جامعة مصر الدولية
MISR INTERNATIONAL UNIVERSITY

Faculty of Engineering

Electronics and Communications Department

Design And Implementation of Satellite Communication System

By:

Mohamed Ezzat	**17/06220**	**Omar Abdel Zaher**	**17/11354**
Youssef Nasser	**17/09152**	**Ahmed Maged**	**17/09070**
Hana Mostafa	**17/09114**	**Michel Maged**	**17/12277**
Yassmine Ashraf	**17/09509**	**Amr Ahmed**	**17/08151**

Under Supervision of:

Dr. Haitham Medhat

Head of Space Telecommunication Department in EGSA

This thesis is submitted as a partial fulfillment of the requirements for the degree of Bachelor of Science in Electronics and Communications

Cairo, Egypt. 2022

Acknowledgements

We would like to express our sincere gratitude to several individuals for supporting us throughout our graduate study. First, a word of appreciation to our supervisor, Dr. Haytham Medhat, for his enthusiasm, patience, insightful comments, helpful information, practical advice and innovative ideas that have helped us tremendously at all times in our project and writing of this thesis. His immense knowledge, profound experience and professional expertise in Satellite communication.

Furthermore, Thanks to Dr. Mehaseb Ahmed for his continuous support and interest in our project by providing all the necessary information about the Egyptian Space Agency (EGSA).

Abstract

This thesis describes the process of implementing the software part of a communication sub-system and control board for satellite (CubeSat).The implementation consists of a communication protocol in the data link layer that utilizes AX.25 frames for communicating with a ground station based on amateur radio equipment , SSP is a simple protocol intended for master-slave communication on single-master serial buses, especially within small spacecraft which is used in internal communication between communication subsystem and OBC (on board computer). The development and usage of these communication protocols are based on data communication theory, link budget calculations, efficiency and availability analyzed together with practical tests.

Also, there is design and implementation of the PCB and electronics for the communication sub-system.

Table of Contents

Acronyms or Abbreviations

ACK	Acknowledged message
ARQ	Automatic Repeat Request
AX.25	Amateur Packet Radio Protocol X.25
BER	Bit Error Rate
COMM	Communication
CRC	Cyclic Redundancy Check
EGSA	Egyptian Space Agency
FCS	Frame-Check Sequence
FEC	Forward Error Correction
FSK	Frequency Shift Keying
GCS	Ground Control Station
GEO	Geostationary Earth Orbit
GFSK	Gaussian Frequency Shift Keying
GMSK	Gaussian Minimum Shift Keying
HDLC	High-Level Data Link Control
I frame	Information Frame
LEO	Low Earth Orbit
LSB	Least Significant Bit
MCU	Microcontroller unit
MIU	Misr International University
NAK	Negative Acknowledged message
OBC	On-board Computer
PCB	Printed Circuit Board
RF	Radio Frequency
SAT	Satellite
SDLC	Software development life cycle
S-Frame	Supervisory frames
SLIP	Serial-Line IP

SNR	Signal-Noise-Ratio
SPI	Serial Peripheral interface
SSID	Secondary Station Identifier
SSP	Simple Serial Protocol
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
UI	Unnumbered Information

List of figures

List of tables

1 Chapter 1 Introduction

1.1 History

Satellite communications are the outcome of research in the area of communications and space technologies whose objective is to achieve ever-increasing ranges and capacities with the lowest possible costs.

The World War II stimulated the expansion of two very distinct technologies – missiles and microwaves. The expertise eventually gained in the combined use of these two techniques opened up the era of satellite communications. The service provided in this way usefully complements that previously provided exclusively by terrestrial networks using radio and cables .

In order to assist colleges throughout the world in making space science and exploration more accessible to students and researchers, California Polytechnic State University at San Luis Obispo (Cal Poly) and Stanford University created a small spacecraft idea beginning in 1999. The goal was to develop an idea that would increase the likelihood of being accepted as a secondary passenger on a space launch while also enabling academic groups to quickly implement a minor space mission. This was made possible by standardizing interfaces and banning or restricting design elements that would be potentially dangerous and lessen the likelihood that they would be permitted to launch alongside larger, more expensive spacecraft. Professors Bob Twiggs of Stanford University and Jordi Puig-Suari of California Polytechnic State University came up with the CubeSat reference design in 1999. The CubeSat was not intended to become a standard when it was first proposed, but over time it evolved into one and eventually became a highly well-known and standardized satellite concept. In June 2003, a Rocket launch vehicle carried out the first CubeSat launches.

1.2 Motivation

The motivation was to find, adapt and test a suitable communication protocol that could be compatible with other EUTS CubeSat missions and compete international projects.

1.2.1 EGYPTIAN UNIVERSITY SATELLITE (EUTS)

This project demonstrated by the Egyptian Space Agency (EGSA) aims at developing the first Egyptian satellite that is locally developed by Egyptian universities students guided by Egyptian space experts of EGSA. The developed satellite will be a 1U CubeSat. The system design, integration, testing, launching, and operation will be performed by EGSA experts, while the satellite subsystems development, the prototype manufacturing will be performed by students as their graduation projects under supervision from their university supervisors and EGSA experts, then the rest of the developing phases are performed by EGSA specialists.

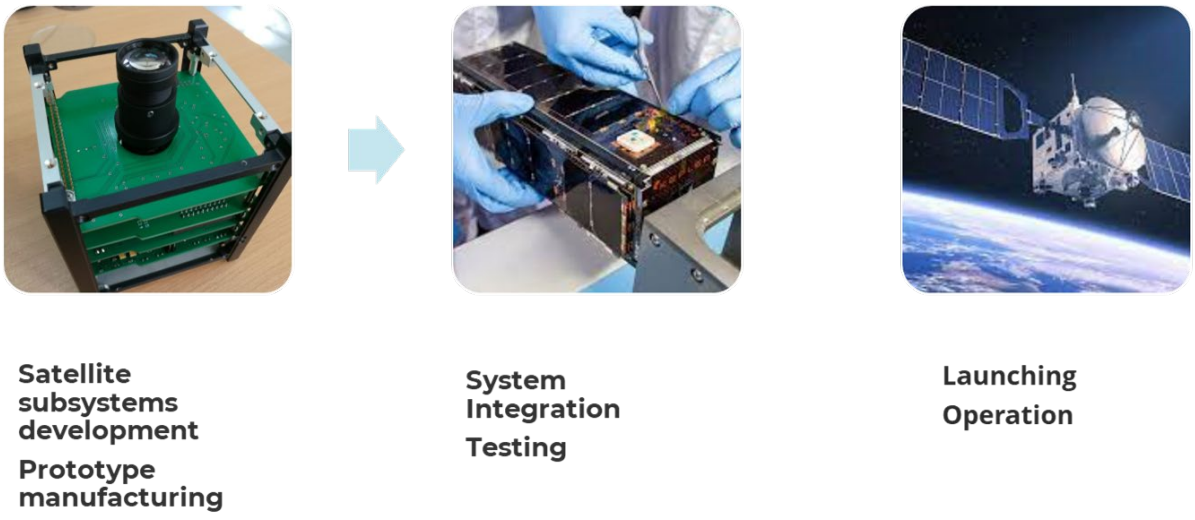

Figure 1 EGSA Egyptian university satellite project

1.3 Problem Statement

The project has main objective and multiple aims need to be reached with success and with certain quality assurance. The objective of our project is to Develop of Cube Satellite communication system Software package & PCB Control Board to

be able to receive command from its ground station and transmit its telemetry back to it, reliably.

1.3.1 EUTS SPECIFICATIONS

Table 1 EUTS specifications

Parameters	**Value**
Category	Cube Sat
Size	1U
Outer Dimensions	$10x10x10cm^3$
Inner Dimensions	$9x9.5x10cm^3$
Mass	< 3Kg
Power Consumption	< 5 W
Transmitted RF Power	> 500 mW
Bit Error Rate	10^{-4}
Antenna Gain	1 to 3 dBi
Solar Panels	Body-mounted
Imaging Payload	Camera
Orbit Altitude	600 — 700 Km
Orbit Type	Sun-synchronous
Inclination Angle	According to orbit altitude
Local Time of Descending Node	10:30 am

1.3.2 COMMUNICATION SUBSYSTEM REQUIREMENTS

Table 2 communication subsystem requirements

Parameter		Value
Transmitter	Transmit power, mW	500
	Modulation	Gaussian Minimum Shift Keying (GMSK)
	Data rate, bit/s	1200
	Data protocol	AX.25
Receiver	Sensitivity, dBm	-100
	Modulation	Gaussian Minimum Shift Keying (GMSK)
	Data rate, bit/s	1200
	Data protocol	AX.25
Operating Temperature range, ^{0}C		-40 to +60
Power Supply, V		3.3
Mass, g		≤ 100
Dimension, mm		96 x 90 x 15
Max. Power Consumption, mW		500 with receiver only 3000 with transmitter ON
RF input Interface		MMCX, 50 Ohm
RF output Interface		MMCX, 50 Ohm
Control interface		RS-485
Control Protocol with OBC		SSP

1.3.3 LINK BUDGET

Is a measurement of all the power gains and losses that a communication signal undergoes in a telecommunication system, from the transmitter to the receiver, using a communication channel which shown in the following figure.

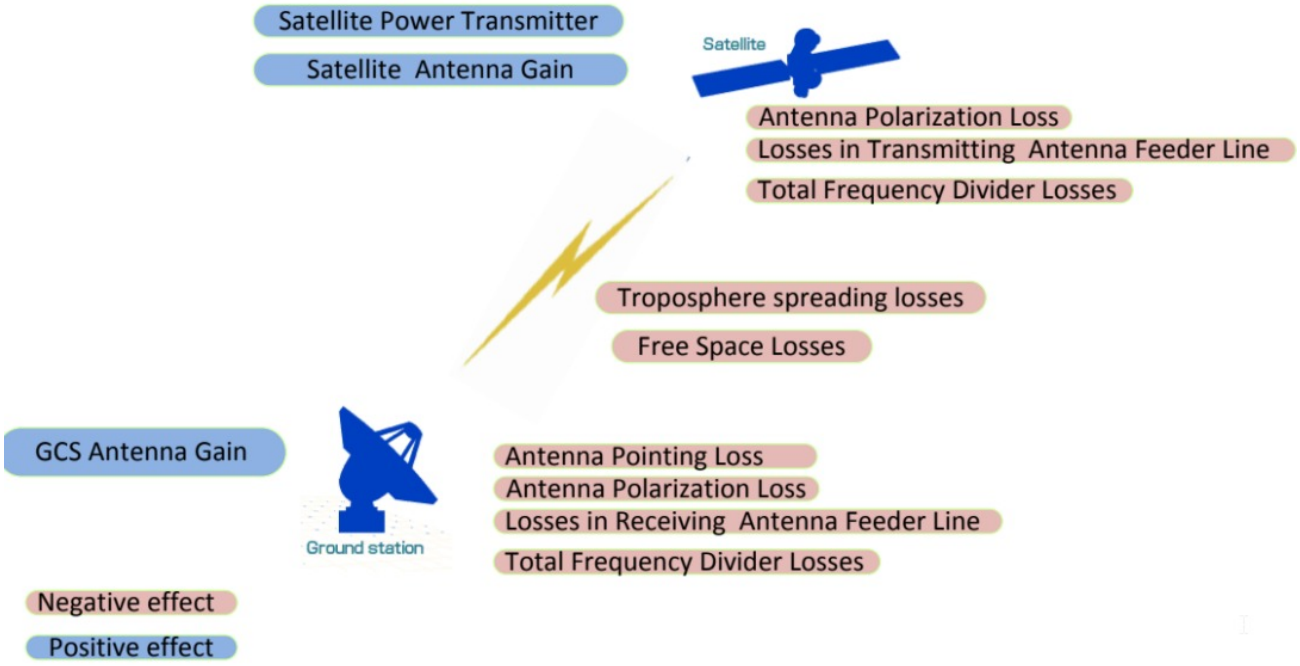

figure 2 link budget diagram

1.4 Project Phases

1.4.1 Project Methodology

1.4.1.1 1st Phase

This phase is the proof of concept , we will use the nRF24L01 transceiver module to make a wireless communication between two Arduino boards using the two nRF24L01 transceiver modules. The nRF24L01 module works with the Arduino through the SPI communication. we will use the nRF24L01 Arduino interfacing, we are going to simply send the data from one Arduino to another Arduino to proof the concept of transmitting and receiving data with the nRF24L01 transceiver module using Arduino (microcontroller board based on the ATmega328P) as in the communication subsystem in the cube satellite what we will develop in the 2nd version.

Success criteria

We will try to send a data from microcontroller board based on the ATmega328P (Arduino) by nRF24L01 to another Arduino board we should get this data at the receiving side displayed on the serial monitor.

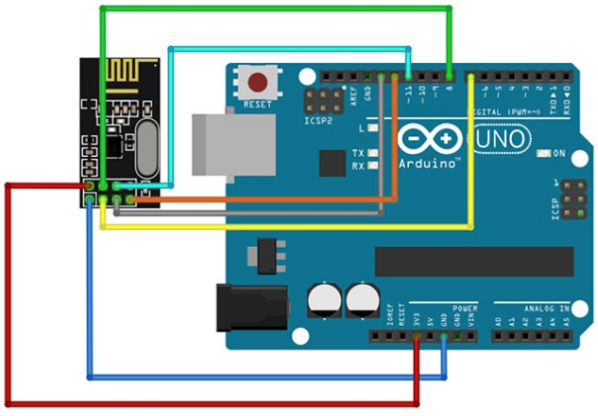

figure 3 version 1

1.4.1.2 2nd Phase

The second phase is prototyping , one of the main requirements of any satellite is the ability to communicate with the ground station reliably by sending and receiving data. In this phase, a design and simulation of the communication system between Ground Control Station (GCS) and the CubeSat for transmitting and receiving the telemetry data is introduced. Two different protocols are used for communication which are: Simple Serial Protocol (SSP) as a control protocol with OBC in the cube satellite and AX.25 for radio transmission. So, we will develop a software code for these two protocols and their layers. The software will do the framing in the transmission side and de-framing at the receiving side by passing throw the control layer to control if the data sent correctly or not. Furthermore, the error of the received data is checked by using the Cyclic Redundancy Check (CRC) technique. The system is implemented by using LabVIEW to verify the communication between CubeSat and GCS. Then by using microcontroller board based on the ATmega328. The microcontroller will accumulate the telemetry data and convert these inputs into a stream of 8-bit binary numbers (bytes). This numerical string is encoded into SSP protocol and then sends it to AX.25 protocol. AX.25 protocol sends the signal to the virtual ground station.

Success criteria

1- For AX.25 protocol we will first send a data and wait to receive the acknowledge if we received it on the graphical user interface (GUI) on LabVIEW we will then try the second condition and try to send incomplete data and wait to receive negative acknowledge if we received it on the GUI then we are done with AX.25 protocol.

2- For SSP we will repeat all the conditions we have done before on the AX.25 protocol.

Figure 4 Version 2

1.4.1.3 3rd Phase

In the final phase, we fulfill the main goal of this project to produce a fully functional communication subsystem. After finishing the 2nd phase, the results will lead to the Final Version of the subsystem. In this version the electrical schematic and layout will be developed for the subsystem then electrical and physical tests will be performed on the Printed Circuit Board (PCB). At this point the software drivers will be finished and tested. We will integrate all the software layers and drivers for SSP and AX.25 protocol with the hardware PCB and test them together to have a fully functional communication subsystem. After the integration of the subsystem, A software program on LabVIEW will be developed to display and simulate the TX and RX data from the satellite to ground station and vice versa.

Success criteria

1- the communication board send and receives frames from the LabView and achieve the expected output from the test cases.

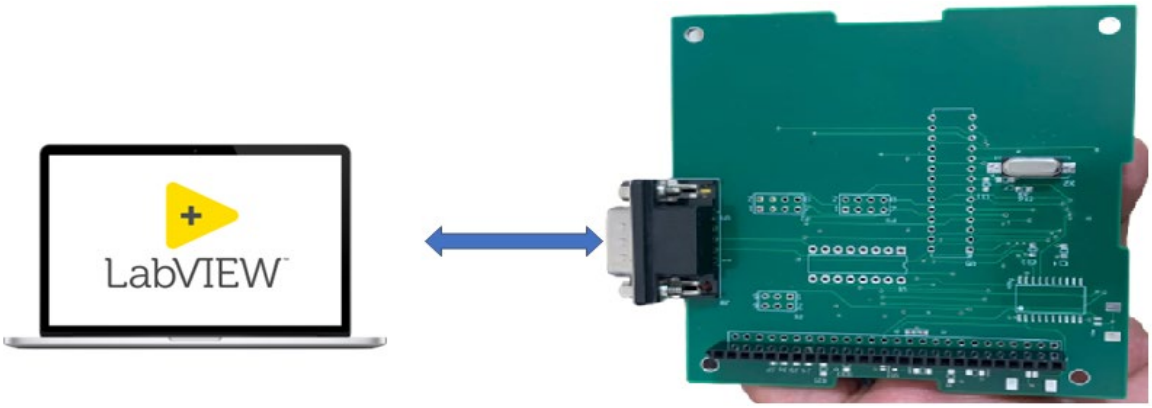

Figure 5 Version 3

2- If the board passed the testing phase, we will install it in EGSA's cube satellite.

Figure 6 EGSA CUBESAT

1.5 Report Organization

The organization of this thesis is as follows:

Chapter 2 introduces the theoretical background and the related works to this project; we will compare the three models of CAL-Poly university and OSLO university finally our proposed work in MIU university and show how our work is distinguished from other works.

Chapter 3 describes the plan and execution of two layered PCB of size 10 cm x 10 cm. Two versions of implementation are introduced to fabricate the communication subsystem board of the CubeSat.

Chapter 4 Describes the implementation of the used communication protocols using the C Programming Language, the program flow for how the communication sub-system uses the communication protocols. communication model in detail. We first describe the architectural model that our communication model is built on, as well as provide a breakdown view on how data flow through this architecture.

Chapter 5 presents the test methodologies of the communication system protocols and the test applications developed for testing the implementation and how these applications are used. these tests will evaluate the performance and functionality of the implementation, ensures the proper program flow and verifies the proper decoding and encoding of AX.25 UI frames and SSP frames.

Last but not the least, Appendix A contains the link budget calculations and Appendix B contains the source code.

2 Chapter 2 Theoretical overview

2.1 Related Works

California Polytechnic 2004:

figure 7 cal poly university

It use AFSK as an modulation which is simple in hardware but on the other hand it is slow data communication and less efficient in both power and bandwidth, it use ax.25 as Protocol between ground and satellite they use this protocol because the upcoming reasons Utilize the AX.25 packet radio protocol in connectionless mode, with an additional, simple handshaking layer, in order to keep communications bursts short and reliable,AX.25 was designed for simplicity and reliability provides a frame check sequence (FCS) which provides a simple way to determine whether the data inside of a packet has been corrupted or not . In addition, a simple command/acknowledge protocol will be implemented within the data field of the AX.25 packet in order to have completed knowledge that a command was received properly by the satellite. Moreover, Cal poly used i2c as a Protocol inside satellite because it's Simplicity and has a low power requirements and power management abilities, Flexibility – The I2C protocol supports multi-master, multi-slave communication, which you can add a lot of functionality to your design, it has better error handling mechanism

Disadvantage of this protocol is the hardware complexity increases when number of master/slave devices are more in the circuit, it has a slow speed because I2C protocol uses pull-up resistors rather than the push-pull so it affects the space too as it requires more space.

As a microcontroller it uses PIC18LF6720 which are profoundly coordinated parts, accessible with on-chip FLASH memory and static RAM just as implicit sequential and equal fringe 16 points of interaction. The PIC18LF6720 was picked

explicitly for its huge measure of FLASH memory (256 Kbyte) for program stockpiling, enormous static RAM (4 Kbyte) for run-time factors, support for the Inter-IC Communication (I2C) transport (the convention used to speak with the super satellite transport) and its very low power prerequisites and power the board capacities.

It uses cc1000 as a modem because it is a true single-chip UHF transceiver designed for very low power and very low voltage wireless applications. The circuit is mainly intended for the ISM (Industrial, Scientific and Medical) and SRD (Short Range Device) frequency bands at 315, 433, 868 and 915 MHz, but can easily be programmed for operation at other frequencies in the 300-1000 MHz range moreover it can be programmed via an easy-to interface serial bus, thus making CC1000 a very flexible and easy to use transceiver, High sensitivity.

OSLO 2010:

1) It uses GFSK as a modulation which is Immune to noise, reduce sideband power, reduce interference with neighboring channels but on the other hand it increases the channel bandwidth, has Bad BER, **increases the modulation memory in the system that causes interference within a symbol.** it uses ax.25 as Protocol between ground and satellite and we mentioned before advantage of ax.25. Oslo use SSP as a Protocol inside satellite because it is simply implemented without any hard hardware and have high efficiency. As a microcontroller it uses ATmega 128A1 which has High capacity (128kb), High speed 32MHZ and support 2 types of memory: EEPROM and SRAM.

