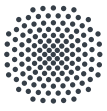

University of Stuttgart
Germany

Institute for Architecture of Application Systems (IAAS)

University of Stuttgart
Universitätsstrasse 38
70569, Stuttgart

Master Thesis

Design of a Software Architecture for Supervisor System in a Satellite

Sarthak Kelapure

Course of Study:	Information Technology (InfoTech) Embedded Systems Engineering
Examiner:	Prof. Dr. Marco Aiello
Supervisors:	Dr.-Ing. Felix Böhringer, Dr.-Ing. Fabian Steinmetz, Ing. Gustavo Ambrosio, Thales Alenia Space, Germany.
Matriculation Number:	3375004
Commenced:	02.06.2020
Completed:	10.11.2020

Acknowledgement

I would like to thank SDR Team at Thales Alenia Space, Germany for giving me this opportunity to pursue my Master Thesis with them. A big thanks to Gustavo Ambrosio, Dr. Fabian Steinmetz, Jens Wieczorek, Michele Belloti, Alexander Pawlitzki, Dr. Felix Boehringer, and the entire SDR team for their constant support and guidance. I would also thank my colleagues Satheesh Konduru and Siddarth Vasudevan for their motivation and support during the thesis.

Thank Prof. Dr. Marco Aiello for examining my thesis work and reviewing my steps during this process.

In the end, I would like to thank my family and my friends for believing in me and helping me through this Master's degree course.

Abstract

Internet of Things (IoT) is not just a word now. With an estimated 30 billion devices in the world by 2020, IoT has already become what it was envisioned when the trend began. But there is still a hustle from companies around the world for better and better user experience because the technology keeps getting upgraded and need for upgrade never stops for the user. Researchers and scientists are trying everyday to improve the experience by improving the involved things and by improving the communication means, "the internet". One such means of communication expected to grow in the future is satellite communication for IoT.

Satellite to be used for this purpose needs to be low-cost, robust, reliable, and future ready. An improvement in satellite architecture is imminent. For making satellite feature rich and robust but still low-cost means increase in mission-life of a satellite. Like human life, this can be achieved by better medical system for the satellites. With introduction of a *doctor on board*, the thesis aims to propose solution for improved mission-life and features for the satellite.

The *doctor on board* in this case is called Supervisor system. This system will need to have a robust and modular software on its designated hardware. Software can be designed and developed to be robust using a standard software architecture that is promising while complementing the requirements. The thesis focuses on designing software architecture for this "Supervisor system".

By the end of this thesis, the author designs a software architecture for the said system after study of similar architectures. The software architecture is used to develop important features of the mission and is tested for its portability and modularity. Future needs and changes to the existing system are also foreseen and discussed in the end.

Contents

List of Acronyms

ADC	Analog to Digital Converter.
API	Application Programming Interface.
ARM	Advanced RISC Machines.
BSP	Board support package.
CAN	Controller Area Network.
CCSDS	Consultative Committee for Space Data Systems.
COTS	Commercial off-the-shelf.
CPU	Central Processing Unit.
CRC	Cyclic Redundancy Check.
DAC	Digital to Analog converter.
DMA	Direct Memory Access.
DtS-IoT	Direct to Satellite IoT.
EDAC	Error Detection and Correction module.
FIFO	First In First Out.
FPGA	Field Programmable Gate Array.
FPU	Floating Point Unit.
GEO	Geostationary Earth orbit.
GPIO	General Purpose Input Output.
GPS	Global Positioning system.
I2C	Inter-Integrated Circuit.
IIoT	Industrial Internet of Things.
IoMT	Internet of Medical Things.
IoT	Internet of Things.
ISM	Industrial, scientific and medical.
JTAG	Joint Test Action Group.

LDO	Low Dropout Regulator.
LEO	Low Earth orbit.
LPWA	Low Power Wireless Area.
M2M	Machine to Machine.
MEO	Medium Earth orbit.
MPSoC	Multiprocessor System on Chip.
MPU	Memory Protection Unit.
MQTT	Message Queue Telemetry Transport.
MRAM	Magnetoresistive random-access memory.
NVM	Non Volatile Memory.
OBC	On-board Computer.
OSI	Open systems interconnect.
PCB	Printed Circuit board.
PWM	Pulse Width Modulation.
RAM	Random Access Memory.
RTOS	Real time operating system.
SEB	Single Event Burnout.
SEE	Single Event Effect.
SEFI	Single-event functional interrupts.
SEL	Single Event Latch-up.
SET	Single Event Transient.
SEU	Single Event Upset.
SoC	System on Chip.
SPI	Serial Peripheral Interface.
SRAM	Static Random Access memory.
SWD	Serial Wire Debug.
TC	Telecommands.
TM	Telemetry.
TTC	Telemetry, Tracking, and Commanding.
UART	Universal Asynchronous Receiver Transmitter.
UML	Unified Modelling Language.
V2V	Vehicle to Vehicle.
V2X	Vehicle to Everything.

List of Figures

List of Tables

Chapter 1

Introduction

Internet of Things (IoT) is nowadays a buzzword and will pertain for the coming time also. Everyday scientist, researchers, and engineers are developing, and testing new ways to make the IoT experience richer, smoother and better for the users. IoT devices demands low power, highly modular, and highly independent functioning.
There is an everyday problem for developers, to make IoT devices capable of reaching millions of users around the world with low latency. Connectivity for these IoT devices can be a bottleneck.
Currently, the world is flooded with IoT devices flourishing the market with various applications. Since communication is one of the most important aspect of any IoT device, lot of research is being put in this field. In the future, it is predicted [1] that satellite communication for IoT device will be heavily in use. IoT device can leverage on the fact that such a satellite communication will be available everywhere in the world with just one installation stage. *But are satellites ready for the the IoT world?*
As the use of satellite increases the need for a robust and long lasting satellite increases too. Software for the satellite remains an important part of the development and continuous research is being done to improve the software and overall robustness of a satellite while keeping the costs of development and deployment low. To achieve low costs, Commercial off-the-shelf (COTS) products are being used as components in space-missions are big driver in cost of a mission. Such products can be visualised as module with hardware and integrated software which have some specific functions with little or no reconfigurations.
For classical satellites, the system is built for maximum availability without any reboot or reset. Hence, hardware used is already latch-up free and radiation hardened. But for the IoT world, data losses are always considered in the system hence here latch-up mitigation techniques are considered. The primary goal shifts from availability to surviving in the space allowing use of cheaper hardware and software. The factor of availability is compensated by increasing the number of satellites.

1.1 Problem statement

Satellite relies on robust hardware and reliable software. For a safe and long lasting space missions, companies spend a lot of money on their hardware and software. The most important software in a satellite is on-board software running on a On-board Computer (OBC) which is characterised by a very secure and well structured software architecture. The software runs on a costly COTS module that is designed with high standards, deep research, and structured development. The software architecture is designed to be comparable to other large scale software architectures, such as air traffic controller. [2]
The software in these COTS modules can be generic as a *Linux* computer which does not allow on the fly configurations and has less focus on health monitoring of the processing board [3]. These COTS modules still costing heavily in development and maintenance. Such problems come with a heavy cost of space debris which is a major problem in space industry. A solution to this does not exactly fit in cost curve for the constrained IoT world.
The hardware known as *"Heritage"* needs improvement and *"New Space Missions"* aim at creating advanced space grade hardware and software, as discussed in this thesis.
A need for change and upgrade persists in this field.

1.2 Methodology

To evaluate and provide solutions to the above said problems, the thesis work will be done at ***Thales Alenia Space, Ditzingen, Germany*** under expert supervision. Various current architectures and middleware structures for Satellites will be studied and compared to the proposed solution. As per requirements for the satellite, a software architecture for the solution will be designed. Few software functions will be assessed, designed, and developed as a scope of this thesis.
This design of the software architecture will employ a "Waterfall model" as software process. The model is a sequential model with each fundamental activity planned and arranged one after other. The work involved in this thesis will be done with "Software Engineering" team following agile methodology. Overall, around 15 people will be working on this product in different teams such as "Hardware Engineering", "Software Engineering", and "Integration, Verification, Validation, and Qualification (IVVQ)".
At the start, requirements were gathered from research study by the team. This will be followed by design and development process, based on the requirements. During the timeline of thesis, the product will be in this stage. After the design and development, the software will be tested and verified followed by maintenance. During the thesis, similar software architectures will be studied to find the best possible architecture. In the end, the designed software architecture and software modules will be evaluated against the baseline requirements from standard software references.

For all proposed solutions, Unified Modelling Language (UML) diagrams have been used.

1.2.1 Tools used

During this thesis, various tools will be used to complete the defined tasks. These tools will be procured and used during the scope of this thesis.

Hardware tools

1. **Vorago M4 Eval Board:** Vorago[4] provides an eval board for testing their radiation hardened microcontroller. This Eval board is supported with its Board support package (BSP) and Technical Reference Manuals. The hardware package contains a single board computer and two daughter cards, *viz.* GPIO board and EBI/Ethernet board. The setup can be accessed and programmed using a USB cable via JTAG.

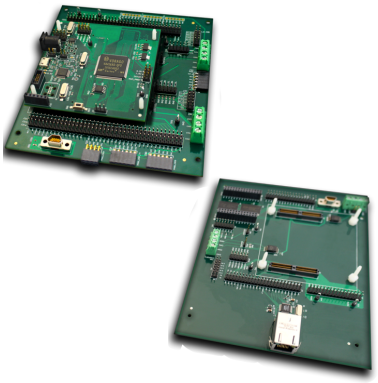

Figure 1.1: PEB1-VA416x0 Vorago Eval board[5]

2. **Zynq UltraScale+ Starter Kit:** As an SoC, starter kit[6] with Zync Ultrascale+ provides by Trenz Electronic was used. This kit has access via PMOD connectors and can be used with Windows/Linux PC using advanced software from Xilinx Inc.[7]

3. **Others:** For simulating the OBC, a standard Linux PC was used. This allowed simulating real time On-board Computer (OBC) experience. Apart from this, USB cables, jumper cables, Logic Analyzer, etc were used to support the planned tasks.

Software tools

1. **Keil IDE:** For using the Vorago-M4 EVAL board, licensed Keil IDE[8] was used. Keil IDE is a standard software development tool for using ARM-based

Figure 1.2: Trenz Zync Ultrascale+ Starter Kit[6]

microcontrollers. MDK version 5 was used during the period of this thesis. The IDE provides in-built compiler, debugger, and some already available software packs.

2. **Xilinx Design Suite:** Xilinx[7] not only provides high computing hardware but also a very comprehensive software suite for using the hardware for its highest computing. The design suite allows user to write low level firmware as well as build applications using a SDK, also provided with the suite.

3. **MODELIO:** As stated above, Unified Modelling Language (UML) diagrams will be used for modeling the software. This was done using Modelio[9], an open source modeling environment. The software allows using UML Modeler, BPMN integration, and export to various other platforms. It supports UML2 standards along others.

4. **Eclipse IDE:** Eclipse IDE[10] on Linux was used to develop the OBC emulator for testing the software on Supervisor system.

1.3 Contribution

This thesis introduces a radiation hardened module named *Supervisor system* which can monitor the health of the payload electronics and also share some tasks with the very important On-board Computer (OBC).The goal of this thesis is to design and present a solution for robust, reliable, and modular software architecture for the Supervisor system in satellites for IoT. This results in high level design and detailed design for the software requirements of the satellite payload electronics.
This software architecture is used to build two important features,

- TM/TC Communication, and
- SoC Software Updates.

The details are discussed in the Chapter 6. This architecture is then proven for it's reliability, robustness, modularity, and portability using a Software Porting test, refer Figure 6.18. The software designed upon this architecture will be used in upcoming satellite missions from ***Thales Alenia Space***.

1.4 Organization of the Thesis Report

This thesis report is organised in several chapters *viz.*

- Background, where introduction about IoT architectures and direct to satellite IoT architecture is discussed.
- State of the art, where study of various currently available solutions and related problems in satellite software/hardware architecture is done as comparison to other similar systems.
- Requirements, where requirements of various involved hardware/software modules is discussed. These requirements will be compared in the Evaluation chapter.
- High Level software architecture, where high-level-design of said *Supervisor system* is discussed and designed. Various possible solutions are discussed.
- Detailed design of software functions with UML diagrams and output statistics of the software function.
- Evaluation, where the discussed solution is evaluated as comparison to the baseline Requirements chapter.

Chapter 2

Background

2.1 Introduction to the Internet of Things

Internet of Things (IoT), simply put, is any device connected to the Internet. IoT is visioned to be a giant network of connected people and things that collect and share data about physical entities. Such devices are as tiny as a shirt button to larger devices like industrial machines, e.g self-driving cars. There are topics like Machine to Machine communication which are being developed for better data sharing while communication medium is being improved to integrate IoT devices in the world. IoT devices are built for home, industries, transportation and also to support other similar smart devices.
A usual IoT device consists of a few sensors and mainly a communication system. An IoT device may or may not have actuators directly linked to it. Communication systems have evolved along with the ever-changing IoT network. Depending on the application, an RF system can be deployed and communication between a user and devices can be established. Inter device communication is also a very important aspect of the IoT world.

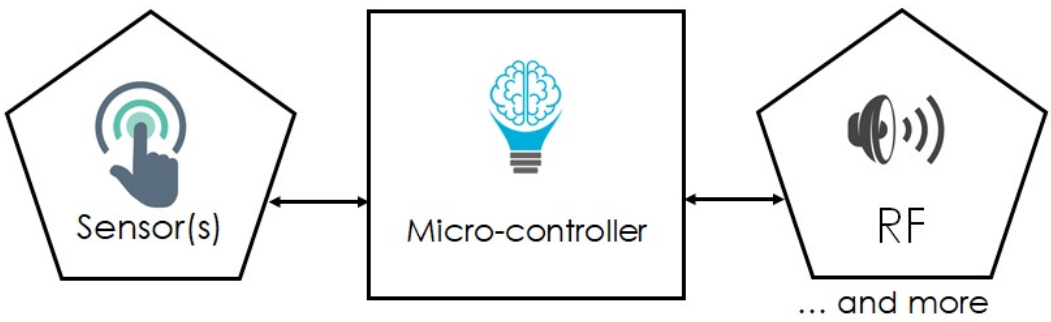

Figure 2.1: An usual IoT device

For an IoT device,

- **BRAIN** = micro-controller
- **Senses** = sensors
- **Voice/Ears** = Internet

Since there is internet involvement, there are also additional components in an IoT infrastructure like servers, databases, user interface, and many others. The overall architecture and infrastructure are heavy and complex. Every day, a new architecture is being tested and checked for a better, efficient and latency-free IoT experience.
The eventual goal of establishing IoT infrastructure is to reduce human intervention, where the devices make their own decisions. This is possible due to information sharing between devices. This involves higher-level learning algorithms built on data generated by the devices.
IoT has largely helped industries optimize cost, waste, and human resources. In day to day life, we are also highly integrated into this process by being a user of smart appliances like watches, cars, refrigerators, television, etc. This has improved user experience and helped companies know the customer better.

2.2 Applications of IoT

Internet of Things has been highly integrated into various domains. While it helps in day to day activities, that is not the limit. There has been a steep increase in IoT devices over the years, this indicates how important it is in today's life and is an indication of its numerous applications.

Smart Cities

Smart city projects are aimed at the development of optimization techniques in a city to have a better infrastructure. The aim should not be mistaken for improvement or optimization of society in any way. In the entire process, every individual is kept involved by the generation of incentives or business opportunities for everyone involved. IoT has become an enabler for all the smart city projects around the world. The efficiency of water supply, waste management, electric supply, traffic management, street lighting, and other necessities can be improved using IoT's data gathering capabilities. From the perspective of a user, the improvement of security and living conditions is a major takeaway.

Medical Systems

The medical field was late to introduce IoT but has seen huge growth since. A new term *Internet of Medical Things (IoMT)* has been introduced in the recent

past. The use of user-generated data is very important since it helps not only the caregivers but supply healthcare providers with actual data to identify issues before they become critical or to allow for earlier invention. Investments in these fields are not only made by techno enthusiasts but also by insurance companies as they are indirectly linked to healthcare systems.
Another segment in medical systems is wearable tech. Such wearables can sense heart rate, number of steps, oxygen levels, body temperature, etc. The majority of medical professionals do not trust the data collected by such devices since they are not certified for medical use.

Connected cars

In the last few years, the segment of the connected car has exploded all because of IoT. But there is no stop to this since companies are constantly pouring in money to have fully automated cars that have high-security component attached. Internet connectivity to components in a car not only improve the user experience but also help the company improve its future builds. Companies can learn from the data and analyze the performance; this can provide major help in predictive maintenance. *Vehicle to Vehicle (V2V)* technologies are being developed as a push to swarm management while *Vehicle to Everything (V2X)* technologies will be a major catalyst for self-driving cars in the future. For all of them near field or far-field IoT is a necessity.

