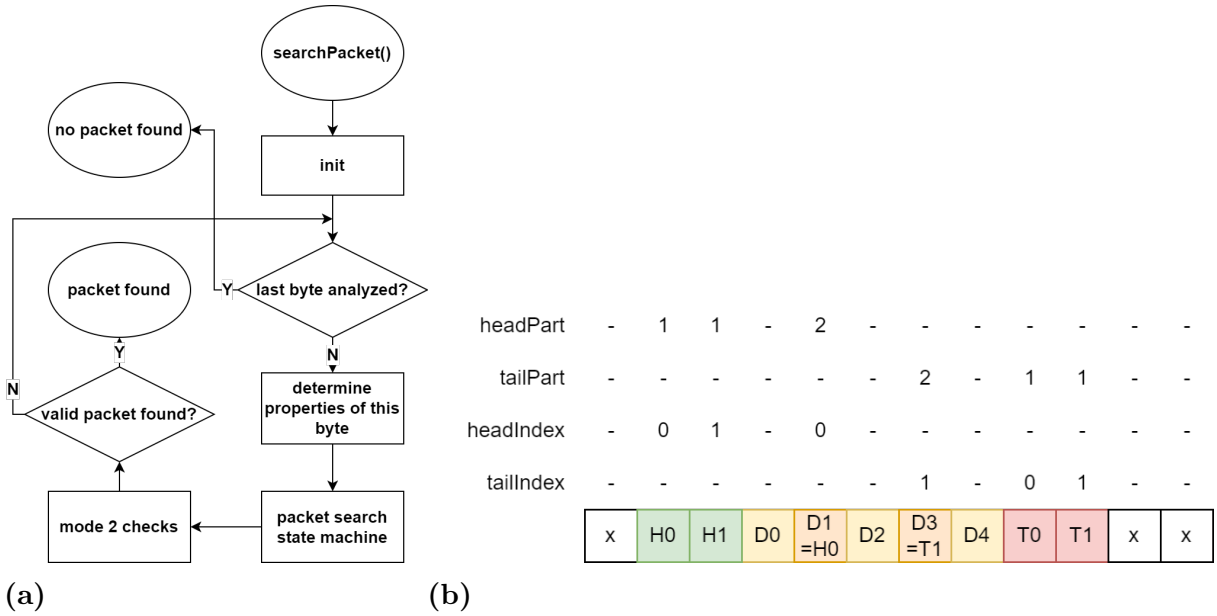



**Figure 6.17:** 6.17a searchPacket() function flow graph. 6.17b Example of byte properties determination, heads are green, tails are red, partial head/tails are orange, other packet bytes are yellow.

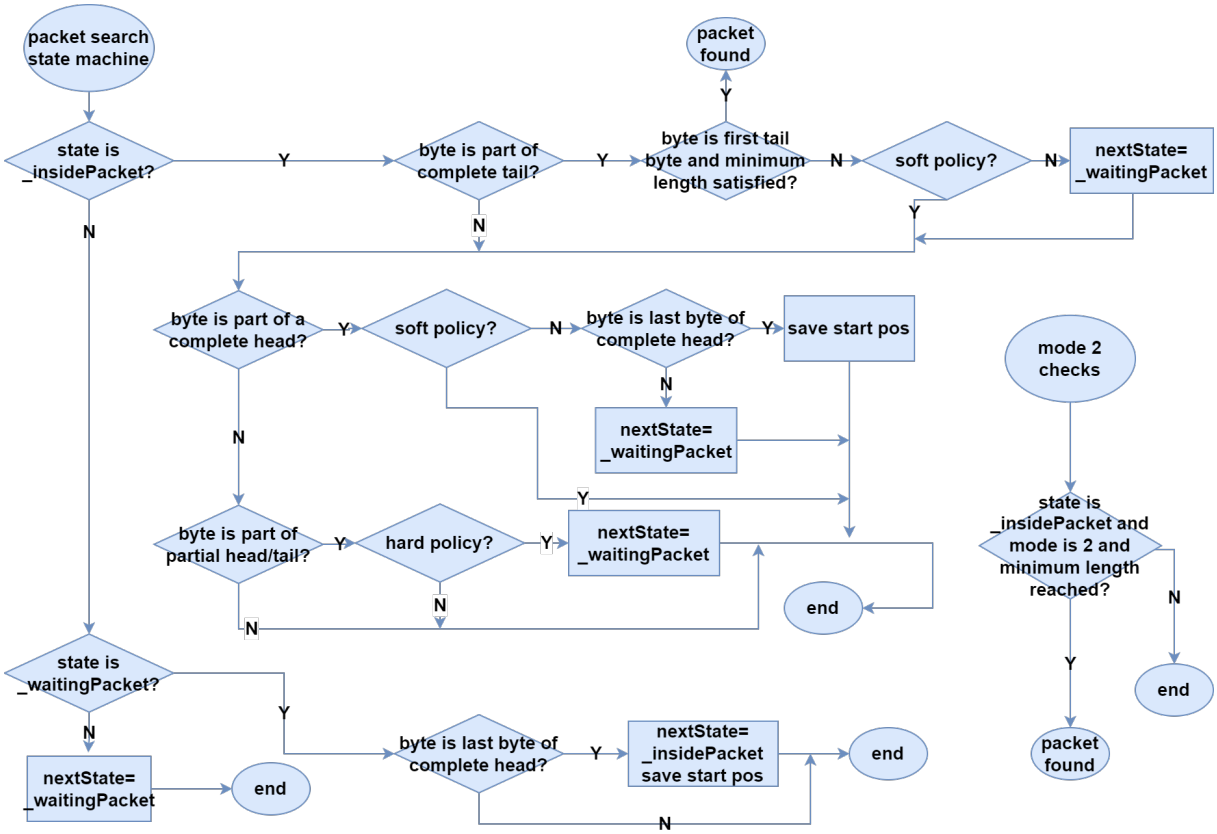



**Figure 6.18:** Search packet state machine and mode 2 check.

The second main job of the library (the automatic flow of data in the buffer) is instead

performed by the searchPacketAdvance() function (header in fig. 6.14), it's core is always the searchPacket() function around which this extra layer of functionalities is added; the searchPacketAdvance() function works on a circular buffer instead of plain, that's because as already anticipated the best option to store bytes arriving from the UART (or a generic serial) stream is a circular buffer, in fact a plain buffer would need continuous shift operations that become computationally intensive with large dimensions. The circular buffer needs anyway to be rotated to convert it to a plain one before the search operation can take place (buffer alignment) but this is done with a buffer rotation algorithm that has a complexity of O(N) with N being the number of buffer elements, instead of the complexity of $O(N^2)$ of the basic shift algorithm, the rotation is also performed only if really needed (so if the circular buffer indexes overflows in memory), as shown in fig. 6.19.

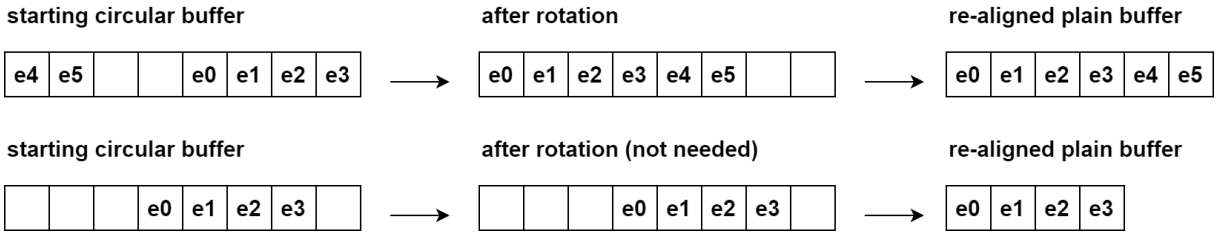



**Figure 6.19:** Example of buffer alignment. (Up) Rotation needed. (Down) Rotation not needed.

A series of flags can be passed to the searchPacketAdvance() function to set the data advancement logic, the possible flags are shown in fig. 6.20 where the logic applied to each one is also explained.

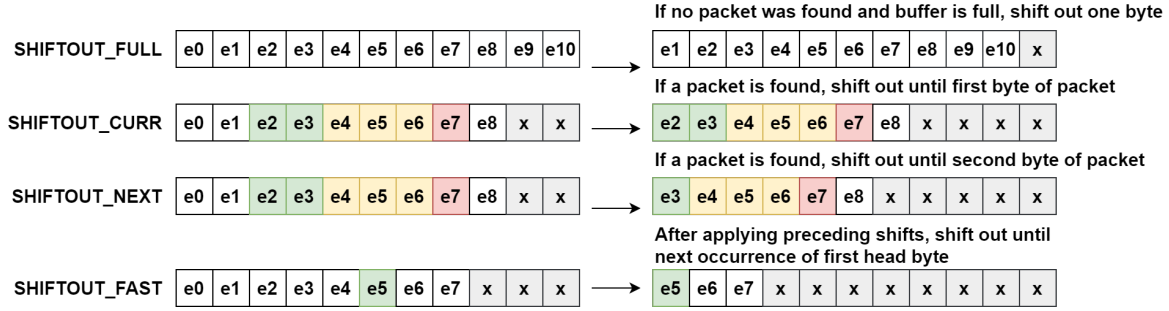



**Figure 6.20:** Buffer advance flags and relative logic. Head bytes are in green, tail bytes in red.

Finally, a step-by-step example is given in fig. 6.21 to show the overall working of the searchPacketAdvance() function in a real-case scenario.

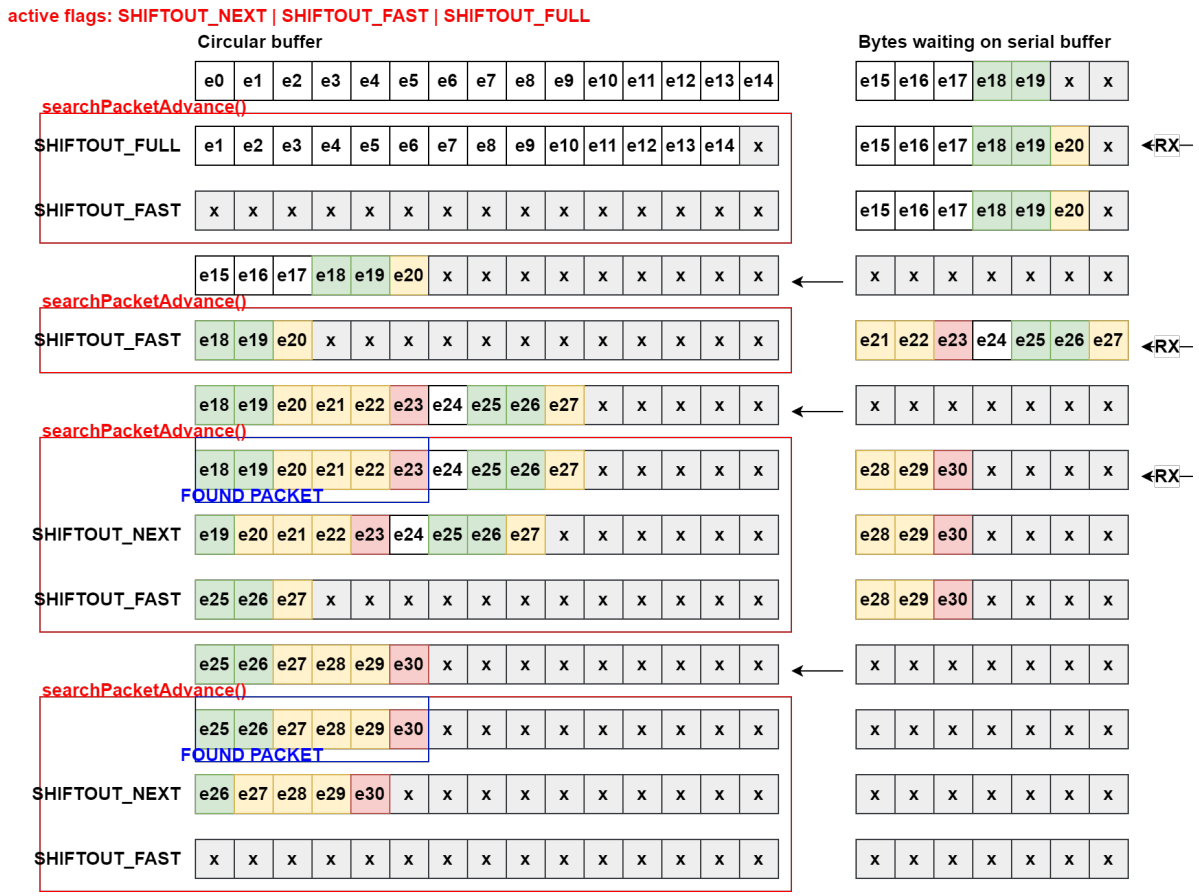



**Figure 6.21:** Real-case example of searchPacketAdvance() applied to an incoming stream of data. Head bytes are in green, tail bytes are in red. Function calls are represented as red squares around the circular buffer, function flags that cause the buffer to advance on each step are shown on the left.

## 6.2.5 IMU driver

### 6.2.5.1 Xbus protocol and IMU op-modes

All Xsens products of the MT line, including our MTi-3 IMU, implement a low level communication protocol (Xbus) that is specified in [62]; this protocol consists of an exchange of packets (messages) with the format shown in fig. 6.22.

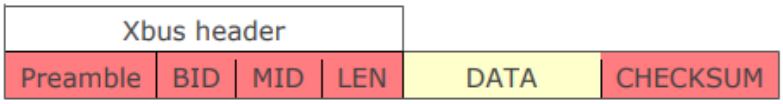



**Figure 6.22:** Xbus message frame, picture from [62].

The message starts with a 4 bytes header composed of the following fields:

- a preamble of value 0xFA;
- a Bus Identifier (BID), identifying the device that sent the message, in our case will always be 0xFF;
- a Message Identifier (MID), identifying the type of data that is sent;

- a message length field (LEN), identifying the number of message data bytes.

The header is followed by:

- a number of LEN bytes of data;
- a 1 byte checksum, the checksum is the 2's complement of the 1 byte-wide sum of all data bytes and basically the message should be considered valid if the 1 byte-wide sum of all data fields including the checksum is equal to zero.

The protocol also specifies an extended length message which is of no interest for our application.
There are various MID values available (for commands or relative Acknowledge messages, or ACK), but for our application we were only interested in the following ones:

- GoToConfig (0x30) and relative ACK (0x31): sends the IMU to configuration state;
- SetOutputConfiguration (0xC0) and relative ACK (0xC1): sets the desired IMU output configuration;
- GoToMeasurement (0x10) and relative ACK (0x11): sends the IMU to measurement state;
- MTData2 (0x36): the message containing sampled values, this message is continuously sent in measurement state and has the format shown in fig. 6.23

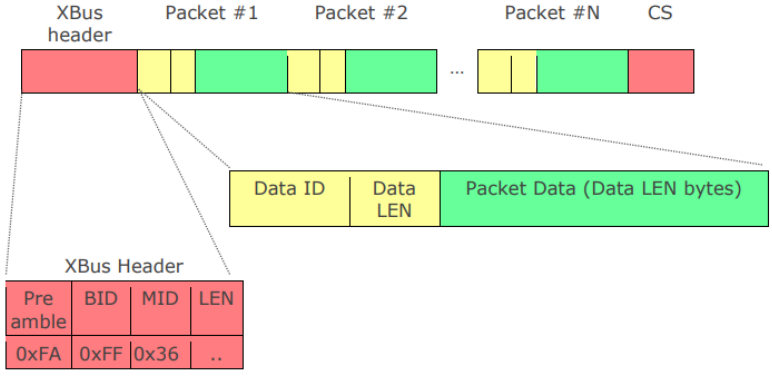



**Figure 6.23:** MT-Data2 message format, picture from [62].

Each category of measure (magnetometer, accelerometer, gyroscope, quaternions, ...) is formatted on a sub-field of the message data, starting with a data identifier and a length field; the frequency of arrival (sample rate) of the MTData2 message, as well as the desired types of measures, are set by the already mentioned SetOutputConfiguration message.
The IMU has two main operative modes, shown in fig. 6.24: Measurement state, where the IMU will continuously send measured samples on the UART at a rate that can be set by the user (in our case we set the minimum possible, 100 sample/s); Config state, where the IMU will not send samples and the SetOutputConfiguration message can set the type of data that we want it to output in measurement state, together with the output sample rate.

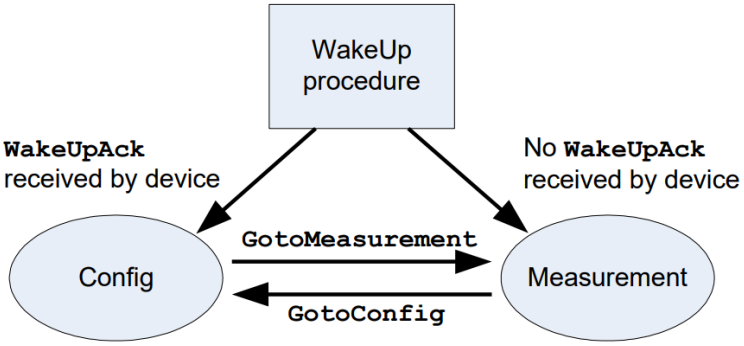



**Figure 6.24:** IMU operative modes, picture from [62].

#### 6.2.5.2 Driver library

The MTi1.c library implements the minimal IMU functionality that we needed for the system, its outline can be seen in fig. 6.25, as can be seen the driver is built on top of the UART driver and uses the buffer and packet utility libraries for analizing the incoming stream of data and search for valid protocol messages.
A series of private functions were defined to execute elementary operations like send and receive messages, perform command-acknowledge transactions, compute the checksum and extract the sampled values from the packet.
The high level interface is composed of two public functions that perform all the necessary operations in a compact way:

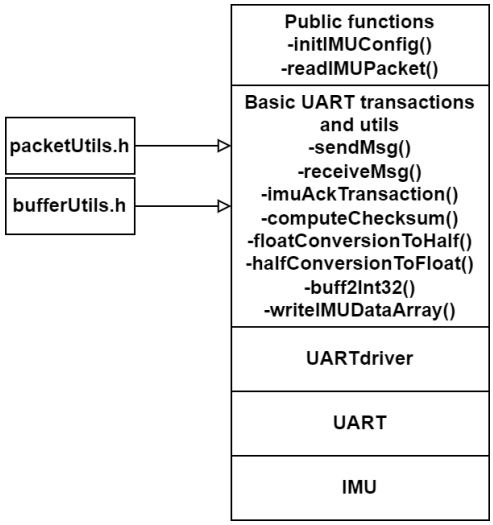



**Figure 6.25:** IMU driver outline.

- the initIMUConfig() function initializes the IMU after reset by sending the GoToConfig message followed by the SetOutputConfiguration command by which the desired output values are set (gyroscope and magnetometer), follows the GoToMeasurement command that sends the IMU to measurement state; the function checks for the reception of ACK messages after each command and otherwise can retry the sequence a predefined number of times before declaring the initialization as failed;

- the readIMUPacket() function will be called to read the last received values an place them in arrays of uint16_t, this function searches for the next complete packet with MID of MTData2 (0x36) and extracts the values from their predefined position inside the message; the function works in blocking (polling) mode and so a timeout is needed to return if no valid packet was found after an amount of time.

### 6.2.6 Parrot communication library

#### 6.2.6.1 Parrot message frame

The packet level communication protocol with parrot consists of an exchange of messages with the format shown in fig. 6.26


**Figure 6.26:** Parrot communication protocol message format.

The message fields are:

- a message head byte with value 0x7E;
- a protocol version field (1 byte), in our application this field will have value 0x01;
- a source address field (2 bytes);
- a destination address field (2 bytes);
- a message identifier (2 bytes);
- an arbitrary number of data bytes (payload);
- a Cyclic Redundancy Check field (2 bytes);
- a message tail byte with the same value of the head byte (0x7E).

The protocol also requires that the head/tail byte (also called frame byte or FRM_BYTE) of value 0x7E never appears inside the message if not for the head and tail, this requires the presence of a byte stuffing algorithm that is applied to the message during encoding/decoding, following this logic:

- if a FRM_BYTE (0x7E) is found, it's replaced by an ESC_BYTE (0x7D) followed by an ESC_FRAME (0x5E), this ensures that no FRM_BYTEs are present inside the message;
- if an ESC_BYTE (0x7D) is found, an ESC_ESCAP (0x5D) byte is inserted after it in the message, this allows having a normal ESC_BYTE+ESC_FRAME sequence as data inside packet and not wrongly interpret it as stuffing of FRM_BYTE.

#### 6.2.6.2 Parrot communication library

Not exactly a board device driver but placed at the same level, the SerialComm.c library implements the packet level communication protocol with the CDHs over the RS422 line, this library was not developed by the author but instead was provided by the team that worked on Parrot and for this reason the implementation details will not be disclosed on this thesis. The library only implements the Parrot message encoding for transmission and reception and it needs to be called on an already aligned and valid Parrot frame, the job of searching the packet on the RX stream is still performed by the packetUtils.c library. The higher level implementation of Parrot communication (so the definition of the various message IDs, the relative payload structure and the addresses of the CDHs and Singer) is directly done on the Parrot Task and for this reason the details will be

disclosed on section 6.3.4.1.

### 6.2.7 printf() implementation

The last building block that makes use of the UART regards the binding of the latter with the standard C library printf() function; this allows exploiting the powerful formatting capabilities of this function to create a debug console on the Access Port UART. Biding the printf() to the UART is quite easy and happens through the definition of the _io_putchar() function that is used by printf() to send characters on the serial line (stdio.h standard library needs to be included in the project), fig. 6.27 shows the code that does just that, as can be seen the character is sent to the UART through the UARTdriver.c sendDriver_UART() function.

```
int __io_putchar(int ch){
    sendDriver_UART(&huart2,(uint8_t *)&ch,1);
    return ch;
}
```

**Figure 6.27:** Binding of printf() to the AP UART through the definition of _io_putchar()

### 6.2.8 ADC driver

Communication with both AD7788 ADCs happens through a shared SPI interface with a data rate of 500 kbit/s.

#### 6.2.8.1 AD7788 interface

This ADC needs an high clock polarity and a second edge clock phase (CPOL=1, CPHA=1, refer to fig. 6.7). The chip has four interface registers:

- Communications register (write only), address '00': each communication must start with a write operation on this register, it contains the 2-bits address of the register that the user wants to read/write, one bit to identify if the operation is a read/write and some other mode flags (write enable bit, measurement channel selection, continuous/single read mode selection); while in idle (after startup or a conversion), the ADC expects a write operation on this register at the next SPI transaction (so the first write-mode byte sent on the SPI is automatically written here). The ADC can be reset by sending 32 consecutive '1's.

- Status register (read only), address '00': this contains status bits like the ADC data ready bit, the ADC error bit, an identifier bit to distinguish between an AD7788 and an AD7789 model and a currently selected channel bit. It shares its address with the communication register but the read/write bit of the latter is used as additional address bit to select between the two (since one is read only and the other write only).

- Mode register, address '01': this register is used to set the ADC to work in single conversion mode, continuous conversion mode or standby mode; it's also used to select the desired coding of the output (unipolar or bipolar, depending on whether the ADC is used in single ended or differential mode).

- Data register, address '11': this register is the only 16-bits wide register and contains

the conversion output.

Exactly describing the bit-by-bit content of these registers is out of the scope of this thesis, the complete description can be found in the ADC datasheet: [55]. In our application the ADC is used in single conversion mode, with the timing shown in fig. 6.28.

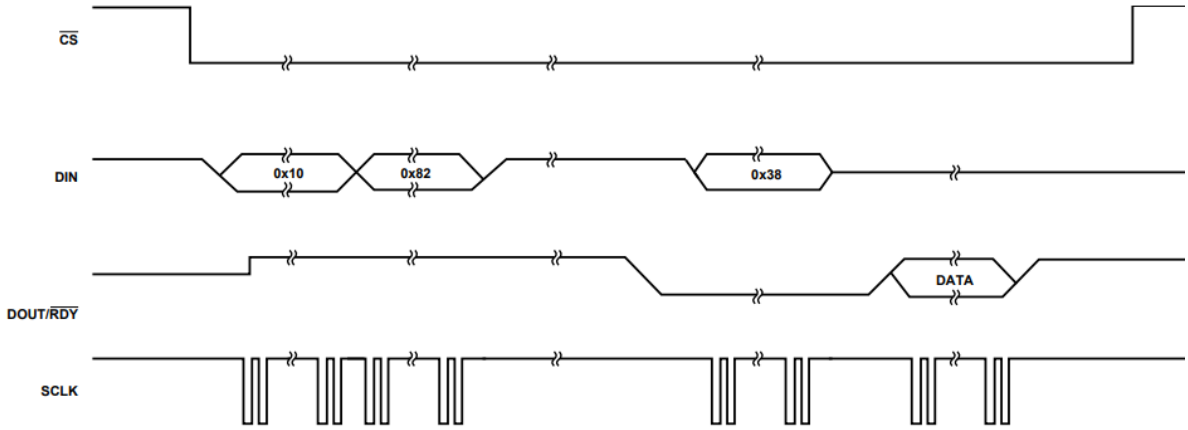



**Figure 6.28:** AD7788 single conversion mode timing diagram, picture from [55].

At first, 0x10 is written on the communications register to command a write operation on the mode register, then 0x86 is written on the mode register to set the conversion mode to single and the output coding to unipolar (in fig. 6.28 a value of 0x82 is shown but this would instead set the output coding to bipolar); the conversion starts and the chip select pin can go high again, in fig. 6.28 the chip select is left low because the DOUT(MISO) line can be used as conversion completed flag since it goes from high to low when the conversion is completed, in our case we needed to send the conversion start command to the other ADC over the same SPI line and for this reason the chip select is disabled during conversion and the conversion is considered as completed by keeping track of the elapsed time; at the end of conversion the 0x38 command will instruct the communications register that a read operation needs to be performed on the data register, whose content is subsequently shifted out by the ADC.

#### 6.2.8.2 ADC logical wiring

The detailed ADC logical wiring is shown in fig. 6.29, as can be seen the ADCs share an SPI bus in multi-drop configuration and the address lines of MUXes are shared so that the same channel is selected on both by a single set of four GPIOs.

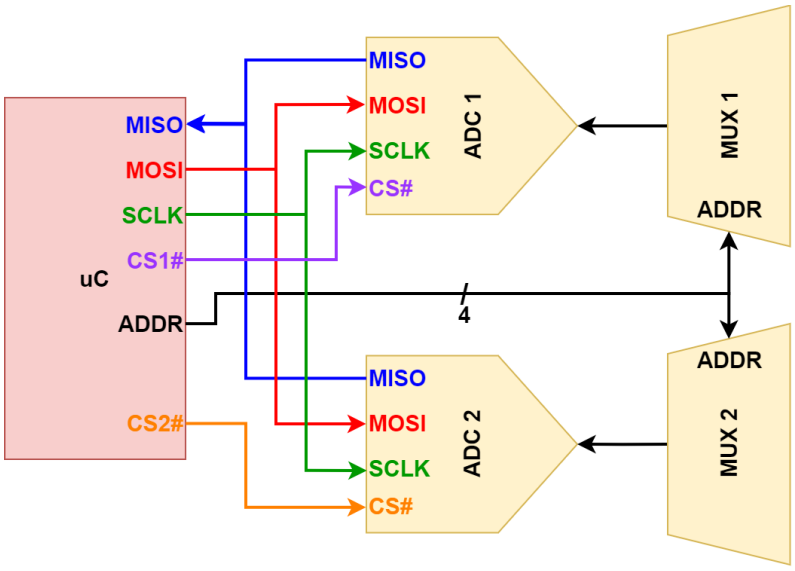



**Figure 6.29:** Logical wiring of the two ADCs.

### 6.2.8.3 Driver library

The ADC driver is implemented by the ADCsinger.c library, the driver outline can be seen in fig. 6.30. This library manages the MUX address and the presence of two ADCs transparently to the user, the conversion happens in parallel on both ADCs on the same MUX channel (as said the two MUXes share the address line). Three high level public functions are provided to the user: the adcInit() function resets both ADCs by sending the 32 '1' sequences; the adcRequestSample() sets the MUX channel to the one requested as function argument and then orders a single mode conversion to both ADCs; the adcReadSample() reads the conversion result from both ADCs and places them in a uint16_t array passed as argument. The user (task) needs to wait for the right amount of time for the conversion to complete (at least 60 ms) before reading the result from the ADC. The library could potentially be reconfigured to manage an arbitrary number of ADC chains.