Figure 8 OSLO university

2) It uses CC1101 as a modem because it has built in hardware crc error correction and address control, support 2-FSK, GFSK, FSK and MSK modulation so it can support Varity in modulation and high sensitivity 1% for packet error rate.

Misr international 2021

1) It uses GMSK as a modulation which has high spectral power (due to constant envelope) efficiency, reduce sideband power, reducing interference with adjacent frequency channels on the other hand it has high power level, Cause isi (inter symbol interference) and require more complex channel equalization like adaptive equalizer. it uses ax.25 as Protocol between ground and satellite and we mentioned before advantage of ax.25. MIU use ssp as a Protocol inside satellite because it is simply implemented without any hard hardware and have high efficiency. As a microcontroller it uses ATmega 328p-pu because it has High capacity (32kb), High speed 20MHZ and support 2 types of memory (EEPROM and flash memory). It uses NRF24l01 as a modem because it has high data rate and support CRC in addition to its inexpensive and reliable, Auto ACK & retransmit (may remove the need for dedicated SPI hardware in a mouse/keyboard or comparable systems, and hence reduce cost and average current consumption).

Figure 9 MIU University

3 Chapter 3 Hardware

3.1 Introduction

We needed to design a custom communication (subsystem/board) to program the code that were made by the software team and then put in the 1u CubeSat so our point as a hardware team to make a PCB to put it in the EUST CubeSat with different subsystems to be Egyptian college satellite developed by Egyptian universities and we will accomplish that under monitoring of Egyptian space agency.

Presently we will discuss the plan and execution of four layers PCB .We have two versions ; the first variant contains Atmega328P-PU microcontroller while the subsequent adaptation contains both Atmega2560 and Atmega328P-PU. we utilized Atmega2560 to solve the issue of memory shortage, the two versions contain headers for ICSP(in circuit serial programming) and we added header for internal interfaces with satellite subsystem (SPI,UARTs and I2C).we also added an external interface (UART) for debugging.

3.2 Hardware design

We will show version1 & version2 consequently

3.2.1 Version 1

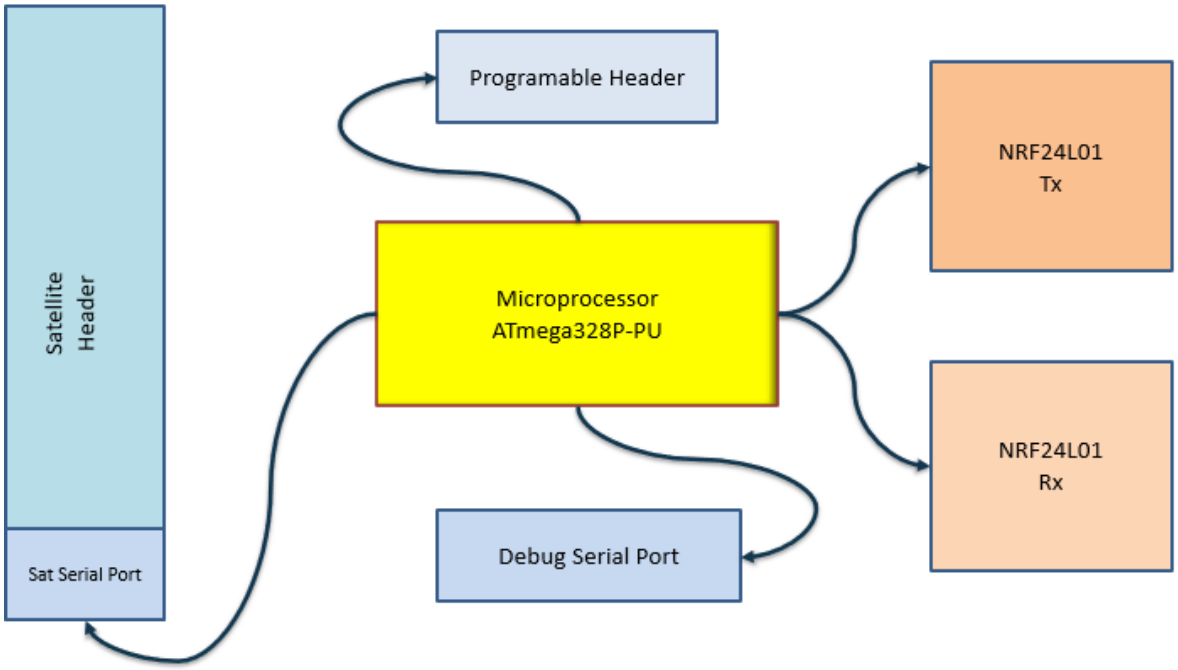

Figure 10 hardware design version 1

Figure 10 shows the different hardware components in the first version of communication subsystem. The satellite communication hardware consists of four parts. The first part is the microcontroller which play an important role in the design , it supports UART , SPI and I2C interfaces. The second part is NRF module which act as transmitter (transmit data to ground station) and receiver (get data from ground station) . The third part is programable header which is used to program the micro controller by SPI interface. Finally, the fourth part is the serial port . we are using at version 1 only 1 serial port which is shared between serial debugging and satellite interfacing. The serial debugging is used to debug the code , on the other hand the satellite serial port is used to connect with all other subsystems of the CubeSat.

3.2.2 Version 2

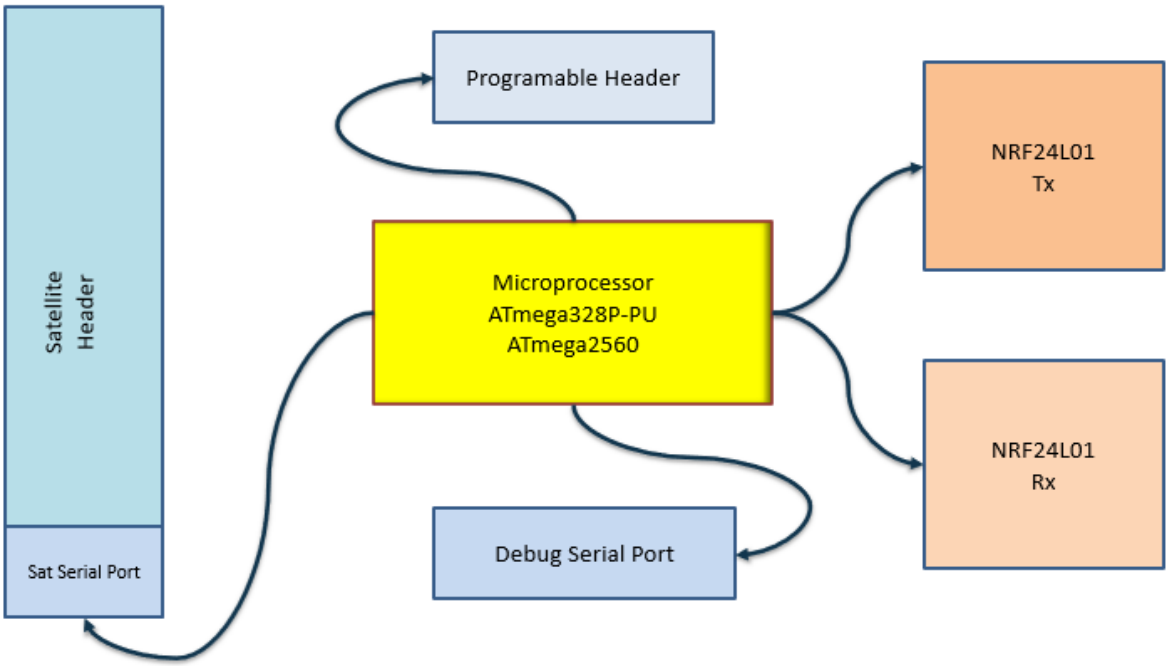

Figure 11 hardware design version 2

due to the problem of memory shortage of ATmega328p-pu so we have added another microcontroller in version 2 (Atmega2560) to overcome this problem , it's the same design of version 1 but the only difference is the new Microcontroller which support 4 UARTs interfaces where the memory is 256kb therefore we were able to use two UARTs instead of one UART in the previous version, the first UART for the Debug serial port and the other for the satellite serial port.

3.3 Main Components

Table 3 list of main components

(Version 1)	Version 2
ATmega328P-PU	ATmega328P-PU and ATmega2560
1 x MAX485	2 x MAX485
Stack through 52 (header)	
MAX3232EEAE	
2 x NRF24L01	

3.4 Data sheet summary

3.4.1 ATmega328P-PU

The ATmega328 is a microcontroller that is assembled on the single-chip and manufactured by the Atmel (who was producers and creator of semiconductors materials) in the megaAVR group of microcontrollers. The processor core of this module is eight-bit RISC (reduced instruction set computer) which has Harvard architecture with some modification. It can support data up to eight bits and has a flash memory of thirty-two-kilo bytes.

Configuration Summary

Table 4 Configuration Summary of ATmega328P-PU

Features	**ATmega328/P**
Pin Count	28/32
Flash (Bytes)	32K
SRAM (Bytes)	2K
EEPROM (Bytes)	1K
Interrupt Vector Size (instruction word/vector)	1/1/2
General Purpose I/O Lines	23
SPI	2
TWI (I^2C)	1

USART	1
ADC	10-bit 15kSPS
ADC Channels	8
8-bit Timer/Counters	2
16-bit Timer/Counters	1

Block diagram

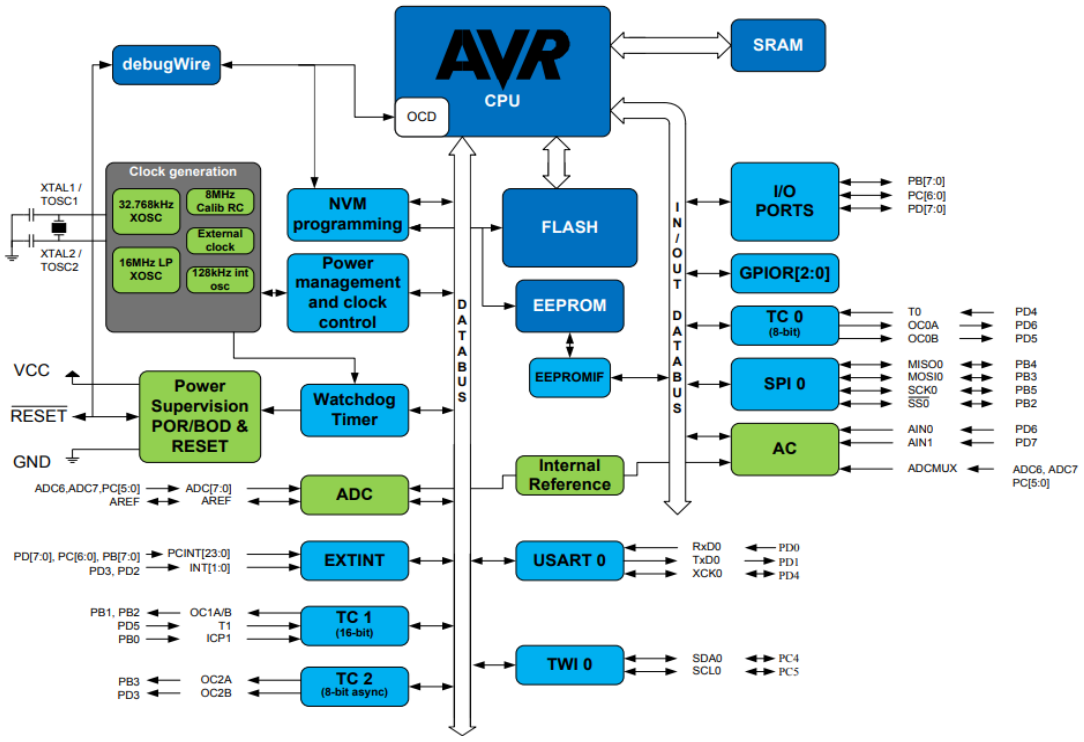

Figure 12 Block Diagram of ATmega328P-PU

Briefly summary about the block diagram

- CPU:8-bit AVR
- Number of pins:28
- Operating voltage:+1.8 V to + 5.5 V
- Number of programmable i/o lines:23
- Communication interface
    - Master/Slave SPI Serial Interface (17,18,19 PINS) [Can be used for programming this controller]

    - Programmable Serial USART (2,3 PINS) [Can be used for programming this controller]
    - Two-wire Serial Interface (27,28 PINS) [Can be used to connect peripheral devices like Servos, sensors and memory devices]
- ADC Module:6 channels, 10-bit resolution ADC
- Time Module
    - Two 8-bit counters with Separate Prescaler and compare mode, one 16-bit counter with Separate Prescaler, compare mode and capture mode Analog comparators
- Analog Comparators :1(12,13 PINS)
- DAC Module: Nil
- PWM channels :6
- External Oscillator
    - 0-4MHz @ 1.8V to 5.5V
    - 0-10MHz @ 2.7V to 5.5V
    - 0-20MHz @ 4.5V to 5.5V
- Internal Oscillator:8MHz Calibrated Internal Oscillator
- Program memory (flash memory)
    - 32kBytes
    - 10000 write/erase cycles
- CPU Speed
    - 1MIPS for 1MHz
- RAM
    - 2Kbyttes internal SRAM
- EEPROM (NVM programing):1kbytes EEPROM
- Watchdog Timer
    - Programmable Watchdog Timer with Separate On-chip Oscillator
- Program lock: yes
- Power saves modes

    - Six Modes [Idle, ADC Noise Reduction, Power-save, Power-down, Standby and Extended Standby]
- Debug Wire
    - The debug WIRE on-chip debug system uses a wire with bi-directional interface to control the program flow and execute AVR instructions in the CPU and to program the different nonvolatile memories.
- SRAM
    - The device is a complex microcontroller with more peripheral units than can be supported within the 64 locations reserved in the Opcode for the IN and OUT instructions. For the extended I/O space from 0x60 - 0xFF in SRAM
    - 2kBytes
- Clock Generation
    - The clock generation logic generates the base clock for the transmitter and receiver. The USART supports four modes of clock operation: Normal asynchronous, Double Speed asynchronous, Master synchronous, and Slave synchronous mode.
- Power management
    - Sleep modes enable the application to shut down unused modules in the MCU, thereby saving power. The device provides various sleep modes allowing the user to tailor the power consumption to the application requirements
- GPIOR
    - General purposed input output register (23 but PC6 is reset so it’s 22 pins)
- Operating temperature
    - -40°C to +105°C (+105 being absolute maximum, -40 being absolute minimum)

ATmega2560

The ATmega2560 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega2560 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.

Table 5 configuration summary OF ATmega2560

Features	**ATmega328/P**
Hardware serial ports	4 UARTs
Operating voltage	5V
Input voltage (recommended)	7-12V
Input voltage (limits)	6-20V
Digital I/O Pins	54 (of which 15 provide PWM output)
Analog Input Pins	16
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	256 KB of which 8 KB used by bootloader
SRAM	8 KB
EEPROM	4 KB
Clock Speed	16 MHz

Block diagram

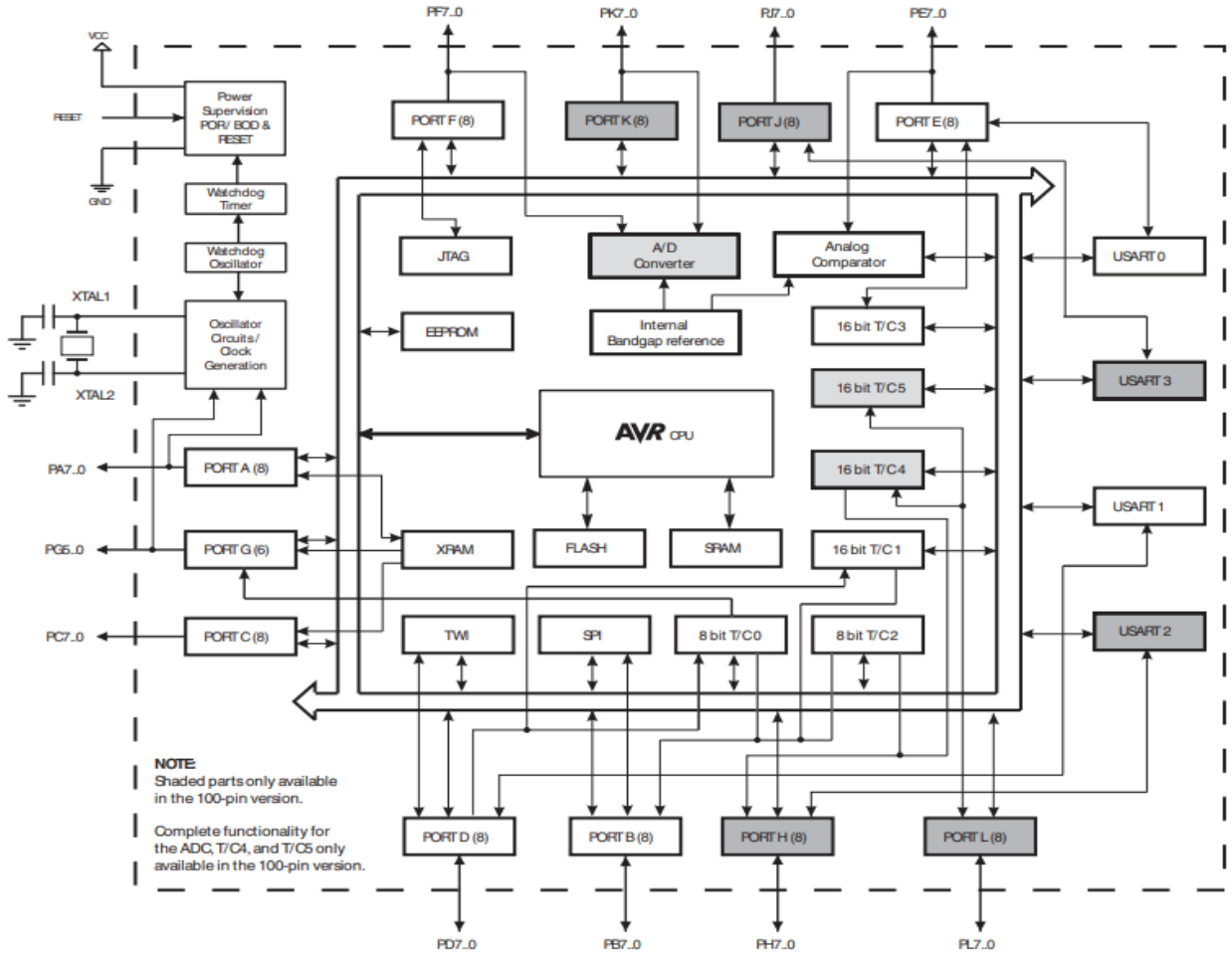

Figure 13 Block Diagram of Atmega2560

Briefly summary about the block diagram

- CPU:8-bit AVR
- Number of pins:100
- Operating voltage:+1.8 V to + 5.5 V
- Number of programmable i/o lines:54
- Communication interface
    - Master/Slave SPI Serial Interface (50,51,52,53 PINS) [Can be used for programming this controller]
    - 4 Programmable Serial USART (0,1 PINS) (19,18 PINS) (17,16 PINS) (15,14 PINS)

    - Two-wire Serial Interface (20,21 PINS) [Can be used to connect peripheral devices like Servos, sensors and memory devices]
- Time Module
    - Two 8-bit counters with Separate Prescaler and compare mode, one 16-bit counter with Separate Prescaler, compare mode and capture mode Analog comparators
- Analog Comparators :1(5,4 PINS)
- DAC Module: Nil
- PWM channels :15
- External Oscillator
    - 0-2MHz @ 1.8V to 5.5V
    - 0-8MHz @ 2.7V to 5.5V
    - 0-16MHz @ 4.5V to 5.5V
- Internal Oscillator
    - 7.3 to 8.1MHz Calibrated Internal Oscillator
- Program memory (flash memory):256kBytes
- CPU Speed: Up to 16 MIPS Throughput at 16MHz
- RAM: 8Kbytes internal SRAM
- EEPROM (NVM programing)
    - 4kbytes EEPROM
    - Write/Erase Cycles:10,000 Flash/100,000 EEPROM
- Watchdog Timer
    - Programmable Watchdog Timer with Separate On-chip Oscillator
    - The WDT is a timer counting cycles of a separate on-chip 128kHz oscillator. The WDT gives an interrupt or a system reset when the counter reaches a given time-out value. In normal operation mode, it is required that the system uses the WDR - Watchdog Timer Reset - instruction to restart the counter before the time-out value is reached. If the system doesn't restart the counter, an interrupt or system reset will be issued.