Smart Grid

Consumers and service providers face a problem of managing power quality, safety, carbon emission, high outages, and reliability of energy supply. Technological advancements in the IoT era provide a way to manage these problems. IoT devices in electrical supply systems can monitor power consumption at various junctions in a city enabling improvement of supply on future days, hence improvement in power quality and reliability. Advanced Metering infrastructure and smart grids are the catalysts for introducing a two-way communication between the conventional power grid and consumers.

Industry 4.0

Industry 4.0 aims to revolutionize business and manufacturing processes to improve efficiency, productivity, and cost. IoT has been inducted in industries in the past decade to help Industry 4.0 become a reality; hence name *Industrial Internet of Things (IIoT)*. IIoT is being used to improve the efficiency of a machine by knowing it's downtime, state of machine parts, and operating environment. Various use-cases in handling and managing the machines remotely have been identified and are being implemented every day[11]. This has been a boost for the economy in various industries. This new concept of Industry 4.0 is considered to

be the fourth industrial revolution, which promotes a more modular and efficient system; hence has an important place in a company's budget .
The linking element between process and implementation is supply chain management and Industry 4.0 focusses highly on this aspect. Automation of supply chain and logistic management is a major concern but is solved by many budding names in the industry like NAiSE GmbH or Bosch Nexeed. Such solutions involve indoor localization collaborating with machines and robots on the shop floor. For example, NAiSE GmbH has solutions that facilitate indoor localization with process automation and digitalization. This all is enabled by IoT and digital networking. Smart retail stores or outlets have also used the services of IoT to know customer needs and make shopping a better experience.

Agriculture

Agriculture is an industry in a way and there is an equal need for process automation and digitalization. IoT facilitates this just like in any industry. Farming is usually remote and a smart farm can be a big help to the farmer especially to monitor cattle, plantations, irrigation, heavy machinery, etc. IoT is the perfect way of helping the above-said cause. Heavy investments in this domain are being made in countries where agriculture products are exported.

Smart homes

Smart home and smart appliances were one of the first-ever tried use-cases of IoT. With the love of wireless technologies, everything is being made wireless; from a switch to security systems. While some researchers find this as under-utilization of what IoT can offer, smart home appliances keep surfacing the marketing now and then. The extent of smart appliances can be realized when *"Smart Fridges"* can now order groceries when inadequate. IoT is not just the enabler in this case but the entire system is an IoT device.

Future applications?

All of the above applications are not perfect yet and there is a lot that can be improved here, e.g. connected cars are perfect in a simulated environment but not present-day road ready while fleet management is only virtual. A lot has been done in the smart appliances section but energy efficiency and security remain doubtful. While these all are improvements to be done, there are a lot of things expected shortly. A fully automated industry with a swarm of robots working in harmony is still a dream for the industrial automation sector. The medical field should see a lot of new devices with wireless sensors and predictive diagnosis coming soon. IoT has the power to have devices that can act by learning from human intuition.

2.3 IoT Architecture

An architecture in IoT reference is a framework where physical entities, their connections, functional organization, operational principle, data format, and network configuration are specified[12]. As seen above, IoT is a large field so, one architecture cannot suffice all possible implementations and applications but a reference model can be a good starting point. This means that several architectures can coexist. Though an architecture to incubate all the above needs must be general enough to adapt ever-changing network infrastructure and should be independent of physical sensors or actuators [13].
General components of such architecture are as below,

1. **IoT Device:** An IoT device comprises a sensor that can read a physical entity like temperature, humidity, presence, light, sound, etc. The sensor has no processing capability and may generate an analog electrical signal. Any configuration or data collection needs a software component which cannot run on a sensor, so an IoT device is bound to have a signal processing component, generally a microcontroller. This processing unit does not need to have complex software running on it. It can just be a synchronous program with two tasks, read sensor and send data. This brings us to the device's capability of sending data via a wired or wireless method, hence the internet. Additionally, the device may also have the task to perform operations like controlling switches or relays to enable controls, this may be visualized by an actuator, also managed by the same processing unit.
   To sum up, an IoT device will have three to four components, a sensor, a processing unit, an actuator, and an internet component.

2. **Network facilitator:** As we have seen above, every IoT device has a network component to help connect to the internet. This is made physically possible by using a network facilitator. For example, when a device uses WiFi there is a need for a WiFi router to fulfill this need. A WiFi router, in this case, will typically connect up to 200+ devices and will have auto-management of device requests using WiFi protocols. But in some cases, there might not be any typical queue handling or protocol management. This might need a synchronization algorithm that can not only manage the network but also provide latency-free and error-free message delivery both ways.

3. **Data exchange and storage:** All the information collected by all the involved devices needs a platform to processed and acted upon. This exchange platform will receive, process and pass-on all the data from IoT devices. A usual platform can be visualized as a server running information exchange protocol and database software. For example, the Message Queue Telemetry Transport (MQTT) broker will receive data from publishers and push data to the subscribers. Subscribers here can be applications running on smartphones, other independent servers or IoT devices themselves.

4. **End-application:** All the data generated needs to be visualized or made sense of, an application software receiving processed data from the data exchange service can perform logical and business operations to manipulate data and gain insights. This can be a user interface or just an algorithm to process data on a higher level for business transactions.

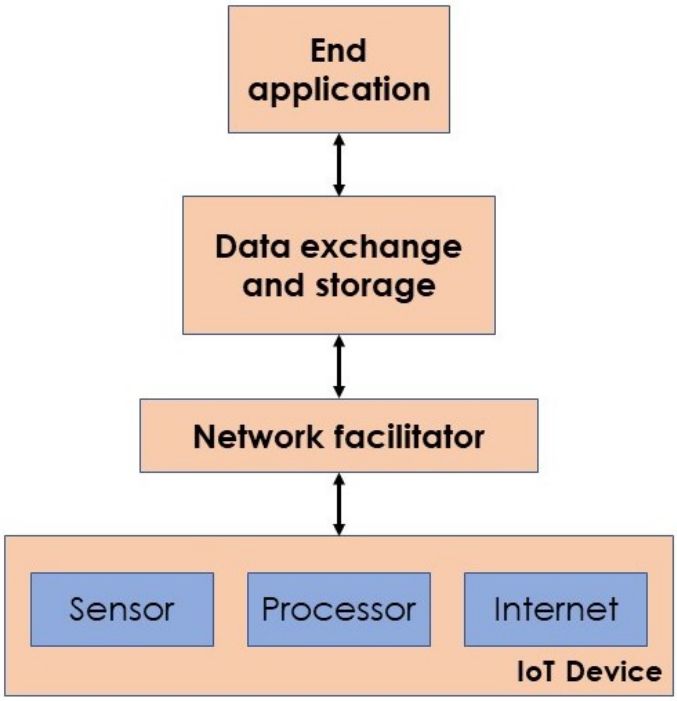

Figure 2.2: Generic IoT architecture

The edge, fog and cloud analogy

To simplify and standardize the architecture, many researchers have brought down IoT architecture to three components with individual functions and complexity. Compared to the Open systems interconnect (OSI) model with five layers, this architecture of edge, fog and cloud computing makes more sense in the IoT world. This makes the architecture more modular and adds up to the reliability of the entire system. Failure of one node does not lead to failure of the system[14].

1. **Edge layer:** Edge layer of devices are the closest devices to the sensors and have direct access to the physical environment. They have the least computation power, complexity, power consumption, and network capacity. They can have a peer to peer connection (edge to edge, E2E) while a connection to the fog layer is a must. This layer may not be directly connected to the internet.

2. **Fog layer:** Devices in this layer are of high importance and should be designed for high reliability. They act as a gateway with computations between edge devices and the cloud. We can visualize these devices as translators between edge and cloud where network connection to edge and cloud may not be the same. A fog device may or may not have decision making properties or data storage properties. A single fog device should be able to handle more than one edge device at a time, this can be facilitated by already available network protocols. To add up to the reliability, multiple clones can be deployed for redundancy. Summary for the tasks of these fog devices is to aggregate all the incoming data with low-latency and forward it to the central cloud for heavy processing.

3. **Cloud layer:** The cloud layer is a data server with the capability to store data and share information. This layer will also host applications for the defined business case. Cloud will have the highest processing power and will run heavy computationally intensive tasks. This is also the hub of data storage, which means this should be designed with the possibility of expansion. By applying this edge, fog, and cloud computing architecture, we can distribute the load, reduce network latency at the device level, reduce the probability of system failure, introduce modularity and improve mobility.

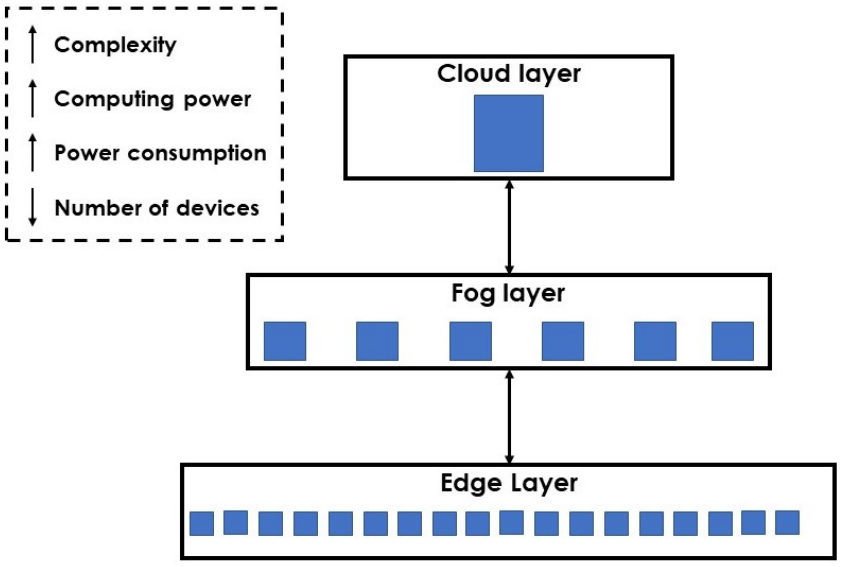

Figure 2.3: Edge-fog-cloud IoT Architecture

2.4 Direct to Satellite Technology

Internet connectivity is possible via direct and indirect ways. In the indirect way, we have a lot of other infrastructures involved but it seemingly fits into

the curve for usage against the power consumption. With the changing times, this architecture will change into direct ways like satellite connections for IoT. This trend has been seen around the world where companies are trying to have satellites launched for internet connection. This method serves for direct internet connection.
The terrestrial networks can cover the streaming requirements of audio and video transfer, but not in remote areas without any terrestrial communication coverage which is the main field of industrial operation of IoT. The global use through the natural borders of every country collides also against the different laws of every involved part. There is no other technical way to overcome this problem besides the use of satellites. LEO satellites suffer from low availability and never had been proposed until now as a possible solution. Implementations for satellite communications to multiple ground users had been done only with expensive MEO or GEO satellites resp. relevant constellations.
Medium Earth orbit or Geostationary Earth orbit constellation requires from the ground transmitter enough energy for successful transmission. The ISM free band restrictions do not permit the usage of these methods, as the received signal is too weak for successful communication. In this case, the business targeting the MEO/GEO satellites has to use private frequency on a global scale. If someone can overcome the legality problems (as it is almost impossible to be the holder of a specific bandwidth in a specific frequency, approved from all over the earth authorities) the energy consumption of transmitting signals, will be insufficient for the building of many years' autonomous devices.
Direct to the satellite[1] is an up and coming field for IoT networks and will change the architecture for IoT devices. Many companies are trying to make such networks available, *viz.* SAT4M2M, Kineis, DLR, and others.

2.5 Space IoT

One of the biggest challenges in the IoT world is to deploy low power devices over wide geographical areas. In such use-cases, satellites can bridge the gap towards a pervasive IoT network that can handle disaster-hit scenarios like earthquakes, tsunamis, and flash floods, etc. In such scenarios, the presence of a resilient back-hauling communications infrastructure is very important. Here, Direct to Satellite IoT (DtS-IoT) connectivity is desired. The space IoT architecture needs no intermediate ground gateway, making IoT networks available easily and reducing infrastructure problems.[15]

Direct access from the sensor and actuator terminals to the satellite is a more appealing solution in challenging scenarios, for example:

- In disaster-prone areas, where infrastructure-less deployments are preferred.
- Areas with a smaller number of devices where the placement of a gateway will not be profitable.

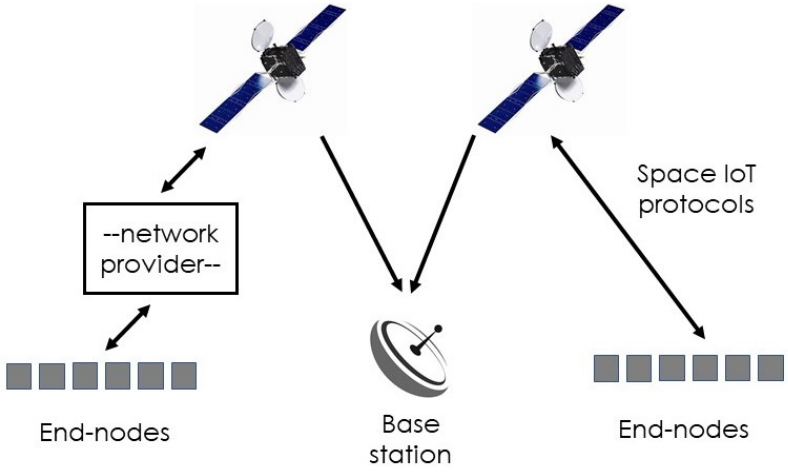

Figure 2.4: Generic vs Space IoT architecture

- In use-cases where the devices have high global mobility covering places with low network connectivity.

However, existing Low Power Wireless Area (LPWA) protocols must be revised as they weren't designed to control over several kilometers in a very ground-to-space link. Similarly, existing satellite protocols were thought to work on highly directional point-to-point topologies and won't be fitted for LPWA applications. Many of existent satellite protocols for scientific and Earth Observation missions are standardized by the Consultative Committee for Space Data Systems (CCSDS). However, these protocol sets aren't thought for networking many devices on sight. Commercial applications on this domain typically depend on proprietary protocols, while only some support Internet protocols. the foremost representative protocols are discussed hereafter, and their applicability challenges are highlighting about the DtS-IoT domain.
CubeSat deployments often employ frequencies within the range of the amateur band, with very low data rates (i.e., within the order of 9.6 kbps to 100 kbps). Although other higher bands are explored in recent developments, the CubeSat deployments are characterized by low data rates and restricted contact times. Traditional satellite protocols tend to behave poorly or just don't work on such constrained devices. As a result, specific CubeSat protocols to serve the particular characteristics of the IoT traffic have been studied within the literature. it's been identified that advanced techniques such as dynamic channelling or precise channel estimation (employed for interference cancellation), require costly resources often not available in low-cost CubeSat deployments. Among the random-access MAC protocols evaluated, only some become near the region where DtS-IoT supported by CubeSats provides a scalable, energy-efficient, and non-complex channel access mechanism to the bottom sensor nodes.
A typical LEO satellite has 4 passes per day at an average though this also depends

on the satellite orbit and device latitude. This allows data transfer opportunities of 7 to 10 minutes when the satellite is just above the spot on the ground. Nearer to the horizon, contact duration is lessened and the channel conditions are worsened. This can happen for around 20 minutes a day at a particular location when the channel resource is likely to be shared among hundreds of devices. Due to this limitation coupled with a limited data link, a single satellite is insufficient for a typical IoT application.
Satellite constellations are a fleet of satellites properly distributed in the orbit to provide continuous coverage around the globe. As a result, data transfers are enhanced, and the waiting time is drastically reduced. For an effective direct to satellite IoT network, a fully connected satellite constellation will be the best fit. However, this adds up to the cost of hundreds of LEO satellites which goes against the need for low-cost IoT architecture. A middle-ground solution of having less number of satellites to provide partial and opportunistic connectivity. In this case, a device will have to wait until a satellite is accessible, this adds to the data latency but such an approach can still satisfy a lot of IoT applications. In such a constellation, precise timing constraints will have to be kept in mind while designing the IoT system.
Kineis a French company with the motto of providing *"IoT everywhere"* is developing a constellation of satellites along with their self-designed chipset to provide network connectivity in use cases like logistics, marine, agriculture, outdoor sports, and many others. This constellation not only provides data relaying but also has an inbuilt Global Positioning system (GPS) like positioning system from their constellation of nanosatellites. A similar approach has been taken by other projects like SAT4M2M, ORBCOMM, and others.

Chapter 3

State of the art

CubeSats come in various sizes and weights, but they are all designed on the standard CubeSat unit, a cube-shaped structure measuring 10x10x10 cms with a weight between 1 and 1.33 kg. This unit is known as 1U. Nanosatellite developed on CubeSat standards promises a relatively inexpensive access to space. It also guarantees a wide range of available launch vehicles and space rocket options.