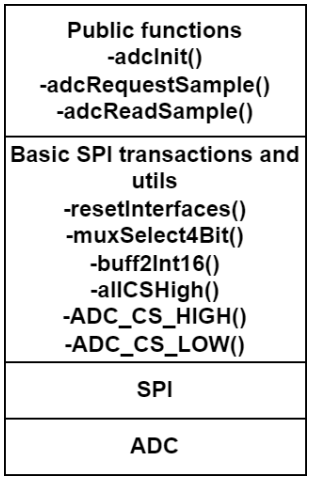



**Figure 6.30:** ADC driver outline.

## 6.2.9 MRAM driver and log system

The MRAM management is performed on two separate library layers: the lower layer is the device driver that implements the basic transactions like read and write operations and so is application-independent; the upper layer is the log system, consisting of a memory organization for logging purposes and a mechanism to keep track of the write position in memory (last written log packet), which is application-specific. SPI communication with the MRAM happens with a data rate of 4 Mbit/s.

#### 6.2.9.1 AS301604 interface

This device's SPI interface doesn't have a requirement on the clock polarity, in any case the bit sampling happens at the rising edge (CPOL=0 and CPHA=0 or CPOL=1 and CPHA=1 are both valid options, refer to fig. 6.7). It supports single/double/quadruple SPI modes, of which the single is the one that we used; it also supports Single Data Rate (SDR) or Double Data Rate (DDR) output modes, of which the single is the one that we used. The memory internal architecture can be seen in fig. 6.31

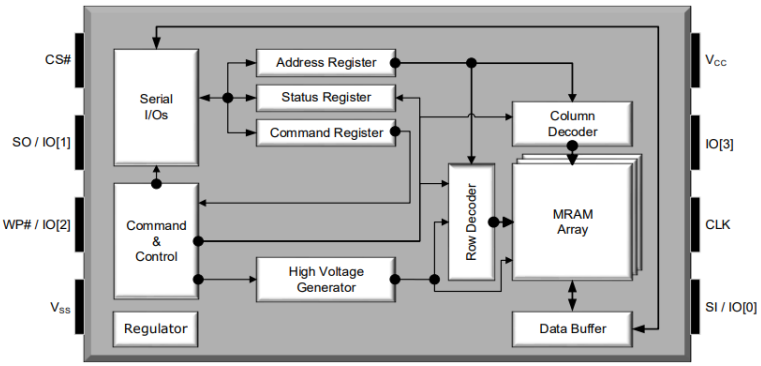



**Figure 6.31:** MRAM internal architecture, picture from [42].

The memory contains two arrays that are addressed with 24 bits of parallelism:

- the Main array having 2MB of storage, it doesn't have a specific alignment and read/write operations can be performed from any address and for any number of bytes; various write protection options are available that can cover fractions of the memory (negative powers of 2) or the entire array; it covers the address space 0x000000 - 0x1FFFFF;
- the Augmented Storage (AS) array having 256 bytes of storage, organized in 8 rows of 32 bytes each; each region can be individually write protected but beside that there is again no specific alignment for read/write operations; it can be addressed in the range 0x000000 to 0x0000FF.

The two arrays have superimposed address ranges, but these address spaces are independent and the array that gets accessed depends on the specific instruction used. Fig. 6.32 shows the two memory arrays and their address space.

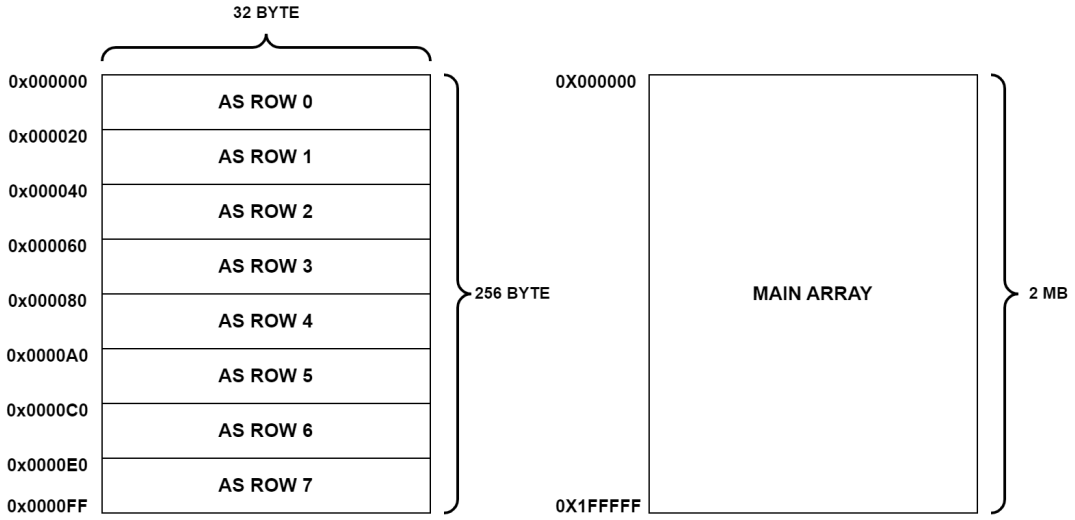



**Figure 6.32:** MRAM arrays and corresponding address space.

The device has a total of 8 interface registers, 7 of them are shown in fig. 6.33.

| Register Name | Address |
|---|---|
| Status Register | 0x000000h |
| Configuration Register 1 | 0x000002h |
| Configuration Register 2 | 0x000003h |
| Configuration Register 3 | 0x000004h |
| Configuration Register 4 | 0x000005h |
| Device Identification Register | 0x000030h |
| Unique Identification Register | 0x000040h |

**Figure 6.33:** MRAM interface registers, picture from [42].

Again the registers' address space is independent from the ones of main array and AS array and it's the specific instruction used to determine which address space is in use. The 8th register, the AS Array Protection Register, is not shown in fig. 6.33 because it cannot be accessed by address but only with a specific instruction; in general all registers have an instruction to access them directly besides the one to access them by address, using the latter is not convenient because it requires 8 clock cycles of latency before the register can be read, versus the 0 latency of reading them directly by instruction.
The memory registers can be written to set every aspect of its functionalities: SPI mode, memory and registers write protection, output drivers strength, memory read latency, memory write address wrapping and so on... describing each register with the associated bits would be of no use for the scope of this thesis, for a complete description the device datasheet can be consulted: [42].
Data transactions with the memory are always initiated with an instruction being sent, then depending on the specific instruction an address and/or data could be sent; the instructions are categorized resorting to a nomenclature with the form:
$< instruction > - < address > - < data >$
where the three fields are replaced with 0 if the field doesn't need to be present on the transaction and 1 otherwise (actually the 1 can also be 2 or 4 to represent the number of SPI lines used to transfer it in double or quadruple SPI modes, but in our case it can only be 0 or 1). In our application we only used 1-0-0, 1-0-1 and 1-1-1 (shown in fig. 6.34) instruction types, see fig. 6.35b for the list of used instructions.

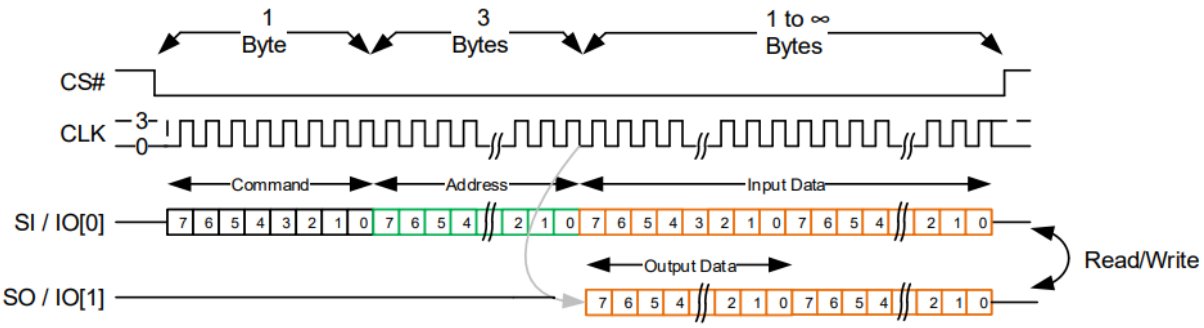
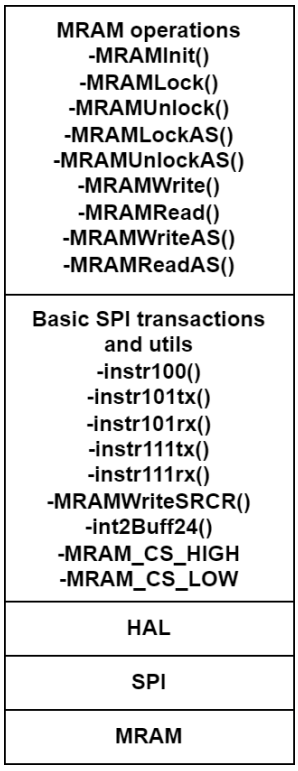



**Figure 6.34:** Timing of 1-1-1 instruction type, picture from [42].

### 6.2.9.2 Low level driver

The MRAM.c low level driver provides a basic set of functions to init the memory, enable/disable the write protection on the main or AS array and read/write them, the driver outline can be seen in fig. 6.35a




(a)

```
//instruction set
//only some of the overall instruction set is used
//on this library
//******* control
#define NOOP 0x00 //1-0-0 No operation
#define WREN 0x06 //1-0-0 Write enable
#define WRDI 0x04 //1-0-0 Write disable
#define DPDE 0xB9 //1-0-0 Enter deep power down
#define HBNE 0xBA //1-0-0 Enter hibernate
#define SRTE 0x66 //1-0-0 Software reset enable
#define SRST 0x99 //1-0-0 Software reset
#define DPDX 0xAB //1-0-0 Exit deep power down
//******* read register
#define RDSR 0x05 //1-0-1 Read status register
#define RDCX 0x46 //1-0-1 Read config registers 1-2-3-4
#define RDAP 0x14 //1-0-1 Read AS protection register
//******* write register
#define WRSR 0x01 //1-0-1 Write status register
#define WRCX 0x87 //1-0-1 Write config registers 1-2-3-4
#define WRAP 0x1A //1-0-1 Write AS protection register
//******* read or write main array *******
#define READ 0x03 //1-1-1 Read main array
#define WRTE 0x02 //1-1-1 Write main array
//******* augmented storage *******
#define RDAS 0x4B //1-1-1 Read AS array
#define WRAS 0x42 //1-1-1 Write AS array
```

(b)

**Figure 6.35:** 6.35a MRAM driver outline. 6.35b MRAM instruction set definition inside the driver.

The driver has a complete definition of the memory instruction set (fig. 6.35b) and of bit masks for every interface register. It's built on two distinct layers: the first implements the basic SPI transactions (1-0-0, 1-0-1 and 1-1-1 both for read and write) by means of the HAL SPI libraries, the second layer makes use of these basic transactions to provide

the upper layers with:

- an init function to be called after system reset, which writes the default value on every register and locks the memory afterwards;
- lock and unlock functions, for both the main and AS arrays, with effect on the entire main array or selectively for the AS array (every row can be locked independently, for security reasons unlocking one row will first lock all the others);
- write and read functions, for both the main and AS array; the main array can be written from any valid address and for any number of bytes, the AS array can be written on any row, starting from the first byte and for a maximum of 32 bytes (this is ensured by the already mentioned security lock of all the other rows).

This basic set of functions gives the user the possibility of using the memory easily and without caring on low level details, the default registers' values can eventually be redefined since they are declared as macros in an user-configurable portion of the code.

#### 6.2.9.3 Memory organization and log system

The memory organization is defined on the MRAMsinger.c library, it consists of a very simple log system that is appointed to store every minute the samples coming from the various Singer data sources and allow their retrieval; the log system makes use of both the main array and the Augmented Storage array, as shown in fig. 6.36.

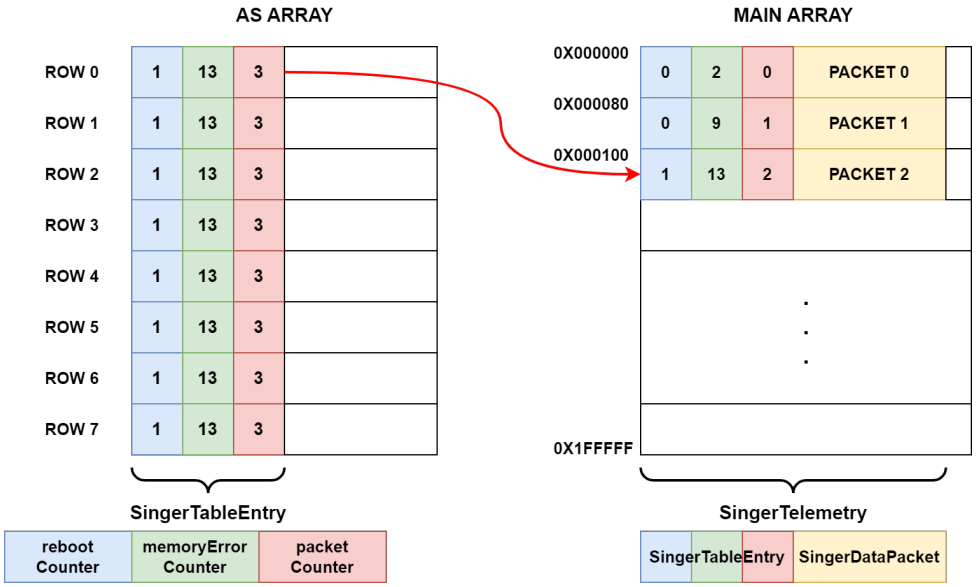



**Figure 6.36:** MRAM log system organization.

- The AS array contains the log system table, which consists of the SingerTableEntry structure (fig. 6.37):

```
//type of memory table fields
typedef uint32_t tablefield_t;
//memory table structure
struct SingerTableEntry
{
    tablefield_t rebootCounter;
    tablefield_t memoryErrorCounter;
    tablefield_t packetCounter;
} __attribute__((packed));
```

**Figure 6.37:** SingerTableEntry structure.

this contains three fields, the reboot counter that is updated after every system reboot, the memory error counter that is updated after every detected flash or RAM error (not implemented) and the packet counter, which is the most important one because it keeps track of the last written packet and is updated every time a packet is written on the main array, this field acts as pointer of the log system head and is everything the system needs in our use case (a data read session from memory will always start from the last written packet and proceed backwards for 1 or more packets); the table is written in eight redundant copies inside the AS, one per each row, a voting algorithms is applied while reading the table (since the number of copies is even, in case of tie the '0' wins the voting) to determine its value. Some space per each row of the AS will obviously remain empty. The 32 bit packetCounter will overflow in more than 8000 years with a sampling period of 1 minute.

- The Main array contains the sample packets, defined in the SingerTelemetry structure (fig. 6.38):

```
//singer telemetry packet
struct SingerTelemetry
{
    struct SingerTableEntry singer_table_entry;
    struct SingerDataPacket singer_memory_packet;
} __attribute__((packed));
```

**Figure 6.38:** SingerTelemetry structure.

as can be seen, this structure contains again the SingerTableEntry structure, in this case the stored table values represent the table content at the time the packet is written on memory, so the main array will contain a log of packets with an increasing packetCounter and an history of the other two table values, as can be seen in fig. 6.36; the total size of a SingerTelemetry packet is 110B. the packet counter uniquely identifies the position of a packet in memory by the formula **ADDRESS = packetCounter*size(packet row) % size(main array)**, for this formula to be true the packet row size needs to be a divisor of the main array size to guarantee the memory alignment after an address overflow, in this case a 128 byte row was chosen (the lowest divisor of the main array size that could contain the SingerTelemetry structure) so even in this case some space per each row remains empty, the packetCounter on the AS table will be incremented after each packet is written and consequently the value on the table will always have a value that is one unit greater than the last packet (basically the packetCounter stored in the main array corresponds to the index on an array while the one stored on the table corresponds to the array size).
The other field in the SingerTelemetry structure is the SingerDataPacket structure, which

actually contains the sampled values and has the format seen in fig. 6.39a

```c
//singer data packet
struct SingerDataPacket
{
    uint32_t timestamp;
    uint16_t CDH_temperature;
    uint16_t gyroscope[3];
    uint16_t magnetometer[3];
    uint16_t data[ADC_CHANN_N * ADC_N];
#ifdef PARROT_SENSORS
    struct ParrotSensorsMemory parrotSens;
#endif
} __attribute__((packed));
```

**(a)**

```c
//parrot sensors data as stored on memory
struct ParrotSensorsMemory
{
    uint16_t eab_temp_sensor_cdh1;
    uint16_t eab_temp_sensor_cdh2;
    uint16_t bat_temp;
    uint16_t bat_disch_volt;
    uint16_t bat_disch_curr;
    uint16_t bat_charge_curr;
    uint8_t vehicleMode_cdh1;
    uint8_t curr_opmode_cdh1;
    uint8_t vehicleMode_cdh2;
    uint8_t curr_opmode_cdh2;
} __attribute__((packed));
```

**(b)**

**Figure 6.39:** 6.39a SingerDataPacket structure. 6.39b ParrotSensorsMemory structure.

The data packet structure contains a timestamp, the gyroscope and magnetometer data from IMU, the 32 temperatures from the ADC and an additional field that stores the temperature measured internally by the microcontroller embedded temperature sensor (as a placeholder, not implemented), it also contains the values from sensors of the Parrot subsystem, which are listed in fig. 6.39b; the Parrot sensors include temperature sensors of the CDHs, battery temperature, charge/discharge current and voltage, vehicle mode and operative mode; as can be seen, the parrot sensors structure is asymmetrical (battery telemetry is only gathered by the CDH 1).

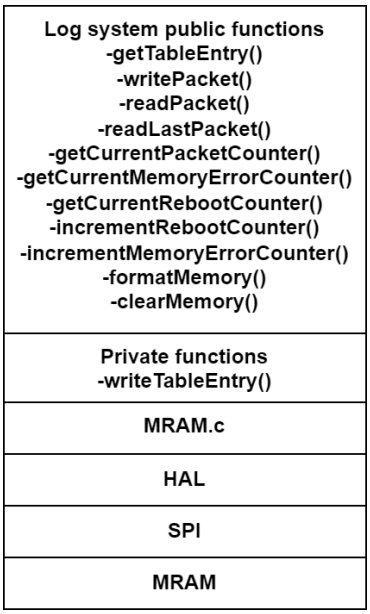


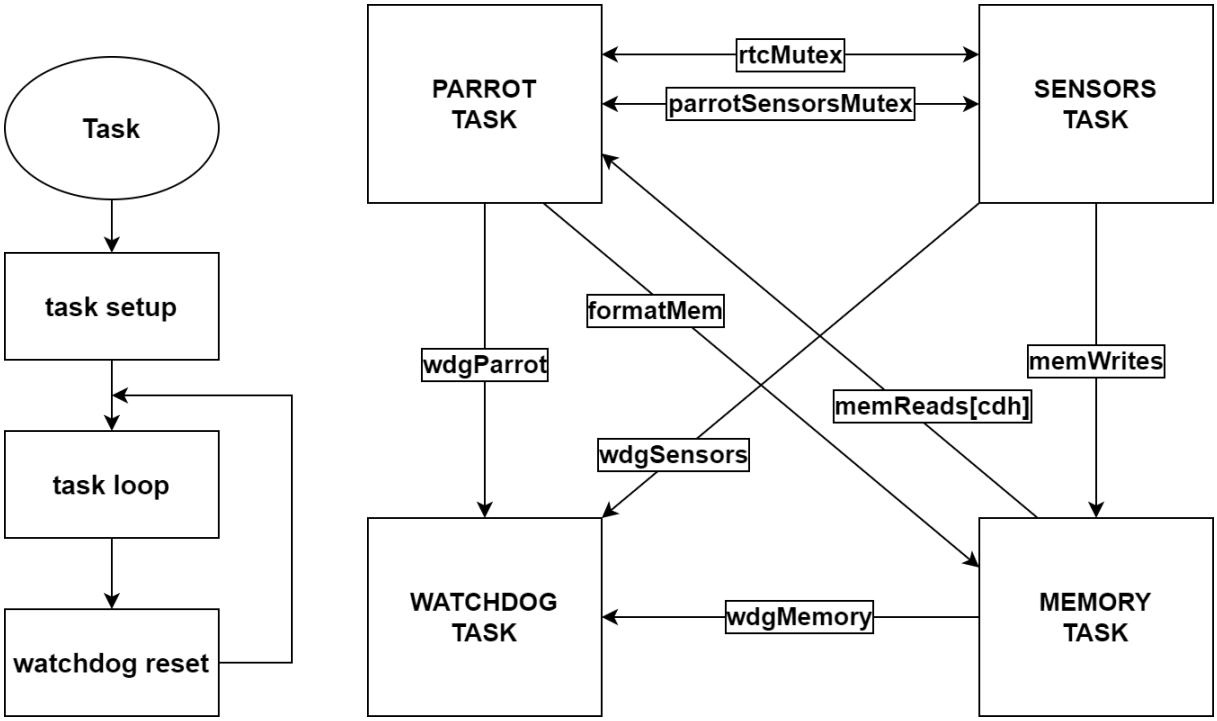

**Figure 6.40:** Log system library outline.

The log system is implemented by the MRAMsinger.c library (of which the outline can be seen in fig. 6.40), this library implements the basic set of functions needed to manage the memory log system, like to read the table and increment single table fields (excluding the packetCounter that is only incremented in the writePacket() function to grant consistency), write new data and read the last packet or one corresponding to a given counter, it also implements functions to initialize the log system and format it: formatMemory() will reset the whole table, while clearMemory() will only reset the packetCounter without touching the other fields.

## 6.3 High level software

The high level software is composed of four tasks:

- Sensors task: gathers data coming from board sensors and parrot sensors, packing them in a SingerDataPacket struct (fig. 6.39a) and sending the packet to the memory task for writing.
- Memory task: manages the memory, dispatching write and read requests coming from the other tasks.
- Parrot task: manages communication with the CDHs, encoding and decoding the messages and implementing the communication state machines.
- Watchdog task: manages the microcontroller hardware watchdog, gathering watchdog flags coming from the other tasks or by checking hardware peripherals state and resetting the watchdog counter if everything is correctly working, it also performs a routine self-reset of the microcontroller.




(a) (b)

**Figure 6.41:** 6.41a Firmware task general form. 6.41b Firmware tasks and data flow paths.

All tasks have the general setup-loop format that is seen in fig. 6.41a, the task loop is executed entirely on each task repetition period, this could potentially be a problem because usually there's the need to wait for an event to take place an this would starve the other tasks; for this reason all tasks (except the watchdog one) are implemented as state machines, which preserve the state between repetitions and only check in non-blocking

manner (with a given timeout) if the waited event took place at each repetition. Fig. 6.41b shows the data flow paths between tasks, inter-task communication is explained more in details in section 6.3.1.

### 6.3.1 Inter-task communication

The high level software analysis will start by the inter-task communication mechanisms because it could help better understanding the following sections about each specific task by defining their I/O interfacing with the rest of the firmware, also these communication mechanisms involve multiple tasks and it would be counterproductive and dispersive to present them in the description of single tasks.

#### 6.3.1.1 Telemetry exchange

Telemetry exchange is done resorting to the TelemetryProdCons structure (fig. 6.42).

```
//data structure for monodirectional telemetry exchange between tasks
struct TelemetryProdCons
{
    uint32_t drdy;                          //data request/ready handshake flag
    struct SingerTelemetry singer_telemetry;//exchanged telemetry packet
};
```

**Figure 6.42:** TelemetryProdCons structure.

This structure contains the SingerTelemetry structure that holds the telemetry packet (fig. 6.38), together with an handshake flag; the usage of this structure for telemetry exchange follows a quite simple handshake mechanism and can only be done monodirectionally:

- the task that requests the exchange sets the handshake flag, this task is the only one that can set the flag and can never reset it;
- the task that serves the request waits until the flag is set, at this point it serves the request (by reading or writing the data from/to the structure) and resets the flag, this task is the only one that can reset the flag and can never set it.

Since on STM32 the load/store operations on 32 bit variables (like the handshake flag) are atomic, this simple mechanism allows exchanging data monodirectionally in thread-safe manner; it's also evident that this mechanism is risky as it can cause a deadlock if not properly handled by the two tasks and so needs caution on its implementation on both involved parts. This structure is the type of the memWrites (used to write sensors data on memory) and memReads[ ] (used to read data from memory to send it to Parrot) variables of fig. 6.41b, the memReads is actually a two-elements array because the two CDHs can be served independently, also in read mode the drdy flag can assume two possible values: a value of 1 stands for single telemetry request (a single packet needs to be read) while a value of 2 stands for a downlink request (the Memory task must start reading from the last written packet going backwards, so it must remember the downlink session position in memory); fig. 6.43 shows the sequence diagram of the data exchange between Sensors, Memory and Parrot Tasks.

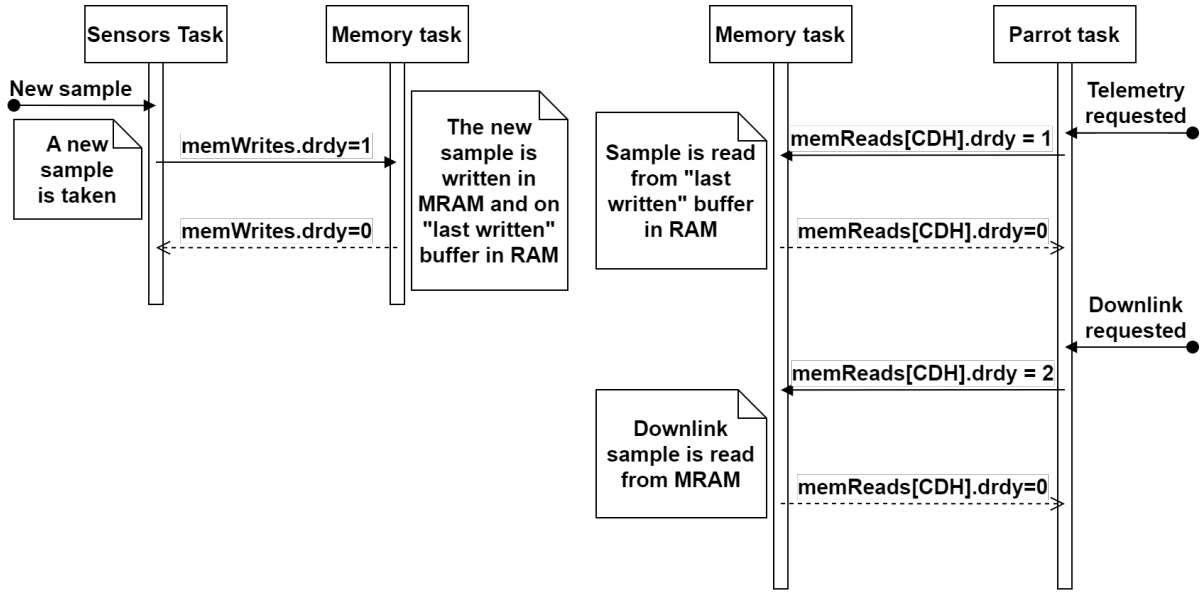



**Figure 6.43:** Sequence diagram of telemetry exchanges.

#### 6.3.1.2 Monodirectional requests

The same exact concept of the handshake flags of telemetry exchange applies to simple requests without exchange of data, referring to fig. 6.41:

- wdgParrot, wdgMemory and wdgSensors are three 32 bits flags that are used by the corresponding tasks to signal the Watchdog task that they are alive an running, the latter waits until all the flags have been set (together with other conditions) to reload the watchdog counter and reset the flags;

- formatMem is also a 32 bit flag that the Parrot task uses to request a memory format to the Memory task, which will serve the request and reset the flag.