- Program lock
    - Yes
- Power saves modes
    - Six Modes [Idle, ADC Noise Reduction, Power-save, Power-down, Standby and Extended Standby]
- Debug Wire
    - The debug WIRE on-chip debug system uses a wire with bi-directional interface to control the program flow and execute AVR instructions in the CPU and to program the different nonvolatile memories.
- SRAM
    - The ATmega640/1280/1281/2560/2561 is a complex microcontroller with more peripheral units than can be supported within the 64-location reserved in the Opcode for the IN and OUT instructions. For the Extended I/O space from $060 - $1FF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used
    - 8kBytes
- Clock Generation
    - The Clock Generation logic generates the base clock for the Transmitter and Receiver. The USARTn supports four modes of clock operation: Normal asynchronous, Double Speed asynchronous, Master synchronous and Slave synchronous mode.
- Power management
    - Sleep modes enable the application to shut down unused modules in the MCU, thereby saving power. The AVR provides various sleep modes allowing the user to tailor the power consumption to the application's requirements.
- GPIOR:32 × 8 General Purpose Working Registers
- Operating temperature:-40◦C to +85◦C

3.4.2 NRF24L01

Having two or more Arduino boards be able to communicate with each other wirelessly over a distance opens lots of possibilities like remotely monitoring sensor data, controlling robots, home automation and the list goes on. And when it comes down to having inexpensive yet reliable 2-way RF solutions, no one does a better job than nRF24L01+ transceiver module that's why we used it to be as a prototype for our connection between satellite and ground station.

Its intendeds to work in 2.4 GHz overall ISM recurrence band which reserved for unlicensed low-powered devices and uses GFSK modulation

The nRF24L01+ handset module of a 4-pin Serial Peripheral Interface (SPI) with a most extreme information pace of 10Mbps.

Specifications of NRF24L01

Table 6 NRF24L01 SPECS

nRF24L01
Frequency Range: 2.4 GHz ISM Band
Maximum Air Data Rate: 2 Mb/s
Modulation Format: GFSK
Max. Output Power: 0 dBm
Operating Supply Voltage: 1.9 V to 3.6 V
Max. Operating Current: 13.5mA
Min. Current (Standby Mode): 26µA
Logic Inputs: 5V Tolerant
Sensitivity: -94dBm

Block diagram

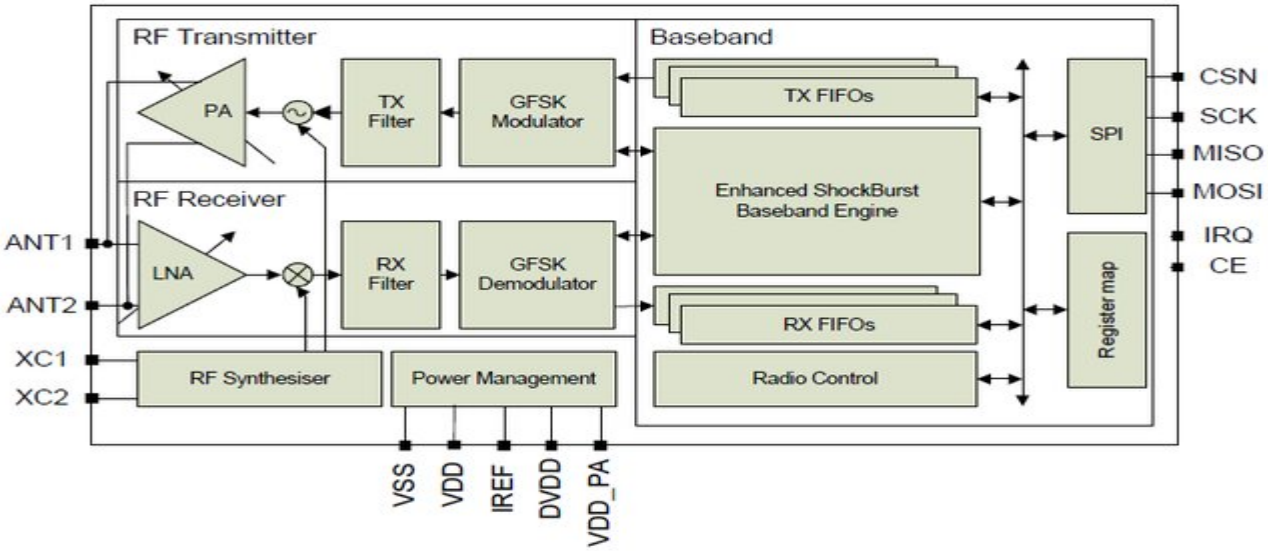

figure 14 block diagram of nrf24l01

The nRF24L01 crystal oscillator is amplitude regulated. To achieve low current consumption and also good signal-to-noise ratio when using an external clock,

The PA stands for Power Amplifier. It merely boosts the power of the signal being transmitted from the nRF24L01+ chip. Whereas, LNA stands for Low-Noise Amplifier. The function of the LNA is to take the extremely weak and uncertain signal from the antenna (usually on the order of microvolts or under -100 dBm) and amplify it to a more useful level (usually about 0.5 to 1V)

The low-noise amplifier (LNA) of the receive path and the power amplifier (PA) of the transmit path connect to the antenna via a duplexer, which separates the two signals and prevents the relatively powerful PA output from overloading the sensitive LNA input

Note we are using GMSK at out project, but this module uses GFSK modulation, so we have to adjust the roll off factor (BT) to be 0.5, so that GFSK will be GMSK.

Table 7 PIN DESCRIPTIONS of NRF24L01

pin	name	Pin function	Description
1	CE	Digital Input	Chip Enable Activates RX or TX mode
2	CSN	Digital Input	SPI Chip Select
3	SCK	Digital Input	SPI Clock
4	MOSI	Digital Input	SPI Slave Data Input
5	MISO	Digital Output	SPI Slave Data Output, with tri-state option
6	IRQ	Digital Output	Maskable interrupt pin
7	VDD	Power	Power Supply (+3V DC)
8	VSS	Power	Ground (0V)
9	XC2	Analog Output	Crystal Pin 2
10	XC1	Analog Input	Crystal Pin 1
11	VDD_PA	Power Output	Power Supply (+1.8V) to Power Amplifier
12	ANT1	RF	Antenna interface 1
13	ANT2	RF	Antenna interface 2
14	VSS	Power	Ground (0V)
15	VDD	Power	Power Supply (+3V DC)
16	IREF	Analog Input	Reference current
17	VSS	Power	Ground (0V)
18	VDD	Power	Power Supply (+3V DC)
19	DVDD	Power OUTPUT	Positive Digital Supply output for de-coupling purposes
20	VSS	Power	Ground (0V)

3.4.3 MAX485

RS-485 is a standard interface of the physical layer of communication, a signal transmission method, RS-485 has been created in order to expand the physical capabilities of RS-232 interface. We choose max 485 to be able to transmit data over very long distances even if there is an extremely harsh environment. The main idea here is to transport one signal over two wires. While one wire transmits the original signal, the other one transports its inverse copy. Such transmission method provides high resistance to common-mode interference. The twisted-pair cable that serves as a transmission line can be shielded or unshielded.

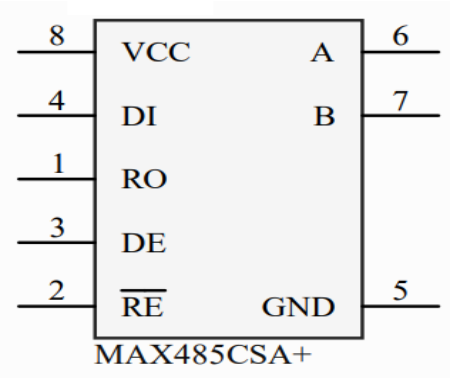

Figure 15 MAX485 IC

Note: Rs is referred to recommended

Features of max485

- Balanced interface:
  - It increases noise immunity and decrease emissions
- Multipoint, bidirectional communication on a single pair of wires
  - so, its lower cabling cost
- Large differential signal, large communication range
  - So, it allows for communication over long distances and with large ground potential difference.
- can achieve signaling rates up to 50Mbps
  - so suitable for wide array of application.
- Low electromagnetic interference (EMI)
- Isolated DC to DC converters which eliminates the need for external isolated power supply

Table 8 Pin description of max485

Pin	Description
VCC	This is the power supply pin. It is connected with 5V that powers up the module.
A	This is the non-inverting receiver input and driver output. It is connected with A on the other module.
B	This is the inverting receiver input and driver output. It is connected with B on the other module.
GND	This is the GND pin. It is connected with the common ground.
RO	This is the receiver output pin. It is connected with the RX pin of the microcontroller.
RE	This is the receiver output enable pin. To enable, it is set at a LOW state.
DE	This is the driver output enable pin. To enable, it is set at a HIGH state.
DI	This is the driver input. It is connected with the TX pin of the microcontroller.

3.4.4 MAX3232EEAE

The 89CS2 of 8051 family of microcontrollers has a built-in serial port that makes it very easy to communicate with the PC's serial port but the 89C52 outputs are 0 and 5 volts and we need +10 and -10 volts to meet the RS232 serial port standard. The easiest way to get these values is to use the MAX232. The MAX232 acts as a buffer driver for the processor. It accepts the standard digital logic values of 0 and 5 volts and converts them to the RS232 standard of +10 and -10 volts. The MAX232 requires 5 external luF capacitors. These are used by the internal charge pump to create +10 volts and -10 volts. It includes 2 receivers and 2 transmitters so two serial ports can be used with a single chip.

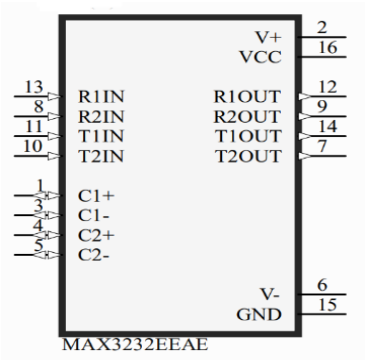

Figure 16
MAX3232EEAE IC

Features of max3232eeae

- Simple protocol design.
- Hardware overhead is lesser than parallel communication.
- Recommended standard for short distance applications.
- Low-cost protocol for development.

Table 9 Pin description of MAX3232EEAE

Pin	**# Of the pin**	**Description**
C1+	1	Positive lead of C1 capacitor
V+	2	Positive charge pump output for storage capacitor only
C1-	3	Negative lead of C1 capacitor
C2+	4	Positive lead of C2 capacitor
C2-	5	Negative lead of C2 capacitor
V-	6	Negative charge pump output for storage capacitor only
DOUT2	7	RS232 line data output (to remote RS232 system)
DOUT1	14	RS232 line data output (to remote RS232 system)
RIN2	8	RS232 line data input (from remote RS232 system)
RIN1	13	RS232 line data input (from remote RS232 system)
ROUT2	9	Logic data output (to UART)
ROUT1	12	Logic data output (to UART)
DIN2	10	Logic data input (from UART)
DIN1	11	Logic data input (from UART)
GND	15	Ground
VCC	16	Supply voltage , Connect to extemal 3 V to 5.5 V power supply

3.4.5 D CONNECTOR 9

D Sub Connectors D-sub or D-subminiature connectors are common types of electrical components typically used in computing to help form connections on a circuit board. Taking their name from the characteristically designed D shaped metal shield, d-sub connectors have various different connection types from Crimp D-subs to Spring Terminal D-sub's.

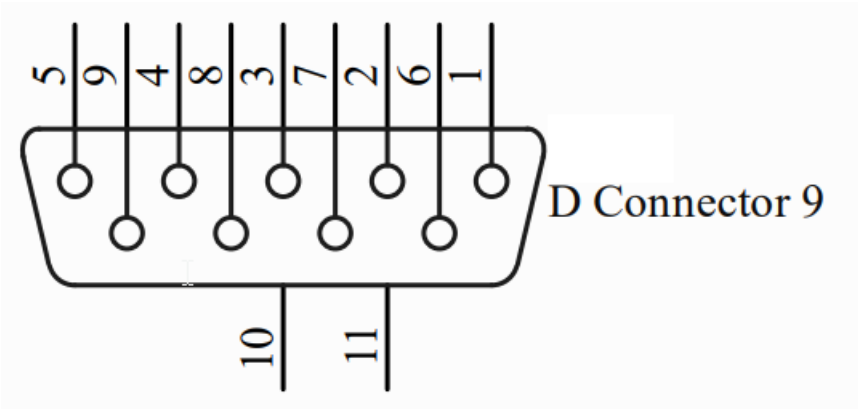

Figure 17 D CONNECTOR JACK

Table 10 PIN description of d-connector

Pin	**Description**
1	GND
3	T1OUT OF MAX3232EEAE
4	R1IN of MAX3232EEAE
2,5,6,7,8,9,10,11	NC

3.5 Schematic

3.5.1 Schematic Version 1

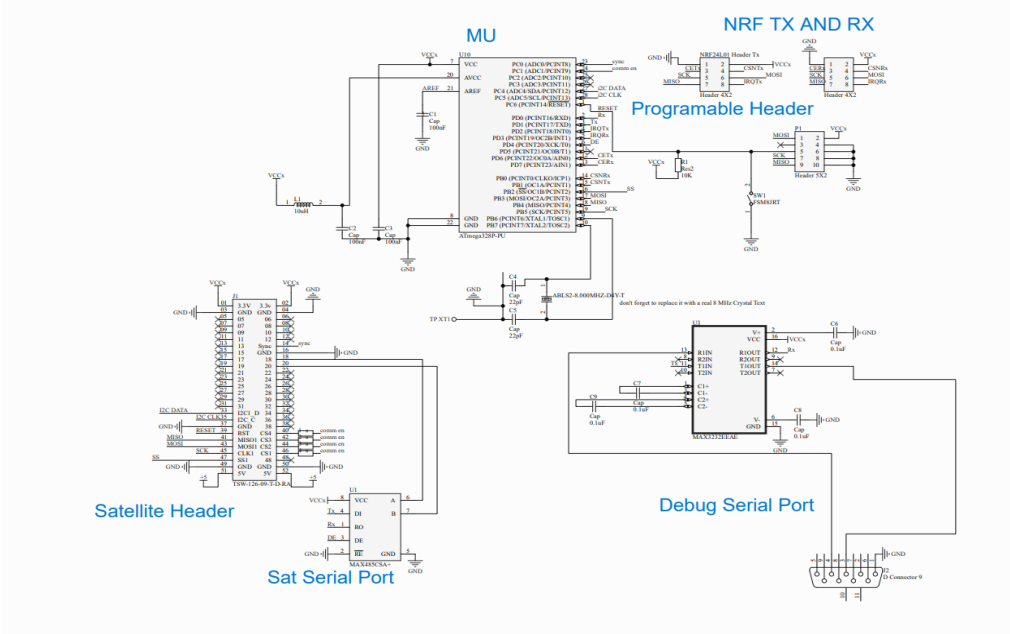

Figure 18 Schematic version 1

- **ATmega328P-PU Microcontroller & programmable header Connection schematic**

The interfaces of the microprocessor are Serial Peripheral Interface (SPI) which used to transmit and receive , IT'S CLEAR THAT WE sharing SPI between the Tx and Rx .Serial1 for sat interface port and serial 2 for debugging , i2c for sat interface , sync is used to adjust clock, communication enable used for restarting the communication system.

we have SPI (which is used in image transmissions) , I2C (which is used as temp sensor) and UART. We have used PULL UP RESISTOR (internal resistance) use instead of connecting an external resistance to connect VCC (to prevent floating node). The coil function to smooth the wave form or make adjust analog power. we have used external crystal 8 MHz to give an accurate clock for chip's operations, SPI acts as dual function for programming (miso) and the other is in-circuit programming .which is use in the programable header.

Satellite Header Connection Schematic

The header contains the following interfaces, SPI interface which are MOSI, MISO,SCK and SS , it used to receive images from the camera , if camera exist. I2C interface which is SCL and SDA (serial clock line , serial data line) it used as temperature sensor.

UART interface which is used to send data to SSP digital input output pins sync to adjust clock communication enable to restart the communication system if needed.

- **NRF**

  We have two NRF modules , one for transmitter and the other for receiver.

  Both NRF modules are sharing SPI of the microcontroller, the modulation technique from the EGSA requirements is GMSK but the NRF module works with GFSK so we have to adjust roll-off factor to be 0.5 , so we can change modulation from GFSK to GMSK.

  Both transmitter and receiver work with SPI interface .

- **Debug and SAT serial port**

For the debugging serial port , we are using max323 convert Transistor-Transistor Logic to RS232, because TTL from 0 to 5 volt , and we need -10 to 10 volts to meet rs232 serial port standard to be able to connect pc .max2323 act as buffer driver for the processor (capacitor act as internal charge pump) and the d connector will be connected with the pc.

For satellite serial port , we are using MAX485 as its specifications of EGSA and for further details , scroll up for MAX485 part.

3.5.2 Schematic Version 2

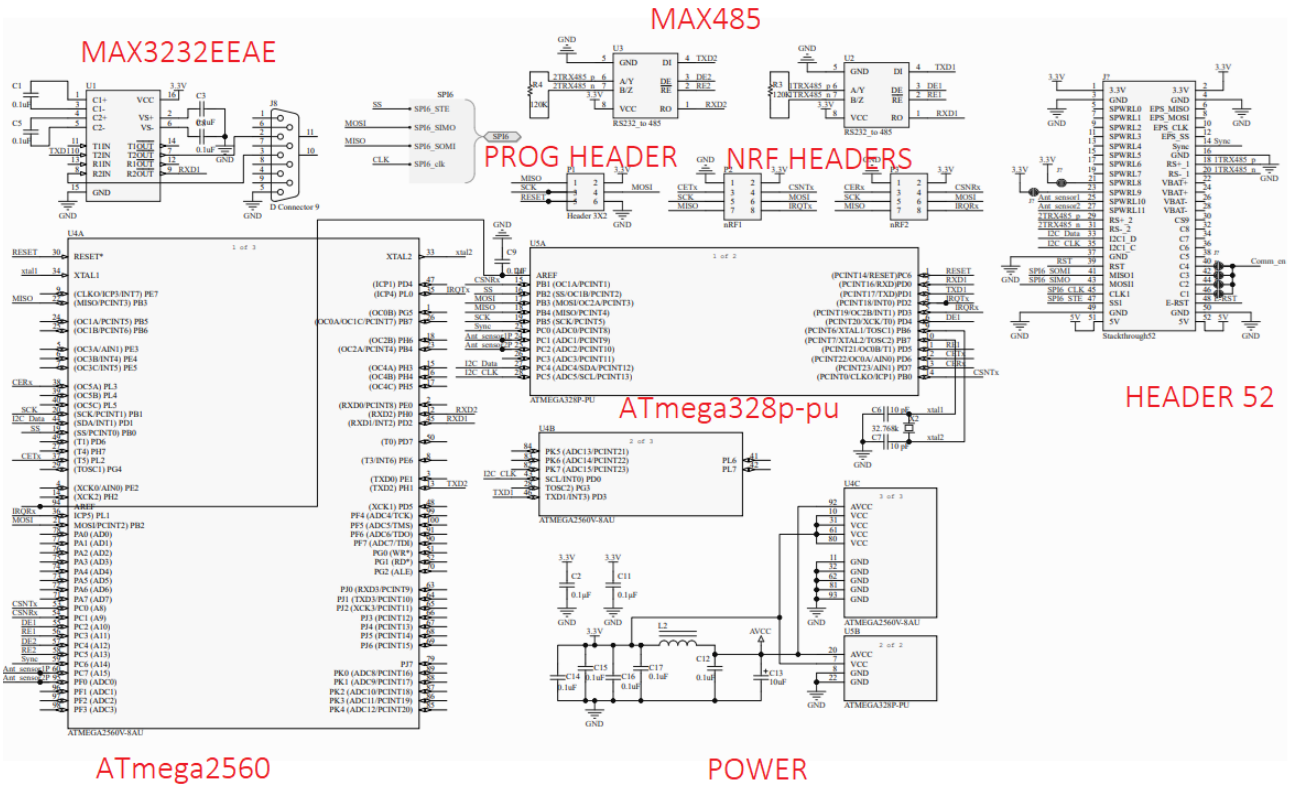

Figure 19 Schematic version 2

The only difference between schematic version 1 and version 2 , that we have used ATmega2560 in addition to ATmega328p-pu in order to solve software memory issue.

In the schematic we have split ATmega2560,328p-pu in order to make it easy while making connection in schematic page only but in the layout each of them is represented as one completed footprint.

Atmega2560 consist of 4 UARTs, I2C and SPI interfaces while ATmega328p-pu consist of UART , I2C and SPI interfaces.

We have used two MAX485 because ATmega2560 support multiple UARTs.

The power circuit contains all VCC , AVCC and GNDS of both ATmega2560 and ATmega328p-pu together , the coil L2 function is to reduce noise.

The programable header is 2x3 while in version 1 it was 2x5 , both do the same function(burning code) by using SPI interface .

3.6 Layout

3.6.1 Placement

We placed the finger print of each component away from each other to keep the noisiest signals away from the highly-sensitive ones. Also, by keeping grouping components according to their function, you'll have better control of their return path.

Place Board-To-Wire Connectors Near the Edge to prevent unwanted contact with other components on the x`.