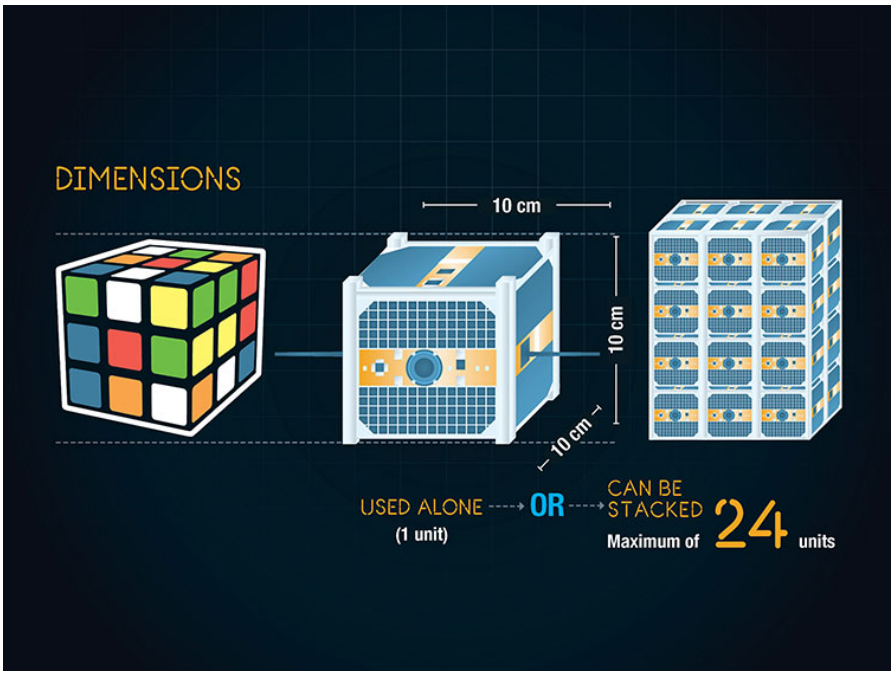

Figure 3.1: CubeSat Dimensions [16]

Advancement in technology has made miniaturization of these CubeSats possible for the space industry. The possible solution now defined is low power, low space, and low cost which boosts the business while doing considerable advancements in science[17]. Such projects have been running since 1957 but now have been boosted due to availability of Commercial off-the-shelf products in the market

which allow faster launch times. An improvement to these COTS components is discussed in this thesis (Section 3.5).

3.1 Satellite Architecture

There is an increasing need of many organizations, like ESA, NASA, ISRO, etc to use fleets of autonomous spacecraft working together to accomplish complex mission objectives. These satellites use distributed architectures because of many advantages that it brings, greater performance, lower cost, and improved fault tolerance, re-configurations and upgradability [16]. Usually satellites, including CubeSats are custom built according to the requirements and application but these three components remain constant:

- A radio communication system with antenna to send and receive data from the earth.
- A computer to receive commands and execute them. This ensures proper functioning of the satellite.
- A power source which can be a solar panel or battery or both.

The cubic structure is made of aluminium and it encompasses the above components along with additional sensors, cameras, etc. The solar panels and antenna can be installed on the exterior structure.

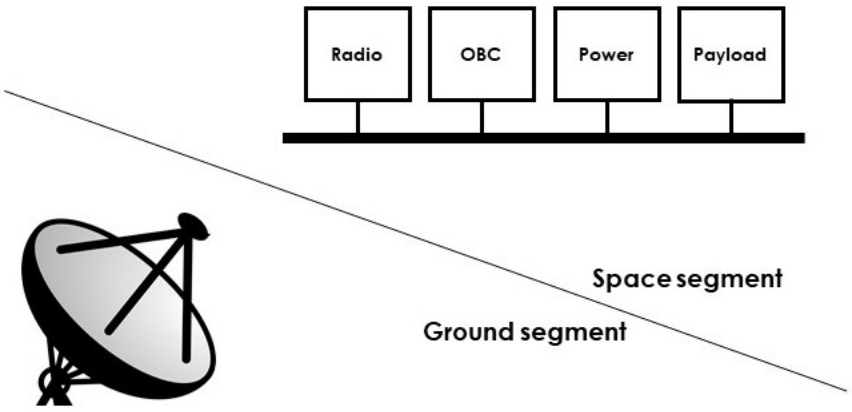

Figure 3.2: System topology of a satellite

Figure 3.2 defines a generic model of a satellite with OBC and Radio module having radio transreceivers for telemetry communication and data communication respectively. The included components can be explained as below,

1. **Radio:** The communication system or radio system is used to receive and send data to the ground station. In standard platform, two separate radio bands are used for the functionality.

2. **OBC:** It is capable of receiving and processing data from the payload. This processed data is then stored as system information which can be later sent to the ground station on request.

3. **Power:** The power section provides regulated power to the satellite for a successful operation. Conventional CubeSats are powered by rechargeable battery supported by solar panels. Additionally, a control unit may be added for battery management and power regulations.

4. **Payload:** The payload is the part of the satellite that allows it to fulfil the mission for which it was designed. It's the reason for the CubeSat's existence. CubeSat payloads are the heart of missions in space. Without payloads, there would be no point in launching a small satellite, since from a practical point of view it would become an object orbiting the Earth with no function to perform [18].

3.2 Satellite Software

Satellite software remains in the background, the design this software architectures needs,

- Resource allocation, management, and exchange.
- Autonomous task creation and allocation.
- Handle and mitigate technical constraints.
- Modular and ready for new mission concepts.
- Mission ready, secure and robust.

The most important software component of a satellite will be the software running on the On-board Computer [3]. This software is designed to be capable of,

- processing the payload data, e.g. images, sensors, star tracker, etc.
- process and execute commands from the ground station.
- manage actuators like reaction wheels.
- manage position, velocity, altitude and rotational rates.
- system status monitoring, failure detection, isolation and recovery.

All of the above has to be designed and developed on the on-board software hierarchy levels. The dynamic software architecture runs by scheduling RTOS threads to encapsulate the above functions. Most of the blocks from application layer will be run as a separate thread in the RTOS. Even though the current CPUs provide

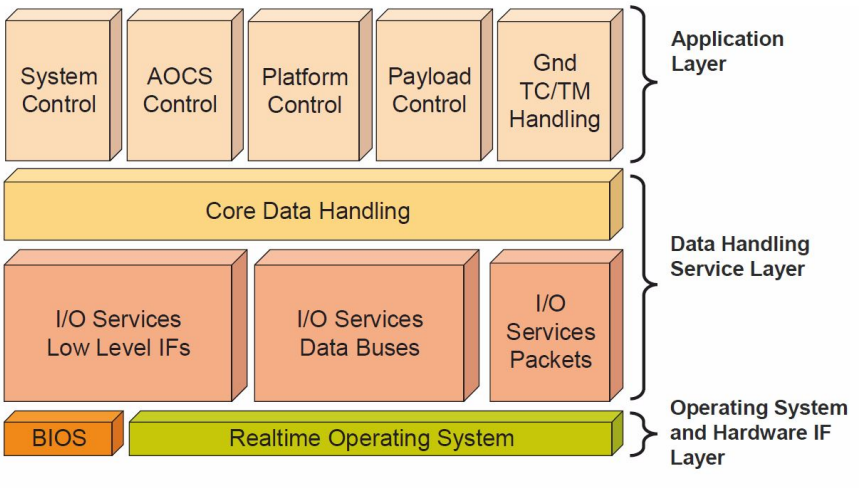

Figure 3.3: Generic Software architecture for an on-board computer [3]

high throughput, the software should perform efficient tuning of the RTOS tasks to use RTOS as baseline in the software architecture. This can facilitate in parallel payload and instrument operations along with parallel ground contact and data handling.

3.2.1 Study of Software Architecture in MTG Satellite

A study was done for On-board Software (OBSW) running on the Meteosat Third Generation (MTG) platform satellite On-board Computer. MTG is a cooperative undertaking between the European Space Agency (ESA) and the European Organization for the Exploitation of Meteorological Satellites (EUMETSAT) and aims at providing Europe with an operational satellite constellation able to support accurate prediction of meteorological phenomena and the monitoring of climate and air composition [19].
As the satellite systems moves towards a distributed architecture, use of COTS components, and reduction in cost the need for distributed software architecture increases. Devoted components perform their devoted functions (power handling, data processing, ground link, etc.). Data and resources are shared between the components who share interfaces for exchange.
MTG platform defines a static software architecture providing all the necessary functions for the satellite along with functions that enable ground station to monitor and supervise the spacecraft during the entire mission. The static architecture is composed of layers as in Figure 3.4.
A study of MTG software architecture is of prime importance to this thesis due to its similarity to the discussed solution. An improvement upon it is intended with a separate module to handle some functions in parallel to the OBC.

The architecture defines three types of components,

- Managers to handle whole spacecraft subsystem and host controllers for

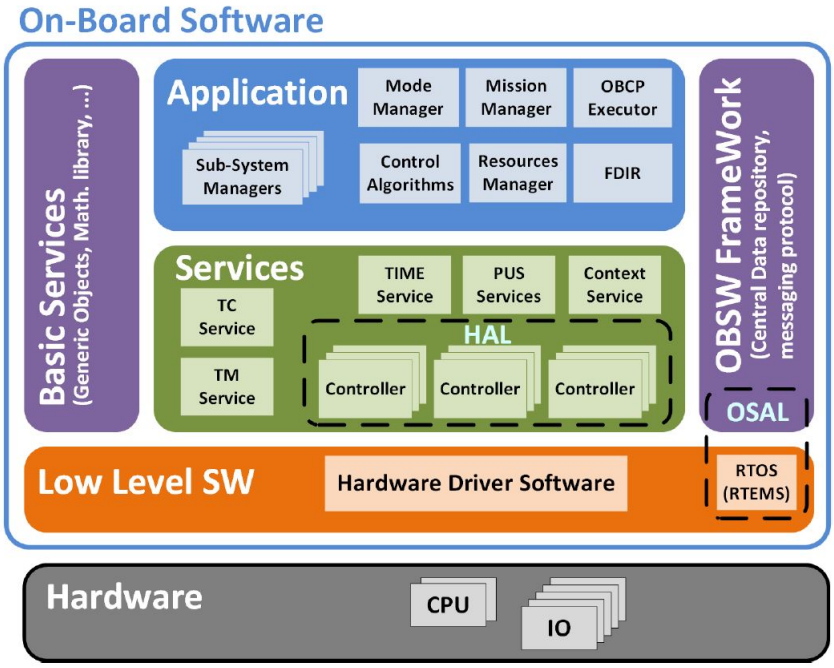

Figure 3.4: Static software architecture of OBC Software[19]

making use of the services.

- Services to provide common data handling functions.
- Controllers to provide hardware access.

The layers separate software functions and tasks to be done by the software in order to have a robust, reusable, and modular software.

Overview of the layers

- Application layer: This layer performs system level, spacecraft mode and mission management functions. This includes Failure detection, identification and rectification tasks also.
- Service layer: This layer contains data handling functions including Packet Utilization Standard (PUS) service. The layer also encapsulates Hardware abstraction layer (HAL) for the hardware access.
- Low level software layer: The layer provides access to the hardware interfaces and drivers for redundant CPUs. RTOS is also a part of this layer.
- On-board software framework layer: This layer provides an essential function of software bus to give access to the central repository for which it provides a messaging protocol. RTOS in also included in this layer to monitor task execution time and find out missed deadlines or over-runs.

- Basic services layer: These functions are generic mission independent to facilitate queue management, data manipulation, mathematical functions, etc.

The software architecture is also studied in Section 6.1.

3.3 Introduction to SAVOIR

SAVOIR[20] means Space Avionics Open Interface aRchitecture. It is an initiative to unite the European Space Avionics Community for improvement in the way Spacecraft avionics are developed. Brains behind this initiative include European Space Agency(ESA), Thales Alenia Space, Airbus, DLR, OHB etc. The aim to have such a reference is improve product orientation, enhance delivery of space systems, and supporting industrial competition. SAVOIR has taken inspiration from AUTOSAR, although the underlying industrial business model is different SAVOIR provides:

- Reference architecture for space grade hardware and software.
- Building blocks for the architecture.
- Specification for internal and external interfaces.
- Specifications of functions involved in the architecture.
- Verification architecture for the functions and interfaces.
- Implementation for selected portions of the architecture.

SAVOIR defines a software architecture for components like OBC and connected Remote Terminal Unit (RTU) along with its interface units. While Supervisor system is seen as a mix of OBC and RTU, some parts of the software architecture can be derived from this reference. For instance, the software architecture defines OBC software reference architecture (OSRA) to be flexible with hardware changes keeping in mind future needs, the software architecture defined below also aims that(More about it discussed in Section 5.4).
Considerations from this reference software architecture are used as baseline when designing software architecture for a satellite system. The reference defines classical software engineering approach but with utmost care for robustness in design.

3.4 Shortcomings in the available solution

The following shortcomings have been studied in the current available and discussed solution,

- Very less amount of re-configurations possible.
- Power and health monitoring is not the top priority amongst many functions of OBC.
- Radiation hardened COTS parts are very costly.
- Heritage COTS are old technology and need upgrade.

Satellites in IoT will need lightweight solution which is cost effective, yet advanced for future technologies. The available solution does not fit in this curve perfectly. Other available solutions can not provide a robust functioning.
This can lead to reduced life of the satellite and sometimes complete failure leading to added space debris. As a solution to this, a novel Supervisor system has been introduced in this thesis (Section 3.5). The solution leads to task distribution reducing the load on the OBC software and hence adding on to the reliability of the satellite system. Additionally, having one central radiation hardened module to manage other modules will reduce the cost, more about it discussed below.

3.5 Novel Supervisor system

The payload processing board is a crucial part for the working of a nanosatellite. The involved electronics can be the bottleneck for a communication system, its design is an important task for an engineer.
A general satellite payload system consists of a Payload processing node (an FPGA or SoC), an on-board computer, and other optional sensors like camera, etc. This thesis introduces an additional component named *"Supervisor system"* which is a radiation hardened component that monitors all the other components in the payload system. The system also applies updates to the SoC with robust logic[refer Section 6.2.1] to not allow any failures. This distributed architecture of the processing board will decrease the number of faults and provide a seem-less internet experience.
Benefits of using an Additional Supervisor system:

- Robust latch-up free health monitoring for the COTS components on board.
- Interrupt free TM/TC communication on board.
- Software and firmware updates for the MPSoC.
- Avoiding redundant design.
- Detection and Mitigation of all on-board power errors.
- Reduction of overall costs for a satellite.

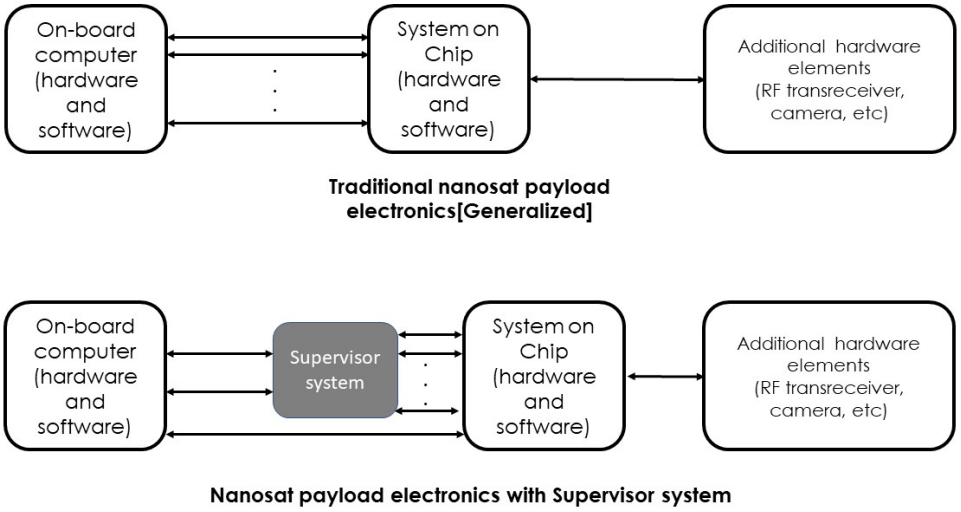

Figure 3.5: Payload structure with and without Supervisor system

Such a design will mitigate on-board errors which can harm the MPSoC seriously and hamper the functioning. When performing these functions, the supervisor system will communicate important system reports in the form of Telemetry (TM) to the on-board computer which in turn sends it to the ground station. At the same time, the system will process and forward Telecommands (TC) from the on-board computer to the MPSoC. This reduces the processing load on the OBC since it does not have to monitor TM/TC communication continuously.
The rad-hard Supervisor system will act as a watchdog for the MPSoC. The watchdog will command a reset (potentially only partial) in case parts of the MPSoC are not reacting, potentially due to a Single-event functional interrupts (SEFI). Since crucial TM/TC communication with the satellite platform ends on the Supervisor system, this service will not be interrupted.
Software architecture of such a system should be comparable to the software architecture of an On-board Computer because of its similarity in functions. Though, it must be compact, decoupled, and modular to ensure flexibility and portability. This changes the usual components on a payload board and hence can have options for selection. A solution trade-off was found out as below,

The vulnerability of COTS parts to radiation is clearly identified as the major design concern. The mitigation of destructive Single Event Effect (SEE) will be achieved by a supervising circuitry that monitors current and voltage supply to the COTS parts. For this circuitry it is mandatory not to be susceptible to destructive radiation effects. The rad-hard island on the PCB will be extended by the supervisor system, a rad-hard microcontroller that is commanding and processing the power supply as well as acting as a watchdog for the MPSoC.