Also in this case the caveat is that the requesting tasks can only set the flag and the serving task can only reset it, the difference between these flags and the ones for telemetry exchange is that in this case the flow of information is truly monodirectional: the requesting tasks have no intereset of checking that the request has been served and so that the flag was reset, they can potentially set again the flag even when the request has not been yet served (and this for example happens continuously for the watchdog flags).

#### 6.3.1.3 Mutual exclusive access

There are some resources that need to be accessed on mutual exclusive manner, in this case a mutex was used to lock the resource by the task that wants to access it, referring to fig. 6.41:

- rtcMutex is used to share access to the microcontroller Real Time Clock (RTC) between Parrot task (who updates the time coming from the CDH) and Sensors task (who reads the timestamp to packet it with data);

- parrotSensorsMutex is used to share access to a ParrotSensorsMemory structure (fig.

6.39b) where Parrot sensors values are stored by Parrot task and accessed by Sensors task during sampling.

### 6.3.2 Sensors task

The Sensors task is responsible of gathering data from ADC, IMU, RTC and Parrot sensors, this data is packed in a SingerDataPacket struct (inside the memWrites struct already mentioned in section 6.3.1.1) and sent to the memory task, which will store it (actually only the SingerDataPacket member of the SingerTelemetry structure of memWrites is used in this case).
The sensors task needs to get access to the RTC in order to get the current timestamp, the timestamp is seen as another scientific information that can be useful for data interpretation and has no functional purpose: the task sampling period is computed by the system tick counter (through HAL_GetTick() function) because the RTC timestamp is considered as unreliable (the latter is updated from Parrot at every received message). The access to the RTC, that is shared with the Parrot task, is done in mutual exclusive manner resorting to a mutex (as explained in section 6.3.1.3).
The Sensors task implements a state machine, the possible states can be seen in fig. 6.44;

```
    // acquisition state variable
    typedef enum
    {
        IDLE_SENSORS,        // task is idle, waiting for new sampling trigger
#ifdef IMU_PRESENT
        WAITING_IMU,         // task is waiting imu sample
#endif
        SAMPLING_ADC,        // task is requesting adc sample
        WAITING_ADC,         // task is waiting adc sample
#ifdef PARROT_SENSORS
        WAITING_PARROTSENS, //task is waiting to access parrot sensors mutex
#endif
        REQUESTING_WRITE     // task is requesting data write on memory
    } acquisition_state;
```

**Figure 6.44:** Sensors task state machine states.

The task is normally in IDLE state, as soon as the sampling is triggered (every 1 minute) the sampling loop starts by taking the timestamp from the RTC, then waiting for the next IMU data (WAITING_IMU state), then the 32 ADC channels (SAMPLING_ADC and WAITING_ADC), finally the Parrot sensors are read and all this data is packed and sent to the Memory task for storage, a new sampling can be always triggered in any state, even if the current sampling has not been completed (but this should never happen), in that case the ongoing sampling is aborted; fig. 6.45 shows the overall Sensors task flow chart and the sampling state machine.

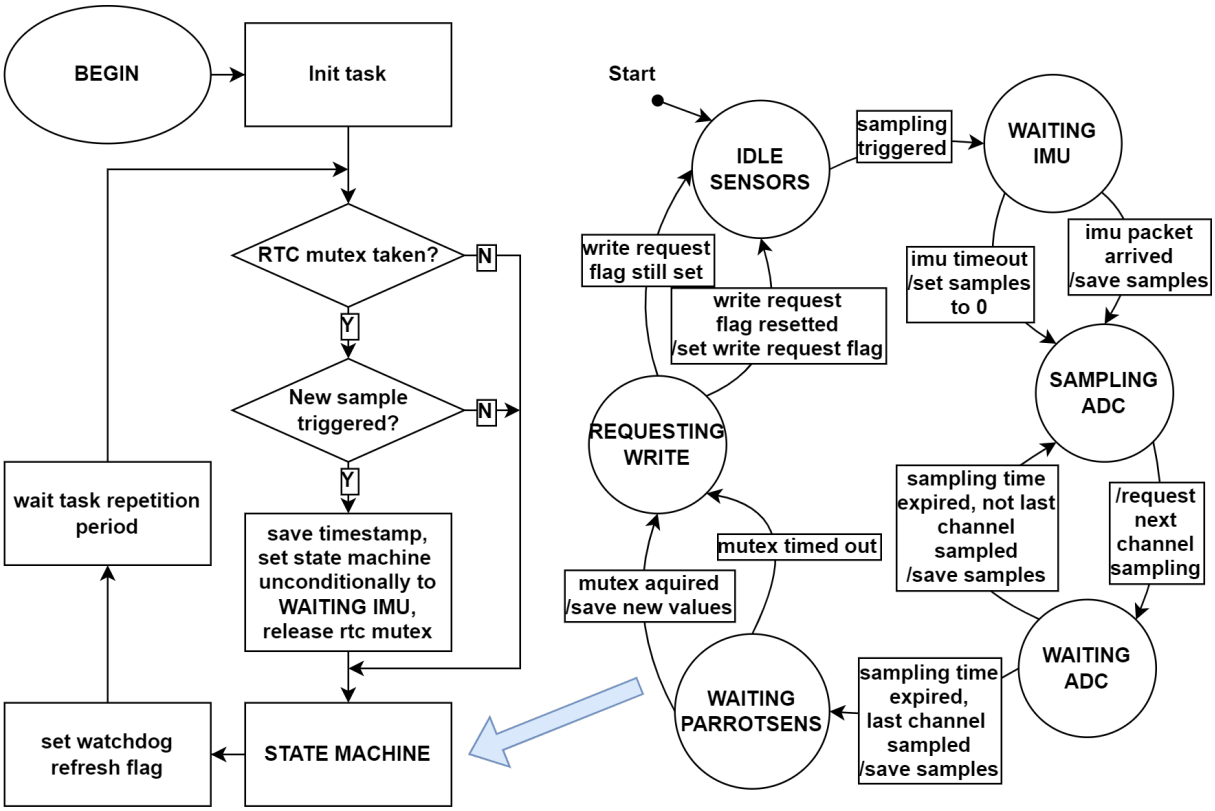



**Figure 6.45:** Sensors task flow chart and sampling state machine.

### 6.3.3 Memory task

The Memory task is responsible of serving write/read memory requests coming from the Sensors task or Parrot task, following the sequence diagram of fig. 6.43; in this case the task state machine is not explicit and consists of a series of flag variables that the task uses to remember the type of session during read (single telemetry or downlink); a write request is performed through the memWrites structure as explained in section 6.3.1.1 by setting the drdy flag with a value of 1, when this flag instead is set to 2 a downlink session starts, in this case the Memory task will remember its position in memory until the downlink session ends; a downlink session consists on sending a number of packets starting from the last written and going backwards, it ends whenever a single telemetry request is issued or the whole memory has been read.
Fig. 6.46 shows the flow chart of the Memory task. As can be seen in the flow chart, during a single telemetry request the packet is not read from MRAM but from a buffer in RAM, this buffer is written every time a new sample is stored in MRAM and ensures that at least the single telemetry can be provided in case the MRAM fails; actually a similar buffer (maybe with a reduced number of packets) could have been implemented also for downlink but was not done for design time and complexity issues.

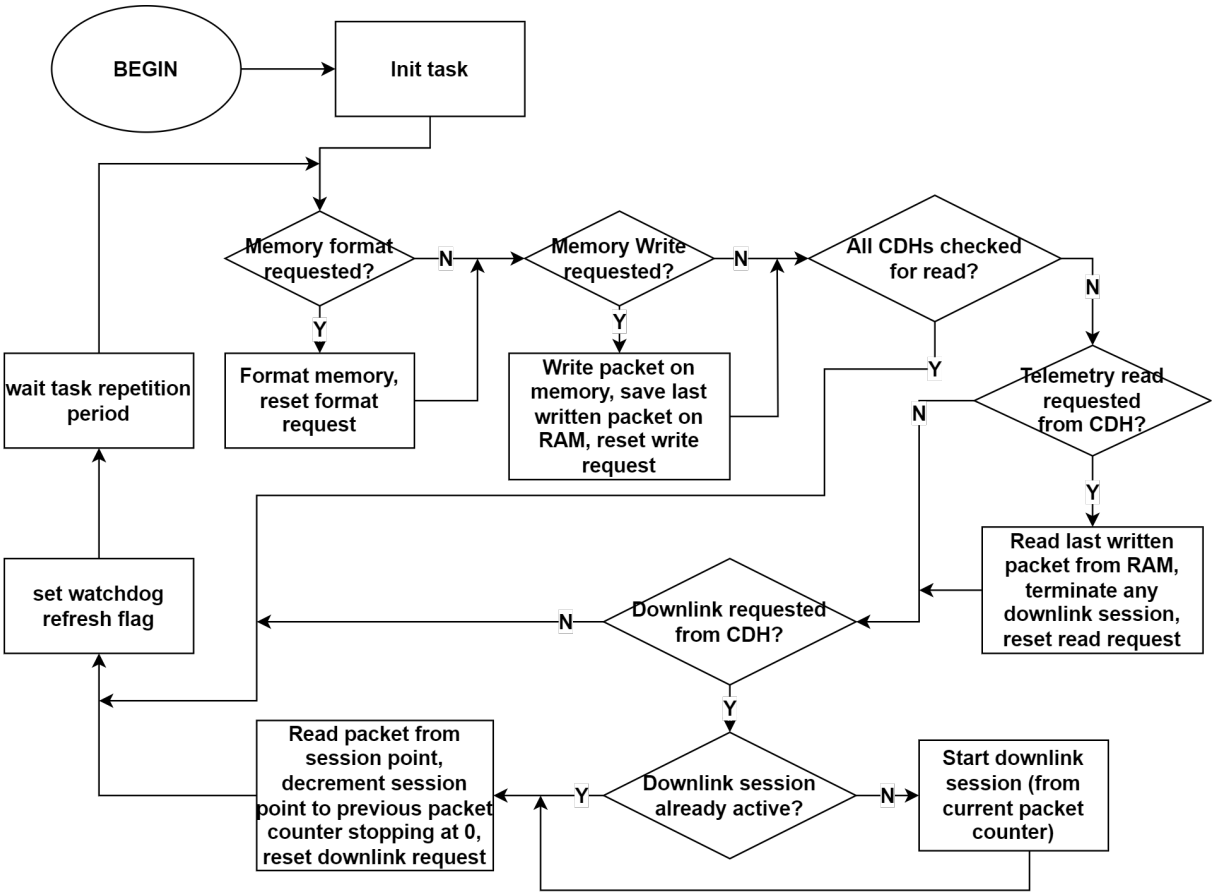



**Figure 6.46:** Memory task flow chart.

### 6.3.4 Parrot task

The Parrot task is responsible of managing the communication with Parrot based on messages with the format described in section 6.2.6.1.

#### 6.3.4.1 Available Parrot messages

While the SerialComm.c library implements the lower layer of communication with Parrot, by providing the functions to encode and decode the parrot message format explained in section 6.2.6.1, the available message IDs and the communication logic state machine is implemented at task level. Tab. 6.1 summarizes the possible Parrot message IDs and corresponding response from Singer; the source and destination fields can contain the two CDHs identifiers (ASCII "P1"=0x5031 or "P2"=0x5032) or the Singer identifier (ASCII "SG"=0x5347), the same logic applies to message identifiers (the table contains the ASCII representation).

**Table 6.1:** List of Parrot/Singer messages.

Parrot to Singer.

| src. | dest. | m. ID | payload | meaning |
|---|---|---|---|---|
| “P1”/”P2” | “SG” | “FM” | - | Format memory. |
| “P1”/”P2” | “SG” | “ST” | <SingerRequest> | Single telemetry request. |
| “P1”/”P2” | “SG” | “DT” | <SingerRequest> | Downlink telemetry request. |
| “P1”/”P2” | “SG” | “SU” | - | Downlink shut-up message. |

Singer to Parrot.

| src. | dest. | m. ID | payload | meaning |
|---|---|---|---|---|
| “SG” | “P1”/”P2” | “SR” | <SingerTelemetry> | Single telemetry response. |
| “SG” | “P1”/”P2” | “DR” | <SingerTelemetry> | Downlink telemetry response. |

Fig. 6.47 shows possible sequence diagrams of messages exchange between Parrot and Singer.

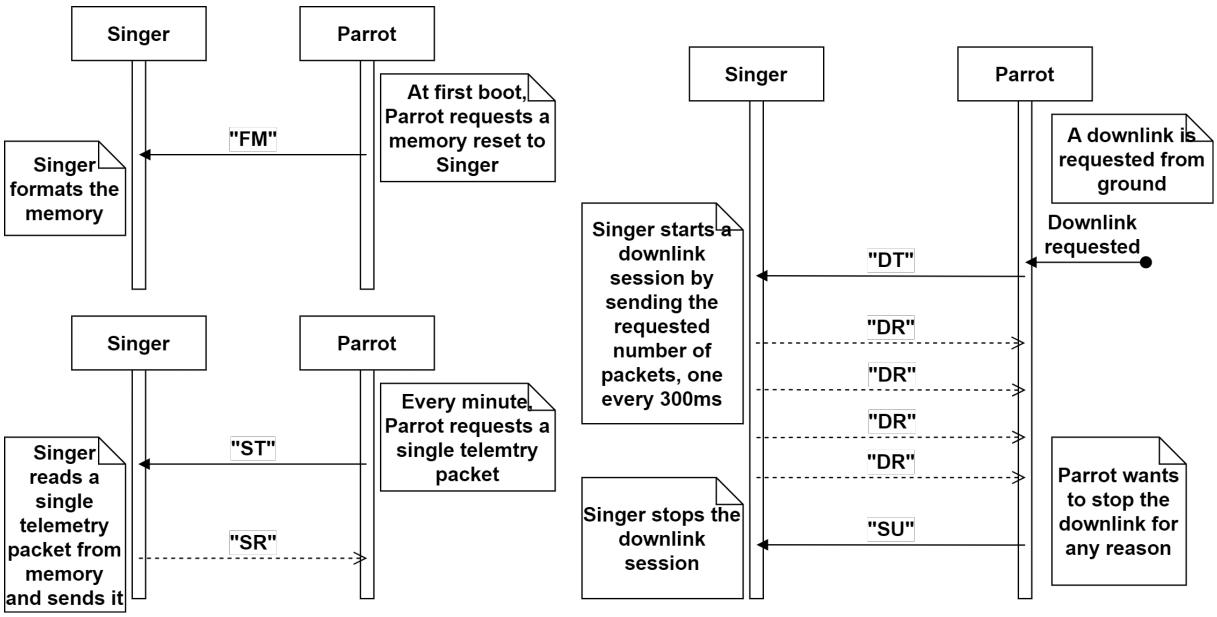



**Figure 6.47:** Parrot protocol messages exchange.

The SingerRequest struct that Parrot sends as payload can be seen in fig. 6.48;

```
    //request struct coming from parrot
    struct SingerRequest
    {
        uint32_t date;
#ifdef PARROT_SENSORS
        struct ParrotSensorsSerial parrotSensorsData;
#endif
        uint8_t number_of_packets;
    } __attribute__((packed));
```

**Figure 6.48:** SingerRequest structure.

this structure contains the current timestamp under the name of “date”, the data from

Parrot sensors and the requested number of packets (which is used only for downlink requests); the Parrot sensors data is inside the ParrotSensorsSerial struct (fig. 6.49).

```
//parrot sensors data received from CDH
struct ParrotSensorsSerial
{
uint16_t eab_temp_sensor;
uint16_t bat_temp;
uint16_t bat_disch_volt;
uint16_t bat_disch_curr;
uint16_t bat_charge_curr;
uint8_t vehicleMode;
uint8_t curr_opmode;
} __attribute__((packed));
```

**Figure 6.49:** ParrotSensorsSerial structure.

The ParrotSensorsSerial structure has a different format with respect to the ParrotSensorsMemory which is stored on memory (fig. 6.39b) and is identical for both CDHs; the Parrot task will then take the useful sensors fields from both CDHs and store them in the mutex protected ParrotSensorsMemory structure that was already presented in section 6.3.1.3, from which the Sensors task will sample the values every minute.

### 6.3.4.2 Task code structure

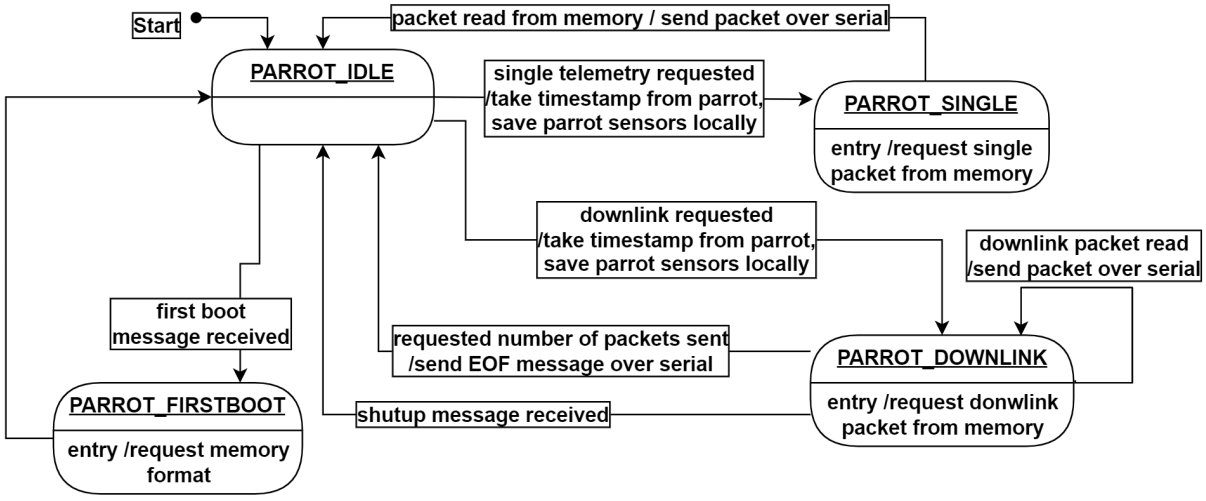



**Figure 6.50:** Parrot task state machine.

The Parrot task is the most complex one:

- it implements the reception of the bytes stream from the UARTdriver.c driver and buffering of the latter resorting to the buffUtils.c buffer library, it then makes use of the packetUtils.c library with its searchPacketAdvance() function that allows searching for a candidate Parrot message (from the message header to the closer) inside the reception buffer; finally it decodes the candidate messages with the SerialComm.c library;

- it implements the state machine that tracks the current communication state (fig. 6.50), the state machine takes as input the received messages and implements the exchange of requests with the Memory task as described in section 6.3.1.1; it also encodes the messages to each CDH with the SerialComm.c library and sends them.

The task executes two separate and (almost) identical state machines, one per each CDH; the only difference between the two state machines is the code that collects Parrot sensors and places them in the mutex protected structure to be exchanged with Sensors task, since one CDH sends battery telemetry and the other doesn't.
The high level task flow chart can be seen in fig. 6.51

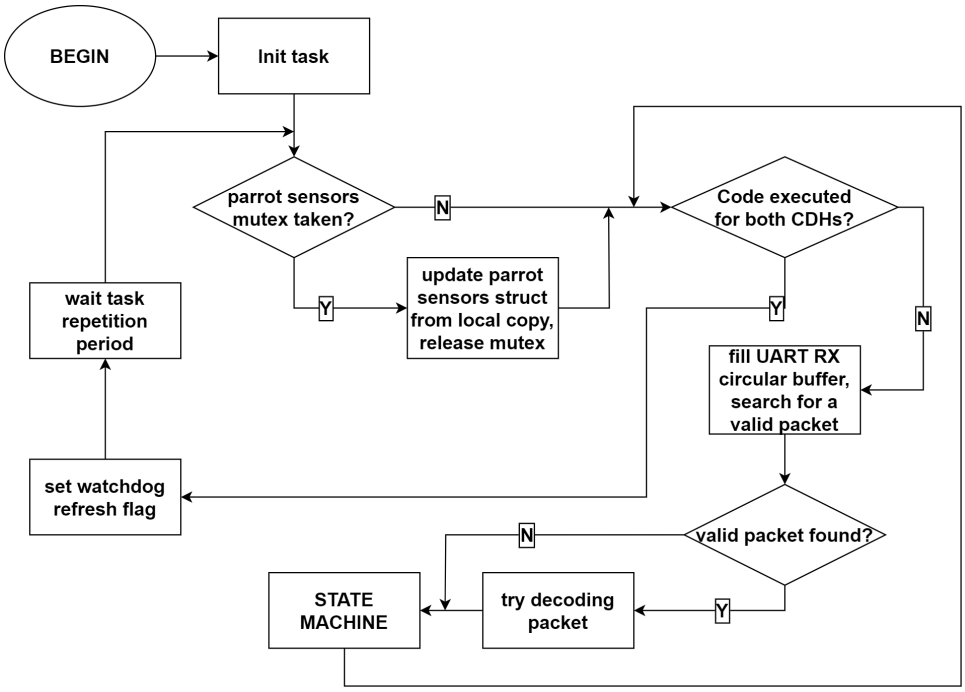



**Figure 6.51:** Parrot task flow chart.

### 6.3.5 Watchdog task

The watchdog task is responsible of managing the microcontroller hardware watchdog, its structure is very simple and can be easily explained as a list of performed checks:

- the task checks that none of the UART peripherals is in error state by calling the HAL_UART_GetError() function on all of them;
- the task checks that none of the SPI peripherals is in error state by calling the HAL_SPI_GetError() function on all of them;
- the task checks that the RTC peripheral is not in error by calling the HAL_RTC_Getstate() on it and verifying that the return is different from HAL_RTC_STATE_ERROR;
- the task checks that all the watchdog reset flags explained in section 6.3.1.2 have been set by the corresponding task;

If all these conditions are verified, the task resets the watchdog counter and then resets all the watchdog reset flags, otherwise the counter is not reset and it will expire in around 4 seconds.

The Watchdog task will also periodically reset the system every 12 hours to refresh the correct firmware states and frequently scan the boot portion of the code in memory to allow its correction through the error correction algorithm embedded in the microcontroller.

### 6.3.6 Tasks timeout management

The whole firmware implements a hierarchical task timeout management system which is based on the concept of "remaining time"; each task has an assigned timeout which limits the loop execution time at each repetition, this is used to compute the timeout of each call to low level functions, which apply the same logic at all levels down to HAL drivers timeouts, as seen in fig. 6.52

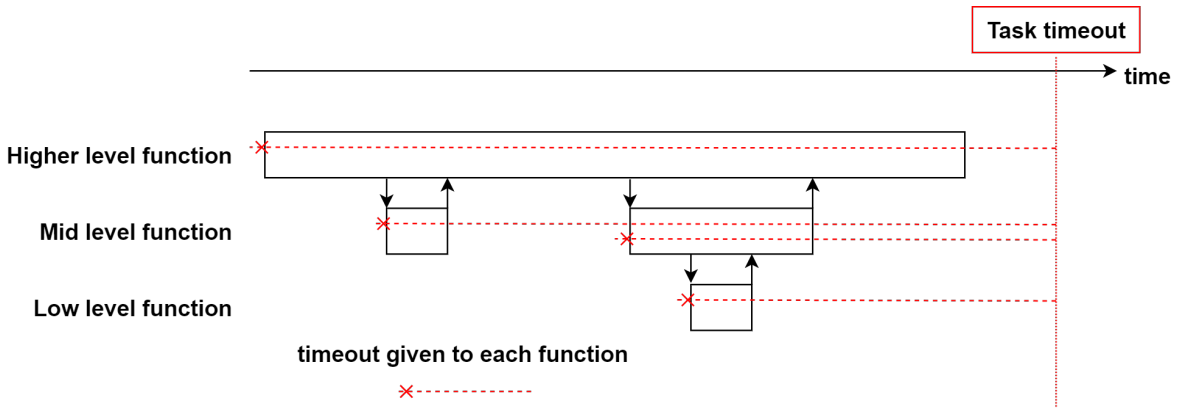



**Figure 6.52:** Hierarchical timeout example.

Each function saves the execution start tick by calling HAL_GetTick() at the beginning, then if a call to a lower level function is needed, it computes the timeout for the lower level function by subtracting the time elapsed from the start to the timeout given to it by the higher level function. This way an hard upper bound is given to the task whose loop execution time cannot span over its given timeout. If the bound is reached any low level function from that point gets a 0 ms timeout and the task quickly terminates its execution.
This mechanism can be exploited in various ways: one way is to give an infinite timeout to the task (HAL_MAX_DELAY) and define a second soft timeout which in case it gets passed will perform some error routine (that can also be a direct reset of the microcontroller), this ensures that even if the task spans over its bound it will complete all operations (under the limits given by the hardware watchdog) and thus is the safer option; a second options, the one that we adopted, is simpler and could be applied because the system doesn't perform vital duties on the spacecraft: the defined task execution bound is directly given as task timeout and that means that the loop execution is abruptly interrupted if the timeout is reached, nothing is done in that case and the timeout has the only purpose of avoid the task blocking and guarantee that the others can continue to be executed normally; it's necessary to give enough margin to the task to ensure that this bound is practically never reached.
A library was set up which defines these timeout management functions and which in debug mode can output a message on the console if a bound is broken.

### 6.3.7 Firmware compile modes

The firmware has a global configuration file which contains macro definitions that can be commented/uncommented in order to change the firmware compile configuration. Some examples of these macros could be spotted in fig. 6.39a, fig. 6.44 and fig. 6.48, which in that case could disable the code associated to IMU (which is not soldered on some boards) or Parrot sensors.