3.6.2 Routing

Routing process is an electrical connection between components. There are no two circuit boards alike. the routing is what makes your design unique.

We have used manual routing and automatic routing from Altium designer.

The large traces are for power connection.

3.6.3 Constrains

- The cube sat size is 1u
    - the board size should be smaller or equal to 10 cm x 10 cm
- NRF module
    - There should be distance between two NRF modules equal to 3 cm , so each one can be set freely on the PCB without collision
    - There should be distance above NRF module equal or more than 4 cm.

3.6.4 Layout version 1

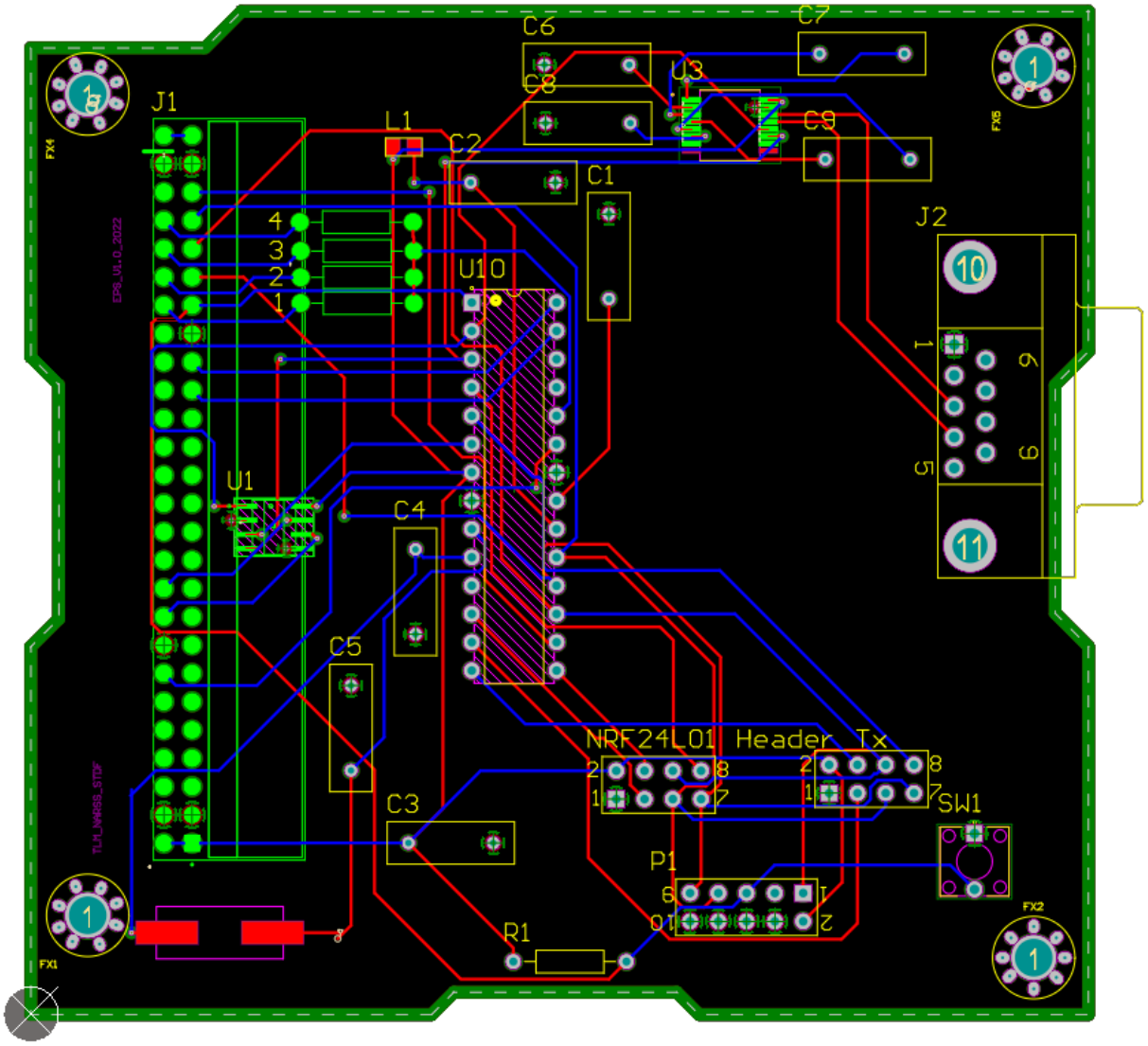

Figure 20 layout version 1

3.6.4.1 Board specifications

Board Size is 95.0000mm x 90.0000mm

Components on board are 29

Number of Layers is 4

3.6.5 Layout Version 2

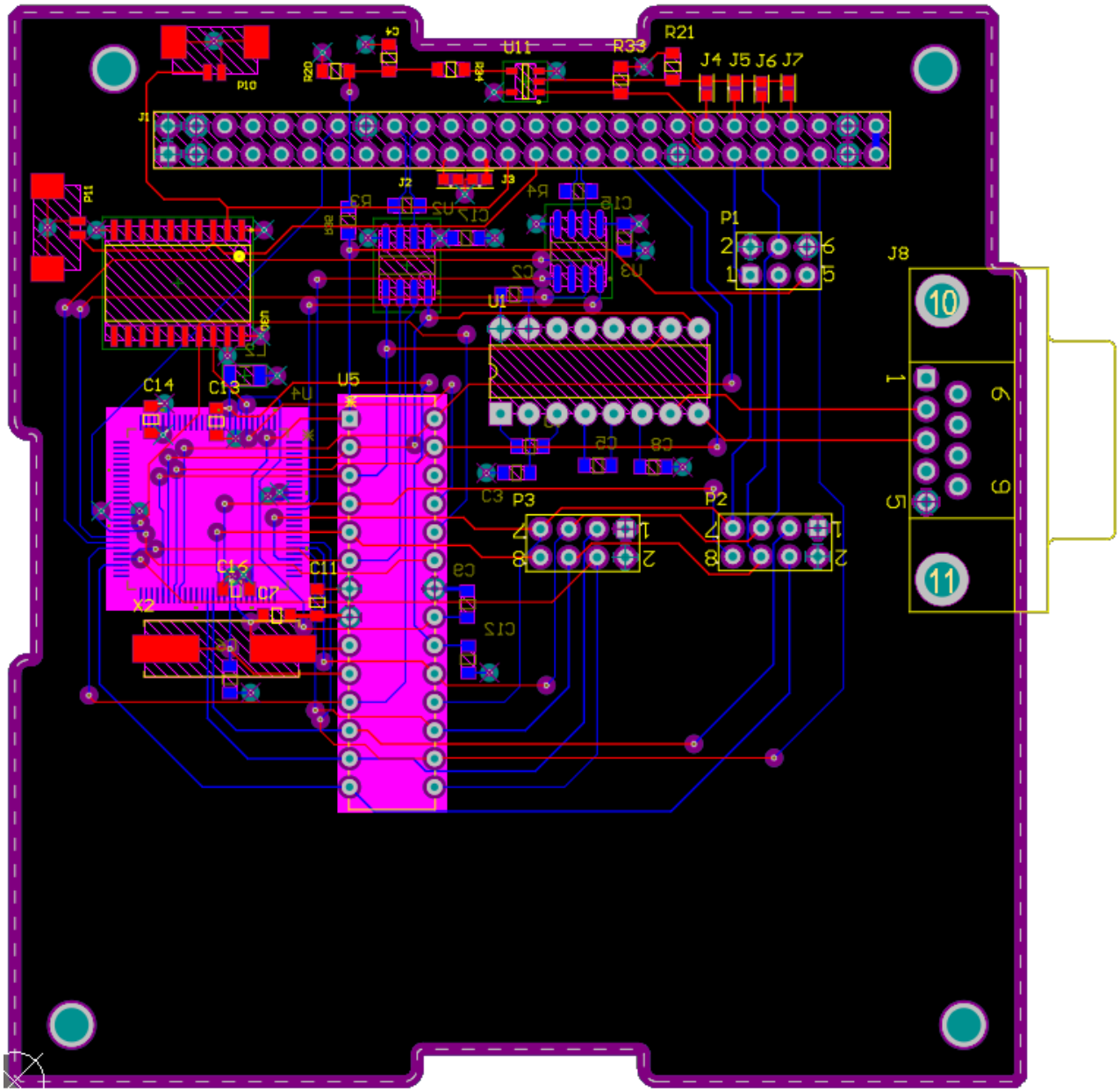

Figure 21 layout version 2

3.6.5.1 Board specifications

Board size is 90.1700mm x 95.8900mm

Components on board are 45

Number of Layers is 4

Gerber file

Gerber File is a software file that is provided to the PCB Manufacturing Company to fabricate a PCB to the required specification. A PCB Gerber file is a two-dimensional pictorial representation of each layer of a PCB board where tracks, pads, and vias are represented by different lines and shapes. It is the universal way of telling a machine how to create a PCB board based on the information provided.

3.6.6 Version 1

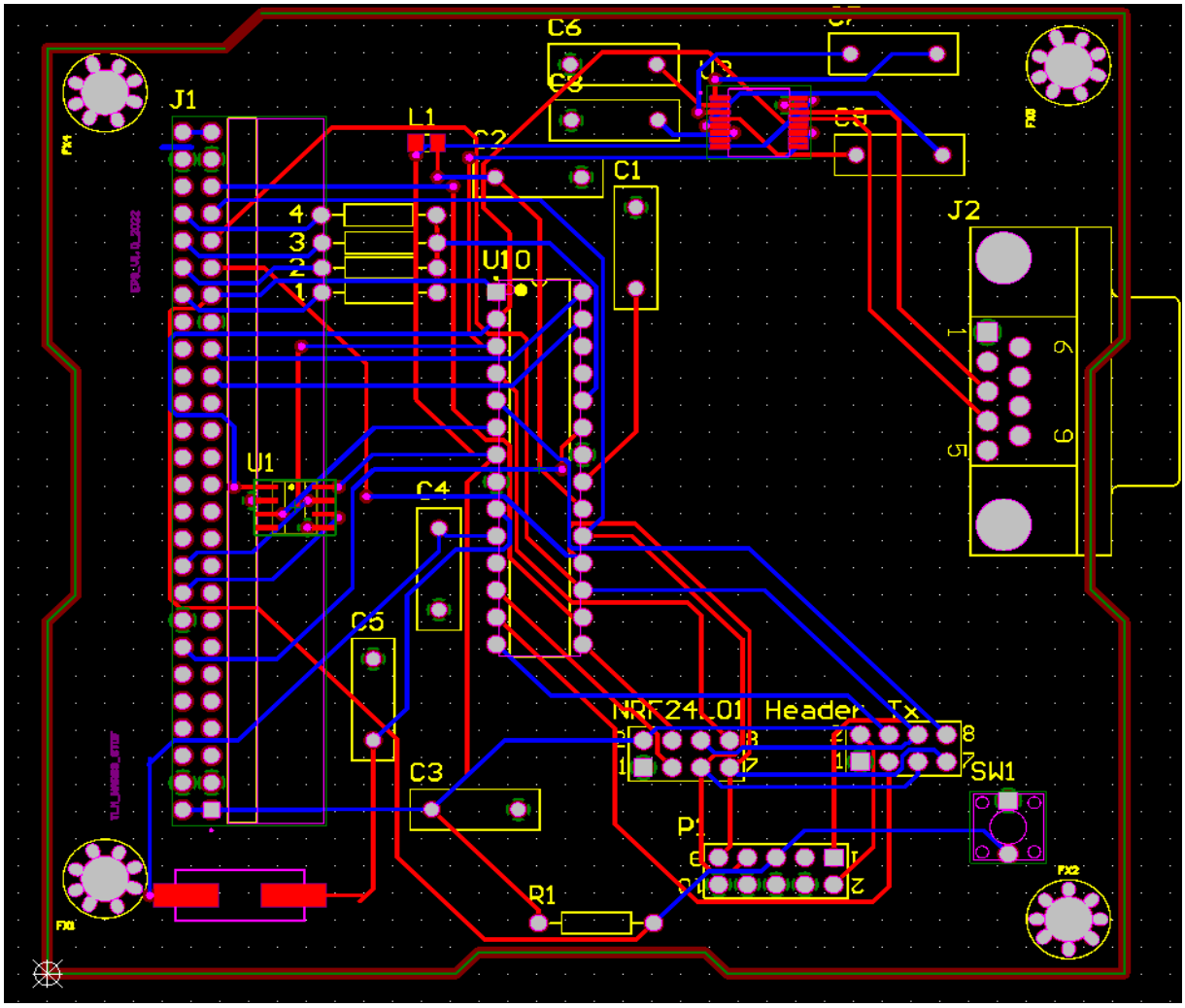

Figure 22 GERBER of version 1

3.6.7 Version 2

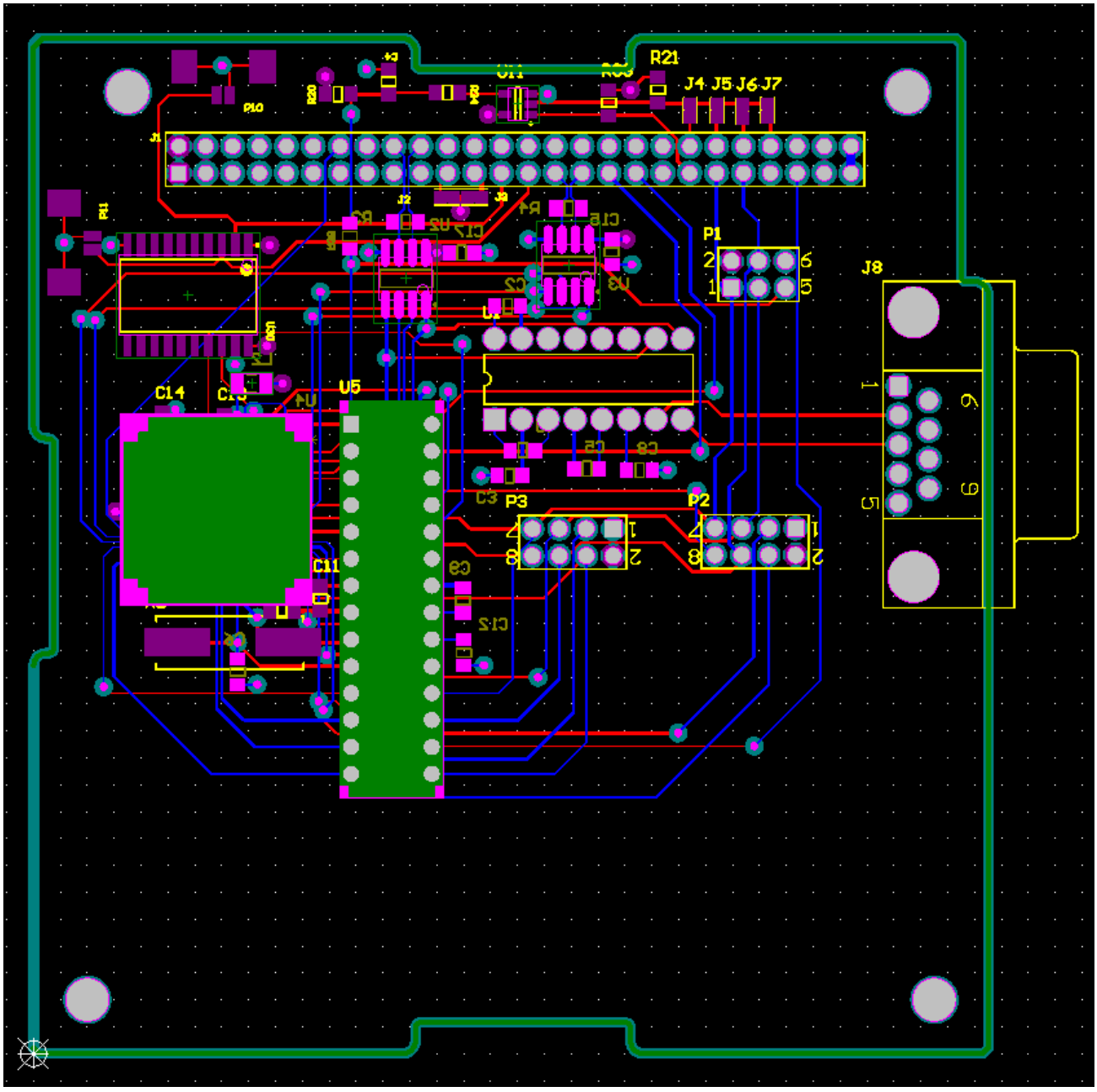

Figure 23 GERBER of version 2

3.7 Drill file

NC Drill File is also known as numeric control drill file. This indicates a file that regulates all the information relating to via or hole drilling requirements. These include hole location and size, and tooling size.

It is very necessary for engineers of PCB design to create the NC drill files. This is because you can avoid so much trouble by converting the PCB files into Gerber files or NC drill files.

However, the truth is that some engineers end up sending the PCB files to the PCB manufacturer directly. They fail to convert into Gerber files or NC drill files.