Solution	Pros	Cons
1x rad-hard microcontroller	Proven space heritage	No access to FPGA performance.
1x rad-hard MPSoC	Rad-hard and space heritage	• Highest price • Less performance than COTS MPSoC
1x COTS MPSoC	Lowest cost	• Reboots more often • High chances of memory corruption • Additional testing time for reliability
2x COTS MPSoC	Low cost	• Voter is also radiation vulnerable • 2x overhead • No COTS SW like Linux, possible common source failure can lead to a total system failure
1x COTS MPSoC and 1x rad-hard microcontroller	• Added robustness • Rad-hard by design • Microcontroller (Supervisor) can also handle essential functions	Two different systems need to be designed.

Table 3.1: Solution trade-off for Supervisor system

Chapter 4

Requirements

In the design process of a product, before design comes requirement capture and analysis. This process can be followed independently for hardware or software. While requirement engineering is a big field in itself, this chapter will try to sum up hardware and software requirements for the discussed solution before design of software architecture.

4.1 Hardware requirements

As discussed above, a microcontroller should be a better option for the Supervisor system (see Figure 3.1). A microcontroller should satisfy the cost to functionality factor for the requirements.

4.1.1 Requirements for a microcontroller as a Supervisor

From the above discussion, we understand that as a supervisor, a microcontroller must fulfill the following requirements:

1. Radiation hardened by design,
2. Memory scrubbing,
3. Availability of Error Detection and Correction module (EDAC),
4. Adequate processing power,
5. More than 50kB of RAM (For running RTOS and BSP),
6. Availability of Serial Peripheral Interface (SPI), at least 3,
7. Availability of Universal Asynchronous Receiver Transmitter, at least 2,
8. Availability of Analog to Digital Converter, at least 6 channels,
9. Availability of Controller Area Network (CAN) with hardware pre-processing,
10. Availability of Ethernet,
11. Internal oscillator, and
12. Floating Point Unit is a nice to have feature.

These are minimum requirements that will allow the hardware to function as a Supervisor module.

4.2 Requirements of the Software Architecture

The Supervisor system is radiation hardened by design but the software running on this module must adhere to standards as in SAVOIR and ECSS. As per software quality metrics derived from ECSS requirements, ECSS-Q-ST-80C [21] the following requirements were finalized:

- The software architecture must account for a real-time behaviour of the software. Definition taken from, ECSS Q80-04 [21] and ISO 9126 [22].
- The architecture must provide a stable functionality for all the mission goals.Definition taken from, ECSS Q80-04 [21] and ISO 9126 [22].
- The architecture must be modular with standard interfaces and independent functionality. Definition taken from, ECSS Q80-04 [21].
- The architecture must allow the modules to be resued for future missions. Definition taken from, ECSS Q80-04 [21] and ISO 9126 [22].
- The architecture must be scalable for future expansion in functionality. Definition taken from, ECSS Q80-04 [21].
- The software architecture should allow change in hardware by simple porting procedure. Definition taken from, ECSS Q80-04 [21] and ISO 9126 [22].

These requirements set a baseline for the Supervisor software and make the processing board future ready. Since, these requirements are built from defined standard, it also allows a track of the quality.

4.3 Requirement for the detailed design

For the functionalities in the scope of this thesis, a baseline has been set so that it fits in the general software architecture. These software functionalities are responsible for stable functioning of the satellite. General requirements for the software design are as below:

- The software must be modelled using UML diagrams.
- The software must be designed in C programming language using a licensed IDE.
- The software design must follow similar structure as the software architecture.

- Documentation of the design must be generated as a Technical Reference Manual (TRM).

4.3.1 Requirements for TM/TC communication

As Supervisor has to handle the TM/TC communication across the mission board, it is necessary to have some standard requirements as below,

1. The Supervisor Software shall handle a communication interface to the OBC via UART, for exchange of Telecommands and Telemetry.

2. The Supervisor Software shall handle a communication interface to the SoC via UART, for exchange of Telecommands and Telemetry.

3. The communication protocol for the Telemetry and Telecommands shall be based on the CCSDS Space Packet standard.

4. The Supervisor Software shall implement a task to handle continuously the Telecommands coming from the OBC.

5. The Supervisor Software shall implement a task to handle continuously the routing of the Telemetry data flow comming from the SoC.

6. The Supervisor Software shall implement a task to generate periodically a TM packet that will be delivered to the OBC via UART0.

7. The Supervisor Software shall implement a task to generate a TM packet on detection of certain events. The TM Packet will be delivered to the OBC via UART0.

8. The Supervisor shall provide a status report of the SoC Boot Process, by means of the generation of the proper TM packets via UART0

9. The Supervisor shall provide a status report of the Software/Firmware Updates Function, by means of the generation of the proper TM packets via UART0

10. The Watchdog status shall be reported to the OBC via a proper TM packet.

These requirements makes the TM/TC communication a complex function of the Supervisor. This function is one of the most important task to be done by the Supervisor. All other tasks must have a link to the communication architecture so that they can share their status telemtry.

4.3.2 Requirements of SoC software update function

The Supervisor system will be responsible for firmware update for the MPSoC. MPSoC can run application from an external connected memory, this concept will be used to perform software update.

1. The Supervisor software shall handle TCs regarding Software update from OBC.
2. The update procedure should be defined in stages with appropriate handling.
3. The Supervisor software should handle a common memory *SPI NOR Flash* for reading and writing.
4. The Supervisor software should implement an independent task for software updates on receiving an *"Update available"* TC.
5. The *"Update available"* TC shall specify update parameters, supervisor software should read those parameters and apply them in action.
6. The Supervisor software should write the memory on reception of *"Data bytes"* TC(s).
7. The Supervisor software shall verify the update image using CRC on reception of *"Update verify"* TC.
8. The Supervisor software shall execute the update using a Boot process on reception of *"Update execute"* TC.
9. The Supervisor software shall send a detailed report about the update to OBC using the UART link.

These are the minimal requirements that the SoC Software update function. They set a baseline for the software design as built upon software architecture.

Chapter 5

High Level Design

5.1 Microcontroller for the Supervisor System

Rad-hard microcontrollers

Radiation hardening is the process of making electronic components or circuits susceptible from any errors on the exposure of ionizing radiation possible in space or high-altitude flights or also in environments nearby nuclear reactors.
Microcontrollers made and tested with such a process is known as Rad-hard microcontroller. These microcontrollers are safe from Single event effects like Single Event Latch-up (SEL), Single Event Transient (SET), Single Event Upset (SEU), or Single Event Burnout (SEB).
All such controllers have the least or zero effect from electronic noise or signal spikes. In related digital signals, the controllers will not be generating inaccurate signals.
Radiation tolerant controllers are also available but they still show some signs of failure especially in space.

5.1.1 Microcontrollers comparison study

To choose the best option for a microcontroller as a Supervisor system, a study was done amongst few competitive microcontrollers. The findings are as below,

From the Figure 5.1 we can see that there are two viable options, Vorago VA41630 and Microchip SAMRH71 due to their process power and availability of interfaces. Microchip SAMRH71[23] does not have Analog to Digital Converters which can be managed using an SPI to ADC bridge which is an extra component on board. Instead, a compromise can be made on the processing power between the two controllers for choosing Vorago VA41630 as the controller for the Supervisor system. The company VORAGO Technologies also provides support for the software and hardware development in the future.
Hence, Vorago VA41630 will be used as a supervisor in this context.

Microcontroller	Architecture	MIPS/FLOPS	Radiation performance	Interfaces						Memory scrubbing	Software toolchain
				SPI	UART	CAN	ADC	SpaceWire	Ethernet		
Vorago VA10820	ARM Cortex M0	46 DMIPS	Rad-hard >300K rad (Si)	x3	x2	Nil	Nil	Nil	Nil	Memory scrubbing and EDAC.	Keil with FreeRTOS support
Vorago VA41630	ARM Cortex M4	125 DMIPS	Rad-hard >300K rad (Si)	x3	x3	x2	x8	x1	x1	Memory scrubbing and EDAC	Keil
Microchip SAMRH71	ARM Cortex M7	200 DMIPS	Rad-hard 100K rad (Si)	x10 (flex)	x10 (flex)	x2	Nil	x2	x1	Memory Scrubbing with EDAC	MPLABX IDE
Texas Instruments MSP430FR5969	16-bit RISC Architecture	0.288 DMIPS/Mhz	Rad-hard 50k rad (Si)	x3	x2	Nil	x16	Nil	Nil	No scrubbing or EDAC	IAR or Code Composer Studio
Cobham UT32MOR500	ARM Cortex M0+	0.95 DMIPS/MHz	Rad-tolerant 50k rad (Si)	x1	x2	x2	x8	Nil	Nil	Memory Scrubbing with EDAC	Keil

Table 5.1: Comparison study for few microcontrollers

5.1.2 Vorago VA41630

Radiation Hardened VA416x0[4] 32-Bit ARM® Cortex-M4 (with FPU) microcontroller manufactured with HARDSIL technology offering radiation performance and latch-up immunity.

Key Features

- Manufactured with HARDSIL® technology
- 32-bit ARM® Cortex-M4 processor
  - Single Precision FPU
  - SWD based debug interface
- Operating Voltages
  - GPIO 3.3 ± 0.33 V
  - Optional 1.5 V core supply voltage
  - Includes on-chip LDO regulator
- Clock rate upto 100 Mhz
- Memory
  - 64 Kbyte on-chip data and 256 Kbyte on-chip program memory SRAM
  - 256 KByte SPI NVM
- Peripherals
  - 104 Configurable GPIO pins
  - 3 UART interfaces
  - 3 I2C interfaces
  - 3 SPI interfaces
  - 2 CAN 2.0B
  - Ethernet 10/100 MAC
  - Full-duplex SpaceWire interface
  - DMA controller
  - 8 Channel ADC (12-bit, 600 ksps)
  - 2 Channel DAC (12-bit)

  - Temperature sensor
- External Asynchronous Parallel Bus Interface
  - 8-bit or 16-bit memory support
  - Up to four memories of up to 16 Mbytes each
- Timer System
  - 24 configurable 32-bit counters / timers
  - Input capture, Output compares
  - PWMs, Pulse Counters, Watchdog timer
- 176 QFP (20 mm x 20 mm) Package
- Total Ionizing Dose (TID) > 300 kRad (Si)
- Soft Error Rate (SER) < 1e–15 errors / bit-day
- Latchup immunity

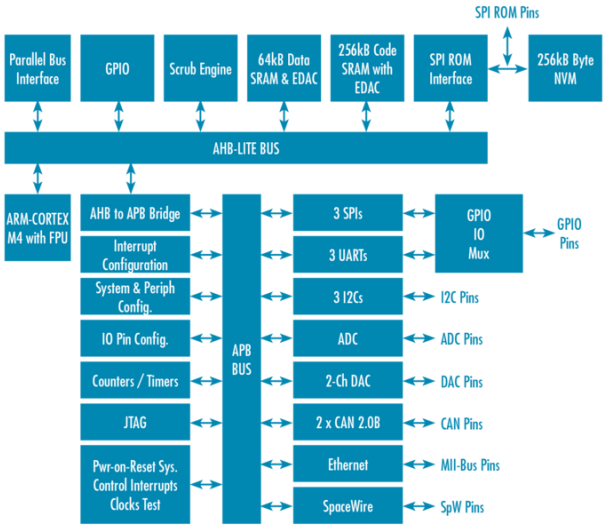

Figure 5.1: Vorago VA41630 block diagram[4]

ARM Cortex M4F

ARM Cortex[24] M family of processors are cost and energy-optimized 32-bit microcontrollers. ARM licenses this architecture to CPU and SoC Manufacturers for license fees and certification.
The core defines various silicon options as Intellectual property, such as SysTick, Memory Protection Unit, Watchdogs, Wakeup interrupt controllers, interrupts, halting debug support, Data cache, etc.
The series contains ARM Cortex M0, Cortex M0+, Cortex M1, Cortex M3, Cortex M4, Cortex M7, Cortex M23, Cortex 33, Cortex 35, Cortex 55.
Cortex M4F core is based on ARM7E-M architecture with 3 stage pipleline and an optional Floating Point Unit. The core has entire Thumb 1 and Thumb 2

instruction sets with 32-bit integer multiply and 32-bit hardware divide.

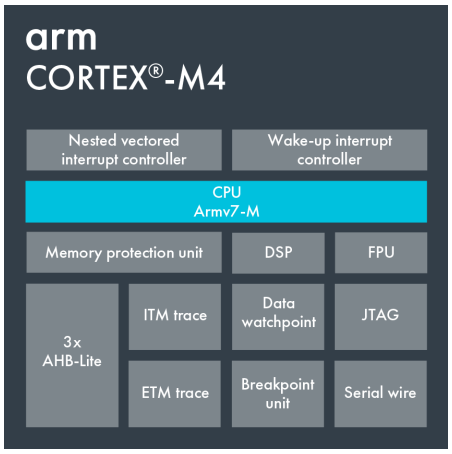

Figure 5.2: Cortex M4F architecture[24]

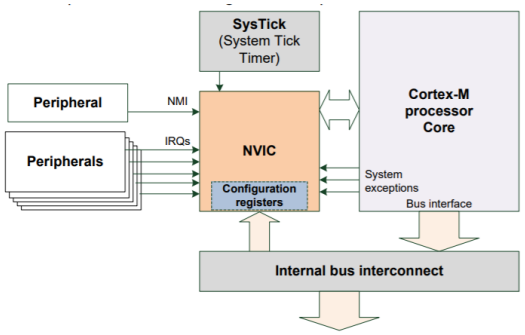

Figure 5.3: Nested Vectored Interrupts in Cortex M4F[24]

The core has DSP instructions with saturation arithmetic support. ARM offers FPU and Memory Protection Unit (MPU) as optional IP along with the core. It can have 1 to 240 Interrupts along with non-maskable interrupts.
The core also provides various sleep modes for improving energy efficiency. This includes Wait for Interrupt (WFI) and Wait for Event (WFE) modes.
The core makes debug available through JTAG or Serial Wire Debug (SWD).

5.2 Block design for processing board

The processing board is the core of a satellite. All functionality relating to RF signals, communication to ground station and control is managed by this processing board. The processing board being developed has three main components,

- FPGA/Processor system
- Supervisor system
- Interface to On-board computer

Additionally various components added for functionality, *viz.* Non Volatile Memory (NVM), camera interface, clock reference, power electronics and interfacing connectors.

Block diagram in Figure 5.4 is very important for the software to function as knowledge of all the components on board will make the software more precise, robust and reliable.

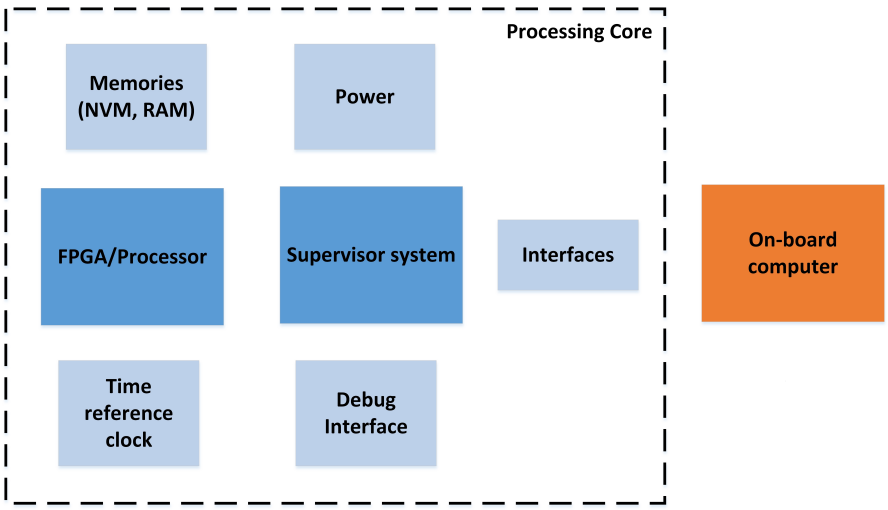

Figure 5.4: Minimal hardware building blocks of the processing board

5.3 RTOS vs baremetal

Bare metal programming means simple loop based programming where the control of connected peripherals is via polling or interrupts. This means following a random schedule based on the use case and implementation structure. This is efficient for simpler problem statements with less number of tasks.
But one major drawback is that code is not scalable. There are two directions of scalability:

- vertical (a.k.a. scaling up): faster CPU, more RAM, more disk space;
- horizontal (a.k.a. scaling out): more cores in CPU, more CPUs, more servers;

On the other hand, RTOS brings the concept of scheduler and tasks. Numerous tasks can be scheduled and run based on algorithms to reach optimum CPU and memory usage.

Why to consider RTOS?

- Guarantee of having real-time performance.
- Multi-threading and Concurrency (Concurrency is achieved by using task scheduler).
- Management of inter task communication.
- Resource management (semaphores/mutex).