#### 6.3.7.1 Access Port debug console

The Access Port serial console is a quite simple output-only serial line that was useful during testing and debug phases, an example of output is shown in fig. 6.53.

```
 _____________________________________
|  SSSSSS S SS    S  SSSSSS SSSSSSS SSSSSS  |
| S       S S S   S S       S       S     S |
|  SSSSS  S S  S  S S   SSS SSS     SSSSSS  |
|       S S S   S S S     S S       S    S  |
| SSSSSS  S S    SS  SSSSSS SSSSSSS S     S |
|_____________________________________|
<initTasks> Current RTC timestamp: 3155766773 , Date: 2070-01-01, 01:52:53
<memoryTask_function> Current MRAM table:
<memoryTask_function>-----------------------------------
<memoryTask_function> Packet counter:       1484
<memoryTask_function> Memory error counter: 0
<memoryTask_function> Reboot counter:       22
<memoryTask_function>-----------------------------------
<sensorsTask_function> IMU correctly configured
<parrotTask_function> Received timestamp from cdh 1 : 6778 Date: 2070-01-01, 01:52:58
<parrotTask_function> Timestamp updated
<parrotTask_function> Requested telemetry from 9a9a with id c770
<memoryTask_function> Reading single packet telemetry for cdh 1
<parrotTask_function> Sending last packet, current singer packet counter 1484
<sensorsTask_function> New sample triggered
<sensorsTask_function> Timestamp: 3155766784 Date: 2070-01-01, 01:53:04
<sensorsTask_function> received IMU packet
<sensorsTask_function> all ADC channel sampled
<sensorsTask_function> Parrot sensors acquired
<sensorsTask_function> Sampling completed, write requested
<memoryTask_function> Written packet 1484
```

**Figure 6.53:** Access Port debug console.

Each task and some drivers write output messages through the printf() function implemented as described in section 6.2.7, the code associated with debug messages can be completely or selectively eliminated for some files or the whole firmware through the already mentioned global configuration file.

#### 6.3.7.2 Time measurement mode

Another configuration macro can enable time measurement mode, in this mode all the output from tasks is disabled and the maximum task loop execution time is computed (with granularity of 1 ms). Fig. 6.54 shows the output of time measurement mode.

```
TIME MEASUREMENT MODE
Maximum task execution times (ms) will be printed if
a new maximum execution time is registered

Legend:
PT:parrot task
ST:sensor task
MT:memory task
WT:watchdog task

PT      ST      MT      WT
0       0       0       0
        1
                        1
                2
        2
2
                        2
```

**Figure 6.54:** Time measurement mode output.

In this mode, a table is printed with each column corresponding to a different tasks, each task is compiled with an additional little piece of code that measures the time to execute the loop at every repetition, if a new maximum time is measured it gets printed and so the last value written on the bottom of the column is the maximum measured execution time; the code is left running for a reasonable amount of time while trying to heavy load it with Parrot requests to test a worst case scenario. The results are used to define the task repetition timeouts discussed in section 6.3.6 and 6.3.8.

### 6.3.8 Final firmware configuration

Tab. 6.2 summarized the configuration for each task in terms of stack size, priority, timing configuration (task repetition period, loop timeout and corresponding CPU usage) and execution time measured in time measurement mode.

**Table 6.2:** Firmware configuration.

| Task | Stack | Priority | TRP | Timeout | CPU use | Max. execution time |
|---|---|---|---|---|---|---|
| Memory | 1024 B | High | 50 ms | 10 ms | 20% | 2 ms |
| Sensors | 1024 B | AboveNormal | 100 ms | 10 ms | 10% | 2 ms |
| Parrot | 2048 B | Normal | 300 ms | 10 ms | 3.3% | 2 ms |
| Watchdog | 256 B | BelowNormal | 500 ms | 10 ms | 2% | 2 ms |

From the table the maximum allocated CPU for tasks is 35.3%, at this percentage we must add the worst case CPU time allocated to UART ISR that, as computed in section 6.2.3.1, is around 22%, yielding a worst case total CPU usage of 57.3% with the remaining 42.7% that can be used by the OS. The results from time measurement mode instead highlight that the actual CPU usage of task is way lower than the maximum, around 7%.
The firmware memory usage can be seen in fig. 6.55, the flash is filled with 61036 bytes (text+data) which means around 12% of the available, the RAM usage for variables is 44024 (data+bss) which means 27% of the available and 117 KB remaining for the stack; it's clear that the memory footprint is quite low and this despite the fact that the firmware memory usage is not optimized at all and so it could potentially be further reduced by a fair amount.

```
arm-none-eabi-size   singer.elf
   text    data     bss     dec     hex filename
  60276     760   43264  104300   1976c singer.elf
```

**Figure 6.55:** Firmware memory usage.

# Chapter 7
# Test campaign, integration and launch

This chapter will give an overview of the test campaign performed on Singer, alone or integrated inside the spacecraft. The operations that were performed for flight preparation and integration on the spacecraft will be presented which finally lead to the launch of SPEISAT.
The testing of Singer and in general of the whole spacecraft required a rigorous planning by the AIV team, whose job was to track the development status of each spacecraft subsystem and planning the integration and testing activities both for the FlatSat and the Flight Model. A view of the AIV plan for the FlatSat can be seen in fig. 7.1, while the same plan for the Flight Model is shown in fig. 7.2, in these pictures each subsystem is represented by a blue parallelogram, each test by a yellow rectangle and each integration activity by a gray rhombus.

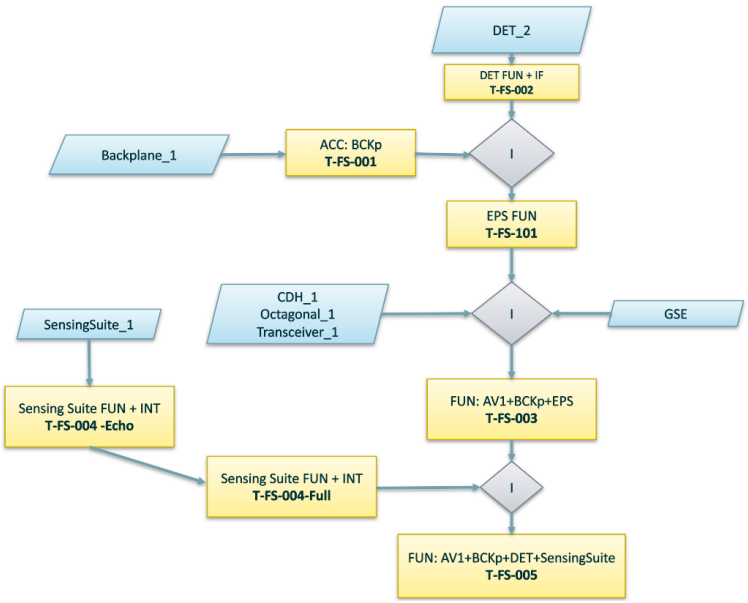



**Figure 7.1:** SPEISAT FlatSat AIV plan.

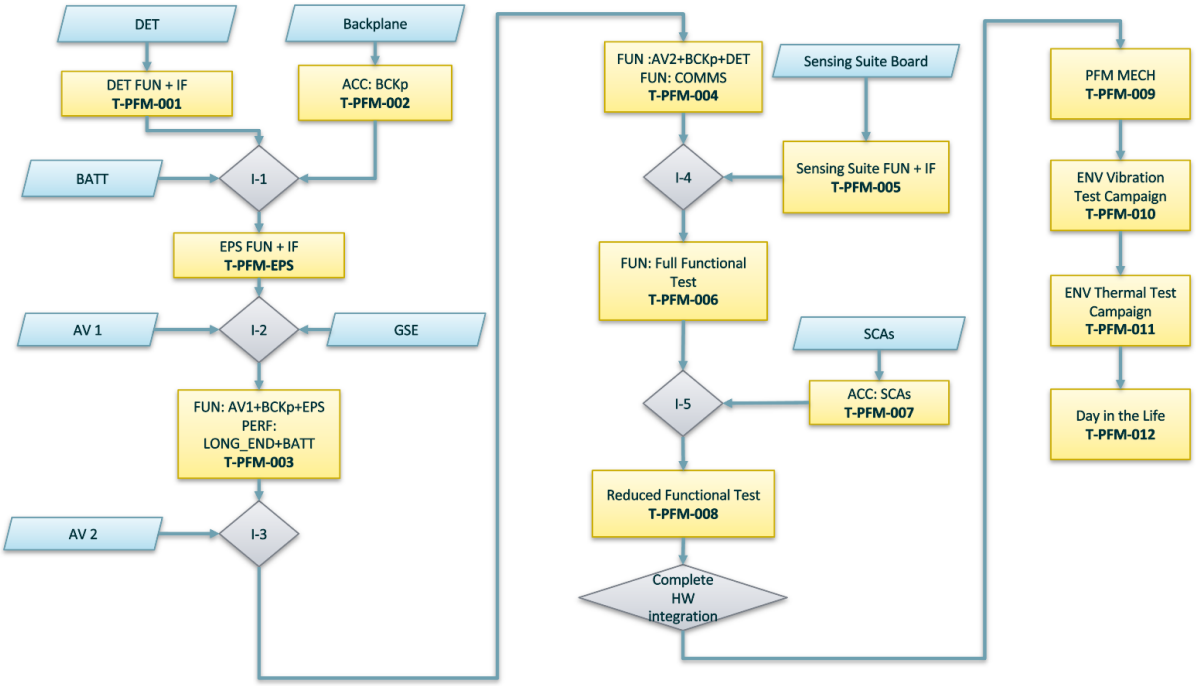



**Figure 7.2:** SPEISAT Flight Model AIV plan.

Any test and integration activity aimed at verifying one or more system requirements and had an associated procedure document, which described the activity step by step in order to be executed by an operator that could technically ignore everything about the specific system involved, the operators for each test were in fact chosen "randomly" by the AIV team from all the group members in order to avoid biases during testing; the operators would perform the test following the procedures and generating a test report document afterwards.
As previously explained in section 4.3.2, the evolution of Singer was characterized by three major phases: the breadboard model, the qualification model and the flight model.
The tests executed on Singer can be mainly divided in three categories:

- Development tests: these tests were informal and not associated with the AIV plan, they can be defined as all the trials which were performed during development to check the correct operation of parts of the system and software (and to verify requirements by Inspection or Demo, see section section 4.2) and that sometimes requested the presence of specific test platforms;

- Acceptance tests: mainly associated with hardware and low level drivers, in these tests any building block was tested individually on the system disconnected by the rest of the spacecraft, to verify its correct operation and hardware requirements;

- Functional tests: mainly associated with firmware, in these tests the overall system was tested to verify the high level functional requirements and the system correct operation under the real use cases, integrated inside the FlatSat or Flight Model spacecraft.

## 7.1 Ground Support Equipment

GSE was necessary to program, debug and test the board, here's a list of the main Singer equipment:

- Laptop: the most obvious equipment is a laptop computer to code, compile, program, debug and test the system; a number of different laptops were used depending on the actual Singer model and if it was necessary to leave it inside the clean room; a remote access was enabled in order to use the laptops remotely from outside the clean room or from home.

- ST-Link: used to interface the laptop with the microcontroller and allow programming and watchpoint debugging; this cheap and versatile programmer (fig. 7.3) comes as a detachable part of any Nucleo board and also features as USB to UART converter that was used to receive debug messages from the Access Port UART.

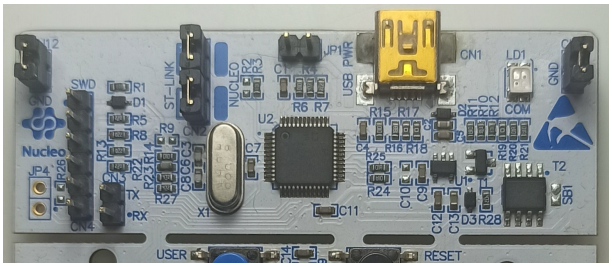



**Figure 7.3:** ST-Link programmer (still attached to the Nucleo board).

- Power supply: which was used to power the system or the whole FlatSat/Spacecraft.

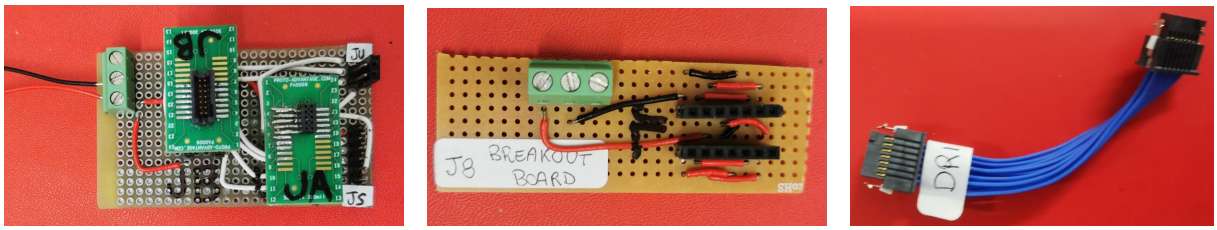
- Multimeter: to measure supply lines voltages and lines continuity.

- Oscilloscope: to visualize signal analog waveforms and timing.

- Logic analyzer: to visualize signal digital waveforms/data and timing.

- Parrot mock-up: realized with a second Nucleo board, this implemented the Parrot message protocol and was used to verify communication with parrot, it wasn't used for the "official" test campaign but only during code development and debug.

- Break-out boards and cables: used to interface the Singer connectors (see section 5.2.7) or the backplane Access Port connector to the ST-Link/Power supply/Parrot mock-up; various adapter boards were built and some were reconfigurable to execute different types of tests, fig. 7.4 shows two examples of breakout boards, fig. 7.4a was used to interface the Access Port/Communication Port of Singer/backplane to the ST-Link and Parrot mock-up, fig. 7.4b was used to implement a loop-back connection of the RS422 lines, fig. 7.4c is the cable model that connects Singer to the backplane or the break-out board, the spacecraft access port (from the backplane) instead had a different connector.




**(a)** **(b)** **(c)**

**Figure 7.4:** Breakout boards. 7.4a Access Port interface board. 7.4b Loop-back board. 7.4c Singer interface cable.

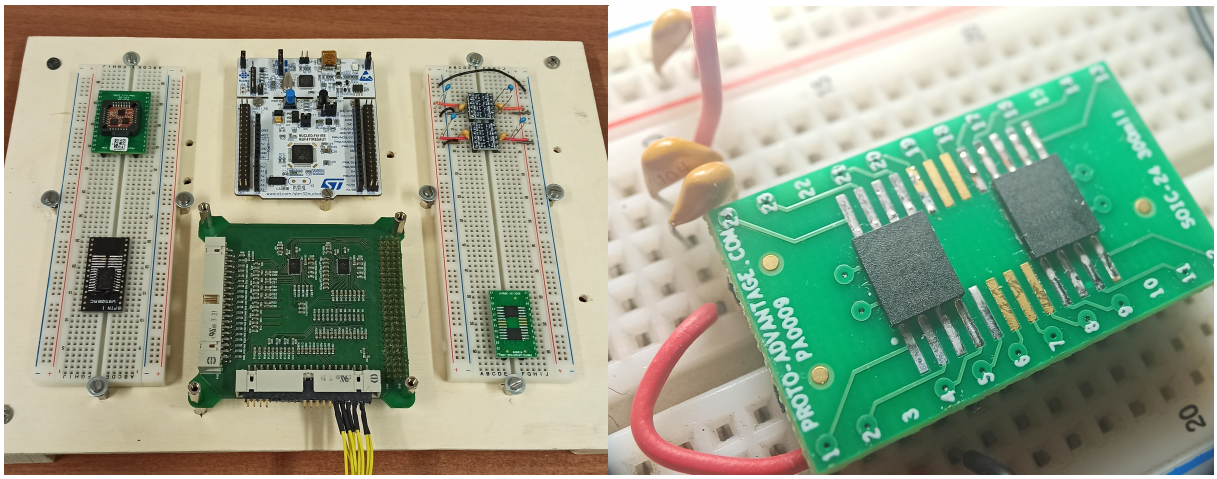
- Anti-static pad and grounding bracelets: these were ubiquitous on any stage of development, they eliminate the risk of electrostatic discharges on the electronics.

## 7.2 Development tests

During all system development phases there was the need to perform informal tests to check the correct operation of parts of the system/software and aid during coding/debug, these usually required setting up different development platforms and utilities of which the most relevant will be presented on this section.

### 7.2.1 Breadboard model

The breadboard model (fig. 7.5a) was the platform for development and testing in the early stages of design, it allowed connecting the microcontroller with the final or candidate/available devices, usually mounted on sockets (fig. 7.5b), to test the interfaces and start developing device drivers.




**(a)** **(b)**

**Figure 7.5:** 7.5a Singer breadboard model, first row from left: IMU, Nucleo, level shifters; second row from left: RS422 transceiver, PC104 board with ADCs, MRAM ICs. 7.5b SMT devices on breadboard socket.

### 7.2.2 Debug console and GDB

The most important utilities for debug, they served different purposes: the console is a serial monitor (see fig. 6.53) which prints the characters sent on the AP UART, in our case we mainly used PuTTY (website: [76]); the GDB debugger is instead configured inside the IDE and makes use of the SWD interface of the microcontroller, it allows inserting whatchpoints on specific code instructions and analyze the order of execution or registers and variables content.
The two debug methods have both pro and cons: the GDB debugger is the most powerful and completely freezes the program when a whatchpoint is reached while peripherals are still running, for this reason it can break the natural flow of the firmware and was mainly used for low level cases in which was necessary to see the memory content, verify specific conditional execution paths or debug the ISRs and drivers; the debug console on the other end is more limited since it can only show data printed from the firmware but preserves its time behavior and in some cases can also be used to mimic a fraction of the functionality of the GDB by printing the content of variables of interest on the monitor, it was mainly used for high level debug and timing verification.

### 7.2.3 Parrot mock-up

The Parrot mock-up was developed as a platform to test the communication protocol, it was simply realized with a second Nucleo board and interfaced with the microcontroller directly (for the breadboard model) or through a breakout board (for the Qualification model). The Parrot mock-up implemented all possible commands which could be sent by hand with specific sequences of clicks on the Nucleo user button, the response message frames from Singer were fully printed in hexadecimal format (fig. 7.6) and this allowed analyzing in details that each message was correctly formatted before connecting the system to the actual CDHs on the FlatSat.

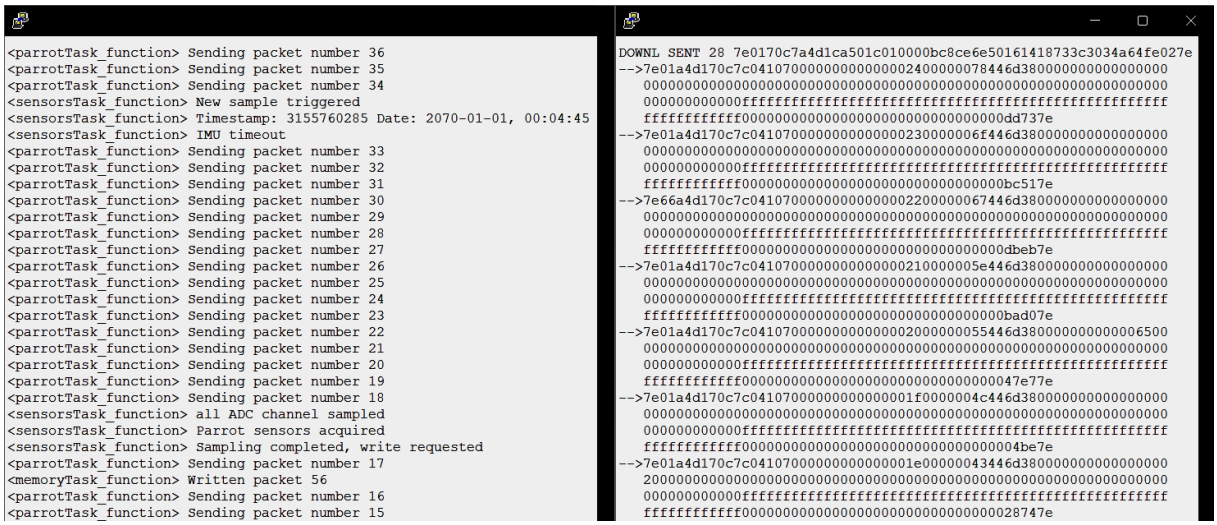



**Figure 7.6:** Singer downlink (left) requested by the Parrot mock-up (right), which then shows the response messages in hexadecimal format.

### 7.2.4 IMU bridge

Xsens provides a PC software suite for all its products (MT Software Suite, website: [63]) which contains a software utility called MT Manager (fig. 7.7); this software provides real time packets visualization, data graphs, calibration utilities and a lot more features that can be useful to test and configure the IMU; MT Manager communicates with the IMU through a serial USB port (COM port on Windows) by means of the Xbus protocol; as already anticipated this is one of the reasons that led to the adoption of the UART interface of the IMU: SPI and I2C interfaces implement a synchronous communication which is different from the Xbus or needs additional layers besides it.

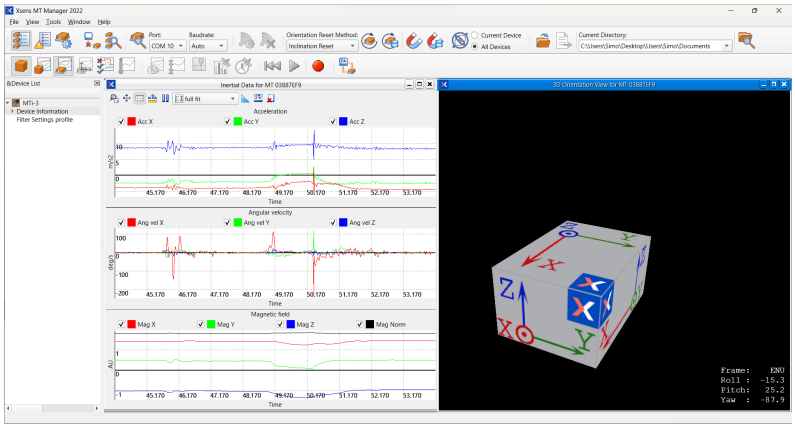


**Figure 7.7:** MT Manager window, showing the data graphs and the IMU orientation view.

By using the UART, the IMU can be directly connected to the PC with an UART to USB converter (already provided by the ST-Link), it's then possible to perform this connection even to the IMU soldered on Singer by using the microcontroller as UART bridge (fig. 7.8), in this configuration a test firmware is loaded on the microcontroller which simply forwards the data between the IMU UART and the AP UART, allowing a simple way to directly connect MT Manager to the device, even when integrated inside the spacecraft.

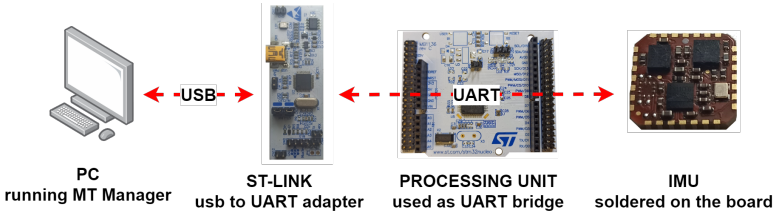



**Figure 7.8:** Microcontroller used as IMU bridge to connect the IMU to MT Manager.

## 7.3 Acceptance tests

As already anticipated, the acceptance tests were made to basically verify each hardware block and the relative device drivers, the reference configuration can be seen in fig. 7.9, with some variations depending on each specific test.
These tests were performed both on the qualification model and the flight model, with minor variations. Acceptance on the qualification model was also helpful to identify errors on the PCB design or production which required some manual corrections.

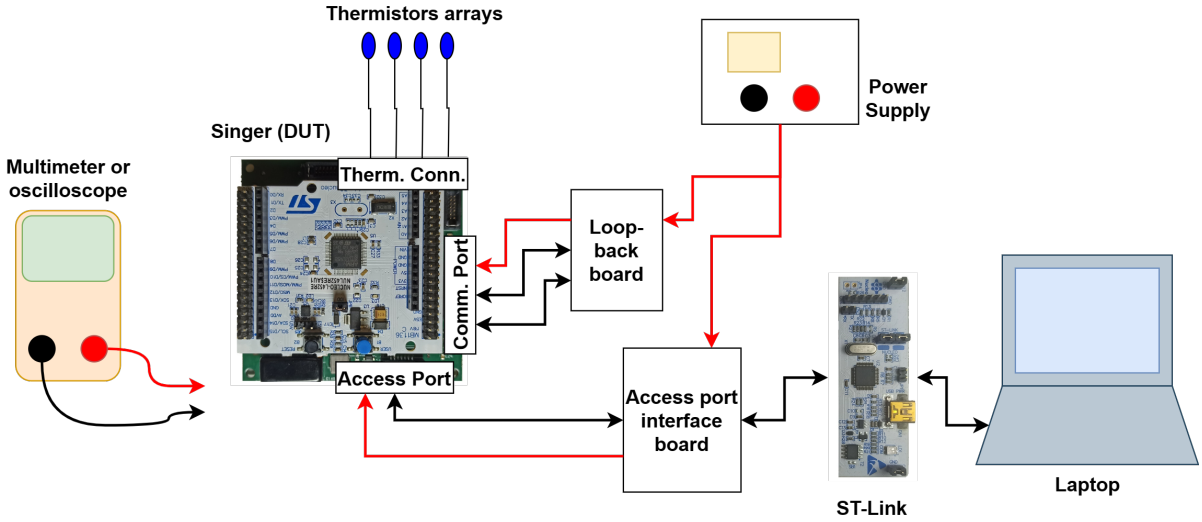



**Figure 7.9:** Acceptance tests reference configuration.

For the majority of tests the microcontroller was programmed with a sketch which would test a specific hardware block, printing the results on the Access Port, this way programming and communication through the AP was also indirectly tested.

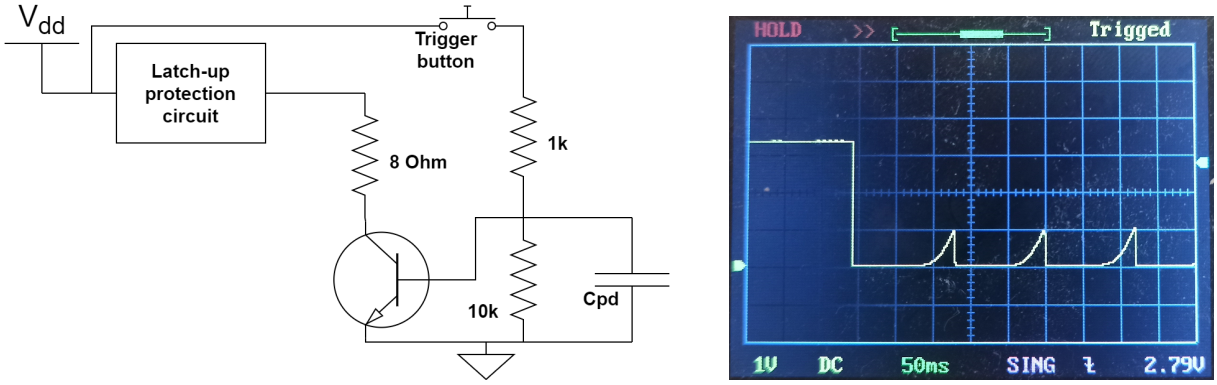
### 7.3.1 Power test

**Table 7.1:** Power test.

| | |
|---|---|
| AIV test (refer to fig. 7.1 and fig. 7.2) | part of T-FS-004, part of T-PFM-005 |
| Main requirements the test was conceived for (refer to section 4.2) | F-P-0, F-P-1 (demo), F-P-2 (demo), P-P-0, I-P-0 |
| Other requirements that were fully/partially verified (refer to section 4.2) | - |

For this test the Nucleo was initially not present on Singer, the board was powered and each voltage domain line was measured with the multimeter to verify its value, the test was repeated by powering the board from both power lines (Access Port and Communication Port) and with different values in the specified range of supply voltage. The Nucleo board was then inserted on the powered off PCB and the current consumption was evaluated by powering the board from both power lines on the specified range extremes. The board power consumption was evaluated from the current measured by the power supply, with a maximum value of 286 mW (at room temperature).
The second part of this test (only performed on the Qualification Model) consisted on simulating a latchup condition on the microcontroller domain, for this purpose the setup was similar to the simulated one (section 5.2.6.2) and can be seen in fig. 7.10a, in this case there was only a single resistor to simulate the low resistance path which is connected as load by an npn transistor turned on as switch by a trigger button, various resistors

values were used to verify the correct activation of the protection circuit; fig. 7.10b shows the voltage at the load using an 8 Ω resistor; differently from the simulated one, the real circuit has a shorter turn-on cycle since it repeats the cycle every approximately 125 ms (while the simulated one was 150 ms).