3.7.1 Version 1

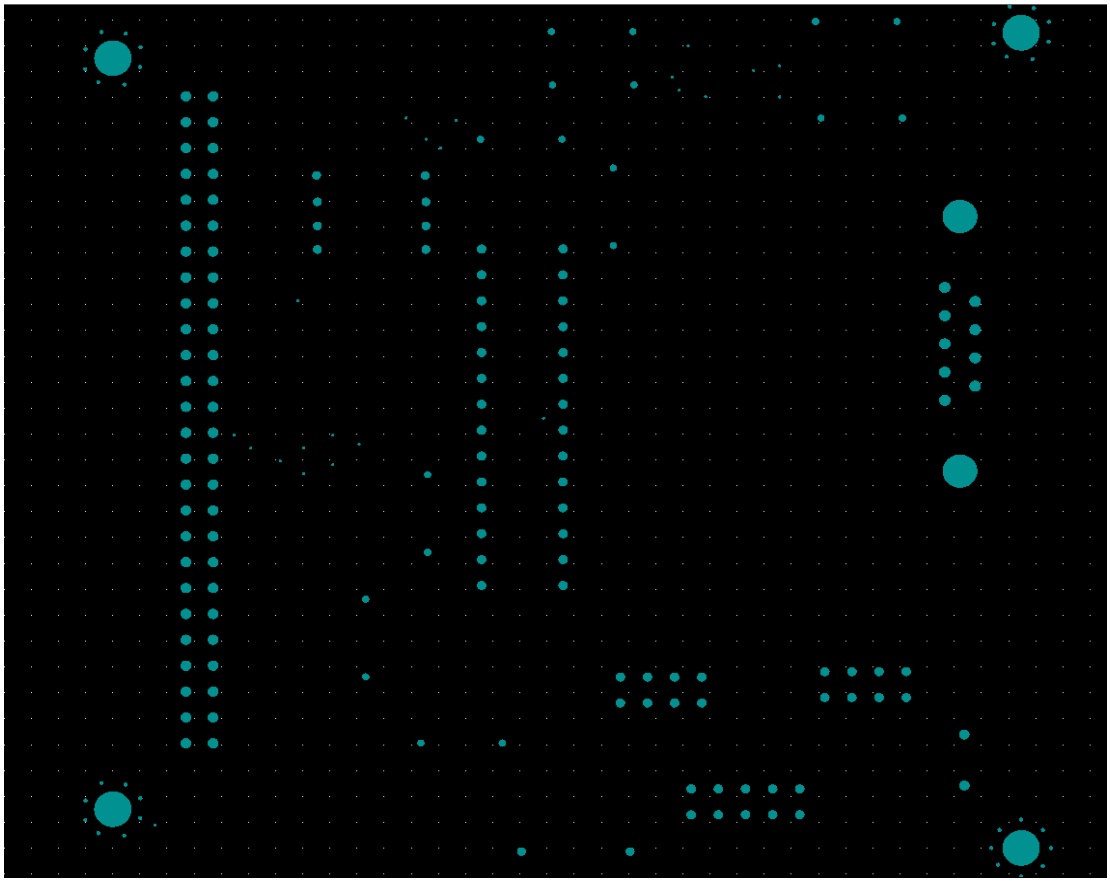

Figure 24 GERBER of version 1

3.7.2 Version 2

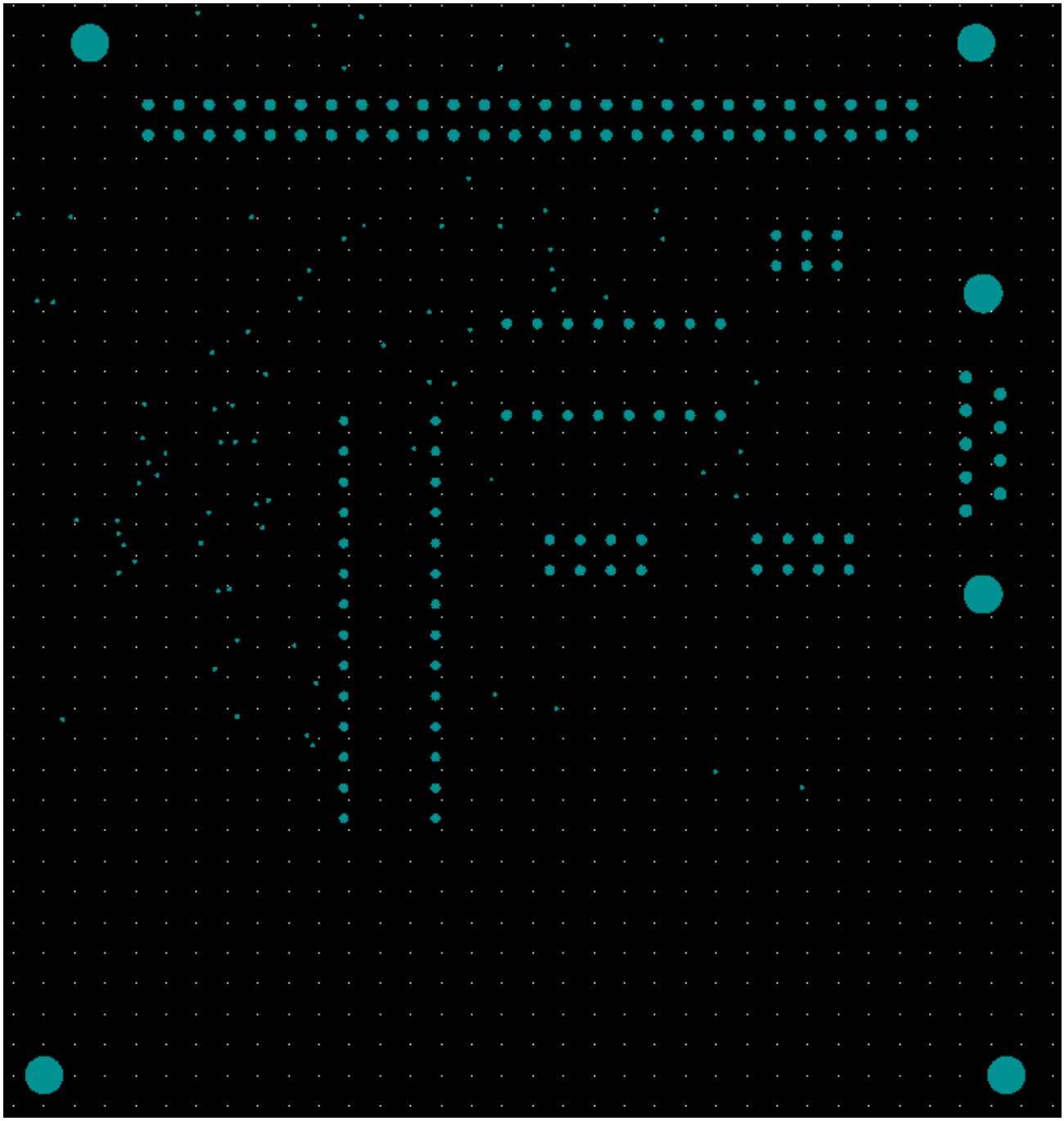

Figure 25 GERBER of version 2

3.8 Assembly

3.8.1 Version 1

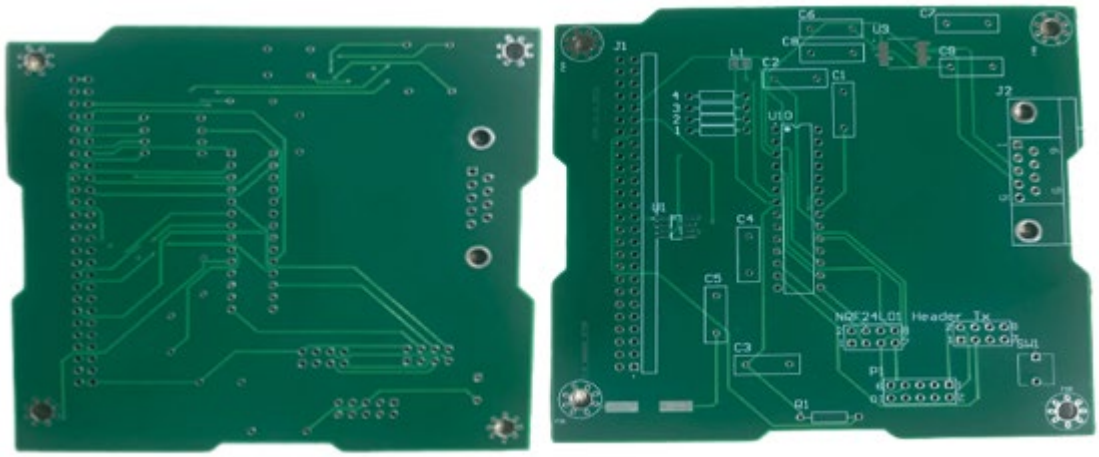

Figure 26 Version 1 PCB

3.8.2 Version 2

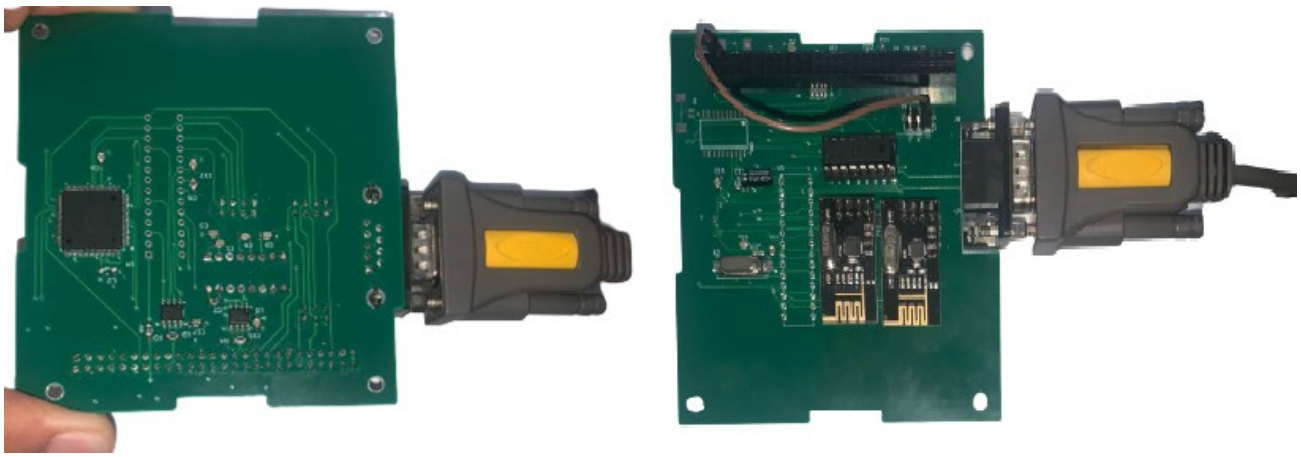

Figure 27 Version 2 PCB

Note : the soldering was at temperature equal to 430 degrees Celsius

3.9 Final result

We have done test on the PCB and we have checked that the controller on the PCB is working also serial interfaces (by using USB-TTL) is working too.

We have done the programming by AVR studio by connecting AVRispmkII to ICSP header.

Figure 28 burning code on the PCB

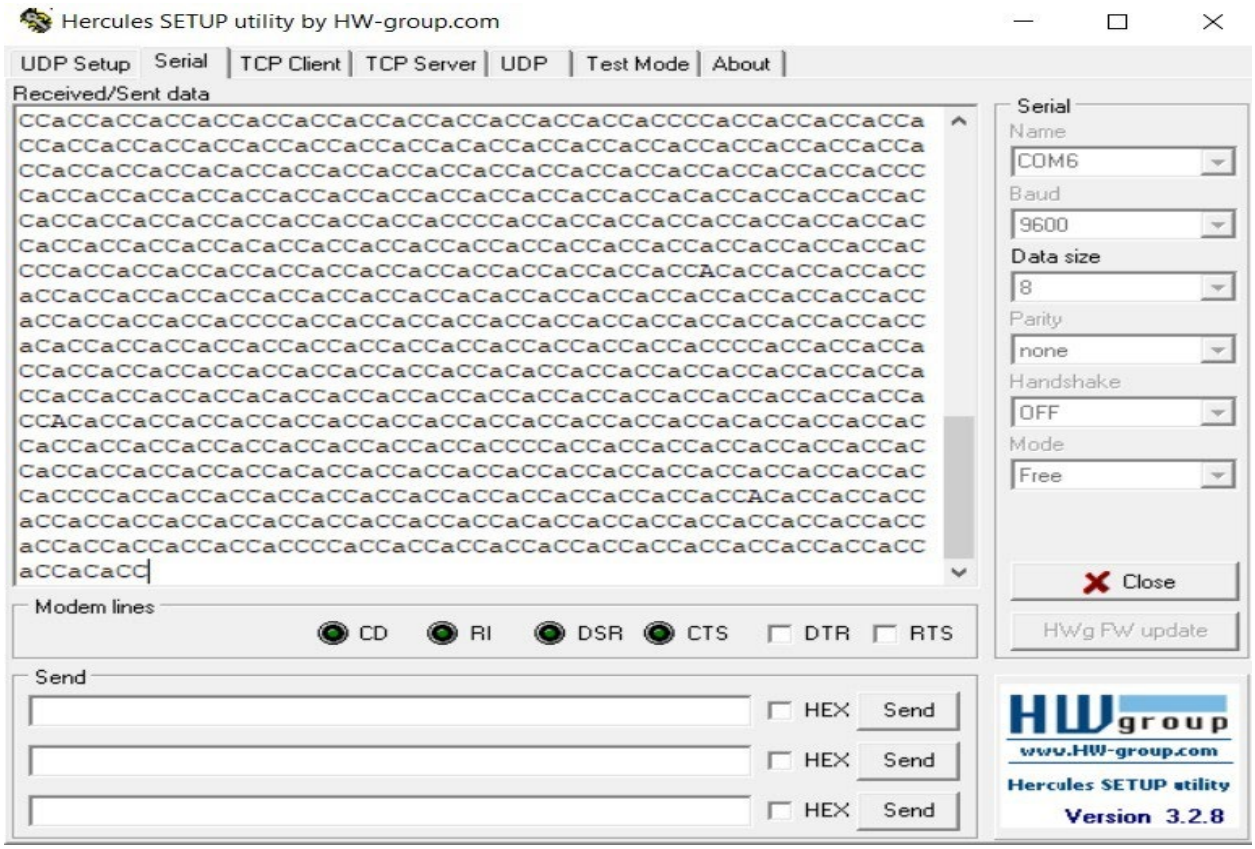

Figure 29 hardware simulation

By using Hercules we make sure that the PCB is working well.

4 Chapter 4 Software

4.1 System Design

The main objective is to establish communication between the satellite and the ground station we do so using two protocols which are the SSP (simple serial protocol) and the AX.25 Protocols, SSP is responsible for communication between OBC (on board computer) and communication board, while AX.25 is responsible for wireless communication between communication subsystem in satellite and ground station.

This is done in four scenarios:

- First:
    - when a command is sent from GCS to satellite. It will be received by communication board in the form of an AX.25 Frame then send a response to GCS and then the SSP Frame inside of it is extracted which then gets passed to the satellite's OBC and then communication board receives a response from OBC.
- Second:
    - when OBC sends a command to the communication board and then the communication board sends back a response to the OBC.
- Third:
    - when OBC sends telemetry data to the communication board, the communication board will send ACK back to the OBC and then the communication board will transmit it to GCS.
- Fourth:
    - when the communication board sends telemetry data to OBC and then the OBC sends a response back.

We implemented the overall system according to the following diagram:

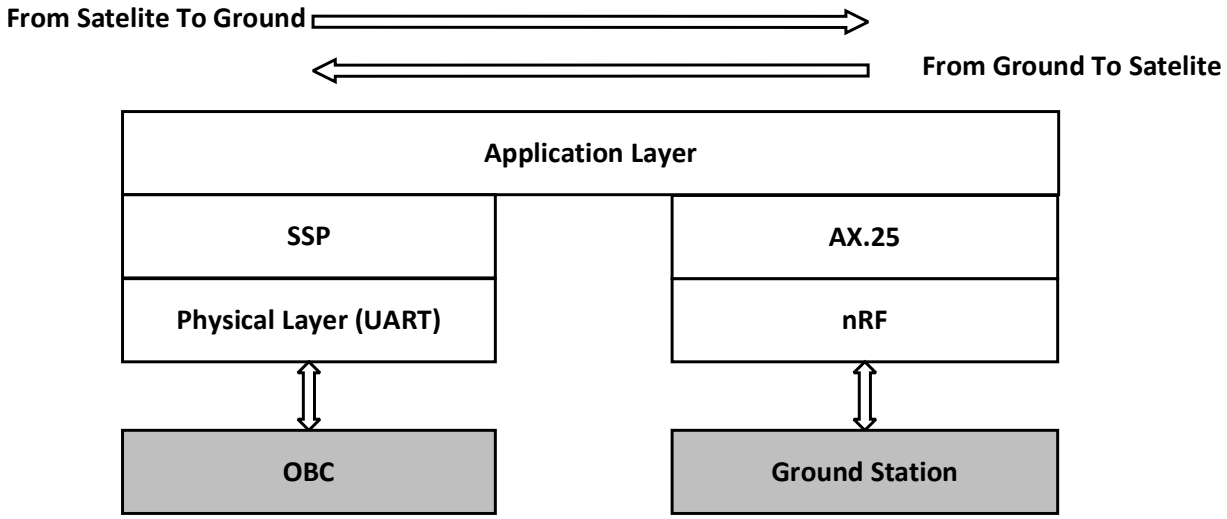

Figure 30 System Diagram

- Application Layer:
    - Interfaces the AX.25 and SSP protocols in order to transfer data between them.
- SSP
    - Responsible for communication between OBC (on board computer) and communication board.
- AX.25
    - AX.25 is responsible for wireless communication between communication subsystem in satellite and GCS.
- Physical Layer (UART)
    - Provides interface between the OBC and the SSP Protocol
- nRF
    - Provides interface for Wireless communication between the AX.25 Layer and the GCS

We note that both of the SSP and AX.25 Protocols are implemented using a State Machine approach which will be discussed in later sections.

4.2 AX.25

4.2.1 Introduction

The AX.25 Protocol is used for the wireless communication between the satellite and the ground station, transferring data encapsulated in frames between nodes, and detecting errors introduced by the communication channel, this protocol features different types of frames for sending data/information and for acknowledgment of frames in order to have reliable data transmission.

In our implementation we divided the protocol into layers as shown in figure below.

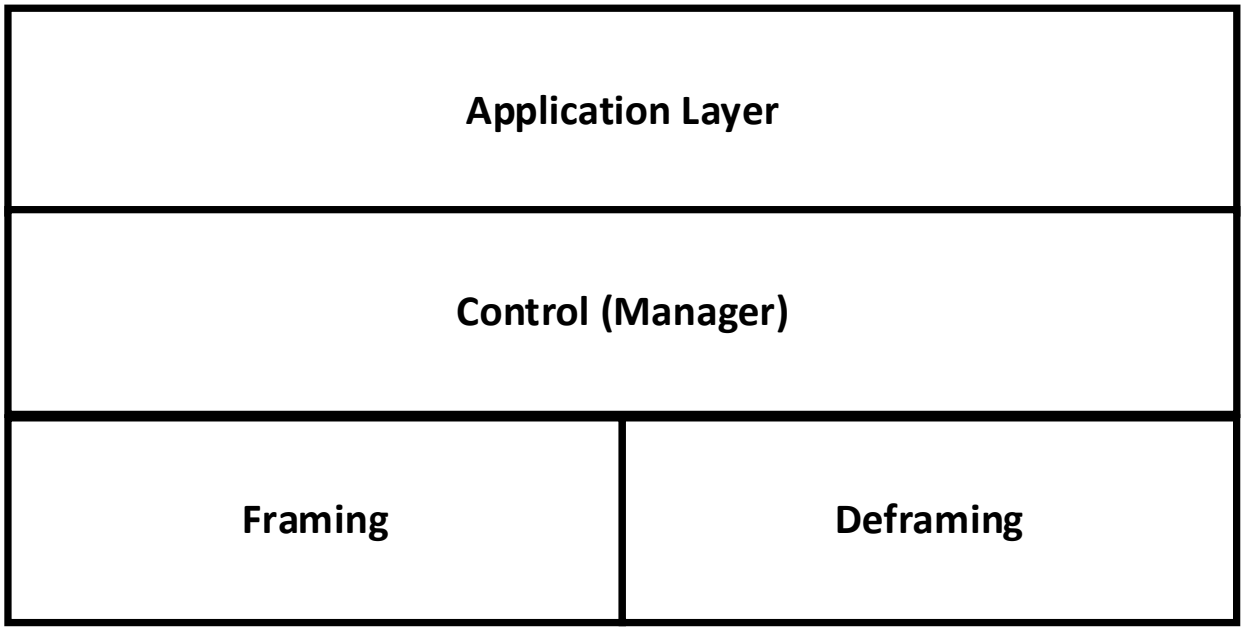

Figure 31 AX.25 Layers

The Application layer is responsible for interfacing between the AX.25 protocol and the SSP protocol, the Manager layer operates the state machine of the protocol (as will be discussed in the next section) and then the Framing layer is responsible calculating the CRC for the frame in order to implement error checking and then building the full AX.25 frame in order to transmit it and then the Deframing layer is responsible for deconstructing the received AX.25 frame into its subfields (which are mentioned in later sections), the layers and the movement of data between them are explained in detail in later sections.

4.2.1.1 Protocol State Machine

The main states of the protocol are ***Idle***, ***Transmit*** and ***Receive*** which are inside the **control** layer shown in the previous figure. The default state is the ***Idle*** state and from then it has two states to go to. If it has data to send then it goes to the ***Transmit*** state where it transmits the data and then waits for the Acknowledgement which will be in the form of an S-Frame. if it has received data then it goes to the ***Receive*** state where it checks on the received data and after verifying the address and the CRC (in the FCS sub-field of the frame)

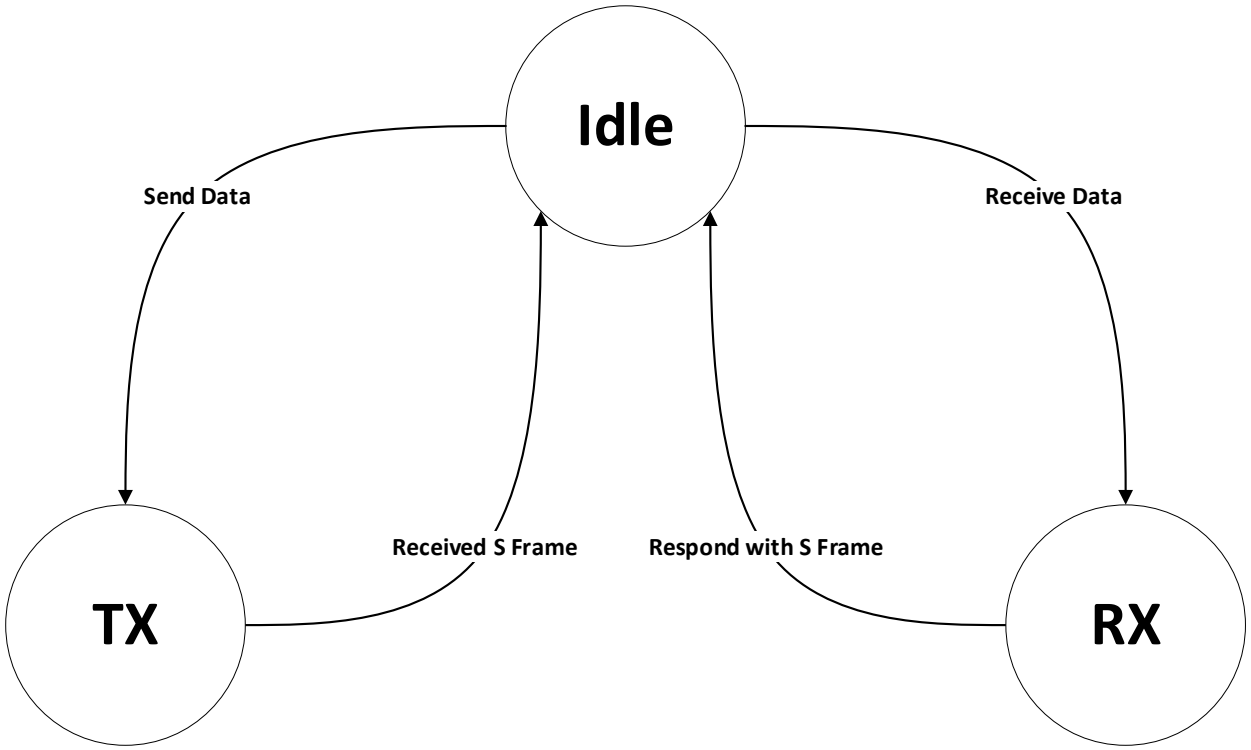

Figure 32 State Diagram of the protocol

4.2.1.2 Frame Response Mechanism

In our current implementation we have managed to implement two types of response namely RR and the REJ, if the TX node receives a RR it continues on sending the next frame, however if a REJ is received the TX node resends the same frame again, and if we get three consecutive REJ we move on to sending the next frame in line.