The choice of running the implementation on baremetal or a RTOS depends on several factors that are to be considered. Same goal can be achieved with either

of them. But, RTOS has lot of advantages to offer for example, it has a scheduler to run tasks concurrently by switching between them using priorities assigned to them, pre-emption of tasks, sharing of resources with ease using mutexes, synchronizing the tasks using semaphores and lot of APIs come with RTOS as a package to be used with the application.
As for our project, since the Vorago MCU as a supervisor is going to handle many tasks requiring specific scheduling, using RTOS would be a better option, since it can schedule tasks with priorities assigned and we can still use the interrupts to pre-empt tasks.
According to preliminary analysis,the following operating systems have support for the ARM Cortex M4 processor and would be possible candidates:

- RTX (Keil Implementation)[25]
- FreeRTOS[26]
- μC/OS [27]
- RT-Thread [28]

Why to choose FreeRTOS?

- Provides a single and independent solution for many different architectures and development tools.
- Contains a pre-configured example for each port. No need to figure out how to setup a project – just download and compile!
- Is known to be reliable. Confidence is assured by the activities undertaken by the SafeRTOS sister project.
- Provides features like memory protection, stack overflow detection, etc.
- Is still undergoing continuous active development.
- Has a minimal ROM, RAM and processing overhead. Typically an RTOS kernel binary image will be in the region of 6K to 12K bytes.
- Provides ample documentation.
- Is very scalable, simple and easy to use.
- FreeRTOS offers a smaller and easier real time processing alternative for applications where eCOS, embedded Linux (or Real Time Linux) and even uCLinux won't fit, are not appropriate, or are not available.

Features	Baremetal	FreeRTOS
Multithreading	Not available	Available Pre-emptive, round-robin, or cooperative scheduling.
Memory Management	Self-made algorithms for stack and heap management.	Static and Dynamic Memory Allocation available. Five options available from FreeRTOS.
Interrupt Management	Handled by Interrupt Controller.	Semaphores available for synchronization between tasks and interrupts.
Hardware abstraction layer	BSP from Vorago	Self created HAL-utility layer
Resource Management (hardware arbitration)	None	FreeRTOS has additional layer for resource management to allocate locks/mutex/semaphores for hardware
Provision of Software Timers	No functionality provided	Provided Software Timers to schedule periodic tasks or one-shot tasks at a specific time.
Hierarchy separation and protection	HAL is the only layer.	Tasks and HAL drivers are on separate layers and are well separated.
Exception Management and Recovery	Watchdog with a handler to switch back to the known state	Same as baremetal. Additionally, for a task-based failure, the handler can have restarting for the task.
Managing of failures / defects / un-availabilities in a way not harming other processes and tasks	Self-made functions to handle errors, defects and infinite loops.	FreeRTOS has hooks for few possible errors.
Failure / Error confinement	Error in one functionality can lead to failure of the entire system.	Since tasks are separated from hardware, failure of one task will have less effect on the failure of the system.
Stackoverflow detection	Not available, has to be developed	Has two methods for checking stack overflow. This is done using Stack-Overflow hooks.

Table 5.2: Comparison between baremetal and FreeRTOS implementation

5.4 Layered software architecture design

Satellite development and its architecture is always application driven. This allows the payload to function aptly and according to the requirements and specifications. Spacecraft operations will succeed only when the Space and Ground Segment are interlinked optimally through appropriate data handling and management concepts. Communication satellites are successful only with proven reliability and flexibility of the on-board systems. The need for a robust and reliable satellites increases with the rules concerning space debris and deorbiting satellites.
The software architecture in figure 5.5 attempts to provide a robust and reliable software which can also perform re-configurations during operations. The architecture designed for this use-case is a layered architecture with hardware dependent and hardware independent layers as in Figure 5.6.

1. Components in the hardware dependent layer are dependent on the microcontroller i.e. Vorago VA41630. The software makes interfaces available for the higher level to latch on and add functionality.

2. The hardware independent software components are modular functions that can be reused in future. They can call functions from the hardware dependent layer to develop application. The middleware layer is a very important layer since it allows smooth interaction between application and hardware.

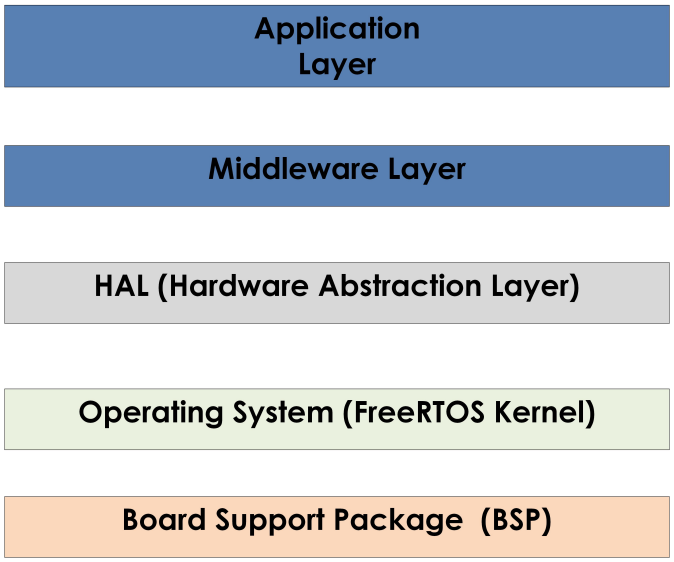

Figure 5.5: Overview of the designed software architecture

This architecture aims to shorten the development time and to have flexible application during operations. Since many tasks are linked to associated commands, an architecture to fulfil requirements in this direction makes sense.

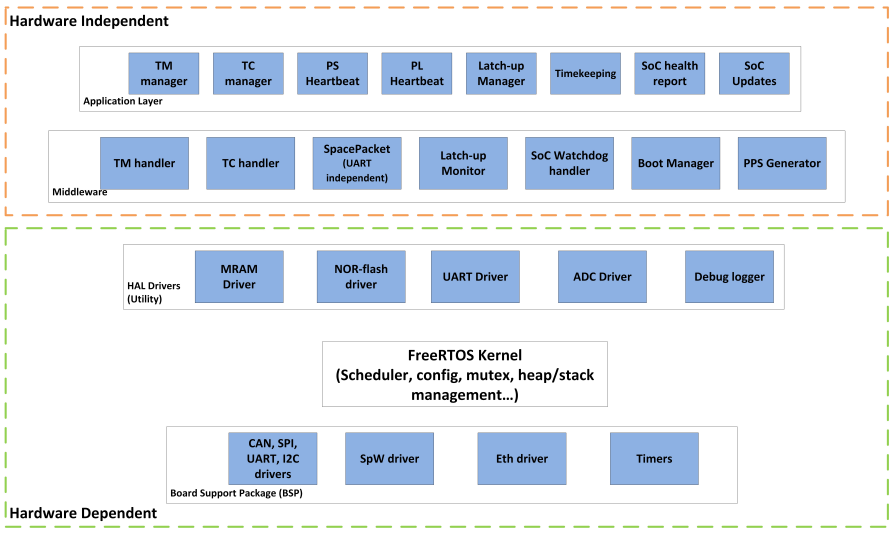

Figure 5.6: Detailed software architecture

This software architecture is an extension and improvement on OBC software architecture in Figure 3.3. Our designed software architecture is a tailor-made architecture for the generic OBC software architecture.

Board Support Package

Board support package from Vorago gives access to the hardware for all the available interfaces– I2C, SPI, UART, ADC, Timer, SpaceWire, Ethernet, etc. This serves as a baseline for the FreeRTOS kernel. The FreeRTOS project needs importing of the BSP to build over it.

BSP Component	Description
UART Driver	**Speed:** Maximum up to 2,000,000 bps, 2000 Kbps **Transfer Size:** FIFO size: 16 bytes each instance **Number of Instances:** three(0,1,2)
SPI Driver	**Speed:** (SystemCoreClock/2), 50Mhz **Transfer Size:** 16 words, each instance **Number of Instances:** four(0,1,2,3).
I2C Driver	**Speed:** Upto 400Khz **Transfer Size:** 16 word FIFO, each instance. **Number of Instances:** three(0,1,2).
ADC Driver	**Speed :** 600k samples/second **Transfer Size:** 16 word FIFO **Number of instances:** 8 external channels
Spacewire	**Speed:** max 100 Mbits/second **Transfer Size:** 1Kbyte receive FIFO, 1Kbyte transmit FIFO, transfer packet upto 256 characters **Number of instances:** 1
Ethernet	**Speed:** 10/100 Mbps **Transfer Size:** 16 Kbytes of data **Number of instances:** 1

Table 5.3: Description of BSP Components

FreeRTOS Kernel

FreeRTOS kernel provides various ways to manage applications and handle them with time deterministic nature. FreeRTOS allows a software application to be written as a set of independent tasks. Each task is assigned a priority and it is the responsibility of the Real Time Operating System to ensure that the task with the highest priority that is able to run is the task that is running.

1. **Scheduling:**[29] FreeRTOS schedules task according to the priority of the task but for tasks with same priority, user can set round robin like time slicing using the FreeRTOSConfig.h file. The scheduling method chosen is pre-emptive with time slicing (round robin).

2. **Synchronization:** FreeRTOS allows task synchronization using Mutex, couting/binary semaphores and most importantly task notification. Each RTOS task has a 32-bit notification value. An RTOS task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value.
   That flexibility allows task notifications to be used where previously it would have been necessary to create a separate queue, binary semaphore, counting

semaphore or event group. Unblocking an RTOS task with a direct notification is 45 percent faster and uses less RAM than unblocking a task with a binary semaphore.

3. **Heap management:**[30] The RTOS kernel needs RAM each time a task, queue, mutex, software timer, semaphore or event group is created. The RAM can be automatically dynamically allocated from the RTOS heap within the RTOS API object creation functions, or it can be provided by the application writer.
   The FreeRTOS download includes five sample memory allocation implementations, each of which are described in the following subsections. The subsections also include information on when each of the provided implementations might be the most appropriate to select. Each provided implementation is contained in a separate source file which are located in the *Source/Portable/MemMang* directory of the main RTOS source code download. Other implementations can be added as needed. Exactly one of these source files should be included in a project at a time [the heap defined by these portable layer functions will be used by the RTOS kernel even if the application that is using the RTOS opts to use its own heap implementation].
   Following below:

   - heap_1 – the very simplest, does not permit memory to be freed.
   - heap_2 – permits memory to be freed, but does not coalescence adjacent free blocks.
   - heap_3 – simply wraps the standard *malloc()* and *free()* for thread safety.
   - heap_4 – coalescences adjacent free blocks to avoid fragmentation. Includes absolute address placement option.
   - heap_5 – as per heap_4, with the ability to span the heap across multiple non-adjacent memory areas.

   heap_4.c is particularly useful for applications that want to use the portable layer memory allocation schemes directly in the application code rather than just indirectly by calling API functions that themselves call *pvPortMalloc()* and *vPortFree()*. The *xPortGetFreeHeapSize()* API function returns the total amount of heap space that remains unallocated when the function is called, and the *xPortGetMinimumEverFreeHeapSize()* API function returns lowest amount of free heap space that has existed system the FreeRTOS application booted. Neither function provides information on how the unallocated memory is fragmented into smaller blocks.

4. **Intertask communication:** FreeRTOS Queues[31] are the primary form of intertask communications. They can be used to send messages between tasks, and between interrupts and tasks. In most cases they are used as thread safe FIFO buffers with new data being sent to the back of the queue,

although data can also be sent to the front.
FreeRTOS allows task notifications which are fast and efficient ways of communicating between tasks. In this case, each task is given a 32-bit task notification value. The flexibility of task notifications allows them to be used where otherwise it would have been necessary to create a separate queue, binary semaphore, counting semaphore or event group. Unblocking an RTOS task with a direct notification is 45% faster and uses less RAM than unblocking a task using an intermediary object such as a binary semaphore [32].
Stream buffers[33] is a task to task, interrupt to task, and interrupt to interrupt communication option. They are optimised for single reader single writer scenarios, such as passing data from an interrupt service routine to a task, or from one microcontroller core to another on dual core CPUs. Data is passed by copy – the data is copied into the buffer by the sender and out of the buffer by the read.
Stream buffers pass a continuous stream of bytes. Message buffers pass variable sized but discrete messages. Message buffers use stream buffers for data transfer.

Higher Abstraction Utility Drivers

There are various components and protocols that are not a part of BSP and need special driver development which is built on the hardware drivers given by Vorago, e.g. MRAM driver can be made using SPI driver from BSP layer. Similarly, for current monitoring ADC Driver is needed. These are the essential utility drivers for a thread-safe environment. They are built on lower level *"BSP drivers"*.
These drivers must make interfaces available for the higher levels to access the hardware. They should have *getter* and *setter* functions.
In future, for porting the application to another hardware, manipulation in this layer is necessary.

Component	Description
MRAM Driver	Magnetoresistive Random Access Memory is used here by the Supervisor system to store configuration related data. In the future, the same memory can be used for software updates for the Supervisor system. As seen in the block diagram, one MRAM is connected to Vorago, NVM3. This driver is developed for thread safe operation using SPI library from Vorago's BSP.
SPI-NOR Flash	SPI NOR flash is one of the most important and most used components on the board. Three of them are on-board and one of them connected to Vorago, sharing SPI lines with MPSoC. This driver is developed for thread safe operation using SPI library from Vorago's BSP.
UART Driver	Vorago provides a BSP UART library though it is necessary to make it thread safe and available with interface to the application.
GPIO Driver	Vorago provides a BSP GPIO library though it is necessary to make it thread safe and available with interface to the application.
ADC Driver	Various power rails on board are being monitored by the Supervisor system, reading a setting threshold is made possible using the ADC peripheral driver.
Debug	Debug is made available using UART2 interface, this interface allows debug prints and generation of .log files using Segger tools.

Table 5.4: Components of HAL Utility Layer

Middleware

The middleware layer has the property to provide services to the application. It provides the running applications access to the hardware through the *HAL layer*. The middleware provides system abstraction to application software to increase application portability. It is important to note that the components in middleware should be free of any hardware-realted APIs.
Here, number of *"Handlers"* are involved. The term handlers refers to their function to handle hardware access requests from application. These handlers interface the hardware with the application, they need to be designed according to interfaces made available by the hardware-dependent layer.
These components need to be vary of available interfaces and available hardware resources, hence fixed interfaces from lower levels must be present.

Component	Description
TC Handler	TC Handler collects data from the UART driver and uses a state machine to generate spacepackets for consumption by the application
TM Handler	Every TC will be acknowledged by a TM. This TM from Supervisor has to be relayed to the OBC on thread-safe UART.
SpacePacket driver	SpacePacket protocol from ECSS Standards is being used. This protocol ensures hardware independent implementation for parsing and generating spacepackets.
Boot Manager	During SoC boot, a very careful execution is expected from Supervisor. The boot manager component ensures that the best-known previous state of the SoC is restored on boot-up.
Latch-up monitor	Current spikes or latches can cause a major damage to connected electronics. Since Supervisor system is designed to protect the payload computer from such latches, a spike must be detected by the Supervisor and an action must be taken to ensure all components on board are kept safe.
SoC Watchdog Handler	Watchdog feature for SoC regularly keeps checking heartbeat signals from the MPSoC and reports/corrects abnormalities.
PPS Generator	To have a time sync between all connected components, a pulse per second signal is generated from Supervisor.
Error handling	All errors on hardware must be checked and reported to the OBC/Ground station.

Table 5.5: Components of the Middleware Layer

Application layer

Application layer has user-defined tasks to build upon the middleware and fulfil the mission criteria. These are all runnable tasks calling run functions from the handlers. The tasks are created and started by using *xCreateTask()* function in the main function.