(a) (b)

**Figure 7.10:** 7.10a Latchup test setup. 7.10b Voltage at load during simulated latchup.

During the test the power lines of all the other domains were observed with the oscilloscope to verify that they were unaffected.

### 7.3.2 MRAM test

**Table 7.3:** MRAM test.

| | |
|---|---|
| AIV test (refer to fig. 7.1 and fig. 7.2) | part of T-FS-004, part of T-PFM-005 |
| Main requirements the test was conceived for (refer to section 4.2) | F-M-0 (only HW), P-M-0 |
| Other requirements that were fully/partially verified (refer to section 4.2) | F-A-0, F-P-0, P-P-0, I-A-0, I-P-0 |

The purpose was to test the most important memory functionalities, for this test the Nucleo was inserted on the PCB, a sketch was loaded on the microcontroller which would perform a complete test of the memory and output on the Access Port the results:

- initialization of the memory registers with the default values;
- total erase of the memory, followed by a complete read to verify the correct erasing;
- trying to write the entire array with the write lock enabled, follower by a complete read to verify that nothing was written on it;
- writing the entire unlocked array with increasing numbers, followed by a complete read to verify the correct execution;

- (before this step a power cycle on the board was performed, with a minute of cool-down before powering the board again) verifying the memory non-volatility, by reading the entire array looking for the numbers written on the previous step;
- repeating the memory erase test (the first time the memory content at startup was unknown and so the actual erase was not really verified, this time we were sure that it was written by the write test).

The test output on the Access Port can be seen in fig. 7.11.

```
TEST MRAM
This program will test MRAM functionalities
Initializing memory...Done
Verifying memory non volatility (fails if this is the first launch)...Failed
Erasing memory...Done
Verifying memory erase...Done
Verifying memory locking...Done
Verifying memory programming...Done
Now turn off and back on supply to test non volatility
Bye!


TEST MRAM
This program will test MRAM functionalities
Initializing memory...Done
Verifying memory non volatility (fails if this is the first launch)...Done
Erasing memory...Done
Verifying memory erase...Done
Verifying memory locking...Done
Verifying memory programming...Done
Now turn off and back on supply to test non volatility
Bye!
```

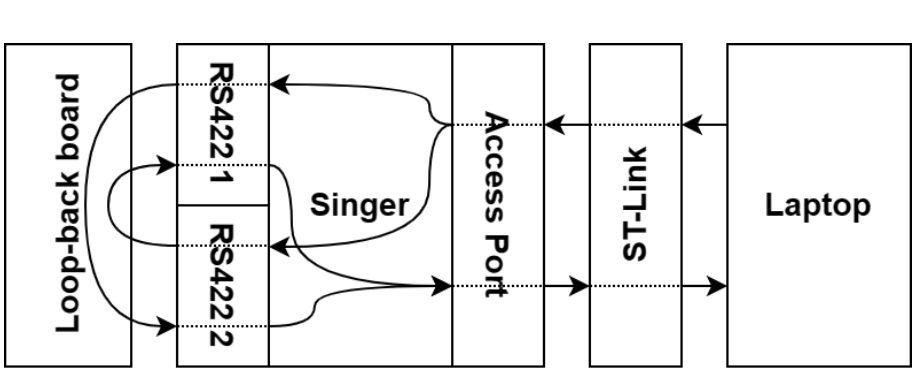
**Figure 7.11:** Memory test AP output.

### 7.3.3 RS422 lines test

**Table 7.5:** RS422 test.

| | |
|---|---|
| AIV test (refer to fig. 7.1 and fig. 7.2) | part of T-FS-004, part of T-PFM-005 |
| Main requirements the test was conceived for (refer to section 4.2) | I-C-0 |
| Other requirements that were fully/partially verified (refer to section 4.2) | F-A-0, F-P-0, P-P-0, I-A-0, I-P-0 |

The purpose was to test the RS422 lines and all the layers of UART driver, for this test the Nucleo was inserted on the PCB; this was also the only test that needed the loop-back board connected, which would close the TX and RX lines of each transceiver in a crossed loop (fig. 7.12a); a test sketch was loaded on the microcontroller, the operator needed to write a string on the Access Port (to also test the transmission of data through it), the inserted string was sent on both RS422 lines and the received data was printed back on the AP (fig. 7.12b), the operator would verify that all bytes were correctly received in the right order.




(a)

```
TEST ECHO
ciao

1 c
2 c
1 i
1 a
2 i
2 a
1 o
2 o
1

2
```

(b)

**Figure 7.12:** 7.12a RS422 test data flow. 7.12b RS422 test AP output. Each number represents an RS422 transceiver, the order of arrival between channels is not important since the messages were sent and received at the same time.

This test was also accompanied by an observation of the waveforms with the oscilloscope to verify the correct implementation of the RS422 standard by inspection.

### 7.3.4 ADC test

**Table 7.7:** ADC test.

| | |
|---|---|
| AIV test (refer to fig. 7.1 and fig. 7.2) | part of T-FS-004, part of T-PFM-005 |
| Main requirements the test was conceived for (refer to section 4.2) | F-S-0 |
| Other requirements that were fully/partially verified (refer to section 4.2) | F-A-0, F-P-0, P-P-0, I-A-0, I-P-0 |

The purpose was to test the correct operation of the ADC chain and the channel assignments, for this test the Nucleo was inserted on the PCB, a test sketch was loaded on the microcontroller which would perform continuous measurements of all the channels and output on the AP the results (fig. 7.13).

```
J3 connector
65535 65535 65535 65535 65535 65535 65535
J4 connector
65535 65535 65535 65535 65535 65535 65535
J6 connector
65521 65522 65522 65522 65522 65535 65535
J7 connector
43020 42630 42963 42925 42548 42572 37387
J5 connector
65522 65522 65522 65522
```

**Figure 7.13:** ADC test AP output. The thermistors cable was connected to J7 at this point, the last thermistor was being heated up by the operator (the code decreased).

This test required some manual work by the test operator: for the qualification model acceptance only one thermistor array was prepared, the operator would move it on every connector and then heat each thermistor with the fingers to verify that the temperature would increase on the right channel. The reconstruction function of the thermistor channel was also applied to the ADC output codes to verify that the measured temperature corresponded to room temperature. The acceptance of the flight model was instead performed with the complete set of flight thermistors in order to also test them.

### 7.3.5 IMU test

**Table 7.9:** IMU test.

| | |
|---|---|
| Main requirements the test was conceived for (refer to section 4.2) | F-S-1 |
| Other requirements that were fully/partially verified (refer to section 4.2) | F-A-0, F-P-0, P-P-0, I-A-0, I-P-0 |

The purpose was to test the interfacing with the IMU, for this test the Nucleo was inserted on the PCB, a test sketch was loaded on the microcontroller which would perform continuous sampling of the IMU output and print on the AP the results (fig. 7.14), when the acceptance test was performed on the qualification model, the IMU was externally connected to the PCB (as previously explained the qualification model didn't have it soldered). The board was slightly rotated in all axis by the operator to verify the sign of the output value (the sign was a good indicator of the correct alignment of the sampled values extracted by the IMU message frame).

```
starting...
IMU: correctly configured
Gyroscope: 193c 9ad4 170c
Magnetometer: b3aa 38e7 a278
Rate of turns [X,Y,Z]: [2.556e-03, -3.334e-03, 1.720e-03]
Magnetic fields [X,Y,Z]: [-2.395e-01, 6.128e-01, -1.263e-02]

Gyroscope: 1959 9a18 163a
Magnetometer: b3a2 38dd a26c
Rate of turns [X,Y,Z]: [2.611e-03, -2.975e-03, 1.520e-03]
Magnetic fields [X,Y,Z]: [-2.385e-01, 6.079e-01, -1.254e-02]

Gyroscope: 194b 9980 12b5
Magnetometer: b386 38df a29d
Rate of turns [X,Y,Z]: [2.584e-03, -2.686e-03, 8.187e-04]
Magnetic fields [X,Y,Z]: [-2.351e-01, 6.089e-01, -1.292e-02]

Gyroscope: 170f 9bfc 1346
Magnetometer: b38e 38e3 a226
Rate of turns [X,Y,Z]: [1.723e-03, -3.899e-03, 8.879e-04]
Magnetic fields [X,Y,Z]: [-2.361e-01, 6.108e-01, -1.201e-02]

Gyroscope: 1816 9aad 1481
Magnetometer: b38e 38e6 a366
Rate of turns [X,Y,Z]: [1.995e-03, -3.260e-03, 1.100e-03]
Magnetic fields [X,Y,Z]: [-2.361e-01, 6.123e-01, -1.445e-02]
```

**Figure 7.14:** IMU test AP output.

## 7.4 Flight preparation and integration

Preparing the flight model required a number of hardware modifications to clean, strengthen and remove possible points of failures from the board.

### 7.4.1 Hardware corrections

Acceptance tests executed on the qualification model highlighted some errors on the PCB that needed to be corrected, this necessity had been considered during design due to the restricted time available and the possibility to produce only one board iteration (as explained in section 4.3.1.1).
As an example some missing pull-up resistors were needed on SPI chip select lines, their absence would have been problematic during power-up or in case of latch-up event on the microcontroller and power cycling performed by the protection circuit, especially for the ADCs that share the SPI bus; these resistors were added externally on the IC packages of domain interfaces. Another corrected error was on the MRAM chip: some QSPI output lines were left unconnected since not needed but these pins had alternative functions in single SPI mode that were found out only after acceptance, one of these functions was a pin that locks writing on registers and for this reason it was connected to the supply rail through a little piece of wire. Fig 7.15 shows both these corrections as performed on the qualification model, the same were done on the flight model but with much more caution and with a better overall quality of the reparations. All these modifications were then secured with glue during flight preparation. Other minor modifications consisted on the substitution of resistors or other passive components.
Some hardware corrections have been necessary to repair mistakes made by the board manufacturer: especially with packages with small pin step (0.5 mm) the manufacturer often caused short circuits (or risk of short circuits) between IC pins due to an improper melting of solder paste under the package legs (as explained in section 7.6.1, this required a reflow process on the paste and in some cases a complete desoldering and replacement of the component.

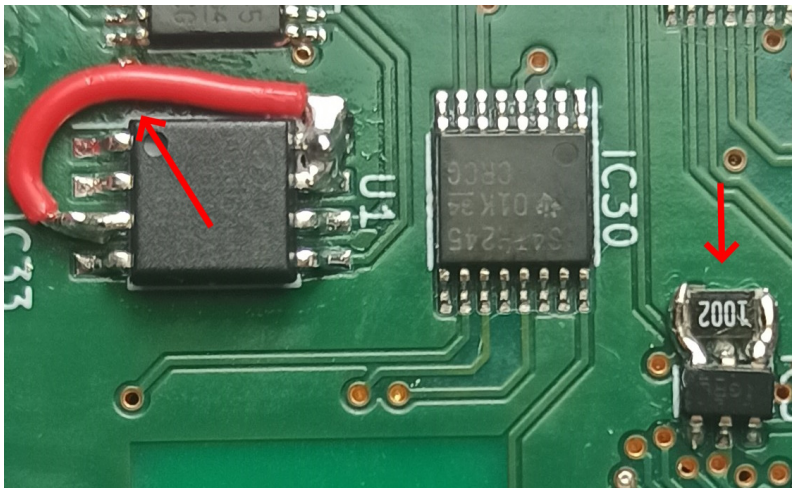



**Figure 7.15:** PCB design error corrections on the qualification model. (Right) Correction on the MRAM. (Left) Added pull-up resistor.

### 7.4.2 Nucleo board modifications

The component that has undergone the highest number of modifications was the Nucleo board (refer to the board schematic [68] and to fig. 7.16):

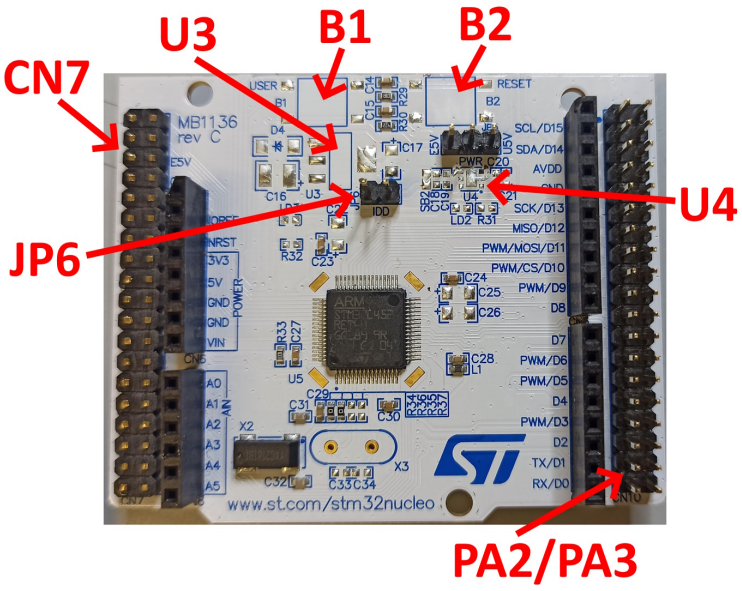
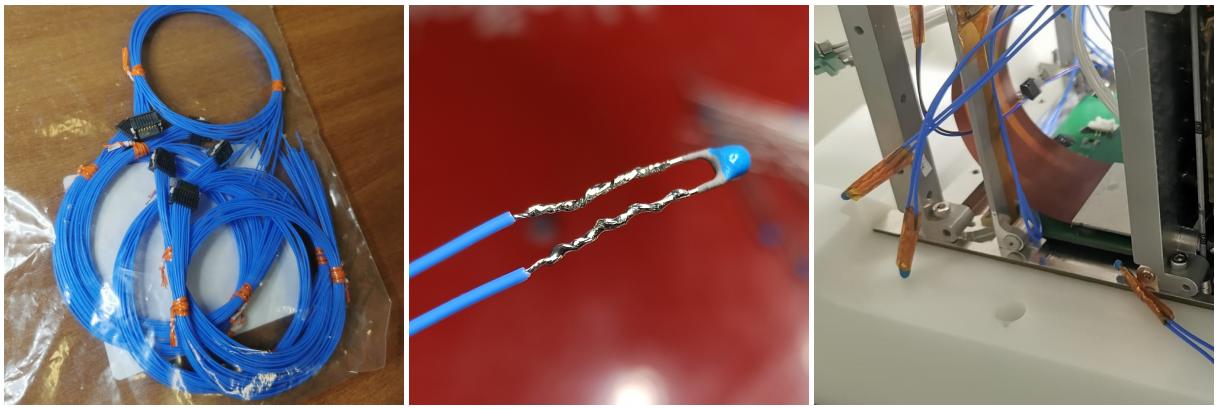



**Figure 7.16:** Nucleo board modified/removed components.

- The ST-Link was detached from the Nucleo board;
- The unnecessary voltage regulators (U3 and U4) and all the associated circuitry was removed from the board, as well as the user and reset buttons (B1 and B2, but leaving the reset button pull-up network);
- The JP6 jumper, which connects the 3.3V line to the microcontroller VDD was shorted with solder instead of relying on the mechanical jumper, this has to be considered another error on the Singer PCB because the microcontroller supply is connected on the 3.3V pin while it could have been connected directly on the VDD pin, eliminating the jumper necessity;
- The UART2 was connected to the corresponding Morpho header pins (PA2 and PA3), this UART is usually connected to the ST-Link attached to the board and when the latter is detached it can be connected to the header by two jumpers (SB62 and SB63);
- All the remaining capacitors were substituted with automotive grade capacitors with flexible termination (the type used on all the Singer PCB) to reduce the possibility for them to break apart or being shorted during launch;
- The CN7 Morpho header pins on the upper side of the board were cut away since cables would need to pass near it inside the spacecraft and we wanted to avoid the risk of piercing through them during launch.

### 7.4.3 Flight thermistors preparation

The complete set of flight thermistors was prepared starting from pre-crimped cables (fig. 7.17a), under instructions from the structure team which provided the necessary length of each thermistor wire. Thermistors' terminations were wrapped around the cables and then soldered (fig. 7.17b), isolation was provided with Kapton tape (fig. 7.17c) to avoid the outgassing of classical heat shrinkable sheaths.

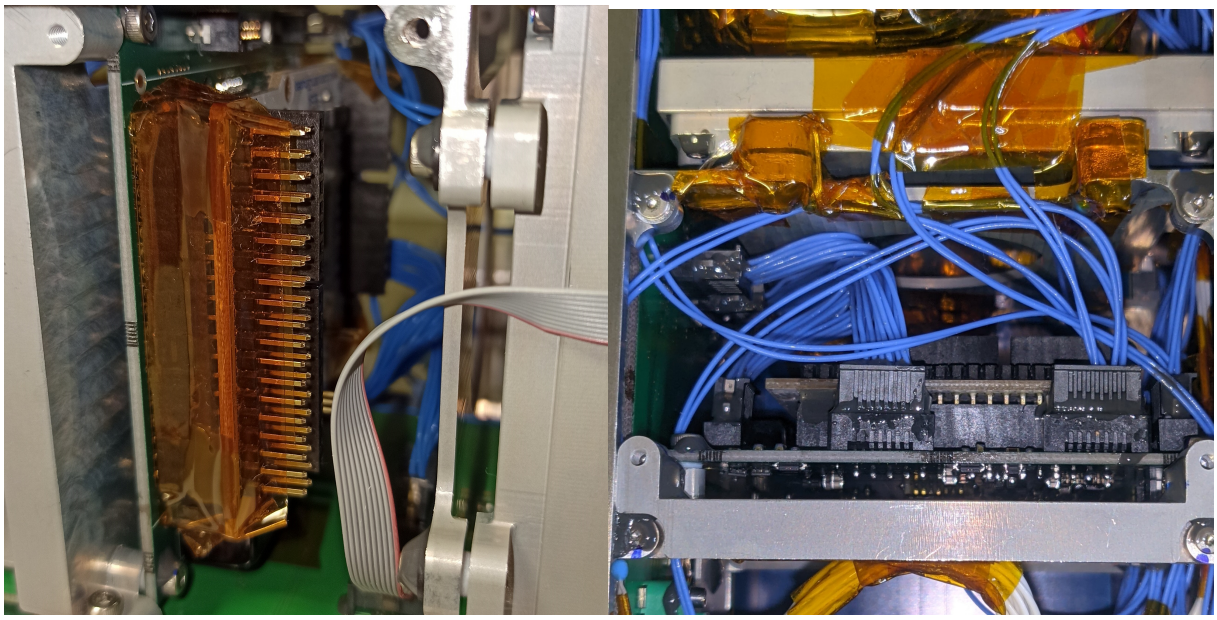


**(a)** **(b)** **(c)**

**Figure 7.17:** 7.17a Pre-crimped cables for thermistors. 7.17b Soldered thermistor. 7.17c Thermistors integrated in the spacecraft, before fixing them to the measurement points.

### 7.4.4 Integration

The AIV team, together with the structure team, created a precise plan and step-by-step procedures for integration, these procedures were developed and tested with the realization of a first exposition model of the spacecraft which was useful to verify the mechanical dimension requirement (I-MI-0). Before proceeding with integration all the flight boards were washed in an ultrasound bath, a specific glue was applied afterwards on all parts to secure the most heavy packages and connectors, glue drops were applied on all Singer's connectors as well as the IMU, the power regulator, the biggest packages on the PCB and the modification described in section 7.4.1. The Nucleo board was inserted on the PCB and glued as well (fig. 7.18a), the same was done on the backplane and thermistor arrays' connectors after insertion (fig. 7.18b).



**(a)** **(b)**

**Figure 7.18:** 7.18a Nucleo board glued on Singer PCB, we can also notice the glue on PCB mounting screws. 7.18b Backplane and thermistor array connectors glued during integration.

The integration took several days and consisted on a series of delicate operations, each part was placed and secured with glue and Kapton, the result can be seen in fig. 7.19.

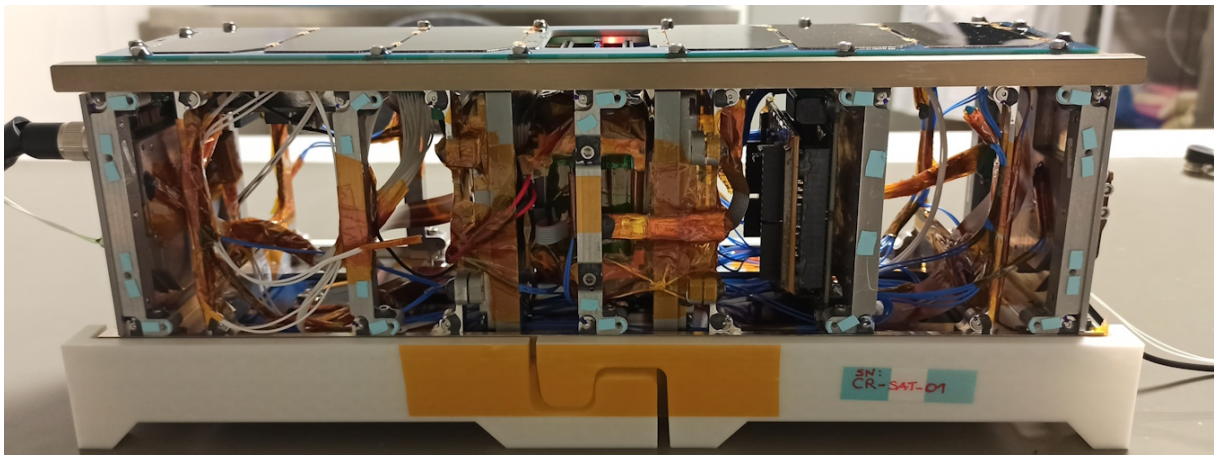



**Figure 7.19:** The integrated flight model without the last two solar panels.

## 7.5 Functional tests

Functional tests were performed both on the qualification model and flight model after integration with the FlatSat or the spacecraft.
Fig. 7.20 shows the reference configuration of functional tests, the picture only shows the point of view of Singer and so only blocks that are directly connected to it while in reality functional tests were usually performed with the FlatSat/Flight Model fully integrated and complete of every subsystem; also in some cases the power supply was still used to power the whole system (and so connected to the battery).

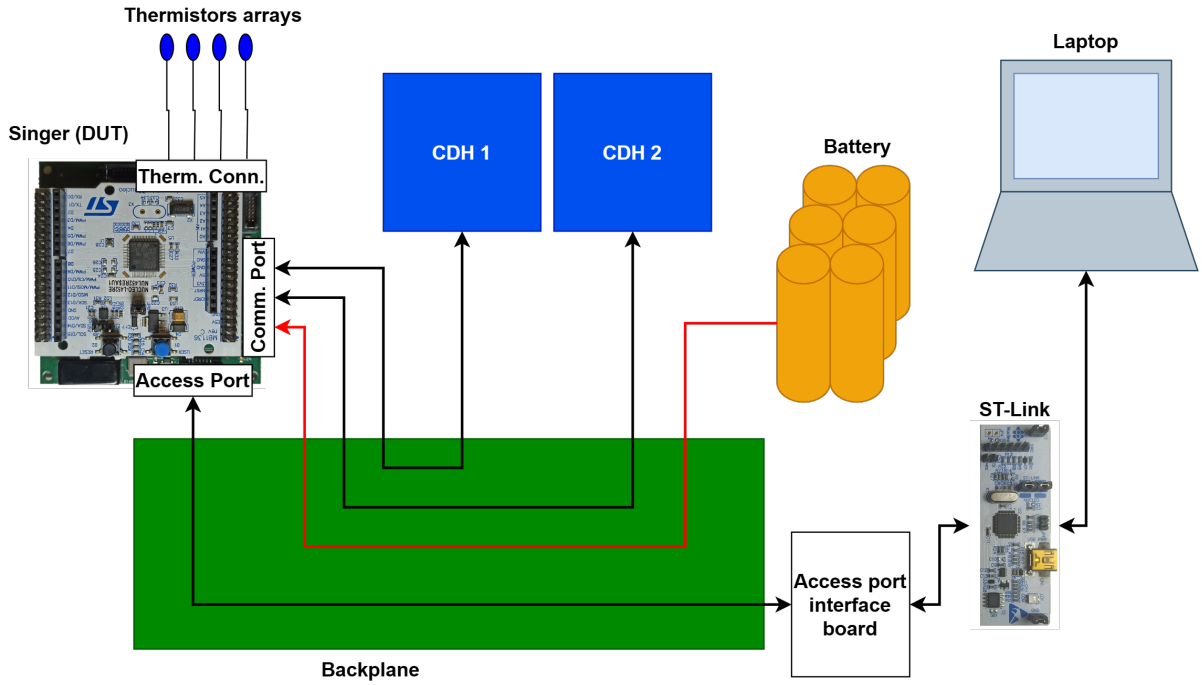



**Figure 7.20:** Functional tests reference configuration (only the point of view of Singer is shown).

In this section the functional tests performed on Singer will be listed.