4.2.1.3 Frame Structure

We have three main types of frames of AX.25 frame, which are:

- Information (I) Frame.
- Supervisory (S) Frame.
- Unnumbered (U) Frame. (Which will not be implemented since EGSA dropped it from the project requirements)

We use I-Frames to send data, and use S-Frames for the acknowledgement mechanism (which are RR, RNR, REJ and SREJ)

All these frames share a common structure as shown

Table 11 AX.25 Frame Structure

Flag	Address	Control	Info	Padding	FCS	Flag
0111 1110	112 bits	8 bits	N*8 bits	N*8 bits	16 bits	0111 1110

4.2.1.4 Flag Field

- The flag has a constant value which is 0x7E, its main function is to delimit the frame

4.2.1.5 Address Field

- In this field the source and destination address of the frame is specified.
- It is split into two subfields, the destination address subfield and the source address subfield.

4.2.1.6 Control Field

This field identifies the type of frame to be sent (I, S or U), it is illustrated as follows:

Table 12 Control field Details

Frame Type	Control Field Bits			
	7 6 5	4	3 2 1	0
I Frame	N(R)	P	N(S)	0
S Frame	N(R)	P/F	S S 0	1

U Frame	M M M	P/F	M M 1	1

The “S” bits are the supervisory function bits. They indicate which type of S frame is being processed, it is encoded as follows.

Table 13 S frame details

Control Filed Type	Control Field Bits			
	7 6 5	4	3 2	1 0
Receiver Ready (RR)	N(R)	P/F	0 0	0 1
Receiver Not Ready (RNR)	N(R)	P/F	0 1	0 1
Reject (REJ)	N(R)	P/F	1 0	0 1
Selective Reject (SREJ)	N(R)	P/F	1 1	0 1

- Receiver Ready (RR) Frame:
  - This type of frame is used to indicate that the sender of this frame is able to receive more “I” frames.
  - It also acts as an acknowledgement for receiving frames up to “N(R)-1” frames
- Receiver Not Ready (RNR) Frame:
  - This type indicates that the sender of this frame is not yet ready to receive more frame
  - It indicates a busy condition
  - It also acts as an acknowledgement for receiving frames up to “N(R)-1” frames
- Reject (REJ) Frame:
  - This type acts as acknowledgement for “N(R)-1” frames
  - it requests retransmission of “I” frames starting from N(R).
- Selective Reject (SREJ):
  - This type is used to request retransmission of a single “I” frame with number “N(R)”

The frame sequence variables are listed as follows:

The Send State Variable "V(S)" which contains the next sequential number that will be assigned to the next "I" frame to be sent.

The Send Sequence Number "N(S)" which contains the sequence number of the "I" frame being sent.

The Receive State Variable "V(R)" which contains the sequence number of the next expected "I" frame to be received.

The Received Sequence Number "N(R)" which exists in both "I" and "S" frames. Prior to sending an "I" or "S" frame N(R) is updated to equal V(R) [to acknowledge the proper reception of frames up to "N(R)-1"].

4.2.1.7 The Information Field

The Information field carries the user data, in our use case this field will be filled with the SSP frame

4.2.1.8 The Padding Field

As mentioned above the information field will contain the SSP frame, and since the SSP frame doesn’t have a fixed length but rather a max size of 236 bytes, while the AX.25 frame has a fixed length, the difference between the two frames is padded with (0xAA).

4.2.1.9 The Frame Check Sequence (FCS) Field

The FCS is a 16-bit number calculated on both ends of transmission (TX and RX) to ensure data integrity. In our use case we use CRC16-CCITT which is explained in the SSP section.

4.2.2 Implementation

4.2.2.1 Protocol Layers and Buffers

Now that we have quickly revised the frame structure, we will discuss our implementation. (Note: for further details the source code is listed in the Appendix)

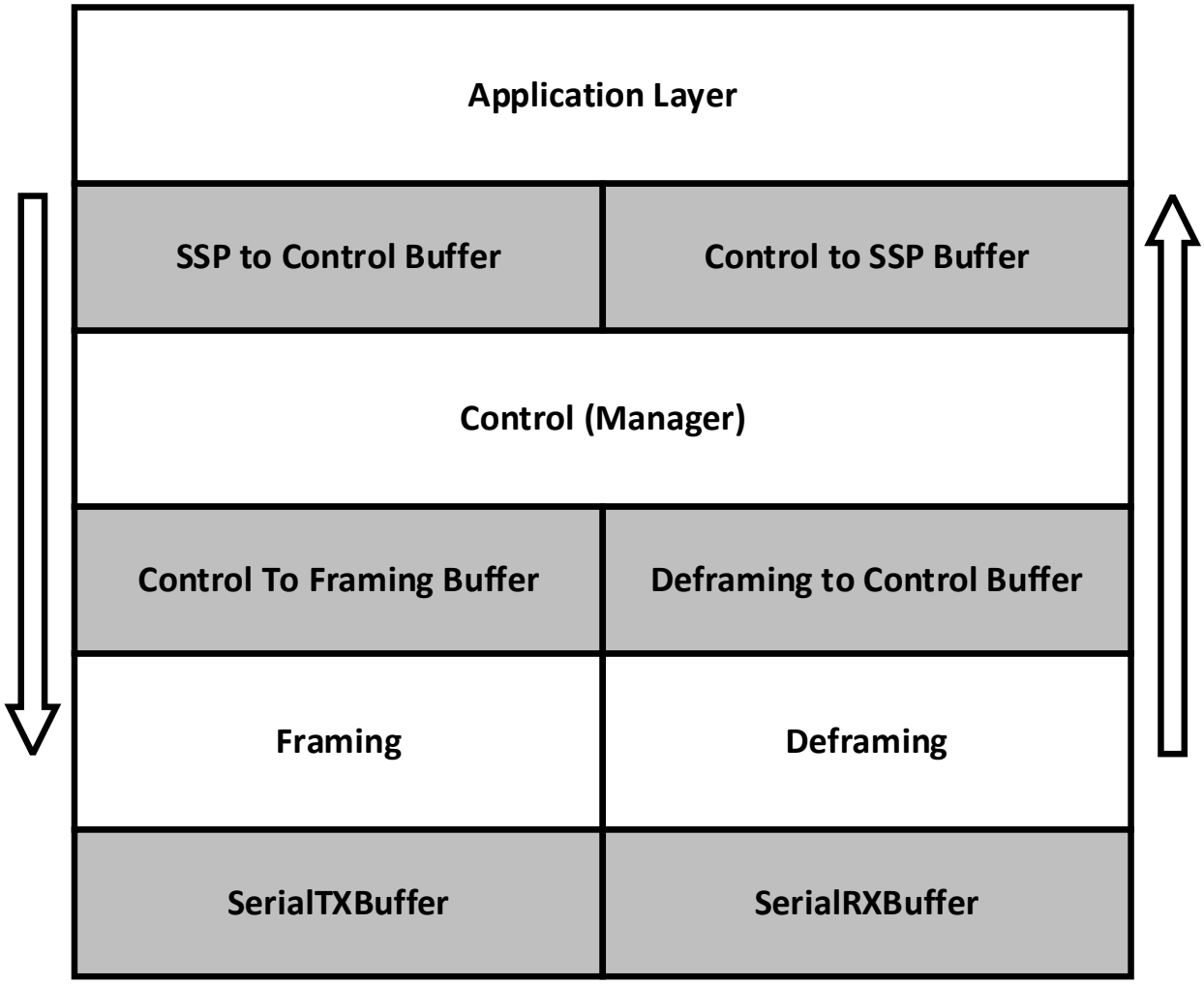

Figure 33 Protocol Layers

Currently the protocol has the following layers that hand each other the data through a shared buffer between each consecutive layers, note that the Application Layer interfaces with the SSP Protocol, and the serial RX and TX buffers represent at will be sent and received from the physical layer (which is will be nRF which is communicates using SPI).

4.2.2.1.1 Layers

- Application
    - It's the interface between AX.25 and SSP protocols, so that AX.25 takes the info field from the SSP Frame, it has two functions depending on whether the node is acting as TX or RX
        - TX: AX.25 take from this layer the SSP frame and put it in the information field
        - RX: AX.25 gives its info field to the SSP layer
- Control (Manager)
    - This layer is responsible for managing the state of the protocol according to flags that are set/cleared by it and by the other layers (the protocol states will be explained later on)
- Framing
    - This layer is responsible for collecting the subfields from the Control to Framing Buffer and it then calculates and inserts the CRC into the FCS field and then finally it delimits the frame with the flags (0x7E)
- De-framing layer :
    - In the receiving side when the receiver received frame its function is to check it by divided it into subfields flag , address , information , CRC . It checks if the CRC is correct. if yes it sets the CRC flag and moves to the next layer if there is a problem in the CRC it will signal the above layer by flag to reject the frame.

4.2.2.1.2 Buffers

- SSP to Control
    - Contains the SSP Frame
- Control to Framing
    - Contains the SSP Frame, Control Byte
- SerialTXBuffer

    - Contains the full AX.25 Frame, which then gets sent to the nRF module
- SerialRXBuffer
    - Contains the full received AX.25 frame
- Deframing to Control
    - Contains the data of the received frame with each field in the frame separated
- Control to SSP
    - Contains the SSP Frame and the Control Byte

4.2.2.2 Protocol Functions' Flowcharts

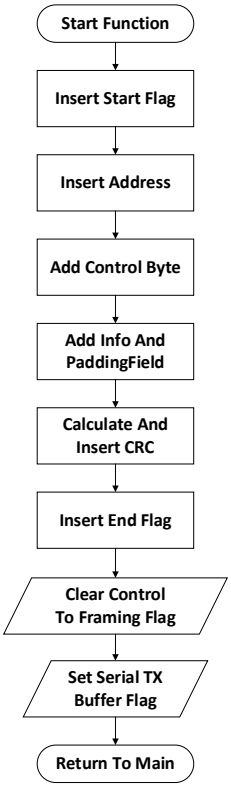

Figure 34 Build Frame Flowchart

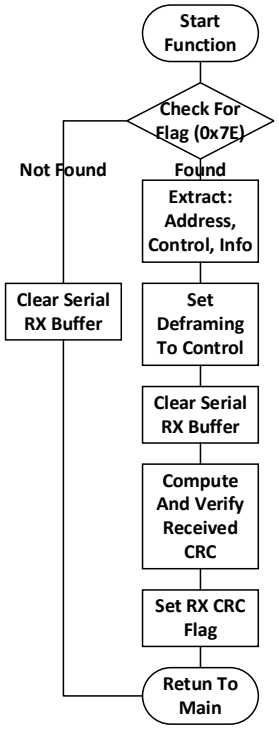

Figure 35 De-frame Flow Chart

Manager (TX)

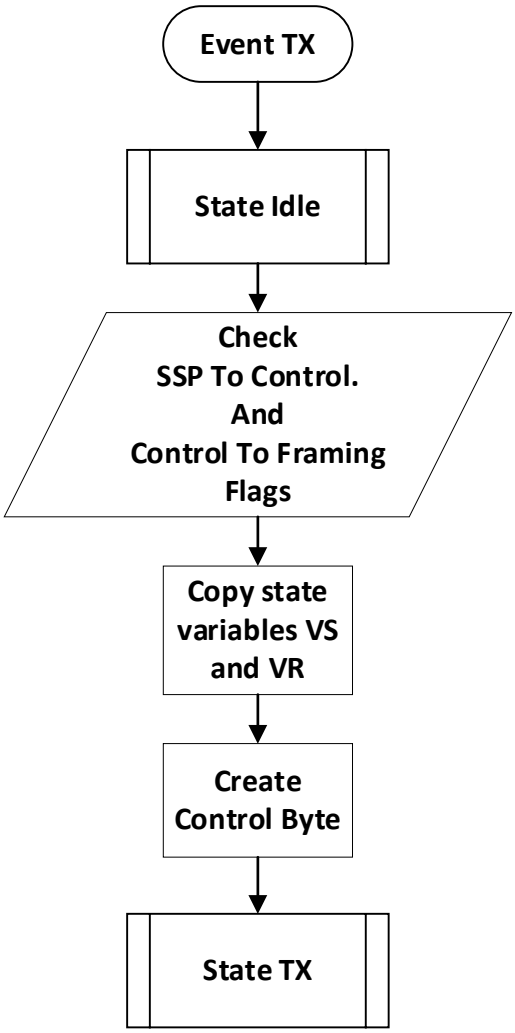

Figure 36 Manager Flow Chart TX 1

Manager (TX) cont…

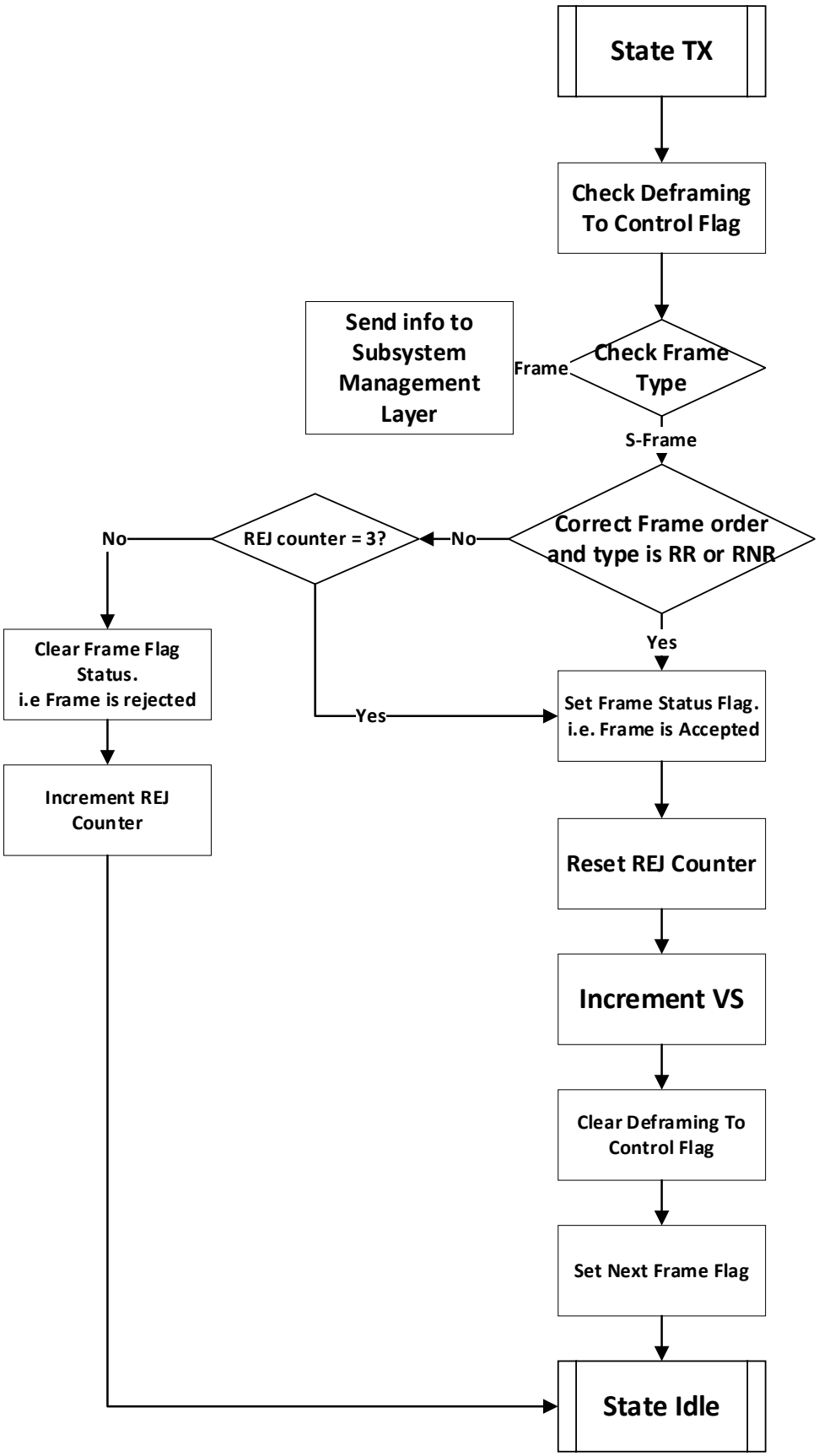

Figure 37 Manager Flow Chart TX 2

Manager (RX)