5.5 List of tasks

Table below summarizes the different FreeRTOS tasks in Application Layer, with a brief description and its priorities. The priorities are assigned from a software criticality study (not in this scope). Higher the criticality factor, higher is the priority. For an example, Latch-up monitoring will be highest priority as it is linked to the on-board power and if there is an abnormality in that, the mission fails hence it is a mission critical task.
The scheduler is able to select the next task to run in every time slice. The length of the time slice is defined by the tick interrupt frequency parameter *(con-*

Task	Description	Priority
TC Manager	TC Handling: receive from OBC, yield and route TC. Waiting for TC from OBC. Gives back an acknowledgment to the OBC.	1
TM Manager	TM Handling: Receive from SoC, yield and route TM, also TM Logging in NVM3. Waiting for TM from SoC and generating TM(ack/reports) for OBC.	1
Latch-up manager	Monitoring current on the power rails and checking for any abnormalities. Polling for current from 6 ADCs.	1
PS Heartbeat	MPSoC-PS heartbeat as an input for the Watchdog system	2
PL Heartbeat	MPSoC-PL heartbeat as an input for the Watchdog system	3
Timekeeping	Generate time sync PPS to PS and PL	3
SoC Health report	Timed reports about SoC health to OBC using TM, also TM Logging in NVM3.	3
Debug	Debug Interface (JTAG,SWD, UART2 serial interface)	4
SoC Updates	Task created only when correct TC is received. Otherwise idle task.	1

Table 5.6: List of Application tasks

figTICK_RATE_HZ) at compile time within *FreeRTOSConfig.h*.
Possible scheduling policies are: pre-emptive, round-robin and cooperative scheduling.
In pre-emptive scheduling, FreeRTOS schedules tasks according to their priority. The FreeRTOS scheduler ensures at every time slice, that tasks in the *Ready* or *Running* state will always be given processor time in preference to tasks of a lower priority that are also in the ready state. In other words, the task placed in the *Running* state is always the highest priority task that can run.
The round robin (time slicing) applies when we have tasks which share the same priority. If *configUSE_TIME_SLICING* is set to 1, then *Ready* state tasks of equal priority will share the available processing time using a time-sliced round-robin scheduling scheme.
The selected scheduling method is a combination of pre-emptive scheduling with round robin (time slicing). This is to ensure that at any point of time higher priority task is given the processor time, while if there are more than one task of same priority, they share the time using time-slicing.
NOTE: Current FreeRTOS configuration Priority based scheduling with round robin in FreeRTOS:

```
#define configUSE_PREEMPTION    1
#define configUSE_TIME_SLICING  1
```

```
#define configTICK_RATE_HZ      ((TickType_t)1000)
```

5.6 Resource Utilization

The following study was done to determine preliminary RAM utilization by each task.

Item	Bytes Used
Scheduler Itself	236 bytes (can easily be reduced by using smaller data types).
For each queue you create, add	76 bytes + queue storage area
For each task you create, add	64 bytes (includes 4 characters for the task name) + the task stack size.

Task	Stack size	Queue size	Task Notification	RAM
TC Manager	1024 + 64	1096 + 76	8	2268
TM Manager	1024 + 64	1096 + 76	8	2268
Timekeeping	256 + 64			320
Read PS heartbeat	256 + 64		8	328
Read PL heartbeat	256 + 64		8	328
SoC Health report	256 + 64		8	328
Latch up monitoring	512 + 64		8	328
Debug	256 + 64			320
SoC Update Task	1024 + 64	1096 + 76	8	2268
FreeRTOS Scheduler				~236
BSP				~36000
Total				**~44992**

Table 5.7: Estimated RAM utilization

There are the following memories available in the Supervisor system:

- 64 Kbyte SRAM data memory: 32 Kbyte on the data bus and 32 Kbyte on system bus
  - SRAM1 is used as Static memory handling *(.bss .data)* and also to allocate the heap for handling dynamic memory allocation, including storage of the "stack" of the FreeRTOS tasks.
  - SRAM2 is allocated currently for DMA.

- 256 Kbyte SRAM instruction memory, loaded from the SPI based memory (256Kbyte serial NVM) or from external memory on the external bus interface at startup. In ARM architecture, this memory can also be used as data memory but it reduces the throughput.
- External memories:
  - NVM1; shared between Supervisor and SoC, and
  - NVM3; Supervisor's configuration memory, can be also used for Supervisor software updates in the future.

5.7 Interfaces

The middleware has to communicate with the lower layers, hence a generic interface has to be set up. This interface will be useful to share data between the hardware and the software.
To define these, an interface diagram is created using the layered architecture as a reference from Figure 5.6.

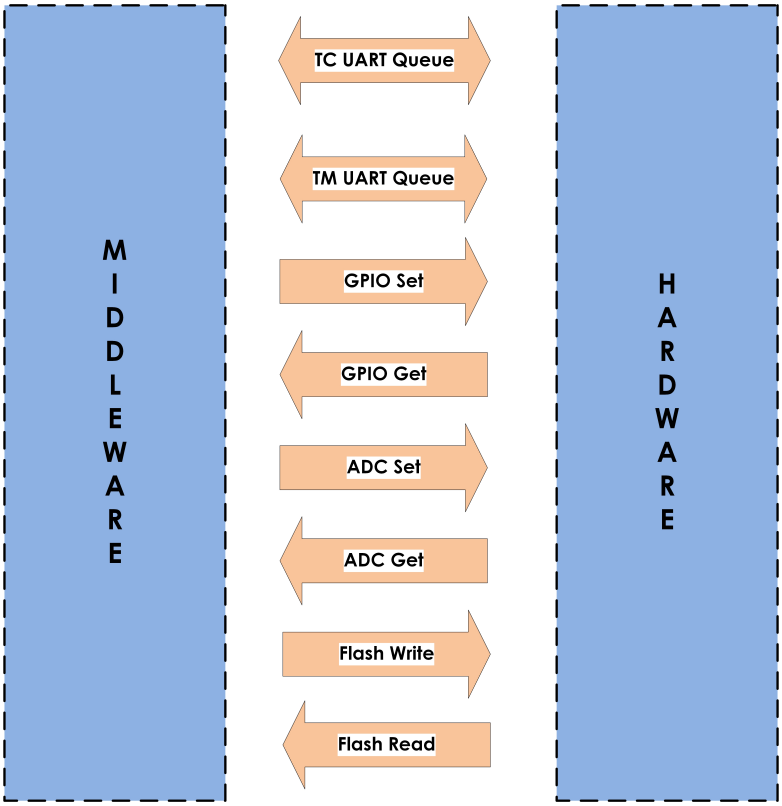

Figure 5.7: Interfaces between hardware and middleware

Chapter 6

Detailed Design

6.1 TM/TC Communication

The processing board receives Telecommands from the ground station using TTC link on S-band. This is collected by the on-board computer using a S-band antenna. This is relayed to Supervisor for further action or response. In this TTC

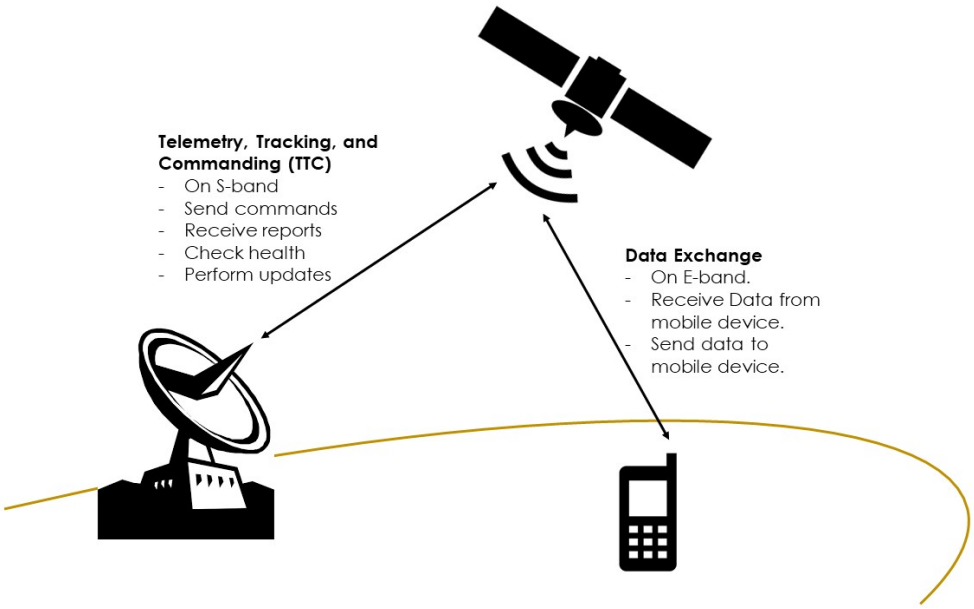

Figure 6.1: TTC Communication

communication, the ground station can send a service request while the satellite can send a regular report and state of the payload electronics.
A **"telecommand packet"** is a data packet carrying a service request from the ground station to the satellite.
A **"telemetry packet"** is a data packet carrying a service report from satellite to the ground station. This can be a reply to a telecommand or just a standalone packet.

In this project, the on-board computer is responsible for the TTC communication. For all requests for the Supervisor or SoC, the on-board computer forwards the requests using a dedicated UART interface.

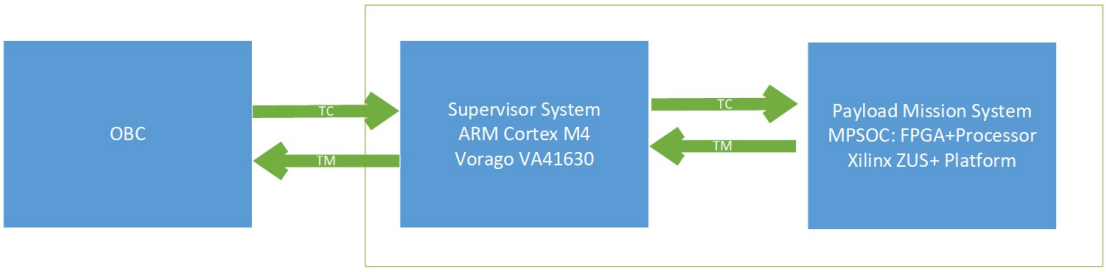

Figure 6.2: TM/TC Architecture

6.1.1 Introduction to SpacePacket Protocol

As seen in the requirements above, space packet protocol from CCSDS must be used to perform communication between the on-board modules. This protocol is defined in the document **ECSS-E-ST-70-41C** [34] specifying a lightweight but descriptive packet structure. The packet based communication architecture is designed for data transfer between nodes, space-to-space, space-to-ground, and for on-board communication. Few saliant features of the SpacePacket protocol are as below,

- 6-byte primary header,
- Optional secondary header,
- Variable length data field (upto 65536 bytes),
- Primary header can be customized for 11-bits of Application ID,
- Start and end of burst packet can be defined,
- Designed to be independent of higher and lower application interface.

packet primary header							packet data field	
packet version number	packet ID			packet sequence control		packet data length	packet secondary header	user data field
	packet type	secondary header flag	application process ID	sequence flags	packet sequence count or packet name			
3 bits	1 bit	1 bit	11 bits	2 bits	14 bits	16 bits	variable	variable
2 octets				2 octets		2 octets	1 to 65536 octets	

Figure 6.3: Standard SpacePacket Protocol from CCSDS [34]

The Figure 6.3 shows the standard packet structure as given by CCSDS document. While, Figure 6.4 represents customized yet standard packet structure to be used in this project. The 11-bits of Application ID have been divided to specify the source, destination, and purpose of the packet.

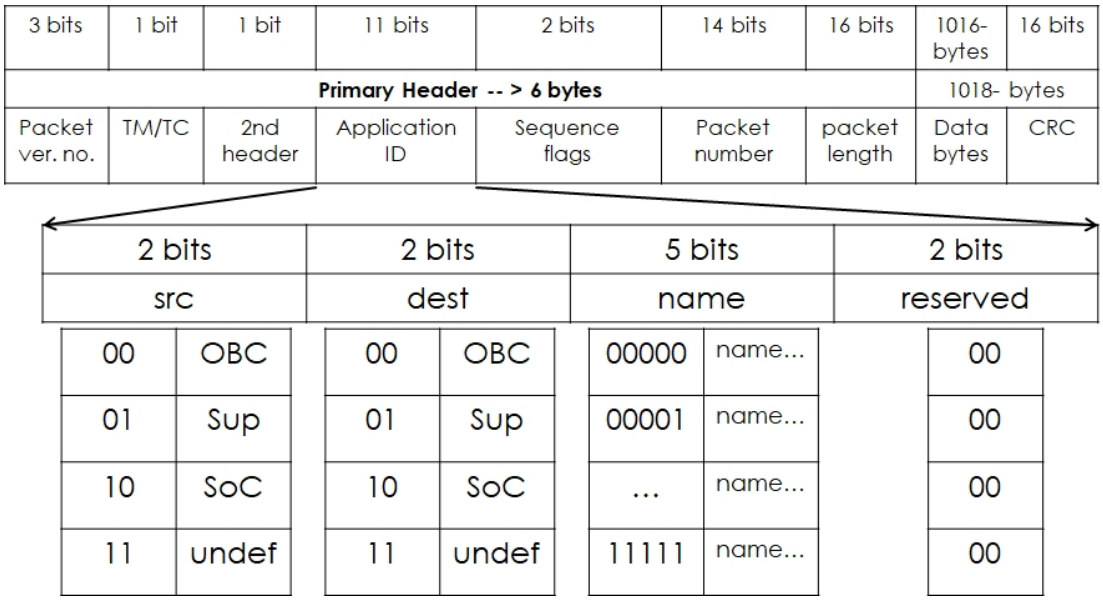

Figure 6.4: SpacePacket protocol with customized Application ID

The SpacePacket protocol restricts the overall packet length to 1024-bytes to save RAM on all modules of the board. An additional 16-bit CRC is involved to keep a check on packet integrity.
This packet definition is derived from a very similar approach being used in standard software of OBC [3].

6.1.2 Design

UART Driver

TM and TC communication on Supervisor is done via UART, *viz.* UART0 for communication with OBC while TM communication to and from SoC. To facilitate the UART Driver must be capable to read, store and process the data on UART(s). The UART Driver can be made with the following three options,

1. **Interrupt based:** Vorago has RX and TX interrupt for UART which can be configured to an registered event in half/full buffer conditions. This condition is raised from STATUS register of the specific UART.

2. **UART Polling:** User can specifically poll the STATUS register for the UART and read data from the DATA register. This eliminates the need of an event of Interrupt Service routine. Vorago provides functions to get the length of RX data and also grab that data.

3. **UART with DMA:** Currently not supported, future use case.

A polling version of the driver is used because interrupts can ruin the round robin scheduling and can generate a non-deterministic nature for the system which must be avoided.
At the fixed UART baud rate of 115200, we can get 14.4 bytes per millisecond. Since Vorago has a 16 byte UART FIFO buffer, we can poll every 1ms to grab all data and never miss any communication on any UART.

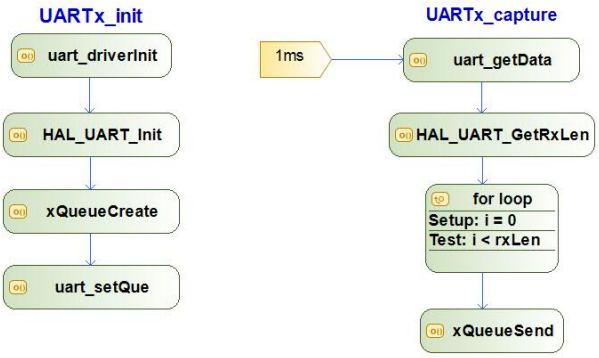

Figure 6.5: UML Diagram for UART Driver

The hardware FIFO for Voargo's UART is 16-bytes and to make sure, we do not miss any data, two tasks are run every 1 millisecond to grab data from the FIFO and put on the respective Queue, as shown in Figure 6.5. **NOTE:** UART capture is run as a task, defined in *HAL* of the software using private members of the driver.

SpacePacket library

As discussed above, a rendition of SpacePacket protocol has been used in this project. To accommodate this, a very generic spacepacket library was developed in C-programming language. This library was designed and developed to be independent of the higher and lower layers of the application. The library is developed to be lightweight in terms of memory and computation time so that it can also be used in microcontrollers such as Vorago, in this case.

TM/TC Handler

The UART receive task grabs data in bytes, this must be packatized to be utilized by the application. As seen from the Figure 5.7, a *Queue* is made available as an interface to be utilized by the TM/TC handlers. A state machine is run over the Queue to receive and parse data in the queue. This state machine also finds and reports *rxError*.
The state machine is designed as in Figure 6.6. This state machine is responsible for parsing data collected by the UART driver from the *HAL layer*.

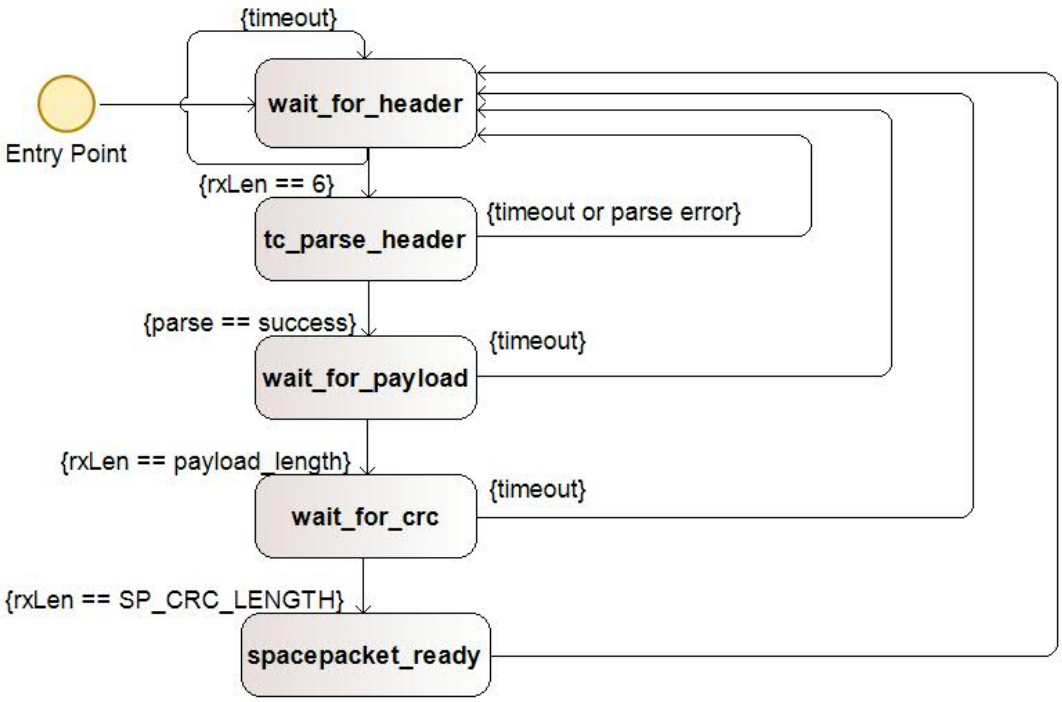

Figure 6.6: State Machine for parsing bytes to SpacePacket

The TM and TC handlers run this state machine as in Figure 6.7 and 6.8. These functions allow the collected data on UART to be used in a SpacePacket and further parsed to allow processing of the respective TC or TM. This process yields in a SpacePacket which is verified and checked for its CRC before passing on to the next layer.