### 7.5.1 Full functional and day-in-the-life tests.

**Table 7.11:** Full functional and day-in-the-life tests.

| | |
|---|---|
| AIV test (refer to fig. 7.1 and fig. 7.2) | T-FS-005, T-PFM-006, T-PFM-008, T-PFM-012 |
| Main requirements the test was conceived for (refer to section 4.2) | F-T-0, F-T-1, F-S-2, F-M-0, F-M-1, F-C-0, F-C-1, F-C-2, P-S-2, P-C-0, P-C-1, I-C-0 |
| Other requirements that were fully/partially verified (refer to section 4.2) | F-S-0, F-S-1, F-A-0, F-P-0, P-M-0, P-P-0, I-A-0, I-P-0 |


**(a)** **(b)**

**Figure 7.21:** fig. 7.21a The flexible antennas after deployment. fig. 7.21b Ground Station front-end.

The purpose of the full functional test was to verify all the spacecraft operative mode and functionalities, including deployment of the real flexible antennas (fig. 7.21a) and communication through them (until that point communication with the radios was tested with other temporary antennas on a secondary connector, since the real ones are quite fragile and deploying them requires a lot of work to fold them back into flight configuration) thus involving also the ground station chain (fig. 7.21b).
As said, the full functional test aimed at verifying every functionality and command that the satellite could receive and the automatic transition between operative modes, these transitions were verified with fictitious delays since the actual mission has delays on the order of days for the automatic transition. The satellite Access Port interfaces were connected during the full functional test and the operators could see the output messages on the various debug consoles of Parrot and Singer.
The day-in-the-life was a similar, reduced version of the full functional and was performed as a last test before the satellite got prepared for shipping to the launch base and consisted on the simulation of the first hours after deployment into orbit to test the automatic operation of the spacecraft and the first communications with it, the aim was not to test the spacecraft completely but more as a last trial of the satellite and ground station operation in nominal conditions and for an extended period of time to check the system stability, the Access Ports were all disconnected during this test and communication with the satellite could be performed only by the radio link and so the ground station chain, simulating the real mission scenario.
From the point of view of Singer, the day-in-the-life did not verify additional requirements with respect to the full functional, on both tests the spacecraft activation procedure was verified which involved requesting a memory reset to Singer, then the automatic transition to commissioning mode and then payload mode was verified, in payload mode the single telemetry request and the downlink request to Singer were both tested (and so also the time word synchronization), finally the satellite was sent into decommissioning mode. The

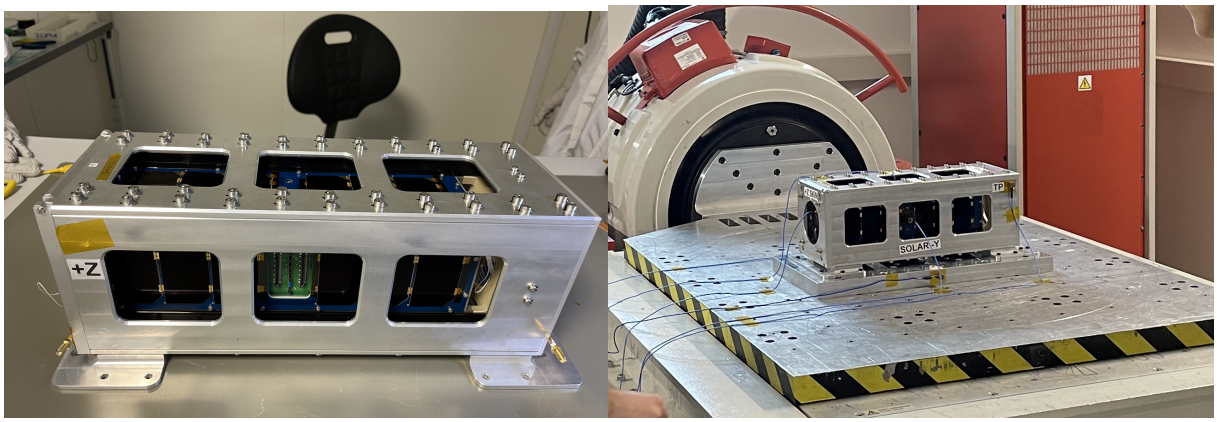
full functional was also a formal verification of some Singer requirements that until that point were only informally verified during development tests, like the correct timing of sampling and the timing (from the software perspective) of commands with Parrot.

### 7.5.2 Mechanical fit and vibrational tests

**Table 7.13:** Mechanical fit and vibrational tests.

| | |
|---|---|
| AIV test (refer to fig. 7.1 and fig. 7.2) | T-PFM-009, T-PFM-010 |
| Main requirements the test was conceived for (refer to section 4.2) | P-MI-0 |
| Other requirements that were fully/partially verified (refer to section 4.2) | - |

Technically not functional tests but mechanical interface ones, the purpose was to verify the mechanical integration of the spacecraft inside the CubeSat deployer (fig. 7.22a) and perform vibrational tests (fig. 7.22b) to verify the compliance with the launcher requirements (and that the spacecraft could sustain the the launch stresses), comparing the results with the previous vibrational test performed on the dummy mass model. The test was performed in the facility of an external company.




**(a)** **(b)**

**Figure 7.22:** 7.22a SPEISAT inside the deployer dummy. 7.22b Vibrational test setup, the CubeSat was placed inside the dummy deployer.

### 7.5.3 Thermal cycling test

**Table 7.15:** Thermal cycling test.

| | |
|---|---|
| AIV test (refer to fig. 7.1 and fig. 7.2) | T-PFM-011 |
| Main requirements the test was conceived for (refer to section 4.2) | P-S-1 (demo), P-TH-0 (demo) |
| Other requirements that were fully/partially verified (refer to section 4.2) | F-T-0, F-T-1, F-S-0, F-S-1, F-S-2, F-M-0, F-M-1,F-C-0, F-P-0, P-S-2, P-P-0, I-C-0, I-P-0 |

The purpose was to verify the thermal behavior of the spacecraft, the test was performed in the facility of an external partner and consisted on the exposure of the spacecraft to the expected temperature variations in orbit inside a thermal chamber (fig. 7.23). The Singer thermistors network was very useful in this phase because it allowed to gather a significative number of temperatures on various points inside and outside the spacecraft.

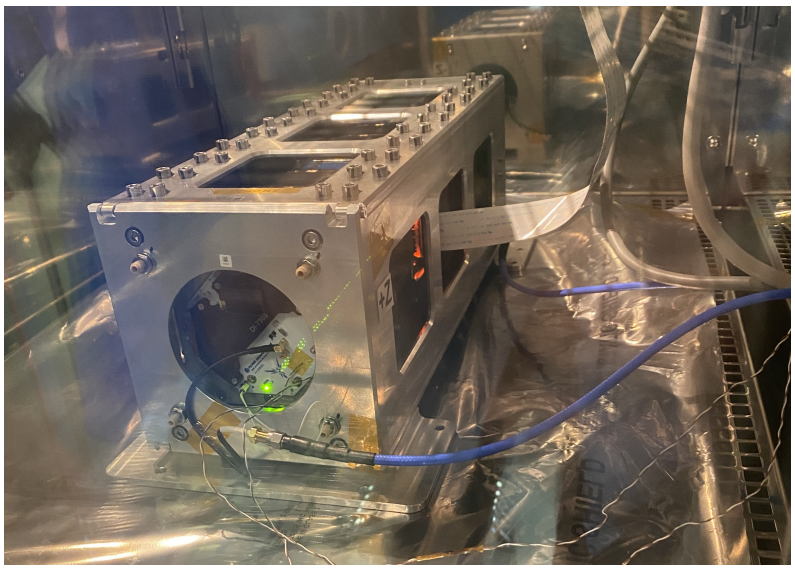


**Figure 7.23:** Spacecraft on the dummy deployer inside the thermal chamber.

## 7.6 Laboratory failures

Some accidental hardware failures occurred during the testing and integration phases which halted the development effort for some time, this section will present the two main problems that were encountered with a brief explanation of the causes and the lessons learned.

### 7.6.1 Qualification model failure

The qualification model boards were visually inspected after production to ensure that the soldering process was correctly performed before powering them for the first time; this visual inspection did not highlight visible process errors and the author proceeded to power the first board to perform the power acceptance test.
During the acceptance test, one of the two ADCs' power domain resulted unpowered,

suggesting that a short circuit on the domain had triggered the PTC fuse, this suspect was confirmed after measuring the PTC resistance and the board was again inspected more in details, revealing that hidden under the device package legs, some tiny solder paste grains were present that shorted the power supply pin to ground (as can be seen in fig. 7.24).

**Figure 7.24:** Solder paste residues from improper manufacturing, the distance between two legs is 0.5 mm.

This error was almost invisible to the naked eye, highlighting the necessity of a better visual/electrical inspection of the boards; other similar errors were found on the other boards, requiring some manual work to remove them, as explained in section 7.4.1.
This incident allowed verifying the correct functioning of the domain separation and protection that was specifically designed to deal with this type of event (requirements F-P-1 and F-P-2), since the other domains were unaffected by the short circuit on this device; it also highlighted the correctness of the design driver of trying to select big packages for devices and choosing to resort on the Nucleo board instead of soldering the microcontroller directly on the PCB (STM32 have the same pin step of the ADCs and desoldering its bigger package would have been much more challenging).

### 7.6.2 Flight model failure

This was the most serious accident which nearly led to the cancellation of the Singer mission. It happened with the spacecraft already integrated and ready for the functional tests; fig. 7.25 shows a reconstruction of the scenario during the accident:

- Two operators were inside the clean room, executing software updates on Parrot and preparing to turn off the satellite supply in view of some final integration activities;
- The singer AP breakout board was connected to the satellite and another operator (the author) was outside the clean room, remotely monitoring the Singer console output;
- Without anybody knowing, the grounding plug of the anti-static pad was poorly inserted on the socket due to a loose mechanical connection, and so the satellite and the operators on the clean room were not grounded to earth;
- The Singer breakout board was not correctly coated and in general the board (made with perfboard due to the short time available) was not up to the important task it had

to perform.

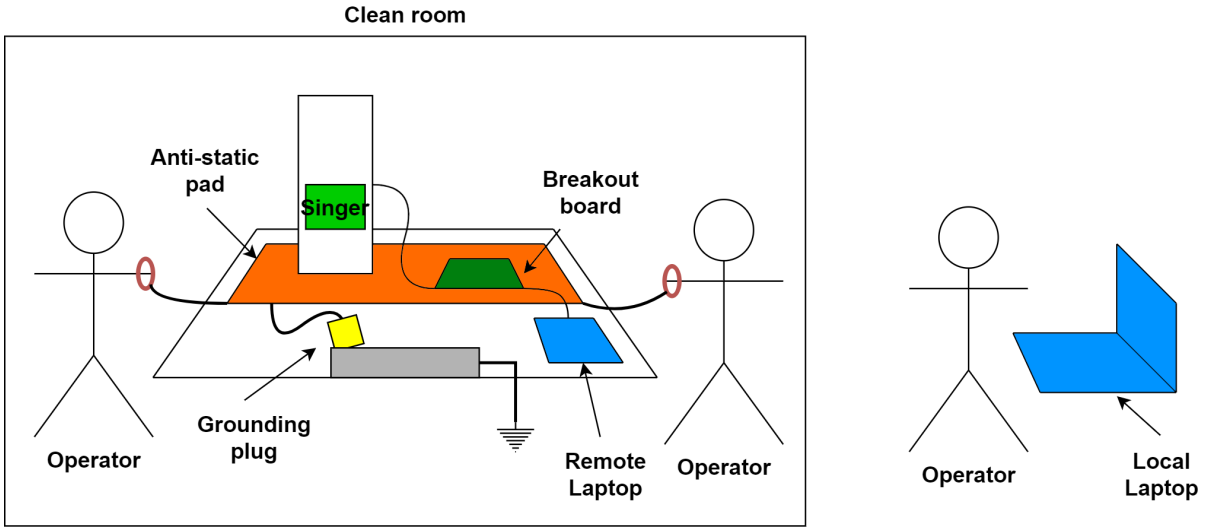



**Figure 7.25:** Flight model accident scenario.

At some time, the author noticed that the console printed some garbage characters and then no more output was getting printed (fig. 7.26), he thought that this was due to the supply being turned off since the garbage output on the UART is common in this case.

```
a5 26 ab 4yd ab a1 ab 13
a9 29 ac de a9 99 a4 77
aa ab ab 37 ac 3b ab 6a
ac e8 aa 68 ac 60 4b 10
4b cc 0a cc 0a 00 02 e0
3f

<encodeDataByteStuff> message encoded, length 115
<sendSerMsg> sending message ID 20912 on serial interface
```

**Figure 7.26:** Garbage output after the incident.

After some minutes of no output from Singer, the author approached the clean room and noticed that the power was still turned on and everything was still connected, at that point it was clear that something had happened. An inspection of the supply voltage of Singer from the Access Port "VDD_target" pin revealed the waveform of the latch-up circuit (fig. 7.10b) and so that the voltage domain of the microcontroller was somehow shorted.
The impossibility to investigate the problem from the spacecraft outside was the topic of discussion for the whole day, two options were on the table:

- completely disconnect Singer from the supply, a possibility that was specifically considered for a case like that and that led to the predisposition of an exposed trace on the access port to be cut;
- open again the satellite and perform a very complex operation to unglue and remove the system and substitute it with the spare, with the risk of damaging part of the primary mission.

After some discussions, the second option was performed and the structure team was involved on this extremely delicate operation that lasted for 48 hours.
The damaged system was later inspected by the author and multiple shorted devices were found:

- the microcontroller supply, along with some GPIOs resulted being shorted to GND;
- the domain interface IC with one RS422 UART had one supply pin shorted (the one on the RS422 transceiver domain, not the one on the microcontroller domain).

Since multiple ICs were shorted on different voltage domains, considering the accident scenario (grounding plug disconnected) and the peculiar fact that in the laboratory there is an unusual high level of static in the air, the most probable cause was determined to be an ESD event that propagated to Singer from the unproperly coated breakout board, maybe from an operator that, unaware of not being grounded, touched the board to move it in preparation of the integration activities.
A second possibility was instead a short circuit of a microcontroller GPIO that reached the inside through the access port, still caused by the unproperly coated and designed breakout board, this was considered less probable for two reasons: it doesn't explain the second IC shorted on a different power domain and it should not produce this catastrophic results on the microcontroller since the latchup protection circuit would activate to disconnect the supply; in any case the author was surely guilty of improperly designing the breakout board, among other unfortunate coincidences.
It's worth noticing that even if two Singer ICs were shorted (one protected by the complex latchup protection and the other with the PTC fuse), this did not affect the rest of the spacecraft in no way, as another confirmation of the correct implementation and functioning of the protection circuits (requirements F-P-1 and F-P-2).
This incident made us learn some valuable lessons:

- the importance of always verifying any grounding connection before operating;
- the importance of properly designing and coating the access port interfacing boards, possibly producing them as PCBs and avoiding the use of perfboards, especially for flight models;
- the importance of reducing as much as possible the time the spacecraft access port is wired, diverting the software testing and monitoring activities to the FlatSat instead;
- the importance of always turning off the power supply before touching in any way the spacecraft or the equipment connected to it.

## 7.7 Spacecraft shipping

The spacecraft was packed (fig. 7.27) and shipped to the launch base in California, a delegation of members of our team was sent to the base to proceed with the integration of the satellite inside the deployer.

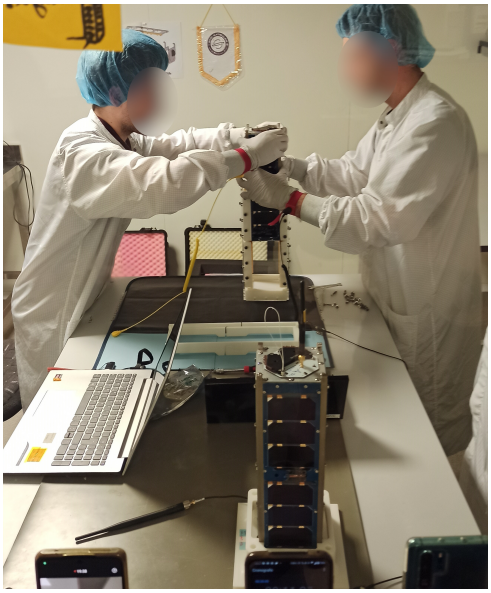
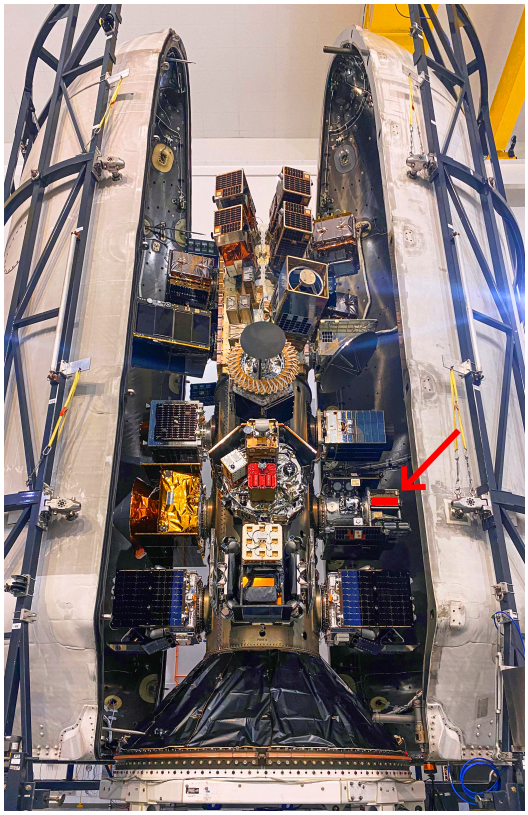


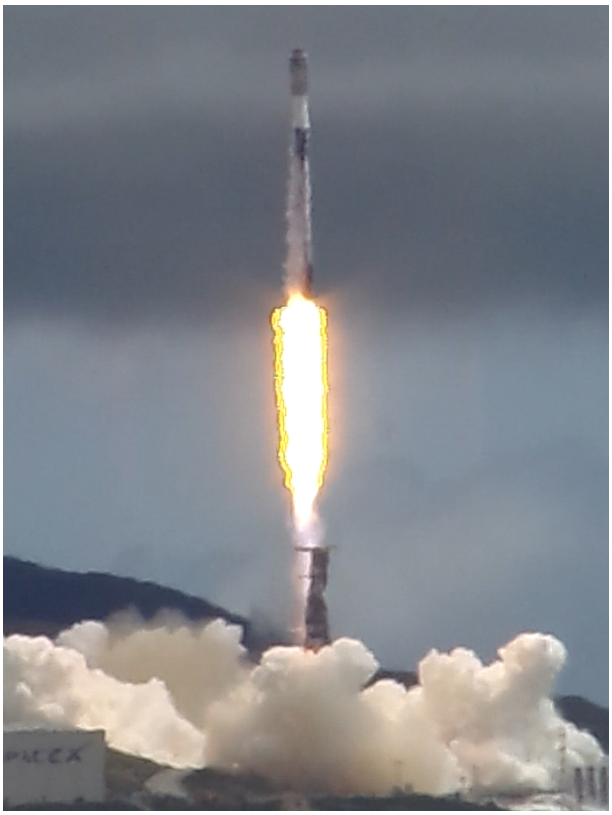
**Figure 7.27:** Operators testing the spacecraft shipping container with the exposition model, the flight model can be seen on the foreground.

# Chapter 8
# Mission results

## 8.1 Launch



(a)



(b)

**Figure 8.1:** 8.1a View inside the Falcon 9 second stage fairing of Transporter-8 launch, the arrow points to the position of SPEISAT, the approximate dimension of the CubeSat is drawn as a red rectangle, Credit: SpaceX. 8.1b Falcon 9 launch with SPEISAT onboard, picture shot by the author. Credit: SpaceX

SPEISAT was launched onboard a Falcon 9 as part of the SpaceX's "Transporter-8" mission; Transporter is a dedicated rideshare program in which dozens of small satellites builders share the launch price (see fig. 8.1a), SPEISAT was technically a tertiary payload,

since it was contained inside a bigger satellite carrier (D-Orbit's ION-SCV) spacecraft and released some days later from it.
The launch took place on June 12, 2023 at 21:35 UTC from Vandenberg Space Force Base (fig. 8.1b), the author and some other team members had the pleasure to be invited to assist the launch in person while a press event was held at the Polytechnic. The ION satellite carrier containing SPEISAT was released at T+01:20:18, the CubeSat was instead released 11 days after, on June 23.

## 8.2 Mission operation

Although the ground station was set up with the effort of a specific sub-group, the operations were also performed by volunteer members from all groups of SPEISAT team (including the author) since they required a significative workforce to follow the satellite passages (some in the middle of the night), decode the received packets and format them to be inserted on the database.

### 8.2.1 Ground station

Our team was working from well before SPEISAT to develop a ground station at Turin Polytechnic, the mission led to an acceleration of the effort but unfortunately it wasn't ready in time. This possibility had already been considered and that's one of the reasons why the spacecraft communicates on a common amateur radio frequency band (UHF) and protocol (AX.25), it was then possible to get support from the community for operations. The precious help from the amateur community came with the downside of not being able to fully exploit the ground station infrastructure that was designed for the Polytechnic station, for this reason a good amount of manual work needed to be performed, especially at the first stages of the mission.


**Figure 8.2:** ARI-Bra amateur radio station. The UHF antenna can be seen in the center.

The main partner supporting the operations was the amateur station of ARI-Bra (website: [7]), having an history of collaborations with the Polytechnic for its preceding missions e-st@r and e-st@r-II, this station is equipped with a directional antenna mounted on a rotator (fig. 8.2) which can follow the satellite on its orbit; the relative satellite position in the sky (azimuth and elevation) is computed by an orbital propagation software, in our case Gpredict (website: [37]) which propagates the satellite position and trajectory (fig. 8.3) from a Two Line Element set (TLE); the TLE is a string of two lines (as the name suggests) which identifies the position and orbital parameters of an object at a precise time instant and can be used to predict the trajectory with a growing error from the TLE timestamp, it must be continuously updated by agencies that perform radar observations. SPEISAT has been assigned with catalog number 56991 and international designator 23084BT (as can be seen on its SatNOGS DB page: [88]).

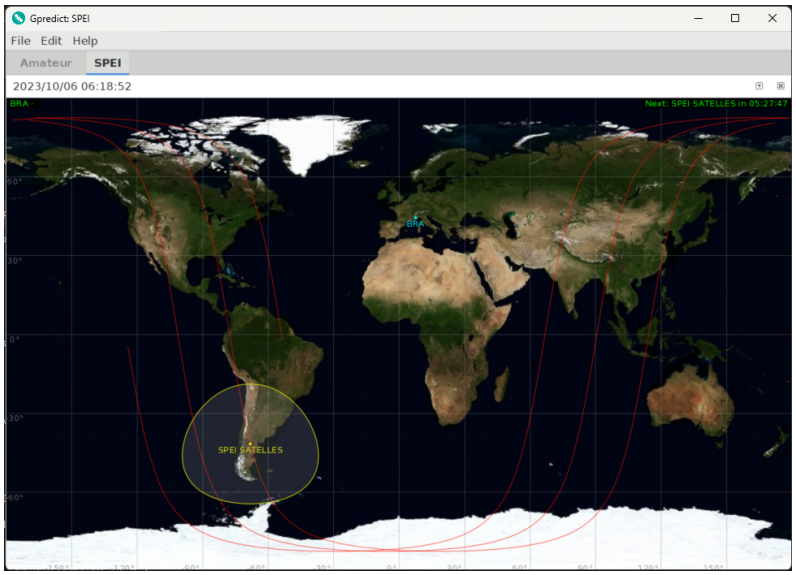



**Figure 8.3:** Main Gpredict window showing SPEISAT position, trajectory and line of sight.

Gpredict also computes the expected Doppler shift from the relative satellite speed, this is used to correct the radio frequency for uplink/downlink.
Fig. 8.4 shows the high level architecture of the Ground Station.

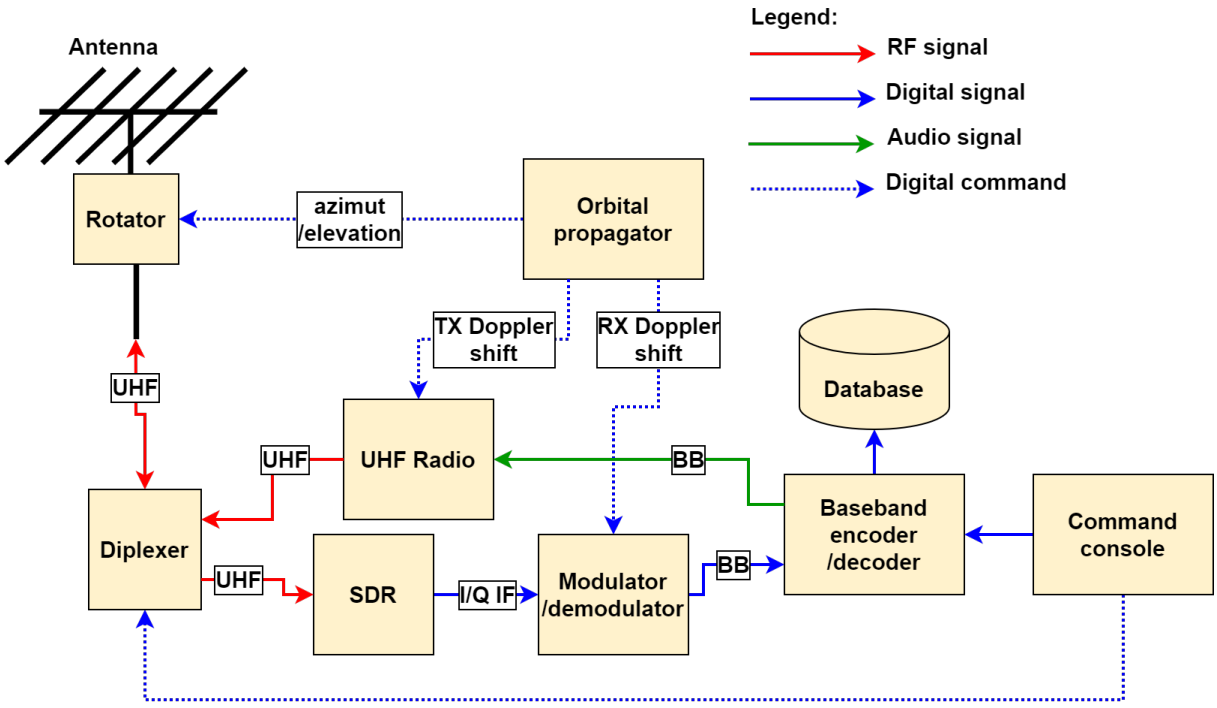



**Figure 8.4:** Ground Station setup at ARI-Bra.

Reception is performed in heterodyne configuration: a Software Defined Radio (SDR) implements the RF filter and Low Noise Amplifier, it mixes and samples in I/Q the received signal on an Intermediate Frequency (IF), a digital FM demodulator is then applied (using Gqrx SDR software, website: [38]) which outputs the demodulated Baseband (BB) signal, the latter is sent to a digital decoder (a flow graph on GNURadio, website: [36])

which finally outputs the digital packet to be stored on the database; the same happens on transmission, with some differences depending on the station that we used (on Polytechnic station there should have been the same chain in transmission but on ARI-Bra station, transmission was performed by a separate UHF radio which was getting the BB signal as audio input, since the SDR didn't have a powerful enough Power Amplifier).

### 8.2.2 Analysis of data

The mission was a success and Singer allowed downlinking days of data (fig. 8.5) from orbit, here an analysis from the sensors data of a single downlink will be presented, this downlink covered a time of around 9 hours or about 5.5 orbits.

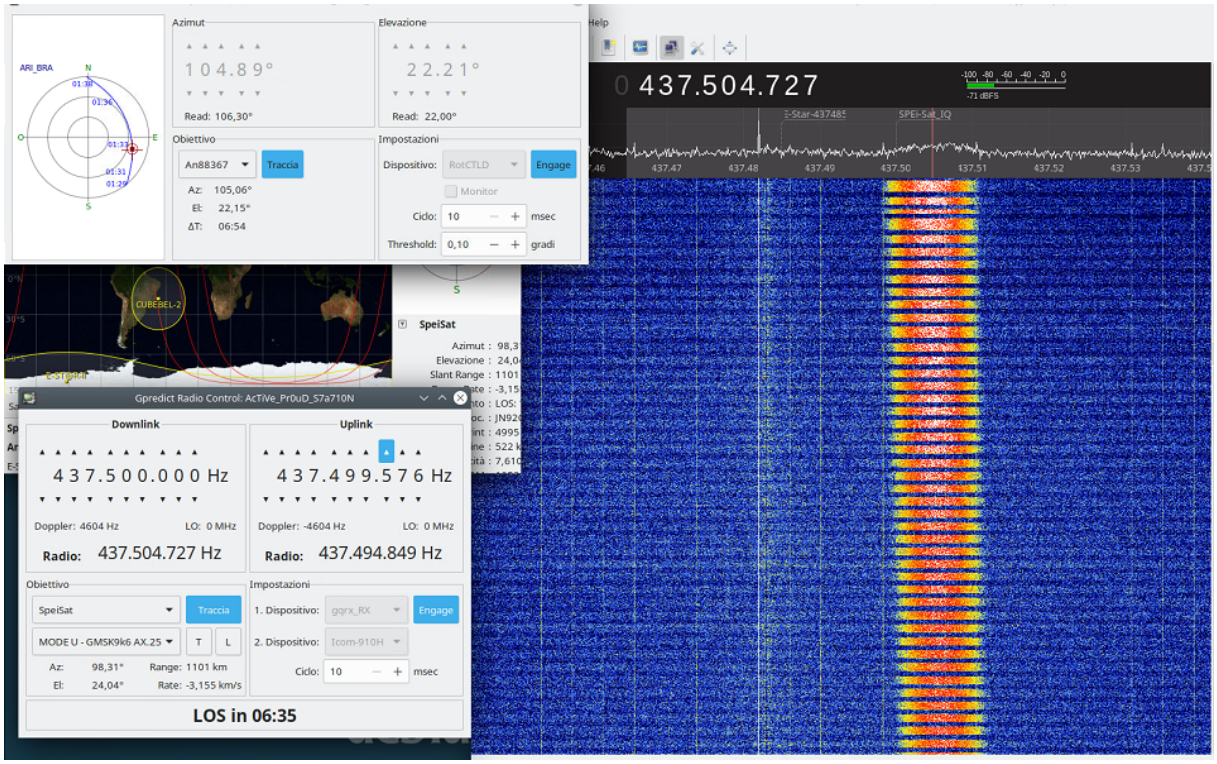



**Figure 8.5:** Singer downlink waterfall shown on Gqrx (right), it's possible to spot the Doppler shift on the received frequency; Gpredict tabs are also visible (left) which were commanding the antenna rotator and correcting the reception frequency on Gqrx.