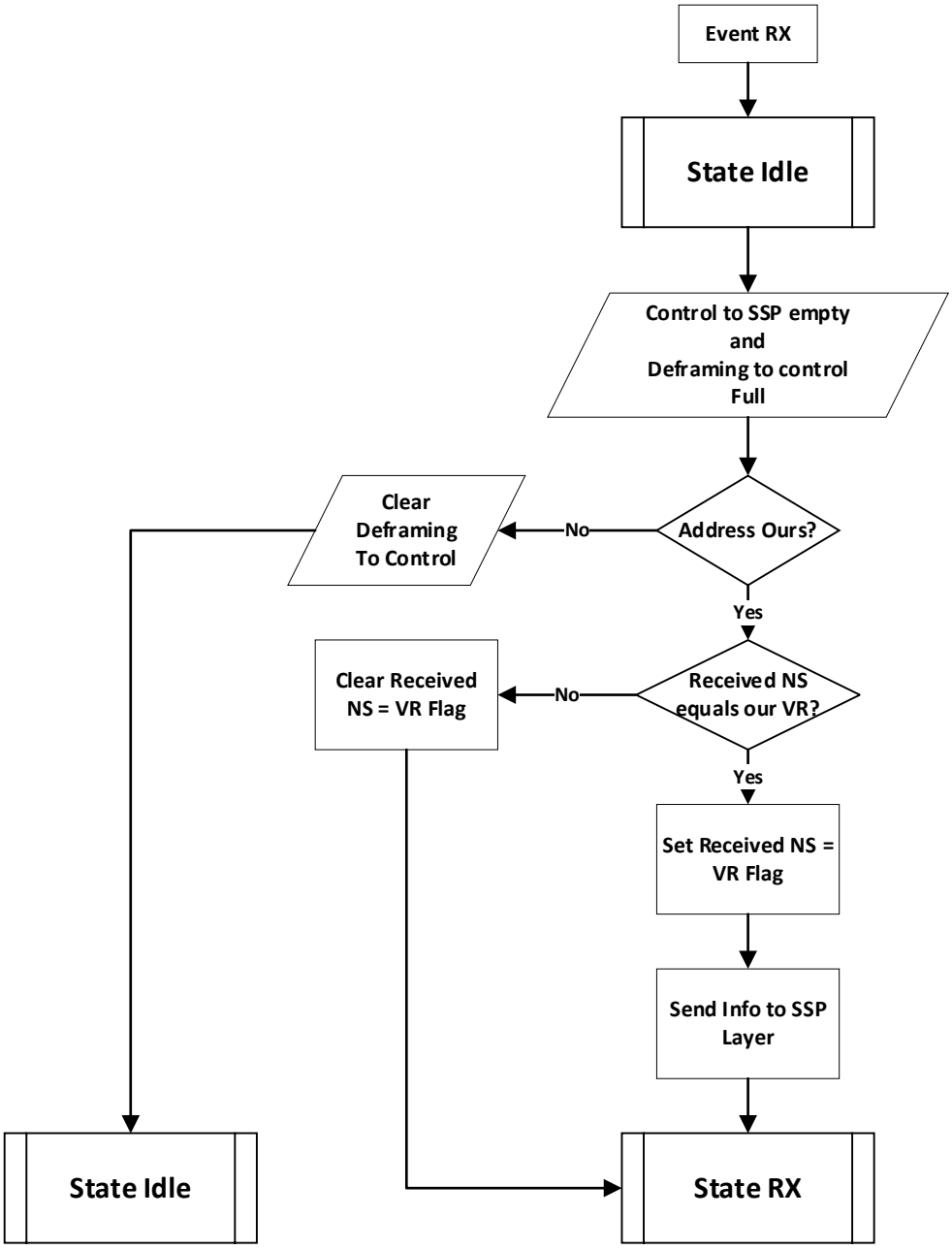

Figure 38 Manager Flow Chart RX 1

Manager (RX) cont…

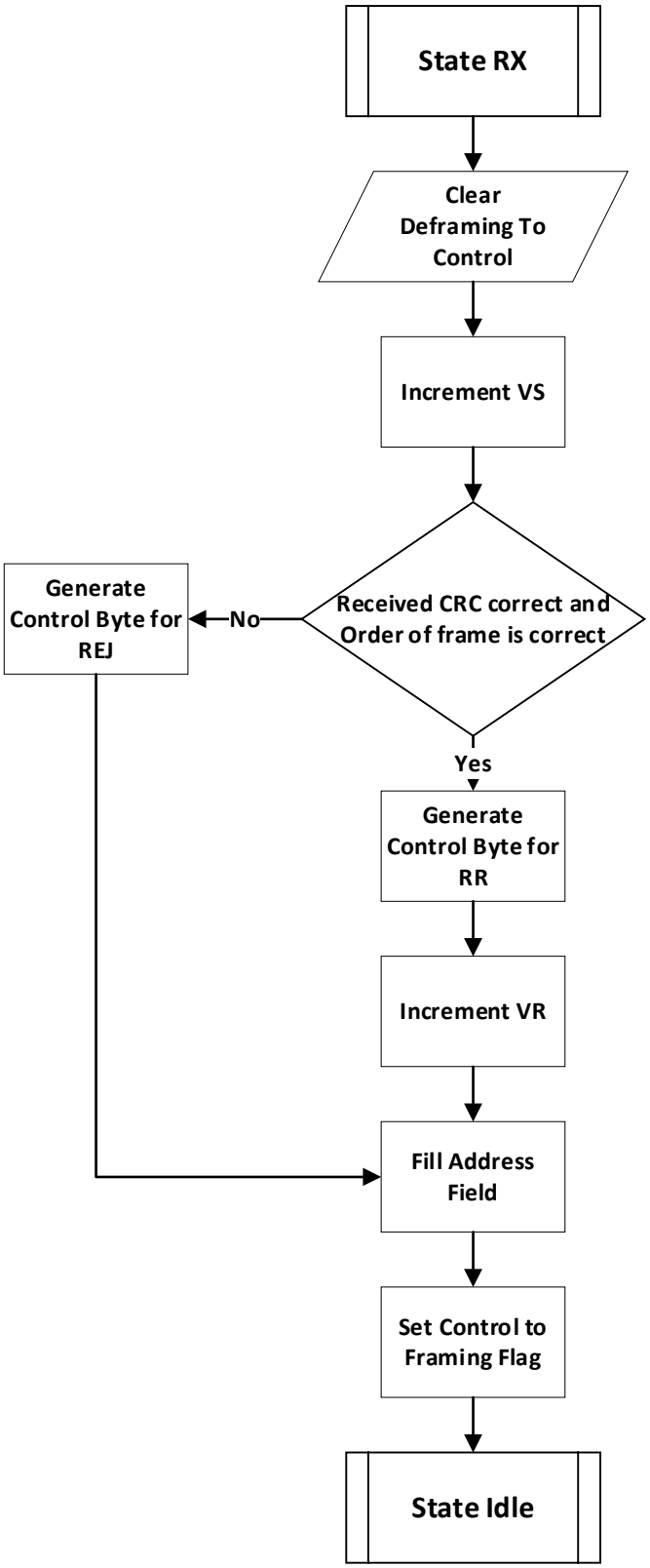

Figure 39 Manager Flow Chart RX 2

4.2.2.3 Protocol Functions

Table 14 deframe funciton

Function Name	AX25_deFrame(uint8 *buffer, uint16 frameSize, uint8 infoSize)	
Function Return	void	
Function Location	Deframing Layer	
Function Description	takes whole frame and splits it into fields (address, control, info).and also checks on the frame flags (namely 0x7E) and the CRC	
Description for each argument		
Name	**Usage**	**Type**
buffer	the buffer that contains the full frame which will be de-framed.	uint8
frameSize	Contains the size of the frame	uint16
infoSize	Contains the size of the info field in that frame	uint8

Table 15 Build Frame Function

Function Name	AX25_buildFrame(uint8 *buffer, uint8 *a_info_ptr, uint16 *frameSize, uint8 *ADDR, uint8 control, uint8 infoSize)	
Function Return	void	
Function Location	Framing Layer	
Function Description	Takes the address, control, info fields then adds padding, start and end flags, and then calculates the crc and then form the complete frame in the correct order.	
Description for each argument		
Name	**Usage**	**Type**

buffer	This is the output buffer of this function where we output the full frame (i.e Serial TX Buffer)	uint8*
a_info_ptr	This is a pointer to the information field of the AX.25 Frame which is the SSP Frame	uint8*
frameSize	This contains the size of the AX.25 Frame	uint16*
ADDR	This is a pointer to the address of the node	uint8*
control	Contains the control byte which was calculated in the Control Layer	uint8
infoSize	Contains the size of the information field of the AX.25 Frame	uint8

Table 16 Manager Function

Function Name	AX25_Manager(uint8 *a_control)	
Function Return	void	
Function Location	Control Layer	
Function Description	This function is responsible for the state machine of the protocol which controls which state the protocol is in	
Description for each argument		
Name	**usage**	**Type**
a_control	This is a pointer to the control byte which we calculate and then insert into the variable which this pointer points to	uint8*

Table 17 increment state variable Function

Function Name	incrementStateVar(uint8 *stateVar)
Function Return	void
Function Location	Control Layer

Function Description	increments the given State Variable (VR or VS) and overflows if reach value of 7.	
Description for each argument		
Name	**Usage**	**Type**
stateVar	This is a pointer to either VS or VR.	uint8*

Table 18 getControl Function

Function Name	**AX25_getControl(frameType frameType, frameSecondaryType secondaryType, uint8 NS, uint8 NR, uint8 pollFinal)**	
Function Return	**uint8**	
Function Location	**Control Layer**	
Function Description	**Computes control byte according to it's given arguments**	
Description for each argument		
Name	**Usage**	**Type**
frameType	Give frame type (I, S, U)	frameType
secondaryType	RR, RNR, REJ, SREJ, SABME, DISC, DM, UA , UI, TEST	frameSecondaryType
NS	Send number sequence	uint8
NR	Receive number sequence	uint8
pollFinal	Determines value of Poll/Final bit	uint8

Table 19 Compute CRC Function

Function Name	computeCRC(uint8 *data_p, uint16 *length)
Function Return	uint16
Function Location	Framing Layer
Function Description	Computes the CRC16-CCITT

Description for each argument		
Name	**usage**	**Type**
data_p	Points to data that we calculate CRC from	uint8 *
length	The length of the output buffer	uint16 *

Table 20 PutCRC Function

Function Name	AX25_putCRC(uint8 *buffer, uint16 *OpArrSize)	
Function Return	void	
Function Location	Control Layer	
Function Description	Puts the calculated CRC in its correct position in the frame	
Description for each argument		
Name	**Usage**	**Type**
buffer	pointer of the frame buffer.	uint8*
OpArrSize	it stores the index of the last inserted element in the array to keep track of size	uint16*

Add flowchart for each function in the code, add input and output of each function.

Explain how main works, then explain each function.

4.2.2.4 Flags

Table 21 AX.25 Flags

Flag	Values		Usage	Type
	Set in	Checked by		
flag_RX_crc	deframe	Manager	used in the Manager function to check whether CRC is correct or not	uint8
flag_SSP_to_Control	fillBuffer, Manager	Manager	Shows whether the SSP to control buffer is full or not	uint8
flag_Control_to_Framing	buildFrame, Manager	Manager	Shows whether the Control to Framing buffer is full or not	uint8
flag_Control_to_SSP	fillBuffer, Manager	Manager, Main	Shows whether the Control to SSP buffer is full or not	uint8

flag_Deframing_to_Control	Deframe, Manager	Manager, Main	Shows whether the Deframing to Control buffer is full or not	uint8
flag_SerialTXBuffer	buildFrame	Main	Shows whether the Serial TX Buffer is full or not	uint8
flag_SerialRXBuffer	deFrame,	Main	Shows whether the Serial RX Buffer is full or not	uint8
flag_next_frame	fillBuffer, Manager, Main, controllayer (in SSP)	Main	Indicates whether we can send a new frame or not	uint8

4.3 Simple Serial Protocol (SSP)

4.3.1 Introduction

SSP (Simple Serial Protocol) is a simple protocol intended for master-slave communication on single-master serial buses, especially within small spacecraft.

We use it as a control protocol with OBC in the cube satellite and as an internal protocol. It provides the interface or link between the OBC and the communication board to send the payload mission data or the telemetry data encapsulated in frames from OBC to the communication board to then pass it to the external protocol to be sent downlink and also to pass a command that was sent uplink from the communication board to OBC to get a response or a telemetry data according to the command.

It provides reliable data transfer between nodes using an acknowledgment message (ACK/NACK). Upon detecting an error or loss of frames, the source can recover by retransmitting the frame.

SSP uses SLIP (also known in the amateur-radio community as KISS), to split a stream of characters into frames.

4.3.2 Serial Line Internet Protocol (SLIP)

SLIP framing, Frame is passed down to SLIP framing, which breaks it into bytes and sends them one by one over the link. After the last byte of the frame, a byte of a unique value is sent to tell the receiving device that the frame has ended. This is called the (SLIP END character) and has a byte value of C0 hexadecimal, 192 decimal, and 11000000 binary. And then it takes the entire frame, sends it one byte at a time, and then sends the byte C0 to delimit the end of the frame.

There is another issue SLIP framing deals with which is if the END character C0 hexadecimal appears in the data itself and the solution for this problem is discussed in detail in section (4.3.3.4.1).

4.3.3 Frame Structure

Table 22 Frame STRUCTURE OF SSP

Fend	Destination	Source	Type	Data	CRC0	CRC1	Fend

4.3.3.1 Destination

A one-byte destination address identifies the process which is the destination of the packet. A value of 0 is reserved for possible future use as a broadcast address. Values of FEND or FESC are forbidden.

4.3.3.2 Source

A one-byte source address identifies the process of sending the packet. For purpose of responses. A value of 0 is reserved for possible future use to indicate an enhanced packet format. Values of FEND or FESC are forbidden.

4.3.3.3 Type

Type is a byte with the fields ss and pktype in it.

The top two bits are supplementary data (ss), which may have significance for particular packet types and we'll always set it to 0, and the bottom six bits are a packet type (pktype) field.

SSP establishes a few standard packet types to ensure those common requirements are satisfied the same way in all implementations. It does not need to define a new type field or perform an additional decoding step because it leaves a number of packet types open for customized use.

The values of the pktype field are predefined to have standard meanings.

Table 23 pktype field

Value	Name	Description	*ss* Bits
0	PING	test packet	0
1	INIT	initialize	flavor
2	ACK	response	flavor
3	NAK	rejection	cause
4	GET	variable fetch	address space
5	PUT	variable store	address space
6	READ	memory fetch	address space
7	WRITE	memory store	address space
8	ID	identify	phase
9		(reserved)	
10 ... 53		custom requests	as needed
54	ADDRT	add route	0
55	DELRT	delete route	0
56	GETRT	read routing table	0
57 ... 63		(reserved)	

ACK: Acknowledgement response, the request has been successfully performed. May or may not include data in response to the request.

NAK: Negative-acknowledgement response, the request could not be performed.

GET: Request to read from one or more variables.

READ: Request to read from memory.

WRITE: Request to store to memory.

PING: Requests a response, with no other action.

4.3.3.4 Data

4.3.3.4.1 Escaping Special Characters

As we mentioned in section () that SLIP framing is used by SSP. SLIP framing addresses another issue to deal with. If the END character is C0 in hexadecimal "FEND" appears in the data itself, and transmitting it that way would fool the receiver into thinking that the frame ended. Therefore, a special Escape character

(ESC) is developed to address and prevent this problem. Its value is DB hexadecimal "FESC," 219 in decimal, and 11011011 in binary. When a symbol is described as "escape," it signifies that it carries the message "this byte and the one after it has a special meaning.". When a value of C0 appears in the data, the transmitter replaces it with the ESC character DB hexadecimal followed by the value DC hexadecimal “TFEND”. Thus, a single “C0” becomes “DB DC” in hexadecimal or “219 220” in decimal “FESC and TFEND”. The recipient translates back from “DB DC” to “C0”.

This leaves the final issue that SLIP framing can deal with. If the escape character itself DB hexadecimal is in the original data that would fool the recipient into thinking that this byte is not the original one and try to translate it back. This is handled by a similar substitution instead of DB hexadecimal we put “DB DD” “219 221” in decimal “FESC and TFESC”.

So, to summarize what SLIP framing deals with in the data field, first if the data contains FEND it will be replaced by “FESC and TFEND” and if it contains FESC it will be replaced by “FESC and TFESC”.

4.3.3.5 Cyclic redundancy check

A technology called error detection or error control makes it possible to transfer digital data with reliability over shaky communication lines. Many communication channels are subject to channel noise, which can introduce errors during transmission from the source to the receiver. Such errors can be detected using error detection techniques.

The precise definition of CRC algorithms is a source of significant debate, with much of it centered on bit order. We'll try to explain how the SSP CRC is determined. The exact combination that was used as a historical accident.

The CRC polynomial is x16 + x12 + x5 + 1, which is a 16-bit CCITT polynomial.

The initial shift-register value is all 1s.

The output is the contents of the shift register; no final XOR operation is performed on the shift register.

The LSB (least significant bit) of each data byte is put into the shift register first. This is the standard mode of communication.

However, this is the opposite of how programmers often think about bit order, and many software CRC implementations start with the MSB and work their way down.

The most obvious technique to check the CRC of a received packet is to compute the CRC of the header and data, then compare the computed CRC's bytes to the received CRC's bytes. The CRC can also be computed over the full received (packet, header, data, and CRC) which will return zero if the received CRC matches the rest of the packet.

Any single-bit error, any two single-bit errors, any odd number of single-bit errors, and any single burst of mistakes up to 15 bits long will be detected with this CRC.

4.3.4 Architecture and Implementation

We designed the protocol to have four layers and a flag between each layer to control the flow between layers. To move to the next layer the flag of this layer must be full and the flag of the next layer must be empty.

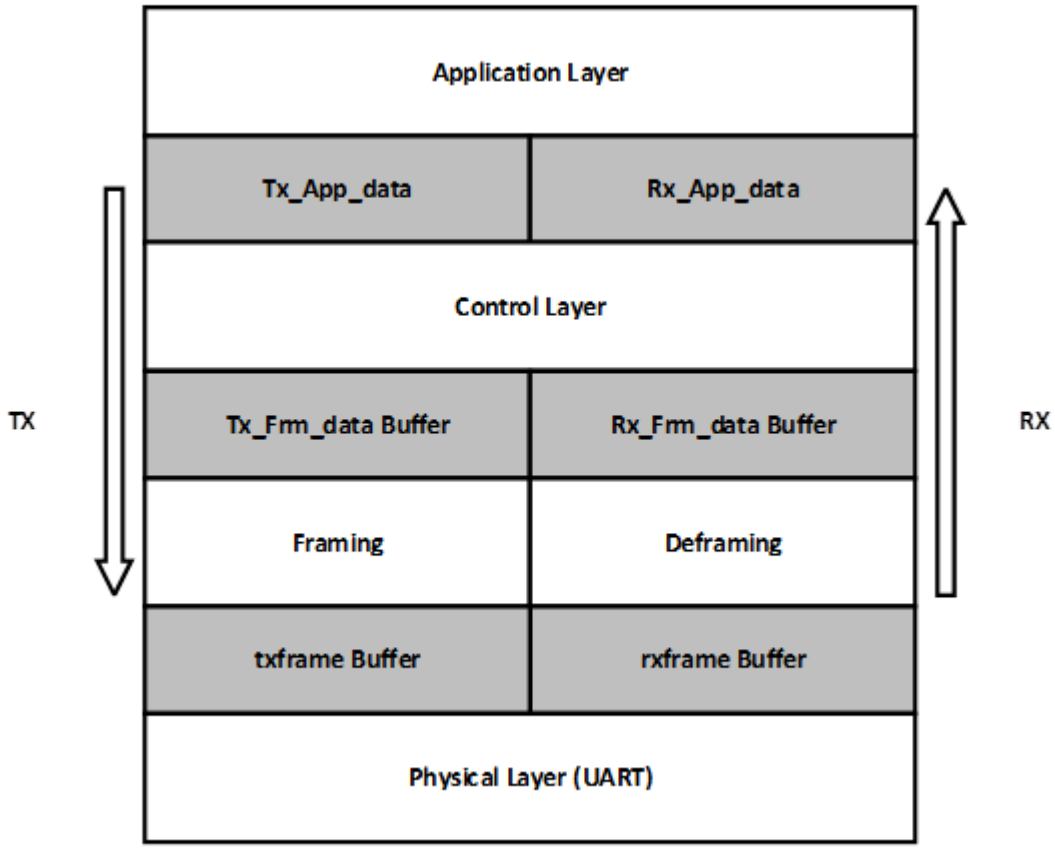

Figure 40 SSP layers and buffers

- Application layer:
    - Responsible for communication between SSP and AX.25.
- Control layer:
    - Responsible for tracking frame and Acknowledgment.
    - Responsible for the determination of communication mode, it has two functions depending on whether the node is acting as TX or RX.
- Framing Layer:
    - Responsible for appending all the fields to build the frame.
- De-framing Layer:
    - Responsible for deconstructing the received frame into its subfields.
- Physical Layer: RS485 protocol constrains (UART)

The satellite uses a half-duplex RS485 bus with the following constraints:

- Data rate = 115200 b/s

- UART configuration = 8N1
- time between the end of TX and switching to Rx less than 0.2 ms.

4.3.4.1 Buffers

- Tx_App_data
  - Contains the data coming from the application layer to be sent.
- Tx_Frm_data
  - Contains the data to be sent to the framing layer.
- txframe
  - Contains the full SPP Frame, which then gets sent serially.
- rxframer
  - Contains the full received SSP frame
- Rx_App_data
  - Contains the data of the received frame to send it to the application layer.
- Rx_Frm_data
  - Contains the data of the received frame after de-framing.

4.3.4.2 Protocol State Machine

The main states of the protocol are Idle, Transmit and Receive which are inside the control layer shown in section(4.3.4.3.3). The default state is the Idle state and from then it has two states to go to. If it has data to send then it goes to the Transmit state where it sends the data and then waits for the Acknowledgement to provide a reliable data transfer. if it has received data then it goes to the Receive state where it checks on the received data and after verifying the address and the CRC.

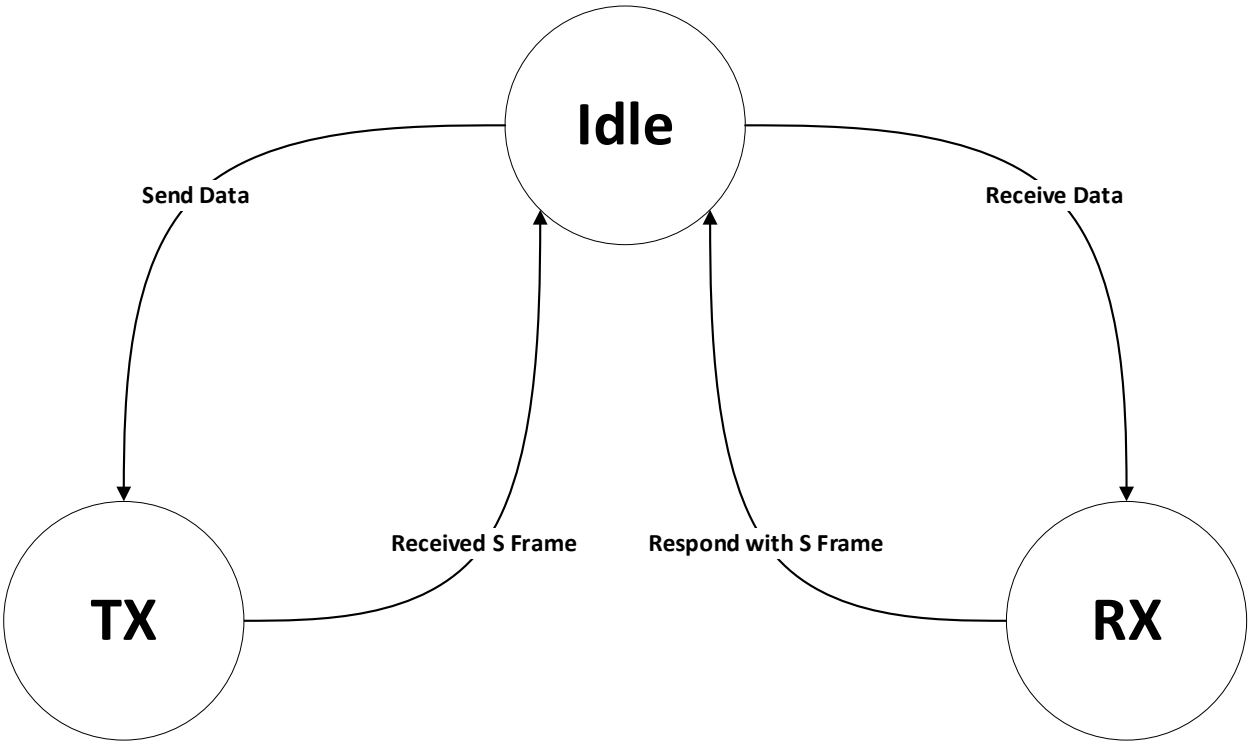

Figure 41 State Diagram of the protocol

4.3.4.2.1 Transmission mode

During transmission mode, it starts by being in the idle state then by checking the data flag which is responsible to check if there is a data to be sent or not and the control to framing flag which is responsible to check if the framing layer is empty and ready to be executed to pass the fields of the frame to it or not.