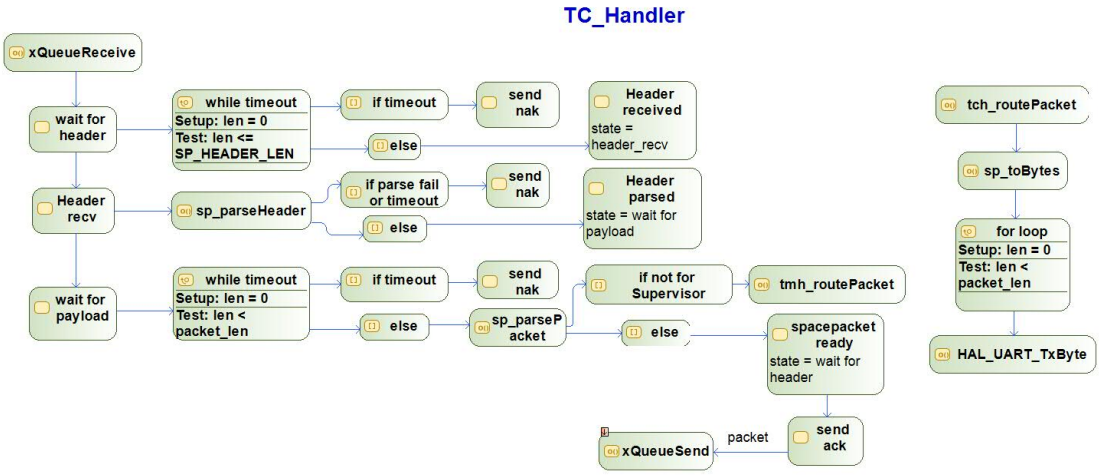

Figure 6.7: TC Handler running activity diagram

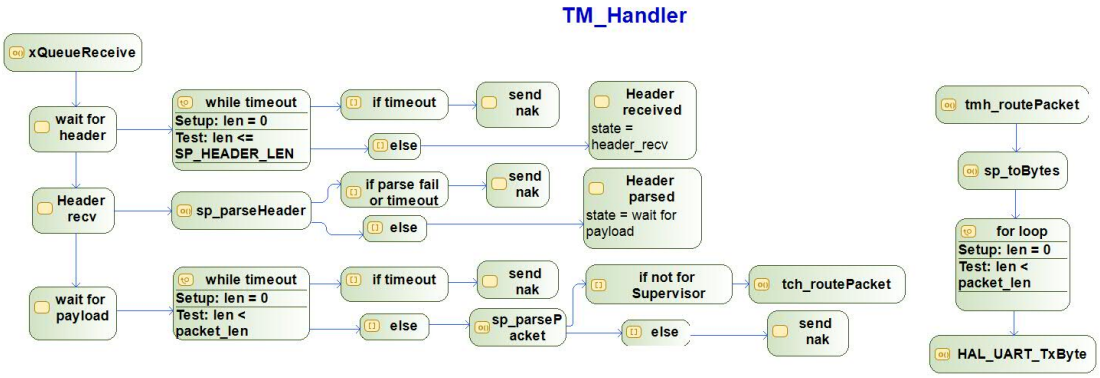

Figure 6.8: TM Handler running activity diagram

TM/TC Manager

Application layer runs task to receive spacepackets as *packets* in a designated queues.These queues are then checked for the source, destination, and purpose for further execution. The execution is shown in Figures 6.9 and 6.10.

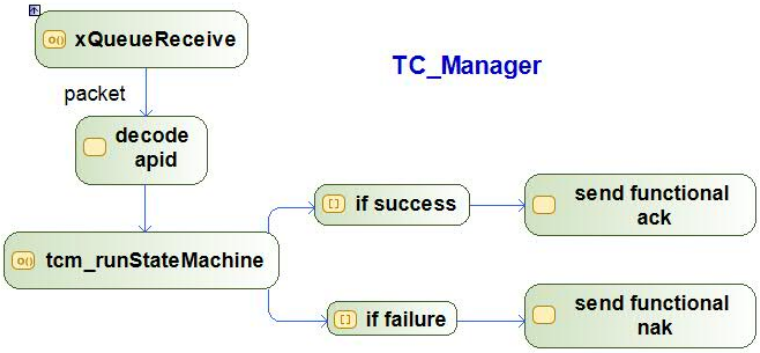

Figure 6.9: TC Manager Task

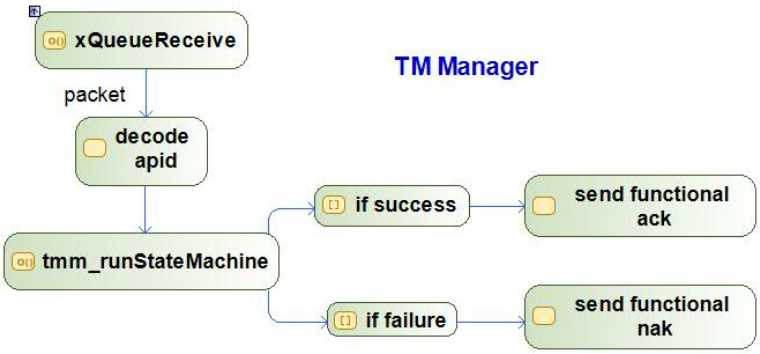

Figure 6.10: TM Manager task

6.2 Software updates

During the mission, the ground station can order a configuration change using a telecommand through the OBC. This can mean change in operations and possibly an update in few features/functions. But a change in algorithm or core functionalities of the SoC is very difficult.
In this case, the processing board allows change in core functionalities of the SoC using an external memory for boot. This will allow change in services and data processing algorithm over fly making the satellite flexible and future ready.

6.2.1 SoC software updates

The OBC receives TTC from the ground station, as in Figure 6.1 allowing the ground station to also send a firmware update over the air. This package includes VHDL firmware and application software for the SoC. The Supervisor module can parse the packets, understand it, and update the boot memory. As a signal, the Supervisor module can generate a *Force Golden* signal to indicate use of updated firmware. This signal is read by the First Stage Bootloader (FSBL) during the boot leading to the correct boot partition. *This boot process is not a part of this thesis.*
Extending the Figure 6.2 following block design can be made for explaining the SoC software update function using the already existing TM/TC Architecture.

The software for this function is also built upon the already existing TM/TC Software as discussed above. The TC reaches the *TC Manager* as discussed in the Section 6.1.2. On parsing the SpacePacket, the software can go through four stages for the update function.

Update Available Stage

A TC from OBC is sent with all the update parameters which includes,

- Update partition,
- Size of update,
- Type of update file,
- 32-bit image CRC.

This packet allows the Supervisor to create a new task and pass these parameters for memory initialization. During this process, any errors lead to a negative acknowledgement to the TC. The new task starts with memory initialization/clean-up and goes on to wait on a queue for data bytes.

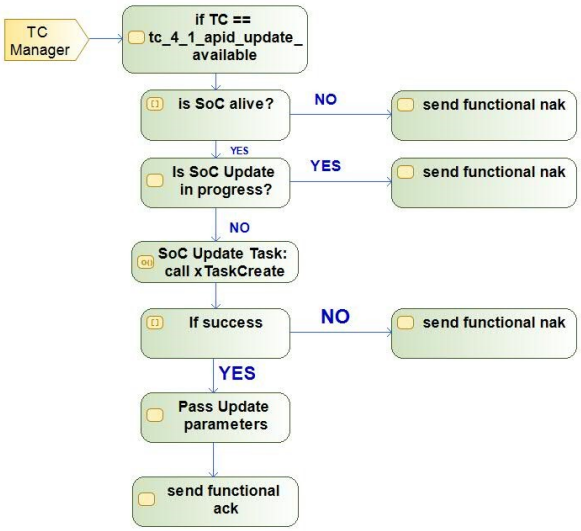

Figure 6.11: SoC Update available on TC Manager task

Update Data Bytes Stage

The update file is sent over packets of 1024-bytes from the OBC to the Supervisor as burst TCs which come through to the *TC Manager*. Here, the packet data is put on a shared queue for the *SoC Update task* to retrieve and use.
The task reads data from the queue and writes it to the memory with correct address pointer. On the last packet, the *SoC Update* task kills itself, saving some memory.

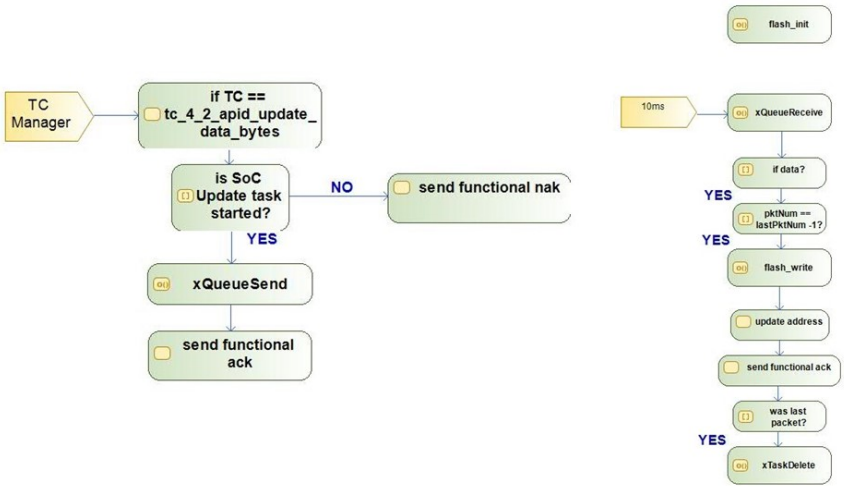

Figure 6.12: SoC Update data bytes on TC Manager and update task

Update Verify Stage

Verification of the update is done using a 32-bit CRC of the image using a TC from OBC to Supervisor system. The Supervisor can then read the corresponding memory and generate it's own 32-bit CRC dynamically for comparison.
On success or failure, a TM is sent back to the OBC with negative/positive acknowledgement.

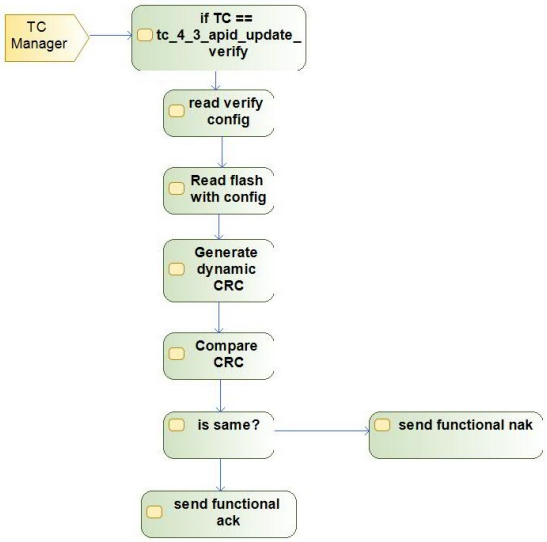

Figure 6.13: SoC Update verify on TC Manager task

Update Execution Stage

The OBC has to order the Supervisor for running the updated firmware on the SoC. This comes to Supervisor as a TC and Supervisor has to then take care of the update procedure. Before passing the control to *Boot Manager*, the *TC Manager* checks for the verification status. The *Boot Manager* handles safe closure of all the interfaces/tasks, signalling the *Force Golden*, and rebooting the SoC.
After the boot-up, *Boot Manager* restarts the respective tasks, interfaces, and sends a TM report back to the OBC.

6.3 Results

Software design and development was done for the the above functions using the above designed software design and standard software development procedures. The test was done by developing a OBC-Emulator on a Linux PC to create a lab environment for the to be satellite software. This OBC emulator was tested for quality check and passed assessment before using with the real hardware. Additionally, USB and jumper cables were used to go through the tests.

Test Setup

To test the software architecture and running software, a test software was developed on a Linux PC emulating an OBC namely *"OBC Emulator"*. This allowed testing the supervisor software with some standard TMs and TCs. Similarly, to emulate an SoC, a test software was developed on a Linux PC namely *SoC Emulator*. This test setup gives an ideal environment for tests since eventually OBC and SoC will run a version of Linux on them. The test software uses same module from Supervisor software *i.e.* SpacePacket library, proving a level of modularity already.

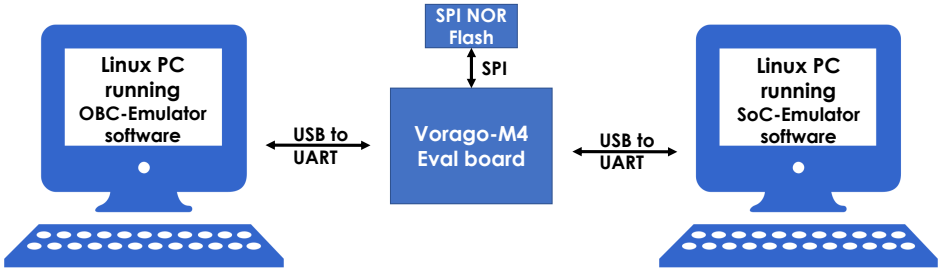

Figure 6.14: Test setup for TM/TC and SoC Software Updates

This setup was made to test the functionality as well as quality of the software architecture and developed software. It allowed for generating results at this stage. In the future, similar will be done on real hardware. **NOTE:** OBC-Emulator and SoC-Emulator software/architecture is not in scope of this thesis.

Outputs

The software package on Vorago-M4 allows debug prints and generation of log files for these debug prints. Source of the below images are these log prints from a standard text editor.

```
00> DEBUG: In vUart1_Recv
00> DEBUG: In vTaskTM_Manager
00> DEBUG: In vTaskTC_Manager
00> DEBUG: In vTaskLatchup_Mon
00> DEBUG: In vUart0_Recv
00> DEBUG: In vTaskPS_Heartbeat
00> DEBUG: In vTaskPL_Heartbeat
00> DEBUG: In vTaskTimekeeper
00> DEBUG: In vTaskHealth_Report_Gen
00> DEBUG: Spacepacket parsed
00> DEBUG: tc_5_2_apid_watchdog_enable received
00> DEBUG: ack tm_1_1_apid_ack sent
00> DEBUG: ack tm_1_7_apid_execute_ack sent
```

Figure 6.15: A TC from OBC for Supervisor; consumed and acknowledged

```
00> DEBUG: In vUart1_Recv
00> DEBUG: In vTaskTM_Manager
00> DEBUG: In vTaskTC_Manager
00> DEBUG: In vTaskLatchup_Mon
00> DEBUG: In vUart0_Recv
00> DEBUG: In vTaskPS_Heartbeat
00> DEBUG: In vTaskPL_Heartbeat
00> DEBUG: In vTaskTimekeeper
00> DEBUG: In vTaskHealth_Report_Gen
00> DEBUG: Spacepacket parsed
00> DEBUG: tc_5_2_apid_watchdog_enable received
00> DEBUG: ack tm_1_1_apid_ack sent
00> DEBUG: TC not for Sup.. Routing now..
```

Figure 6.16: A TC from OBC for the SoC; routed and acknowledged

The above images, show starting of the FreeRTOS tasks and reception, handling and consumption of a Telecommands from OBC. The Telecommands is replied using acknowledgements, which are a Telemetry with status code, reference packet number, and other details.
For the SoC Update task, the figure below shows all the states of the update. During this test, a 1000 byte file was transferred from the OBC to the Flash memory.

```
00> DEBUG: In vUart1_Recv
00> DEBUG: In vTaskTM_Manager
00> DEBUG: In vTaskTC_Manager
00> DEBUG: In vTaskLatchup_Mon
00> DEBUG: In vUart0_Recv
00> DEBUG: In vTaskPS_Heartbeat
00> DEBUG: In vTaskPL_Heartbeat
00> DEBUG: In vTaskTimekeeper
00> DEBUG: In vTaskHealth_Report_Gen
00> DEBUG: Spacepacket parsed
00> DEBUG: tc_4_1_apid_update_available received
00> DEBUG: ack tm_1_1_apid_ack sent
00> DEBUG: ack tm_1_7_apid_execute_ack sent
00> DEBUG: SoC Update Task started..
00> DEBUG: Spacepacket parsed
00> DEBUG: tc_4_2_apid_update_imageData received
00> DEBUG: ack tm_1_1_apid_ack sent
00> DEBUG: ack tm_1_7_apid_execute_ack sent
00> DEBUG: Update Ended.. Killing SoC Update task
00> DEBUG: Spacepacket parsed
00> DEBUG: tc_4_4_apid_update_verify received
00> DEBUG: ack tm_1_1_apid_ack sent
00> DEBUG: CRC Verification successful..
00> DEBUG: ack tm_1_7_apid_execute_ack sent
```

Figure 6.17: SoC Update debug prints for all states

6.4 Architecture Porting

The software architecture is designed and developed such that it can be reused and ported for future space products, allowing software modularity. This means adapting to foreseen hardware changes for the Supervisor system. This can be done using porting of the *HAL* to accommodate the changes in respective *BSP*.
To prove this, a similar microcontroller was selected for a test. The chosen microcontroller was STM32F407VG[35] with its development board DISC1 - STM32F407 [36] in a breadboard condition. This microcontroller was selected due to its similarity with Vorago-M4. Few notable features of this microcontroller are,

- ARM Cortex-M4F architecture.
- Upto 168Mhz clock speed.
- SPI/I2C/UART/ADC/DAC/CAN Interfaces.
- 1Mb internal flash memory, 196kb SRAM.
- No radiation hardening.