Fig. 8.6 shows all the temperatures measured by Singer, the temperature fluctuation due to the alternating sunlit/eclipse is clearly visible, the highest temperature (red in the image) corresponds to the Direct Energy Transfer (DET) board introduced in section 3.2.1 which dissipates the excess power from the solar panels to limit the voltage, reaching a peak temperature of around 50°C; the lowest temperatures correspond to points on the spacecraft outer faces which reach temperatures as low as around -20°C; a curious phenomenon can be observed when the spacecraft exits the eclipse and starts to be illuminated (highly noticeable at around 13:30 or 16:30 on the plot), the -X skins of the spacecraft experience an abrupt temperature rise which reaches a peak in around 15 minutes and then oscillates back to lower values, this is probably due to the orientation of the

spacecraft at the poles and is a demonstration of the fact that the spacecraft temperatures could potentially be used to estimate the satellite attitude beside the data from the IMU. We can also notice a line in the center with way lower oscillations, actually these are two different superimposed lines representing the temperatures of the two CDHs' aluminum cases, highlighting the fact that the satellite thermal insulation is doing its job.

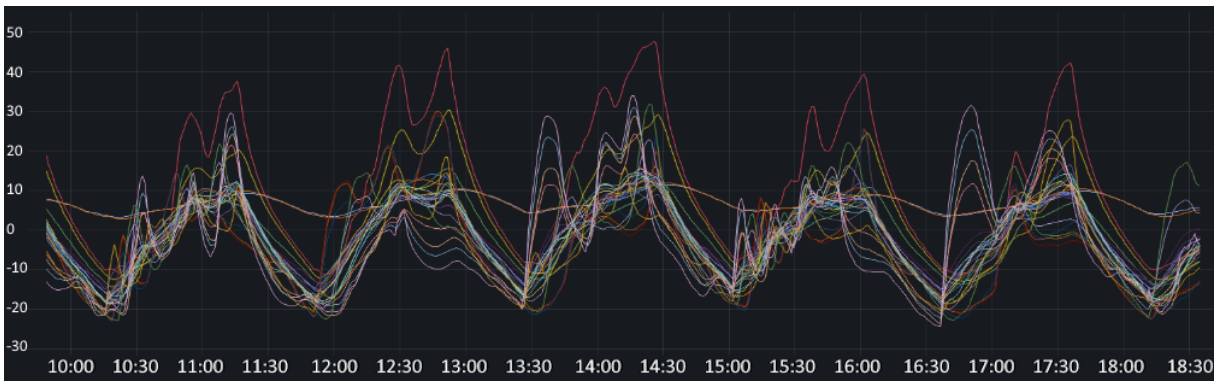



**Figure 8.6:** Singer temperature measurements versus time, y axis is in [°C].

Another source of temperature measurements comes from Parrot sensors (fig. 8.7), which are sent to Singer from the CDHs and contain the temperatures measured internally by the CDHs and of the battery. This values helped producing two theses up to date, in which the measurements are analyzed in detail and compared against the values predicted by the spacecraft thermal model by team members Davide and Francesco ([20], [58]).

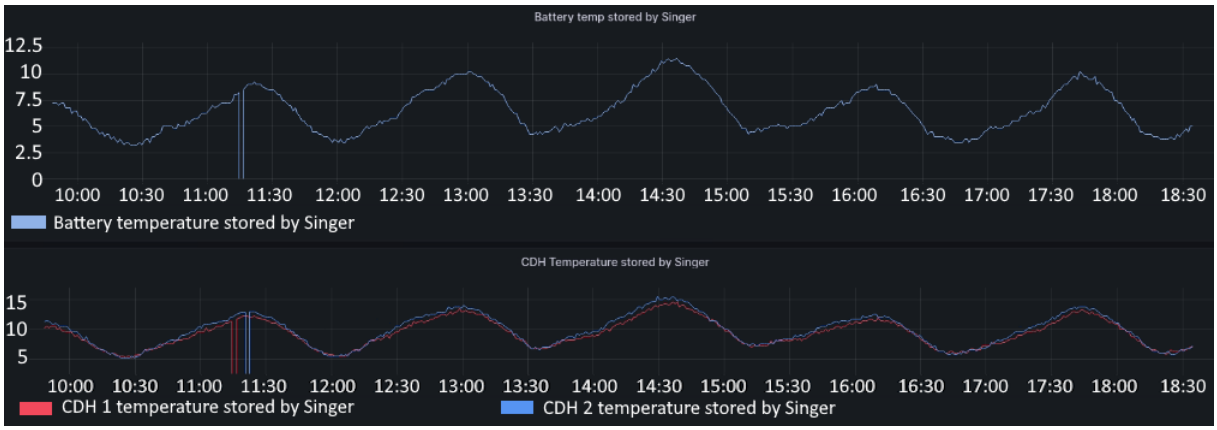



**Figure 8.7:** Parrot sensors temperature measurements of battery (up), and the two CDHs (down), y axis is in [°C] for both.

Remaining on Parrot sensors, Singer also receives and stores telemetry about the battery voltage and charge/discharge currents (fig. 8.8); as can be seen, the battery voltage remains always above 12.2 V, with a discharge current remaining stable at around 220 mA and peaks due to radio transmission or battery heaters activating, the charge current obviously follows the sunlit/eclipse cycle and so has a trend similar to temperatures (fig. 8.6 and fig. 8.7) with peaks of up to 800 mA. Performing numerical computations on this sets of data, the author obtained an average power consumption of 3 W, with an average charge power of 3.2 W, resulting in a battery efficiency of 93 %.

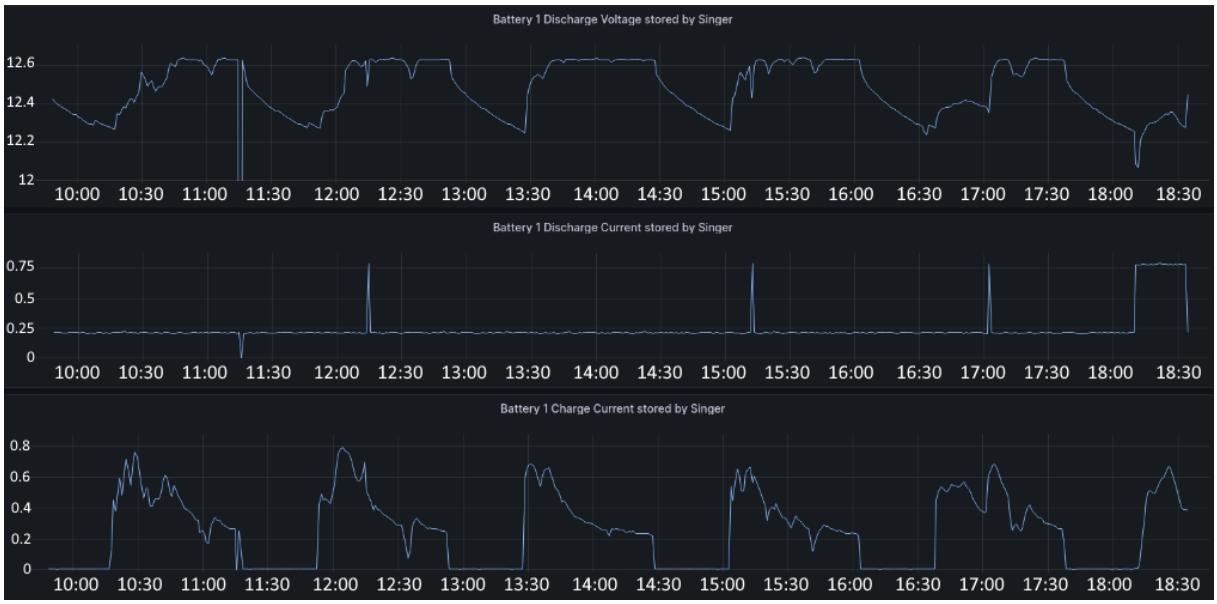



**Figure 8.8:** Parrot sensors battery voltage (up, [V]), discharge current (middle, [A]) and charge current (down, [A]) versus time.

In fig. 8.7 and fig. 8.8 we can also notice that there's a glitch on data at around 11:15, it's unclear what caused it and actually the CDH temperature shows two different glitches on two separate time instants, a SEU event in the MRAM can be excluded because the glitch only appears in Parrot sensors data and as said the MRAM should not suffer from upsets, according to the downlink data neither CDH nor Singer was rebooted but the time of 6 minutes between these glitches suggests that it can be related with Parrot (it was never addressed on this thesis but the Parrot system was designed in such a way as to turn on the CDH 2 after 6 minutes from the CDH 1 post deployment, so any time-driven event on the two CDHs should retain this delay) but the motivation remains unknown.
The magnetometer and gyroscope data can be seen in fig. 8.9, the magnetometer from MTi-3 outputs a vector with no unit normalized with respect to the magnetic field used for calibration, so the absolute value information is not given but only the field direction and a relative intensity.

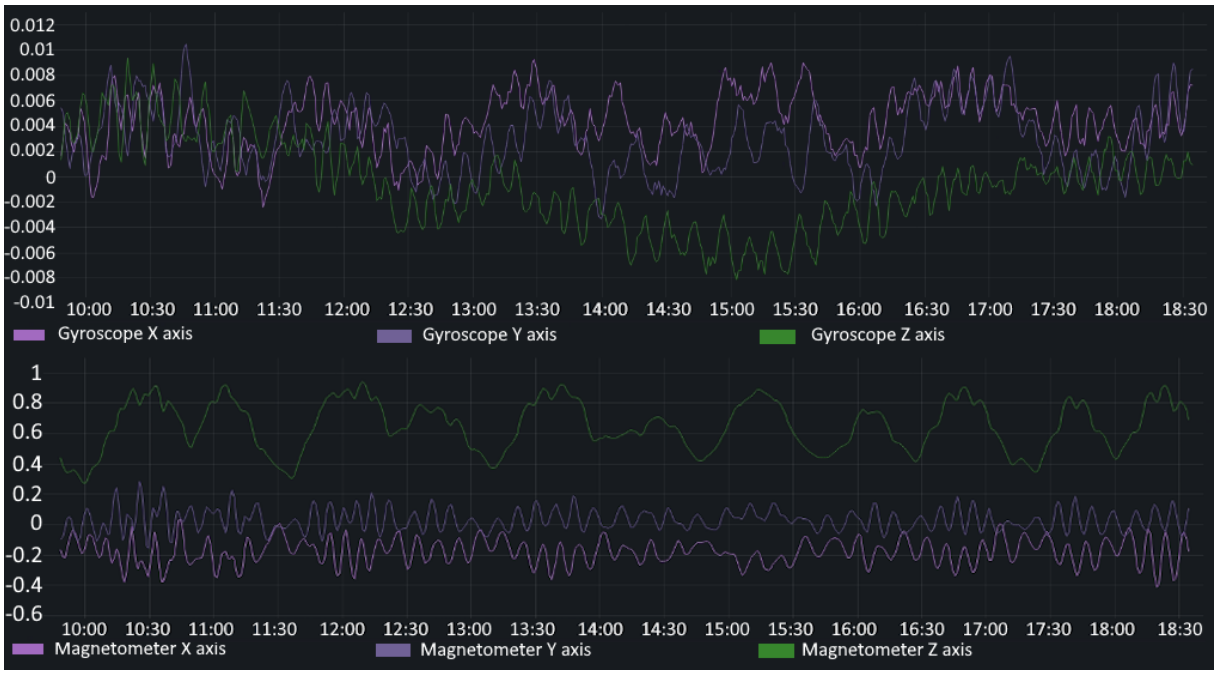



**Figure 8.9:** Singer IMU data versus time: gyroscope (up, [rad/s]) and magnetometer (down, normalized no unit).

Some things are quickly visible from the data: the passive ACS is doing its job of aligning the Z axis of the spacecraft to the magnetic field lines and in fact the Z component of the magnetic field is way higher than the other components, from the Z component we can also notice the difference in intensity with peaks near the poles and valleys near the equator, we can also see that the minimum at the equator has a slightly lower value in eclipse than in sunlit, probably due to the magnetic field deformation by the solar wind; at the same time the gyroscope (and also the magnetometer) data highlights that the attitude stabilization is not perfect and the spacecraft continuously oscillates with a period of around 7 minutes, not a problem for our application but surely impractical in cases in which an higher stability is needed, the gyroscope data is somehow chaotic and needs a deeper analysis to extract information about the spacecraft attitude behavior, what could be said is that (and this is true also for the other downlinks performed with weeks of distance) the angular velocities reach peaks of around 0.01 rad/s, or about 0.6 °/s (10:30 minutes per revolution), the gyroscope and magnetometer data follow low frequency trends on the X and Y components that could be somehow correlated with the orbits, with higher frequency oscillations superimposed, what's interesting to notice is that the average rotation around the Z axis changed direction over time. For a detailed analysis on the data from these sensors, the paper from the ACS team can be consulted: [24].
Lastly, many other data was retrieved which is useful to monitor the satellite health status, like the Parrot operative mode evolution over time and telemetry regarding Singer. Fig. 8.10 shows the Singer reboot counter over a period of around 1 month, this counter is increased after every reset, including the periodical automatic reboot performed every 12 hours, this revealed itself as a bad decision by the author, since the counting of the

automatic reboots makes it difficult to quickly find anomalous resets due to latchup events without analyzing the data in details, a better approach would have been to decrease the counter before every automatic reset or even better reserving a flag field on the memory table to signal the firmware that it was booting after an automatic reset, it wasn't implemented for design time constraints.

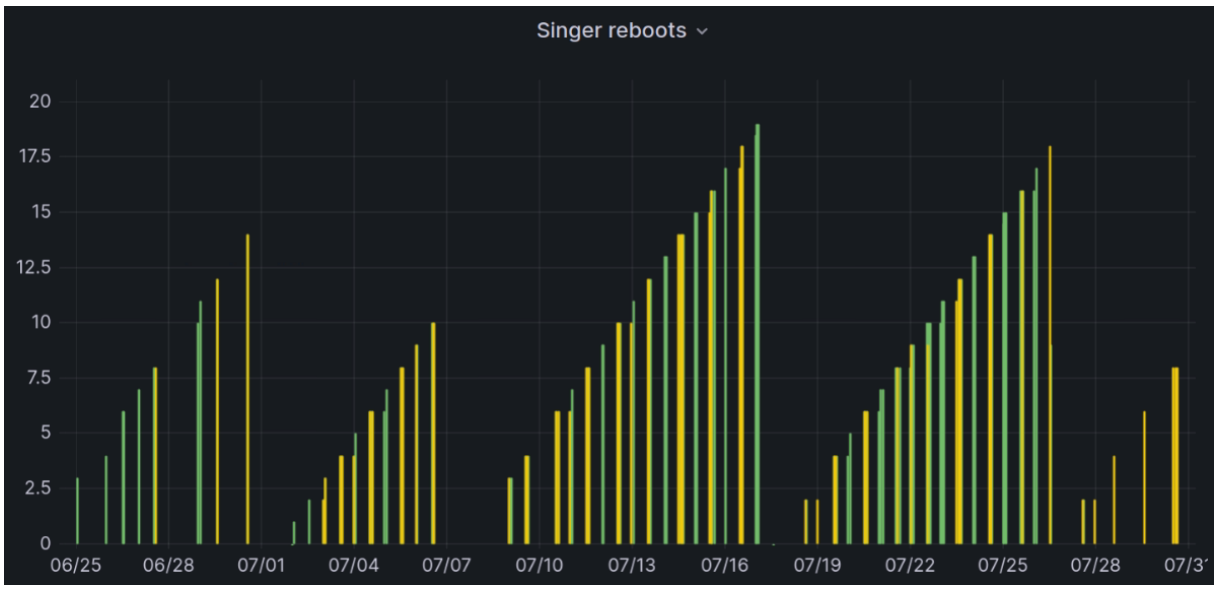



**Figure 8.10:** Singer reboots over the period from 25 June 2023 - 31 July 2023.

The data shows the reset counter going back to zero many times, this was due to the performed operations, which required setting Parrot operative mode to activation mode a number of times and each time it sent the memory reset command to Singer, it was then a proof of the correct working of this command, another view of this can be seen in fig. 8.11 and fig. 8.12, which show the CDH 1 and CDH 2 operative modes stored by Singer over the same time period.

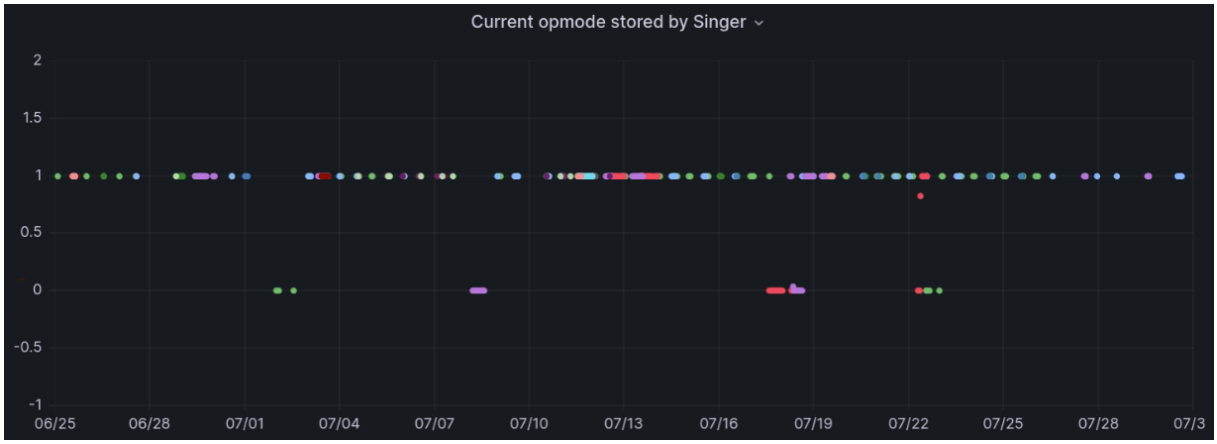



**Figure 8.11:** CDH 1 operative mode stored by Singer over the period from 25 June 2023 - 31 July 2023.

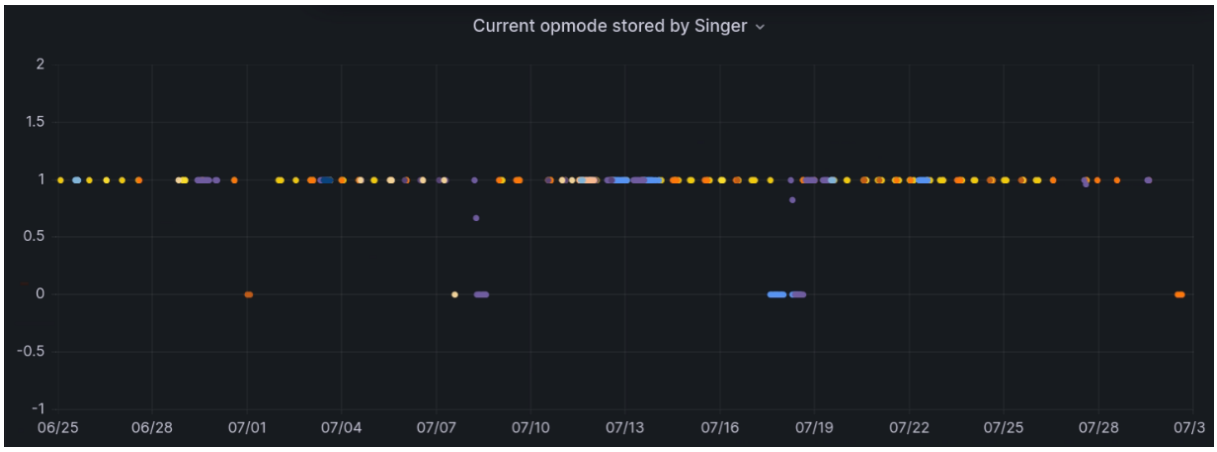



**Figure 8.12:** CDH 2 operative mode stored by Singer over the period from 25 June 2023 - 31 July 2023.

We can see how the opmode changed various times between 0 (activation, with consequent reset request to Singer) and 1 (commissioning); we can also notice that one time the Singer reset command failed (around 22 July on fig. 8.11), since the reboot counter was not reset; this can be related to an unfortunate reset of Singer during the message transmission from the CDH 1 or an error on the transmission channel, in any case the memory reset was just a nice-to-have to clear the memory after the first boot in orbit and for this reason we did not implement an acknowledge / resend mechanism that would be needed otherwise. Also in this case there are some corrupted data points, these have a decimal value and for this reason it's impossible for them to be a valid operative mode; again it's unclear if this is due to a SEU on memory, a sampling of Parrot data after a Singer or Parrot reboot or an error on the communication channel, in any case it's easy to recognize them as outsider points.
This was just an example of the useful support that Singer can provide to the mission management.

# Chapter 9
# Conclusions

The work of this thesis reached the objectives of designing and validating in space a low-cost system built entirely from COTS with the application of techniques for external radiation hardening and system resilience improvement, the system allowed collecting scientific data about the space environment and supported the satellite health monitoring. The system also demonstrated the usage of COTS MRAM in orbit for non-volatile storage of data, a memory technology which is promising for space applications due to its inherent resistance to radiation induced effects.
The mission demonstrated the feasibility of fast development of a CubeSat mission, confirming the Polytechnic of Turin as one of the leading italian universities on this front. Up to date the project produced two papers ([24], [47]) for the International Astronautic Congress (IAC) and some theses have already been written which made use of the data collected by Singer ([20], [58] from the thermal team students) with more to come in the near future.
At the same time this work has to be intended as a starting point for future development, the author identified some aspects that should be further developed:

- While the work did validate the developed On Board Computer in orbit, there was a lack of precise figures about the expected reliability of the system which requires a deeper research about the behavior in high radiation environment of each specific device; in this case that aspect was not a major concern due to the secondary nature of the system and the short development time but it would become so in case of future missions that plan to use this technology and know-how for vital elements of the spacecraft.

- Due to the strict time constraints, the system missed some opportunities to better exploit the hardware capabilities in favor of software and interface simplicity: the simple method for retrieval of packets, which can only be read backwards from the last measured, left out the possibility to address specific time instants of interest, like the post-deployment window; the in-orbit performance of MRAM banks could have been better investigated by running some dedicated experiments on portions of the memory, like for example a basic periodical scan to identify bit flips and derive the upset probability, a similar experiment was planned for the microcontroller memories but not implemented due to insufficient development time.

There are some possible improvements of the board that could increase its reliability

starting from the existing hardware:

- The first improvement could be to store the microcontroller code on the MRAM and so run the firmware from it, the MRAM should reduce the risk of Single Events Functional Interrupts (SEFI) due to its intrinsic higher resistance to SEU, again this should come with a deeper analysis on the risk of latch-up of the CMOS part of the memory and the feasibility of such configuration for the microcontroller, alternatively the MRAM could be used to store only a copy of the code to correct eventual upsets on the flash, exploiting the error detection capability of the STM32L4 series.

- Another big room for improvements is offered by the latch-up protection circuit, the actual implementation is quite basic and can be largely improved by: implementing a better load switch with a second n-type MOSFET for fast discharge of the supply capacitance, this could shorten by at least one order of magnitude the turn-off time and reduce the damage from a latch-up event by quickly starving the parasitic SCR; implementing a latching mechanism on the circuit to keep the domain powered off indefinitely in case of latch-up, signaling it to the microcontroller, this would allow a better control of voltage domains and gathering of data about latch-up events, as well as reduce the power consumption of latched-up or shorted domains; another improvement offered by a latching mechanism on the protection circuit would be to allow turning off some blocks voluntarily while not needed (like for example the MRAM, IMU and ADC that are basically needed for some seconds every minute), this would completely eliminate the possibility of latch-up on these blocks while turned off, greatly increasing their life expectancy (this was investigated during design and in fact at a point there were npn BJTs controlled by the micro in parallel with the open collector output of the comparators, but this solution was later discarded since there wasn't a latching mechanism and every reset of the microcontroller would have turned on again the domains).

- The design discussed on this thesis used an external dev-board for the microcontroller, basically doubling the board height; in our specific case space was not a major concern and we enjoyed the flexibility offered by this solution, in other cases this could be a problem and the natural evolution of the board would be to embed the microcontroller directly on the PCB, increasing the system compactness.

On the other end, some important lessons were learned from the negative events that happened during design:

- Failures most likely related with ESD events highlighted the importance of handling flight hardware with the maximum possible caution, especially with the integrated spacecraft, redundantly checking safety measures before every activity.

- The necessity to manually correct PCB errors highlighted the importance of resorting to multiple prototyping cycles before coming up with a clean final product.

- The bad experience with some manufacturers and component providers, which occurred not only for the Singer system, highlighted the importance of careful inspection of the hardware and avoid blindly trusting hardware provided by third parts.

In general, all this points highlight the necessity of a continuous R&D effort in order to be prepared for an eventual future mission without suffering the low development time problem.

Lastly, this mission had high educational implications and was an incredible opportunity for students to experience a real space mission, which is the important objective that was initially set at the birth of the CubeSat standard.
Personally speaking, the author was really enriched by this design experience and felt privileged of getting involved on a project of this relevance, he hopes that this thesis will help and encourage future students on their pursuit of space.

# Bibliography


[3] (Picture) The CSSWE CubeSat and PPOD just prior to integration, CSSWE at English Wikipedia, Accessed on 16 October 2023. [Online]. Available: https://commons.wikimedia.org/wiki/File:CSSWE_CubeSat_and_PPOD_prior_to_integration.png.

[6] E. Areda, M. Cho, J. R. Cordova Alarcon, and H. Masui, "Development of Innovative CubeSat Platform for Mass Production," *Applied Sciences*, vol. 12, Sep. 2022. DOI: {10.3390/app12189087}.

[7] ARI-Bra website, Accessed on 16 October 2023. [Online]. Available: https://www.aribra.it/.

[9] H. J. Barnaby, M. L. Mclain, I. S. Esqueda, and X. J. Chen, "Modeling ionizing radiation effects in solid state materials and CMOS devices," in *2008 IEEE Custom Integrated Circuits Conference*, 2008. DOI: {10.1109/CICC.2008.4672075}.

[10] R. Barry, Mastering the FreeRTOS™ Real Time Kernel - A Hands-On Tutorial Guide. 2016. [Online]. Available: https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf.

[11] E. L. Bella, "Development of an Electrical Power System for a 3U educational CubeSat," Politecnico di Torino, Dec. 2022. [Online]. Available: http://webthesis.biblio.polito.it/25150/.

[12] N. Bianco, "Design, development and verification of the Electric Propulsion Interface System for the CubeSat Test Platform," Politecnico di Torino, Apr. 2022. [Online]. Available: http://webthesis.biblio.polito.it/22718/.

[13] L. Bottini, A. Boschetto, F. Veniali, and P. Gaudenzi, "Next Generation CubeSats and SmallSats," in Jan. 2023, ch. Additive manufacturing for CubeSat structure fabrication, pp. 153–180, ISBN: 9780128245415. DOI: {10.1016/B978-0-12-824541-5.00025-X}.

[14] A. Busso, M. Mascarello, S. Corpino, F. Stesina, and R. Mozzillo, "The communication module on-board E-ST@R-II cubesat," in *7th ESA International Workshop on Tracking, Telemetry and Command Systems for Space Applications, TTC 2016*, 2016. [Online]. Available: https://hdl.handle.net/11583/2704674.

[15] CESI involved in the Vatican's first satellite, Accessed on 16 October 2023. [Online]. Available: https://www.cesi.it/news/2023/cesi-involved-in-the-vaticans-first-satellite/.

[16] C. Z. Chen, D. Y. Hu, and H. Wu, "Analysis of ESD Effect and Ionizing Radiation Particles in Gate Oxide," in *2020 China Semiconductor Technology International Conference (CSTIC)*, 2020. DOI: {10.1109/CSTIC49141.2020.9282445}.

[17] F. C. Ciccotti, "Design and development of the optical navigation system for Cubesat in support of the Martian rover mission," Dec. 2022. [Online]. Available: http://webthesis.biblio.polito.it/25147/.

[18] C. Conigliaro, D. Calvi, L. Franchi, F. Stesina, and S. Corpino, "DESIGN AND ANALYSIS OF AN INNOVATIVE CUBESAT THERMAL CONTROL SYSTEM FOR BIOLOGICAL EXPERIMENT IN LUNAR ENVIRONMENT," in *International Astronautical Congress*, 2018. [Online]. Available: https://hdl.handle.net/11583/2765530.

[19] S. Corpino, S. Chiesa, F. Stesina, and N. Viola, "CubeSats development at Politecnico di Torino: The e-st@r program," in *61st International Astronautical Congress 2010*, 2010, pp. 8321–8328. [Online]. Available: https://hdl.handle.net/11583/2885392.

[20] D. Cosenza, "Development of a Tool for the Design and Verification of Thermal Control Systems of Small Sats," Politecnico di Torino, Jul. 2023. [Online]. Available: http://webthesis.biblio.polito.it/27928/.

[22] CubeSat official website, Accessed on 16 October 2023. [Online]. Available: https://www.cubesat.org/.

[23] CubeSatShop FAQs page, Accessed on 16 October 2023. [Online]. Available: https://www.cubesatshop.com/frequently-asked-questions/.

[24] A. D'Ortona, F. Manconi, F. Stesina, and S. Corpino, "Passive attitude stabilization strategy for a 3U student CubeSat," IAC-23,E2,4,12,x79994, 74th International Astronautical Congress 2023, Oct. 2023.

[25] Daisy-Chaining SPI Devices, AN3947, Maxim Integrated, Dec. 2006. [Online]. Available: https://www.analog.com/media/en/technical-documentation/tech-articles/daisychaining-spi-devices.pdf.

[27] P. Dodd and L. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Transactions on Nuclear Science*, 2003. DOI: {10.1109/TNS.2003.813129}.

[28] e-st@r Gunter's Space page, Accessed on 16 October 2023. [Online]. Available: https://space.skyrocket.de/doc_sdat/e-star.htm.

[29] e-st@r-II SatNOGS DB page, Accessed on 16 October 2023. [Online]. Available: https://db.satnogs.org/satellite/41459.

[30] D. Eakman, R. Lambeck, M. Mackowski, and J. Slifer L., "Small spacecraft power and thermal subsystems," National Aeronautics and Space Administration, Tech. Rep., 1994. [Online]. Available: https://core.ac.uk/download/pdf/42781666.pdf.

[31] Electrical Short Circuits due to Tin Whiskers, NASA Lessons Learned system, lesson 6956, 2013. [Online]. Available: https://llis.nasa.gov/lesson/6956.

[32] Encyclopedia Astronautica, Accessed on 16 October 2023. [Online]. Available: http://www.astronautix.com/.

[33] ESA Radiation Test Database, Accessed on 16 October 2023. [Online]. Available: https://esarad.esa.int/.

[34] FreeRTOS website, Accessed on 16 October 2023. [Online]. Available: https://www.freertos.org/index.html.

[35] A. Gili, "Mission Analysis and Trajectory Design for Space Rider Observer Cube," Politecnico di Torino, Jul. 2023. [Online]. Available: http://webthesis.biblio.polito.it/27930/.

[36] GNURadio website, Accessed on 16 October 2023. [Online]. Available: https://www.gnuradio.org/.

[37] Gpredict website, Accessed on 16 October 2023. [Online]. Available: http://gpredict.oz9aec.net/.

[38] Gqrx website, Accessed on 16 October 2023. [Online]. Available: https://gqrx.dk/.

[39] GSFC Radiation Data Base, Accessed on 16 October 2023. [Online]. Available: https://radhome.gsfc.nasa.gov/radhome/raddatabase/raddatabase.html.

[43] G. Holzmann, "The power of 10: rules for developing safety-critical code," *Computer*, vol. 39, no. 6, pp. 95–99, 2006. DOI: {10.1109/MC.2006.212}.

[44] IEEE Workshop on Radiation Effects Data, Accessed on 16 October 2023. [Online]. Available: https://ieeexplore.ieee.org/xpl/conhome/8086/proceeding.

[46] L. Iossa, A. Gili, D. Parrinello, *et al.*, "3U CubeSat mission to assess vegetation hydration status and hydrological instability risk," IAC-22,E2,3-GTS.4,1,x72036, 73rd International Astronautical Congress 2022, 2022.

[47] L. Iossa, T. Giovara, V. Calabretta, *et al.*, "From design to delivery in three months: the fast development of a 3U CubeSat," IAC-23,D1,4B,10,x79829, 74th International Astronautical Congress 2023, Oct. 2023.

[48] H. Jason, MRAM Technology Status, JPL-Publ-13-3, NASA Jet Propulsion Laboratory, Feb. 2013.

[49] M. Johnson, R. Cline, S. Ward, and J. Schichl, Latch-Up, SCAA124, Texas Instruments, Apr. 2015. [Online]. Available: https://www.ti.com/lit/wp/scaa124/scaa124.pdf.

[50] A. Johnston and S. Guertin, "The effects of space radiation on linear integrated circuits," in *2000 IEEE Aerospace Conference. Proceedings (Cat. No.00TH8484)*, 2000. DOI: {10.1109/AERO.2000.878509}.

[51] A. Ju, H. Guo, L. Ding, *et al.*, "Analysis of Ion-Induced SEFI and SEL Phenomena in 90 nm NOR Flash Memory," *IEEE Transactions on Nuclear Science*, 2021. DOI: {10.1109/TNS.2021.3105998}.

[52] KiCad EDA website, Accessed on 16 October 2023. [Online]. Available: https://www.kicad.org/.

[53] C. Leonard, Challenges for Electronic Circuits in Space Applications, Analog Devices, 2017. [Online]. Available: https://www.analog.com/media/en/technical-documentation/tech-articles/thought-leadership/challenges-for-electronic-circuits-in-space-applications.pdf.

[54] LICIACube mission ASI webpage, Accessed on 16 October 2023. [Online]. Available: https://www.asi.it/en/planets-stars-universe/solar-system-and-beyond/liciacube/.

[56] LTspice website, Accessed on 16 October 2023. [Online]. Available: https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html.

[57] LTspice: Worst-Case Circuit Analisys with Minimal Simulations Runs, Accessed on 16 October 2023. [Online]. Available: https://www.analog.com/en/technical-articles/ltspice-worst-case-circuit-analysis-with-minimal-simulations-runs.html.

[58] F. Lucia, "Development of a tool for thermal analysis of small spacecrafts.," Politecnico di Torino, Jul. 2023. [Online]. Available: http://webthesis.biblio.polito.it/27931/.

[59] C. Lughi, "Development and validation of a test bench for Attitude Determination and Control System for small satellite," Politecnico di Torino, Apr. 2023. [Online]. Available: http://webthesis.biblio.polito.it/26470/.

[60] MarCO (Mars Cube One) website, Accessed on 16 October 2023. [Online]. Available: https://science.nasa.gov/mission/marco.

[61] MATLAB website, Accessed on 16 October 2023. [Online]. Available: https://ch.mathworks.com/products/matlab.html.

[63] MT Software Suite website, Accessed on 16 October 2023. [Online]. Available: https://www.movella.com/support/software-documentation.

[66] NTC Thermistors, 29053, Revision: 27-Jan-2021, Vishay BCcomponents, Jan. 2021. [Online]. Available: https://www.vishay.com/docs/29053/ntcappnote.pdf.

[69] G. Obiols Rabasa, S. Corpino, R. Mozzillo, and F. Stesina, "Lessons learned of a systematic approach for the e-st@r-II CubeSat environmental test campaign," in *66th IAC International Astronautical Congress*, 2015. [Online]. Available: https://hdl.handle.net/11583/2625714.

[70] PC/104 Consortium website, Accessed on 16 October 2023. [Online]. Available: https://pc104.org/.

[74] PicPoT Gunter's Space page, Accessed on 16 October 2023. [Online]. Available: https://space.skyrocket.de/doc_sdat/picpot.htm.

[75] C. Poivey, Radiation Hardness Assurance for Space System, NASA GSFC, Jul. 2002. [Online]. Available: https://ntrs.nasa.gov/api/citations/20020080842/downloads/20020080842.pdf.

[76] PuTTY website, Accessed on 16 October 2023. [Online]. Available: https://www.putty.org/.

[77] Python web page, Accessed on 16 October 2023. [Online]. Available: https://www.python.org/.

[78] Qseven standard webpage, Accessed on 16 October 2023. [Online]. Available: https://sget.org/standards/qseven/.

[79] Raspberry Pi onboard computer for AAReST mission (Raspberry Pi website), Accessed on 16 October 2023. [Online]. Available: https://www.raspberrypi.com/news/compute-module-cubesats/.

[80] RTD embedded technologies website, PC/104 page, Accessed on 16 October 2023. [Online]. Available: https://www.rtd.com/PC104/.

[81] M. Sejera, T. Yamauchi, N. C. Orger, Y. Otani, and M. Cho, "Scalable and Configurable Electrical Interface Board for Bus System Development of Different CubeSat Platforms," *Applied Sciences*, vol. 12, no. 18, 2022. DOI: {10.3390/app12188964}.

[84] Software UART using ST7LITE0 12-bit autoreload timer, AN1753, STMicroelectronics, Mar. 2013. [Online]. Available: https://www.st.com/resource/en/application_note/an1753-software-uart-using-st7lite0-12bit-autoreload-timer-stmicroelectronics.pdf.

[85] SOLIDWORKS website, Accessed on 16 October 2023. [Online]. Available: https://www.solidworks.com/.

[86] Spei Satelles Gunter's Space page, Accessed on 16 October 2023. [Online]. Available: https://space.skyrocket.de/doc_sdat/spei-satelles.htm.

[87] Spei Satelles mission website, Accessed on 16 October 2023. [Online]. Available: https://www.speisatelles.org/.

[88] SPEISAT SatNOGS DB page, Accessed on 16 October 2023. [Online]. Available: https://db.satnogs.org/satellite/RINJ-3108-9789-8075-5047.

[89] S. Speretta, "Project solutions for low-cost space missions," PhD thesis, Politecnico Di Torino, 2010. [Online]. Available: https://zerorobotics.polito.it/app/uploads/2019/04/thesis-1.pdf.

[90] State-Of-The-Art Small Spacecraft Technology, NASA/TP-2021-0021263, NASA Ames Research Center, Small Spacecraft Systems Virtual Institute, Oct. 2021.

[91] State-Of-The-Art Small Spacecraft Technology, NASA/TP-2022-0018058, NASA Ames Research Center, Small Spacecraft Systems Virtual Institute, Jan. 2023.

[92] F. Stesina, "Validation of a Test Platform to Qualify Miniaturized Electric Propulsion Systems," *Aerospace*, vol. 6, p. 99, 9 2019. DOI: https://doi.org/10.3390/aerospace6090099.

[93] F. Stesina, "Tracking Model Predictive Control for Docking Maneuvers of a CubeSat with a Big Spacecraft," *Aerospace*, vol. 8, p. 197, 2021. DOI: https://doi.org/10.3390/aerospace8080197.

[94] F. Stesina and S. Corpino, "In Orbit Operations of an Educational Cubesat: the e-st@r-II Experience," *International Review of Aerospace Engineering*, no. 13, pp. 40–50, 2020. DOI: https://dx.doi.org/10.15866/irease.v13i2.18317. [Online]. Available: https://hdl.handle.net/11583/2837831.

[95] F. Stesina, S. Corpino, R. Mozzillo, and G. Obiols Rabasa, "Design of the Active Attitude Determination and Control System for the e-st@r cubesat," in *63rd International Astronautical Congress*, 2012. [Online]. Available: https://hdl.handle.net/11583/2503388.

[97] STM32CubeIDE website, Accessed on 16 October 2023. [Online]. Available: https://www.st.com/en/development-tools/stm32cubeide.html.

[98] STM32CubeMX website, Accessed on 16 October 2023. [Online]. Available: https://www.st.com/en/development-tools/stm32cubemx.html.

[100] M. A. Swartwout, "A brief history of rideshares (and attack of the CubeSats)," in *2011 Aerospace Conference*, 2011. DOI: {10.1109/AERO.2011.5747233}.

[101] M. A. Swartwout, "A statistical survey of rideshares (and attack of the CubeSats, part deux)," in *2012 IEEE Aerospace Conference*, 2012. DOI: {10.1109/AERO.2012.6187008}.

[102] M. A. Swartwout, "Cheaper by the dozen: The avalanche of rideshares in the 21st century," in *2013 IEEE Aerospace Conference*, 2013. DOI: {10.1109/AERO.2013.6497182}.

[103] M. A. Swartwout, "Secondary spacecraft in 2016: Why some succeed (And too many do not)," in *2016 IEEE Aerospace Conference*, 2016. DOI: {10.1109/AERO.2016.7500791}.

[104] M. A. Swartwout, "You say "Picosat", I say "'CubeSat": Developing a better taxonomy for secondary spacecraft," in *2018 IEEE Aerospace Conference*, 2018. DOI: {10.1109/AERO.2018.8396755}.

[105] M. A. Swartwout, "Cubesats/Smallsats/Nanosats/Picosats/Rideshare(sats) in 2022: Making Sense of the Numbers," in *2022 IEEE Aerospace Conference (AERO)*, 2022. DOI: {10.1109/AERO53065.2022.9843832}.

[106] "The evolution of cubesat spacecraft platforms," NATO S&T organization.

[110] F. Topputo, F. Ferrari, V. Franzese, *et al.*, "The Hera Milani Cubesat Mission," in *7th IAA Planetary Defense Conference (PDC 2021)*, 2021, pp. 1–2. [Online]. Available: https://hdl.handle.net/11311/1171933.

[112] T. Tumenjargal, S. Kim, H. Masui, and M. Cho, "CubeSat bus interface with Complex Programmable Logic Device," *Acta Astronautica*, 2019. DOI: https://doi.org/10.1016/j.actaastro.2019.04.047.

[114] Using the Serial Peripheral Interface to Communicate Between Multiple Microcomputers, AN991/D, Rev. 1, 1/2002, Freescale Semiconductor, Jan. 2002. [Online]. Available: https://www.nxp.com/docs/en/application-note/AN991.pdf.

[115] S. Vartanian, F. Irom, G. R. Allen, W. P. Parker, and M. D. O'Connor, "Single Event Latchup Results for COTS Devices Used on SmallSat Missions," in *2020 IEEE Radiation Effects Data Workshop (in conjunction with 2020 NSREC)*, 2020. DOI: {10.1109/REDW51883.2020.9325824}.

[116] Visual Studio Code website, Accessed on 16 October 2023. [Online]. Available: https://code.visualstudio.com/.

[117] R. Walker, CUBESAT EVOLUTION: FROM EDUCATIONAL TOOLS TO AUTONOMOUS SPACE DRONES & BEYOND, Presentation to $8^th$ ECS, Sep. 2016. [Online]. Available: https://a3space.org/wp-content/uploads/2017/10/CubeSat-Evolution.pdf.

[118] A. C. Watkins, S. T. Vibbert, J. V. D'Amico, *et al.*, "Mitigating Total-Ionizing-Dose-Induced Threshold-Voltage Shifts Using Back-Gate Biasing in 22-nm FD-SOI Transistors," *IEEE Transactions on Nuclear Science*, 2022. DOI: {10.1109/TNS.2022.3146318}.

[119] Why Space Radiation Matters (NASA website), Accessed on 16 October 2023. [Online]. Available: https://www.nasa.gov/analogs/nsrl/why-space-radiation-matters.

[120] World's largest database of nanosatellites, almost 3500 nanosats and CubeSats, Accessed on 16 October 2023. [Online]. Available: https://www.nanosats.eu/.

[121] M. Yingqi, H. Jianwei, S. ShiPeng, *et al.*, "SEE Characteristics of COTS Devices by 1064nm Pulsed Laser Backside Testing," in *2018 IEEE Radiation Effects Data Workshop (REDW)*, 2018. DOI: {10.1109/NSREC.2018.8584271}.

[122] R. Zeif, A. Hormer, M. Kubicka, M. Henkel, and O. Koudelka, "From OPS-SAT to PRETTY Mission: A Second Generation Software Defined Radio Transceiver for Passive Reflectometry," in *2020 International Conference on Broadband Communications for Next Generation Networks and Multimedia Applications (CoBCom)*, Jul. 2020, pp. 1–8. DOI: {10.1109/CoBCom49975.2020.9174103}.

# Technical documents


[1] .050 X .050 TERMINAL STRIP, TFM-1XX-XX-XXX-D-XXX-X-X, Rev. FT, Samtec, 2014. [Online]. Available: https://suddendocs.samtec.com/prints/tfm-1xx-xx-xxx-d-xxx-x-xx-mkt.pdf.

[2] .050[1.27] DISCRETE WIRE INSULATOR & LATCH ASM, ISDF-XX-D-X, Rev. Q, Samtec, 2006. [Online]. Available: https://suddendocs.samtec.com/prints/isdf-xx-d-x-mkt.pdf.

[4] 16-channel analog multiplexer/demultiplexer, 74HC4067; 74HCT4067, Rev. 8, Nexperia, 2021. [Online]. Available: https://assets.nexperia.com/documents/data-sheet/74HC_HCT4067.pdf.

[5] 3.3V 20Mbps RS485/RS422 Transceivers, LTC2850/LTC2851/LTC2852, LT 0615 REV E, Linear Technology, 2007. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/285012fe.pdf.

[8] ARM® Debug Interface v5 - Architecture Specification, IHI0031A, ARM Limited, Feb. 2006. [Online]. Available: https://developer.arm.com/documentation/ihi0031/a/.

[21] Cubesat Design Specification, CP-CDS-R14.1, Rev. 14.1, The cubesat Program, Cal Poly SLO, 2022. [Online]. Available: https://static1.squarespace.com/static/5418c831e4b0fa4ecac1bacd/t/62193b7fc9e72e0053f00910/1645820809779/CDS+REV14_1+2022-02-09.pdf.

[26] Description of STM32L4/L4+ HAL and low-layer drivers, UM1884, Rev. 9, STMicroelectronics, Sep. 2021. [Online]. Available: https://www.st.com/resource/en/user_manual/um1884-description-of-stm32l4l4-hal-and-lowlayer-drivers-stmicroelectronics.pdf.

[40] Hardware Integration Manual - MTi 1-series, MT1503P, Revision 2019.A, Xsens Technologies, 2019. [Online]. Available: https://www.xsens.com/hubfs/Downloads/Manuals/Hardware_Integration_Manual_MTi_1-series.pdf.

[41] HEXFET® Power MOSFET, IRF7342PbF, Rev. 2016-5-26, Infineon Technologies, 2016. [Online]. Available: https://www.mouser.it/datasheet/2/196/Infineon_IRF7342_DS_v01_01_EN-3166065.pdf.

[42] High Performance Serial Persistent SRAM Memory, AS1001204, AS1004204, AS1008204, AS1016204, AS3001204, AS3004204, AS3008204, AS3016204, Rev. S, Avalanche Technology, 2022. [Online]. Available: https://www.avalanche-technology.com/wp-content/uploads/1Mb-16Mb-Serial-HP-MRAM-S_SD-10_20_2022.pdf.

[45] INA1x8 High-Side Measurement Current Shunt Monitor, INA138, INA168, SBOS122E Rev. December 2017, Texas Instruments, 2017. [Online]. Available: https://www.ti.com/lit/ds/symlink/ina138.pdf?ts=1695306220346.

[55] Low Power, 16-/24-Bit,Sigma-Delta ADCs, AD7788/AD7789, Rev. C, Analog Devices. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/AD7788_7789.pdf.

[62] MT Low Level Communication Protocol Documentation, MT0101P, Revision 2020.A, Xsens Technologies, Jun. 2020. [Online]. Available: https://www.xsens.com/hubfs/Downloads/Manuals/MT_Low-Level_Documentation.pdf.

[64] MTi 1-series Datasheet, MT0512P, Revision 2019.A, Xsens Technologies, 2019. [Online]. Available: https://www.xsens.com/hubfs/Downloads/Manuals/MTi-1-series-datasheet.pdf.

[65] MTi User Manual - MTi 10-series and MTi 100-series 5th Generation, MT0605P, Revision 2020.A, Xsens Technologies, 2020. [Online]. Available: https://www.xsens.com/hubfs/Downloads/usermanual/MTi_usermanual.pdf.

[67] NTC Thermistors, Radial Leaded, Accuracy Line, NTCLE203E3, 29048 Rev. 05-Jul-2022, Vishay, 2022. [Online]. Available: https://www.vishay.com/docs/29048/ntcle203.pdf.

[68] NUCLEO-XXXXRX, MB1136, Rev. C-05, STMicroelectronics, Aug. 2022. [Online]. Available: https://www.st.com/content/ccc/resource/technical/layouts_and_diagrams/schematic_pack/group2/5a/85/d6/9a/34/e2/47/1d/MB1136-DEFAULT-C05_Schematic/files/MB1136-DEFAULT-C05_Schematic.pdf/jcr:content/translations/en.MB1136-DEFAULT-C05_Schematic.pdf.

[71] PC/104 Specification, Version 2.6, PC/104 Embedded Consortium, Oct. 2008. [Online]. Available: https://pc104.org/wp-content/uploads/2015/02/PC104_Spec_v2_6.pdf.

[72] PCI-104 Specification, Version 1.0, PC/104 Consortium, Nov. 2003. [Online]. Available: https://resources.winsystems.com/specs/PCI-104Spec_v1_0.pdf.

[73] PCI/104-Express™ & PCIe/104™ Specification, Version 3.0, PC/104 Consortium, Feb. 2015. [Online]. Available: https://pc104.org/wp-content/uploads/2015/03/PCI104_Express_v3_0.pdf.

[82] SN74AVC4T245 Dual-Bit Bus Transceiver with Configurable Voltage Translation and 3-State Outputs, SN74AVC4T245, SCES576G Rev. November 2014, Texas Instruments, 2014. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74avc4t245.pdf?ts=1695371287154.

[83] SN74AXC1T45 Single-Bit Dual-Supply Bus Transceiver With Configurable Voltage Translation, SN74AXC1T45, SCES882D Rev. October 2021, Texas Instruments, 2021. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74axc1t45.pdf?ts=1695390961054.

[96] STM32 Nucleo-64 boards (MB1136), UM1724, Rev. 14, STMicroelectronics, 2020. [Online]. Available: https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf.

[99] STM32L41xxx/42xxx/43xxx/44xxx/45xxx/46xxx advanced Arm®-based 32-bit MCUs, RM0394, STMicroelectronics, Oct. 2018. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0394-stm32l41xxx42xxx43xxx44xxx45xxx46xxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.

[107] The FreeRTOS™ Reference Manual - API Functions and Configuration Options, Version 10.0.0 issue 1, Amazon Web Services, 2017. [Online]. Available: https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf.

[108] TL331B, TL391B and TL331 Single Comparators, TL331, TL331B, TL391B, SLVS238J Rev. November 2020, Texas Instruments, 2020. [Online]. Available: https://www.ti.com/lit/ds/symlink/tl331.pdf?ts=1695362356100.

[109] TL431LI / TL432LI Programmable Shunt Regulator with Optimized Reference Current, TL431LI, TL432LI, SLVSDQ6A Rev. November 2018, Texas Instruments, 2018. [Online]. Available: https://www.ti.com/lit/ds/symlink/tl432li.pdf?ts=1695359027933.

[111] Traco Power TMR 3WIR web page, Accessed on 16 October 2023. [Online]. Available: https://www.tracopower.com/int/it/series/tmr-3wir.

[113] Ultra-low-power Arm® Cortex®-M4 32-bit MCU+FPU, 100DMIPS, up to 512KB Flash, 160KB SRAM, analog, audio, ext. SMPS, STM32L452xx, DS11912 Rev. 7, STMicroelectronics, 2020. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32l452re.pdf.

# Acknowledgements (Italian)

Innanzi tutto, un doveroso ringraziamento al Professor Stesina, che mi ha accolto nella famiglia dello STARLAB, credendo fin da subito nel mio potenziale e mettendo a disposizione la sua esperienza per tutti i progetti di cui ho fatto parte, senza il quale Singer non avrebbe mai visto la luce; così come la Professoressa Corpino, a cui devo non solo l' incredibile opportunità di lavorare al progetto SPEISAT ma anche l' enorme regalo di poter assistere al lancio di persona in una delle esperienze più incredibili della mia vita.
Voglio ringraziare di cuore i miei genitori, che mi hanno permesso di seguire questo percorso sostenendomi sempre, così come i miei fratelli e i miei nonni, che sono sempre stati fieri di me e hanno sempre saputo capire il mio essere particolare, così come tutto il resto della mia famiglia. Un ciao in particolare a mia nonna Gabry, che non c'è più da molti anni ma ha piantato in me i semi della creatività e manualità, spingendomi a diventare quello che sono oggi.
Ringrazio gli amici di una vita, con cui sono cresciuto giocando a calcio nel fango fino a raggiungere tutti quanti importanti risultati.
Un saluto,un ringraziamento e un "in bocca al lupo" a tutti i ragazzi del CubeSat team e del team Diana, con cui abbiamo affrontato qualcosa che va ben oltre il semplice percorso di studi e l'abbiamo sempre fatto tra mille risate.
Infine, il grazie più grande va ad Alessia, sempre al mio fianco in questo percorso difficile mentalmente e fisicamente, tu sei stata la mia forza e mi hai aiutato nei momenti più difficili, non so come avrei fatto senza di te, questo risultato è dedicato a te.