During this test, the function of TM/TC was ported to STM32 architecture by making changes to the *UART Driver*[refer Section 6.1.2]. The *Middleware* and *Application* layer remains unchanged due to it's *Hardware-independent* nature. The *Operating System* layer remains the same due to similar hardware architecture. While, the *BSP* will change to the one provided by STM32[36].
NOTE: Changing the Operating system will mean change in the interfaces also.

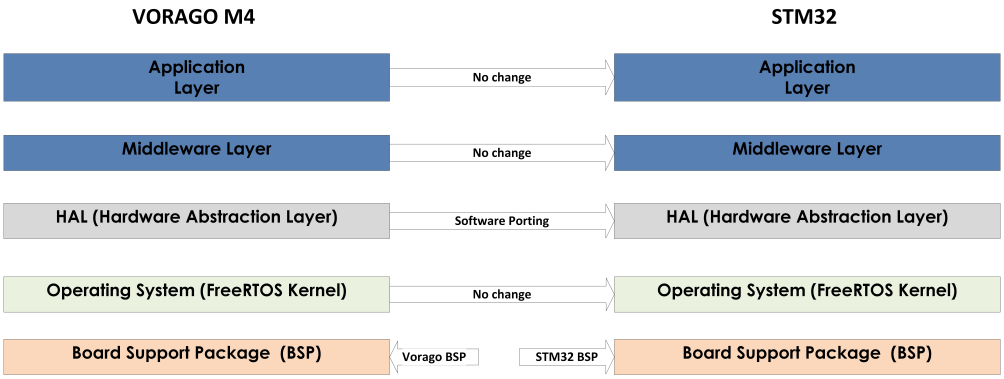

Figure 6.18: Porting of software architecture to STM32

As a result of such porting, the following were the findings. The image below, shows results from Vorago-M4 and STM32 side-by-side for comparison.

```
00> DEBUG: In vUart1_Recv
00> DEBUG: In vTaskTM_Manager
00> DEBUG: In vTaskTC_Manager
00> DEBUG: In vTaskLatchup_Mon
00> DEBUG: In vUart0_Recv
00> DEBUG: In vTaskPS_Heartbeat
00> DEBUG: In vTaskPL_Heartbeat
00> DEBUG: In vTaskTimekeeper
00> DEBUG: In vTaskHealth_Report_Gen
00> DEBUG: Spacepacket parsed
00> DEBUG: tc_5_2_apid_watchdog_enable received
00> DEBUG: ack tm_1_1_apid_ack sent
00> DEBUG: ack tm_1_7_apid_execute_ack sent
```

Figure 6.19: TM/TC debug prints on Vorago-M4

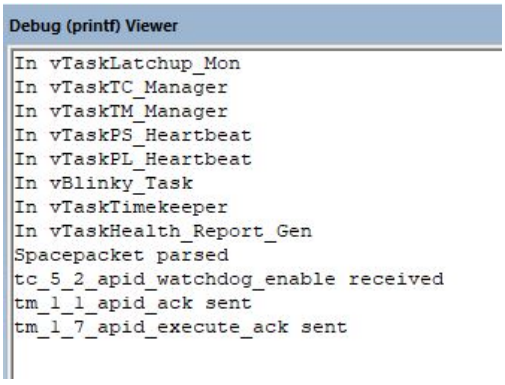

Figure 6.20: TM/TC debug prints on STM32F4

NOTE: UART capture tasks are not seen in the STM32F4 version of debug prints because, STM32 version of capture process is done using interrupts. This change is only made in *HAL* layer and does not affect the other layers in anyway.

From the above test it can be concluded that designed software architecture is modular and can be ported to different hardware in the future. This is an important factor for this specific mission as the product is in its initial stages and future hardware changes are foreseen. It can also be seen how such *Supervisor system* (hardware and software) can be ported to similar products.

Chapter 7

Evaluation

On studying the state of the art, reference architectures, and software architecture standards for satellites a software architecture was designed for the Supervisor system. As part of the thesis few software functions were also developed based on the software architecture. These functions already prove stability of the architecture but in this chapter, detailed evaluation of the software architecture is done.
The hardware requirements as discussed in Chapter 4 were met by using Vorago-M4 microcontroller as Supervisor module. The comparison study can be seen in Section 5.1. But since the thesis focuses on software architecture, we study in detail about the designed architecture as compared to baseline discussed in Section 4.2.

1. **Real-time:** The goal for this requirement is to have real time performance for the system with near to instant reaction to an event. This can be achieved by using schedulers in the software. Various schedulers are available in the market, they were studied, compared, and FreeRTOS [26] was chosen. This has been discussed in Section 5.2 at length. The software architecture is designed with a dedicated layer for inclusion of an Operating system. The scheduler in this RTOS is proven for its real time performance. The RTOS allows definition of tasks that perform functions independently. During the tests, all the tasks were seen to run independently.

2. **Functional stability:** The Supervisor system has some predefined minimal functionality, as discussed in Section 3.5. The software architecture should allow development of these functionalities. The software architecture is designed in layers to allow these functions to be built using the existing hardware and hardware dependent layer. Hardware dependent *BSP* layer allows optimal use of hardware and its functionalities. This has been seen from the above developed functionalities and their results as in Section 6.3.

3. **Modularity:** The architecture needs to be designed such that it is broken down in modules to allow independent and interchangeable functionality. The product requirements can change rapidly and having such modules will allow the software to be modified and reconfigured to allow the said change.

In design, the software architecture is divided in layers and independent modules in those layers to fulfil the requirement. The architecture is designed keeping in mind standard interfaces as discussed in Section 5.7.

4. **Reusability:** The software architecture must be designed in a way that it can also be used in other similar products when needed. The designed software architecture has *Middleware* and *Application* layer which allows the software to be reused in similar systems. This is supported by the designed software interfaces as in Figure 5.7.

5. **Scalability:** The product in discussion is a new concept and future expansion is foreseen. In this case, the software architecture has to be designed such that addition of features in the future is possible. The designed architecture allows build up of any additional functionality in the *Application* layer is possible as long as hardware supports it. Currently, the software in development is being expanded to include more features using the software architecture as base.

6. **Portability:** The processing board is at its initial development phase and in the future improvement or change in hardware is also foreseen. This must be accounted in the software architecture allowing the architecture to be reused when the hardware changes. In the design, architecture is such that the application layer will have no effect of the change. The hardware dependent layers may change with hardware but the hardware independent layers should be unaffected. Porting of *HAL* layer is possible while keeping the interfaces to *Middleware* constant. To try such a porting, an experiment was carried on by changing the hardware and porting the *HAL* components to find that the architecture can remain stable in this situation also. This experiment is discussed in Section 6.4.

For all other functional evaluation, the respective sections explain more.

Requirement	Goal	Evaluation
Real-time	Achieve real-time performance.	By using RTOS as a dedicated layer, real-time performance can be guaranteed.
Functional stability	Achieve minimal functionality for Supervisor system.	Functionalities separated as tasks, independent layer for applications to be built.
Modularity	Generate modules and also apply them in future	Architecture is made layered with each layer having its independent components.
Reusability	Software to be resued in similar product.	Architecture has separate layers for middleware and application with standard interfaces.
Scalability	Allow addition of additional features in the future.	The application layer is a separate entity, allowing expansion in the future.
Portability	Software architecture remains same even after a hardware revision.	Hardware dependent and independent layers allow separation. Only change in HAL layer for porting.

Table 7.1: Summary of the evaluation

To sum up the evaluation, the software architecture is built upon the requirements from the reference software architecture. It satisfies all the requirements and quality standards defined in Chapter 4.
Hence, the software architecture and software is characterised by quality standards defined by ECSS-Q-ST-80C [21].

Chapter 8

Conclusion

Satellite based communication for IoT when put in practice will need a fast, error-free and lightweight solution. The discussed solution not only promises answers to all of the above problems but provides a platform for future satellites to build upon. Such a Supervisor system in satellites will help to have a robust solution in space of unknowns. With inclusion of a Supervisor system will also help build on services for the satellite functionality. In the future, it is foreseen that the Supervisor system replaces the OBC as the controlling module, taking over the satellite bus.

The software architecture for this Supervisor system is designed and developed such that it can be reused and modelled for future satellites to be used in the IoT world. To have such an architecture gives freedom to the user/buyer to utilise the services and customize for given use-case. Modularity makes the architecture portable and yet robust. The current state of the software architecture and software is rigid and designed to be fail-safe.

Health reporting from Supervisor provides an important insight for the ground station about the satellite. The TM/TC communication from/to On-board Computer (OBC) helps to collaborate all the functions and setup a smooth communication amongst all the modules. Mission power monitoring will preserve the payload electronics for longer duration, hence adding on to the robustness and mission-life.

Functional re-configurations and software updates will make sure the satellite is future-ready and can take on problems that arise during the mission. This leads to continuous improvements in the functioning and steady maintenance of the satellite.

As evaluated, the software architecture builds upon the requirements from standards and is designed such that it can be reused in future missions and payloads. The architecture is also designed to be portable from one hardware to another, this allows for the product to evolve and improve over time. The built software will allow real-time performance and functional stability.

The development of this system is still in progress using above design and implementation. In the future, the Supervisor software will evolve and improve.

Few future functions that can be foreseen are as below,

- Design and Development of Supervisor software updates to make the software future ready. This will allow more flexibility and re-configurations.
- Development of Packet Utilization Standard (PUS) as a *Middleware* component to comply with any future needs of ground station.
- Development of microlatch-up algorithm using the already built *HAL* component, *Current Monitoring*. With such detection mechanism, slightest of power anomaly can be detected and mitigated.

Bibliography

[1] J. A. Fraire, S. Céspedes, and N. Accettura, "Direct-To-Satellite IoT - A Survey of the State of the Art and Future Research Perspectives," in *ADHOC-NOW 2019: Ad-Hoc, Mobile, and Wireless Networks*, (Luxembourg, Luxembourg), pp. 241–258, Oct. 2019.

[2] J. Garland and R. Anthony, *Large-Scale Software Architecture: A Practical Guide Using UML*. Wiley Publishing, 1st ed., 2002.

[3] J. Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations*, pp. 85–165. Springer, 2012.

[4] "Vorago VA41630." https://www.voragotech.com/products/va41630. Accessed: 2020-06-21.

[5] "PEB1-VA416x0 Development Kit." https://www.voragotech.com/products/peb1va416x0-development-kit. Accessed: 2020-10-01.

[6] "TE0803-03-3BE11-AS Starter Kit with Zynq UltraScale+." https://shop.trenz-electronic.de/en/TE0803-03-3BE11-AS-TE0803-03-3BE11-AS-Starter-Kit-with-Zynq-UltraScale-ZU3-FPGA-Module. Accessed: 2020-10-04.

[7] "Xilinx Design Suite." https://www.xilinx.com/products/design-tools/vivado.html. Accessed: 2020-10-10.

[8] "Keil IDE with MDK5." http://www2.keil.com/mdk5. Accessed: 2020-10-04.

[9] "MODELIO for UML." https://www.modelio.org/. Accessed: 2020-10-04.

[10] "Eclipse IDE for C/C++ Developers." https://www.eclipse.org/downloads/packages/release/2019-09/r/eclipse-ide-cc-developers. Accessed: 2020-10-04.

[11] L. Belli, L. Davoli, A. Medioli, P. L. Marchini, and G. Ferrari, "Toward industry 4.0 with iot: Optimizing business processes in an evolving manufacturing factory," *Frontiers in ICT*, vol. 6, p. 17, 2019.

[12] M. A. Jabraeil Jamali, B. Bahrami, A. Heidari, P. Allahverdizadeh, and F. Norouzi, *IoT Architecture*, pp. 9–31. Cham: Springer International Publishing, 2020.

[13] J. Guth, U. Breitenbücher, M. Falkenthal, P. Fremantle, O. Kopp, F. Leymann, and L. Reinfurt, *A Detailed Analysis of IoT Platform Architectures: Concepts, Similarities, and Differences*, pp. 81–101. Springer, 2018.

[14] N. Mohan and J. Kangasharju, "Edge-fog cloud: A distributed cloud for internet of things computations," 02 2017.

[15] S. K. Routray, R. Tengshe, A. Javali, S. Sarkar, L. Sharma, and A. D. Ghosh, "Satellite based iot for mission critical applications," in *2019 International Conference on Data Science and Communication (IconDSC)*, pp. 1–6, 2019.

[16] "What is a CubeSat?." https://www.asc-csa.gc.ca/eng/satellites/cubesat/what-is-a-cubesat.asp. Accessed: 2020-08-17.

[17] A. Poghosyan and A. Golkar, "Cubesat evolution: Analyzing cubesat capabilities for conducting science missions," *Progress in Aerospace Sciences*, vol. 88, pp. 59 – 83, 2017.

[18] "What is satellite payload?." https://info.alen.space/cubesat-payloads-what-can-you-put-in-a-small-satellite. Accessed: 2020-08-13.

[19] R. Wenker, C. Legendre, M. Ferraguto, M. Tipaldi, A. Wortmann, C. Moellmann, and D. Rosskamp, "On-board software architecture in mtg satellite," pp. 318–323, 06 2017.

[20] "Introduction to SAVOIR." https://savoir.estec.esa.int/. Accessed: 2020-10-06.

[21] "ECSS-Q-ST-80C Software Quality Standards." https://ecss.nl/standard/ecss-q-st-80c-rev-1-software-product-assurance-15-february-2017/. Accessed: 2020-07-08.

[22] "ISO/IEC 9126-1:2001 – Software engineering, Product quality." https://www.iso.org/standard/22749.html. Accessed: 2020-07-08.

[23] "Microchip SAMRH71." https://www.microchip.com/wwwproducts/en/SAMRH71. Accessed: 2020-06-25.

[24] "Arm Cortex M4." https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4. Accessed: 2020-06-21.

[25] "Keil's RTX." https://www2.keil.com/mdk5/cmsis/rtx. Accessed: 2020-07-17.

[26] "FreeRTOS." https://www.freertos.org/. Accessed: 2020-08-13.

[27] "μC-OS II RTOS." https://www.micrium.com/rtos/. Accessed: 2020-07-17.

[28] "RT-Thread RTOS." https://www.osrtos.com/rtos/rt-thread/. Accessed: 2020-07-17.

[29] "FreeRTOS Scheduling." https://www.freertos.org/implementation/a00005.html. Accessed: 2020-07-17.

[30] "FreeRTOS Memory Management." https://www.freertos.org/a00111.html. Accessed: 2020-09-17.

[31] "FreeRTOS Queues." https://www.freertos.org/Embedded-RTOS-Queues.html. Accessed: 2020-07-17.

[32] "FreeRTOS Task notification." https://www.freertos.org/RTOS-task-notifications.html. Accessed: 2020-10-08.

[33] "FreeRTOS Stream and Message Buffers." https://www.freertos.org/RTOS-stream-message-buffers.html. Accessed: 2020-07-17.

[34] ECSS Secretariat, ESA-ESTEC, Requirements and standard division, "Telemetry and telecommand packet utilization," *Standard*, vol. ECSS-E-ST-70-41C, p. 656, April 2016.

[35] "STM32F407VG Specification." https://www.st.com/en/microcontrollers-microprocessors/stm32f407vg.html. Accessed: 2020-10-10.

[36] "STM32F407G-DISC1 Specification." https://www.st.com/en/evaluation-tools/stm32f4discovery.html. Accessed: 2020-10-04.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 10.11.2020 Sarthak Kelapure