So, if the data flag were full and the control to framing flag is empty it will send the subfields of the frame from the control layer to the framing layer and then to the physical layer and wait for a response, it resends the same frame when the response is NACK until the counter reaches the maximum counts which are three successive times of NACK response or it receives an ACK after that it will escape this frame and return to the idle state but if it receives ACK, it clears the NACK counter and returns to the idle state to be able to send another frame.

It also has a response timeout which occurs after transmitting the data and waiting for a response to overcome the problem of getting stuck waiting for a response and losing frames.

So, we made a 3-sec timeout to receive a response within this time after that it won't wait for a response and will go back to the idle state to send another frame.

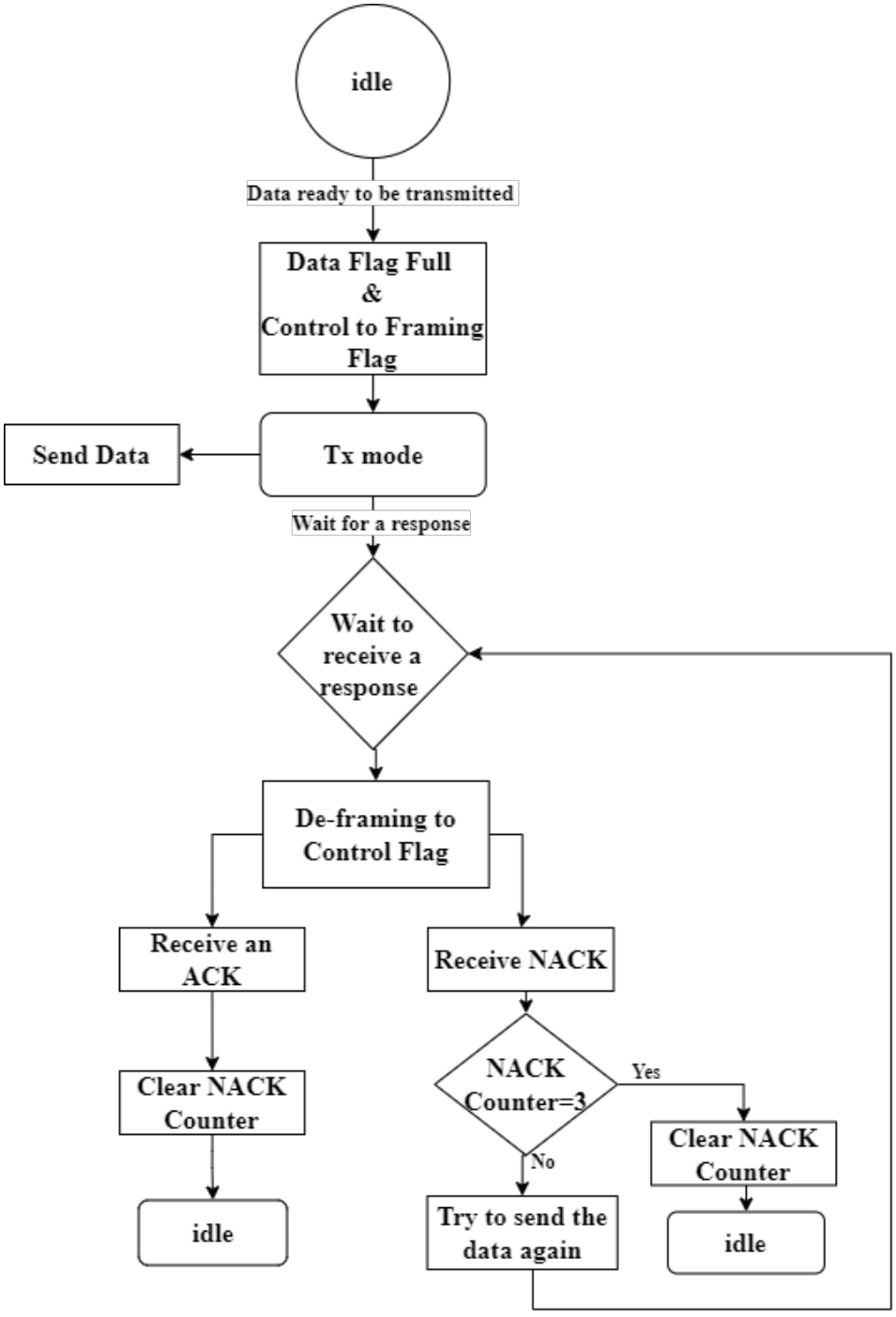

Figure 42 Transmission mode flowchart

4.3.4.2.2 Receiving mode

During receiving mode, it starts by being in the idle state then by checking the receive flag which is responsible to check if there is a frame received or not, and the application layer flag which is responsible to check if it's empty and ready to send the data or not.

So, if the receive flag is full and the application layer is empty this means it's ready to begin the receiving mode, we start by checking the destination field in the received frame to check if it's equal to the layer source this means that the frame was coming to the right destination. it continues to check the CRC flag and if it was correct, it will send ACK and pass data to the application layer else send NACK.

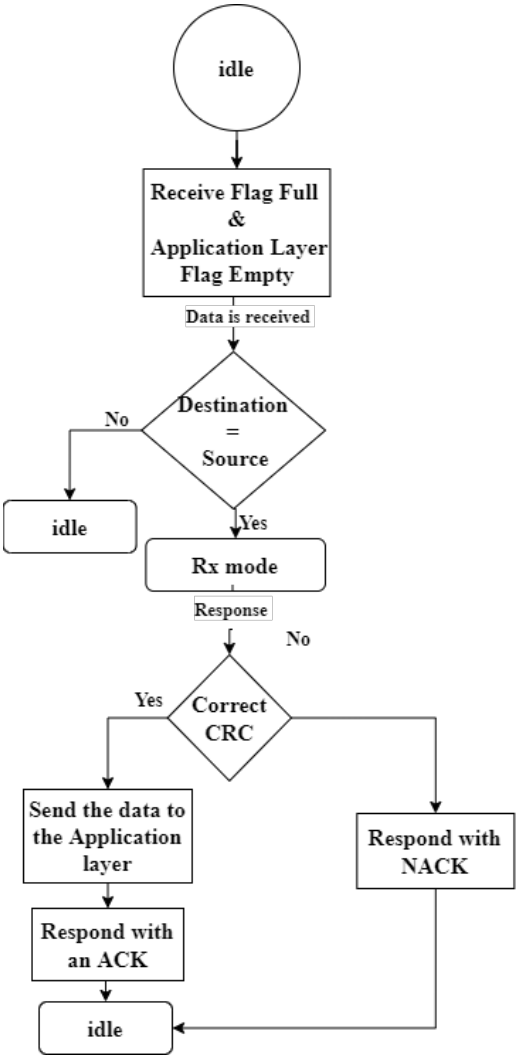

Figure 43 Receiving mode flowchart

4.3.4.3 Functions

These are the functions used to achieve the purpose of each layer

4.3.4.3.1 Framing

This function takes all fields of the frame; it has another function in it that is responsible to compute CRC for these fields to then be appended in the frame to build a full frame.

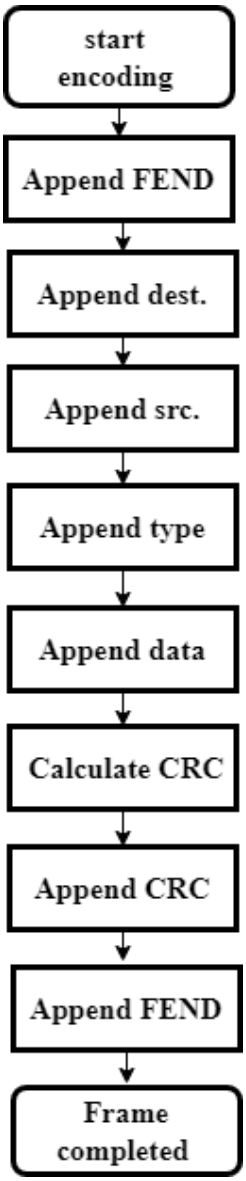

Figure 44 Framing Layer Flow Chart

Table 24 SSP framing function

Function name	ssp_build_frame	
Function arguments	txframe, data, desti, srce, typee, tx_size	
Function Return	Void	
Function description	It's responsible for appending all the fields of the SSP frame to build it.	
Description of each argument		
Name	**Usage**	**Type**
txframe	Output buffer to store the full built frame in it and transmit it.	uint8
data	Input buffer of the data that was passed out from the layer above which is the control layer.	uint8
desti	Input variable of destination address that was passed out from the control layer be appended in the second byte of the frame.	uint8
srce	Input variable of source address that was passed out from the control layer to be appended in the third byte of the frame.	uint8
typee	Input variable of the type of the frame that was passed out from the control layer to be appended in the fourth byte of the frame.	uint8
tx_size	Input variable of the size of the input data that was passed out from the control layer and its extendable to do the replacement of the C0 and DB in the data.	uint16

Flags of the function

txflag: Flag for the framing layer to check if it's EMPTY and ready to be executed or FULL and busy.

4.3.4.3.2 De-framing

This function received a full frame and extracts each field of the frame after checking the CRC.

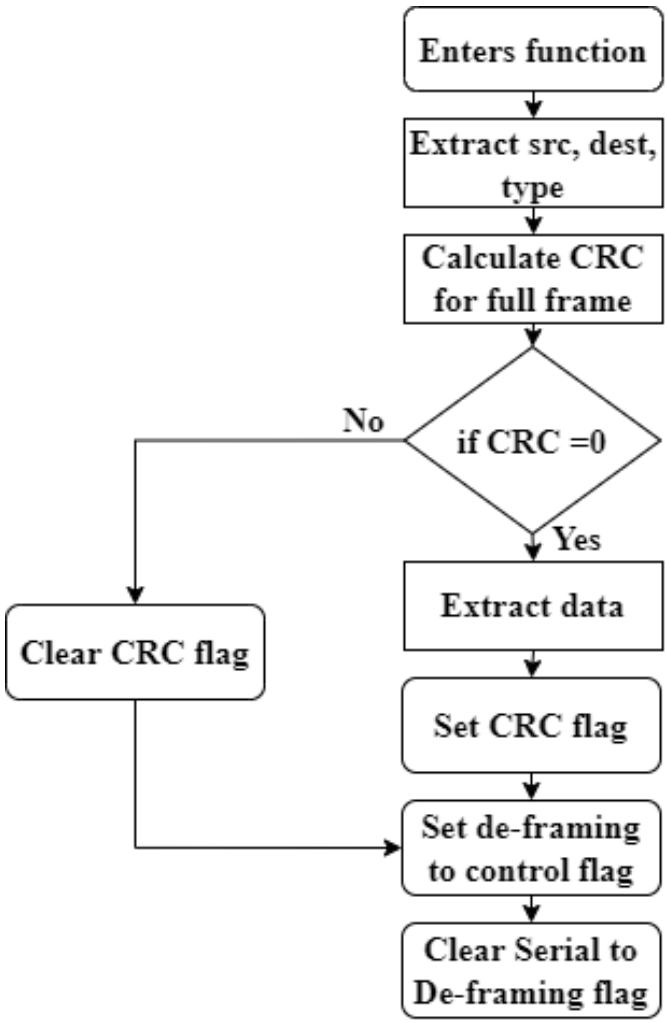

Figure 45 De-framing Layer

Table 25 SSP De-framing function

Function name	ssp_deframing	
Function arguments	rxframe, adddest, addsrc, type, Rx_data, length	
Function Return	Void	
Function description	It's responsible for deconstructing the received frame into its subfields and checking if the received data is correct or not by calculating the CRC to the received data and comparing it with the sent CRC.	
Description of each argument		
Name	**Usage**	**Type**
rxframe	Input buffer to store the received frame in it.	uint8
adddest	Output variable of the destination address to store the destination in it after the deconstruction of the frame.	uint8
addsrc	Output variable of the source address to store the source in it after the deconstruction of the frame.	uint8
type	Output variable of the type of the frame to store the type in it after the deconstruction of the frame.	uint8
Rx_data	Input variable of the type of the frame that was passed out from the control layer to be appended in the fourth byte of the frame.	uint8
length	output variable of the size of the data to be calculated after returning the replacement that was done in it.	uint16

Flags for the function

rxflag: Flag for the function to know if it's ready to receive a frame or not.

crcflag: Flag of the CRC to know if the received frame is the correct frame or not.
deframeflag: Flag for the de-framing layer to check if it's EMPTY and ready to be executed (receive a frame) or FULL and busy.

4.3.4.3.3 Control

It's responsible for tracking the frame and acknowledgment.

It has 3 main states idle, TX, and RX it enters any state of these states based on the existence of the data to be transmitted or the received frame.

TX mode → It's responsible to take the fields of the frame from the application layer to send it to the framing layer to be transmitted and wait for a response to check first, if the destination received was the source of the control layer to know if it was for the communication board or to another subsystem.

Second, if it's in the right destination it will check if the received response was an ACK, then the frame transmitted was correct or NACK, then the frame transmitted was wrong or the communication channels are subject to channel noise or it remained unacknowledged So, to overcome these errors it has to transmit it again until the timer expires after three successive times then its timeout and it will back again to the idle state to be ready to send data again.

RX → It's responsible for the decision to reply with an ACK or NACK to the received frame dependent on the CRC flag that comes from the de-framing layer and also to check first if this frame received destination was its source to know if it was for the communication board or the ground station or to another subsystem to make a right decision to send ACK or NACK.

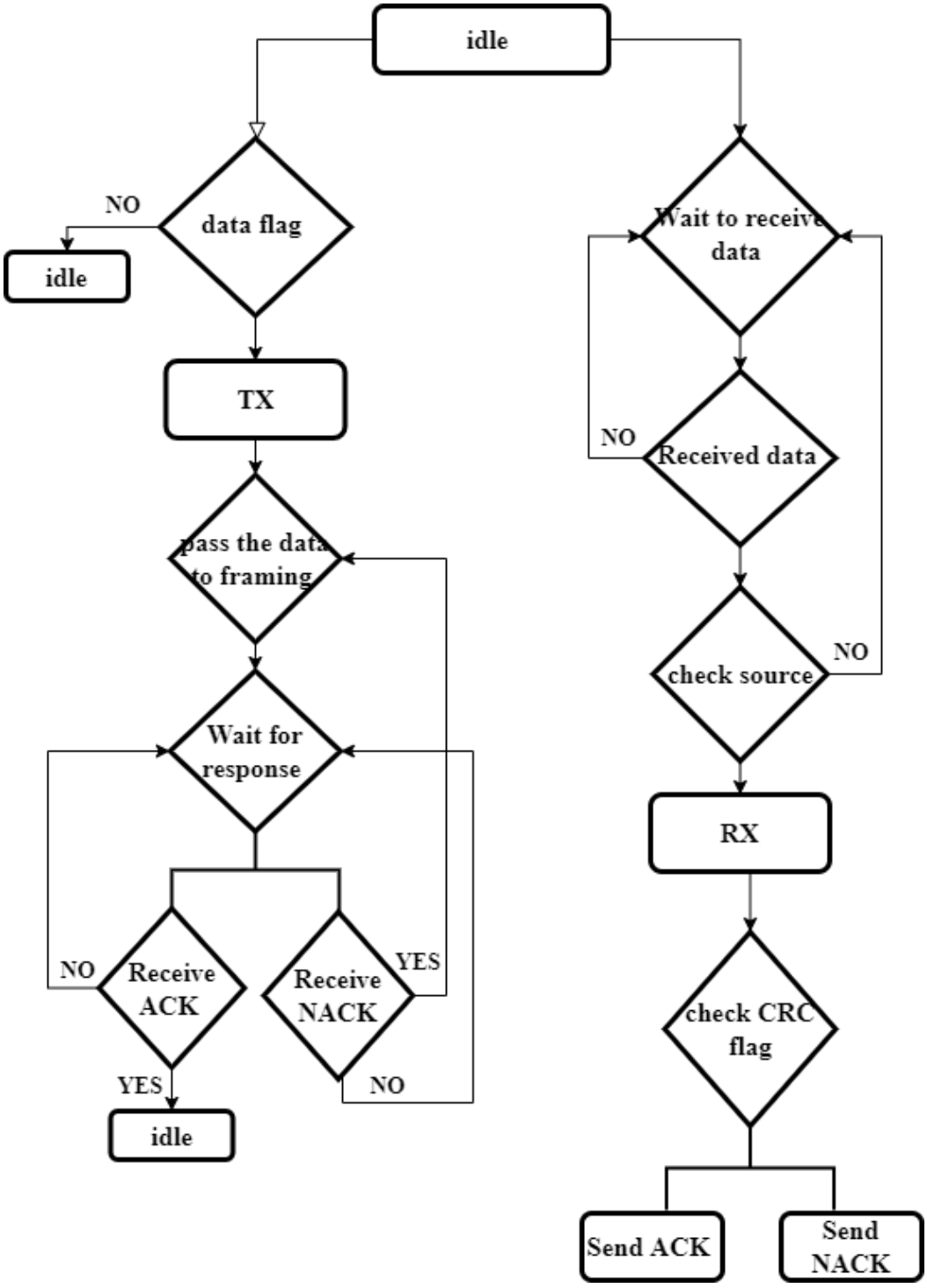

Figure 46 Control Layer Flow Chart

Table 26 SSP control function

Function name	control_layer	
Function arguments	Tx_App_data, data_length, Tx_App_desti, Tx_Frm_srce, Tx_App_type, Tx_Frm_type,Tx_Frm_data, Tx_Frm_desti, Rx_Frm_type,Rx_Frm_data, Rx_Frm_dest, Rx_length, Rx_App_data, tx_size, Rx_Frm_src, checkcontrol, dest_to_framing, src_to_framing, type_to_framing	
Function Return	Void	
Function description	It's responsible for tracking the frame and acknowledgment. This function takes two separate roles the first during transmitting mode, it takes data, source, and type from the application layer to the framing layer and waits for a response from the destination if it's NACK it will resend the same frame until it receives an ACK which refers to a right and complete frame or the counter of NACK reaches its maximum which is three successive wrong frames. The second role is during receiving mode, it takes the received frame as its subfields and checks its destination to know if the frame was sent to the communication board or another subsystem then check the CRC flag to send ACK or NACK according to these checks.	
Description of each argument		
Name	**Usage**	**Type**
Tx_App_data	Input buffer of the data that was passed out from the above layer.	uint8
Tx_App_desti	Input variable of the destination address that was passed out from the application layer.	uint8

Tx_App_type	input variable of the type that was passed out from the application layer.	uint8
data_length	Output variable of the length of the data to be passed out to the framing layer.	uint16
Tx_Frm_srce	Output variable of the source to be passed out to the framing layer.	uint8
Tx_Frm_type	Output variable of the type to be passed out to the framing layer.	uint8
Tx_Frm_data	Output buffer of the data to be passed out to the farming layer.	uint8
Tx_Frm_desti	Output variable of the destination address to store the destination in it to be passed out to the framing layer.	uint8
Rx_Frm_type	Input variable of the received type that was passed out from the de-framing layer.	uint8
Rx_Frm_data	Input buffer of the data received that was passed out from the de-framing layer.	uint8
Rx_Frm_dest	Input variable of the destination received that was passed out from the de-framing layer.	uint8
Rx_Frm_src	Input variable of the source received that was passed out from the de-framing layer.	uint8
Rx_length	Input variable of the size of the received data buffer.	uint16
tx_size	Output variable of the size of the data buffer to be passed out to the framing layer.	uint16
Rx_App_data	Output buffer of the data to store the data in it to be passed out to the next layer.	uint8

dest_to_framing	Output variable of the destination to store the destination in it to be passed out to the next layer.	uint8
src_to_framing	Output variable of the source to store the source in it to be passed out to the next layer.	uint8
type_to_framing	Output variable of the type to store the type in it to be passed out to the next layer.	uint8

Flags of the function

checkcontrol: Flag to check if there is data to be sent or not.

txflag: Flag for the framing layer to check if it's FULL and busy or EMPTY and ready to pass out the fields to it to enter Tx mode.

layerflag: Flag for the application layer to check if it's FULL and busy or EMPTY and ready to be executed to send data to it.

crcflag: Flag of the CRC to know if the received frame is the correct frame to send an ACK or wrong to send NACK.

deframeflag: Flag for the de-framing layer to check if it's FULL to enter the RX mode or not.

4.3.4.3.4 Compute CRC

This function takes (Destination, source, type, and data) to compute CRC as the formula described above.

Table 27 SSP Compute CRC function

Function Name	compute_crc16
Function arguments	*data_p, length
Function Return	uint16
Function Location	Framing Layer, De-framing layer
Function Description	Computes the CRC16-CCITT
Description of each argument	

Name	Usage	Type
data_p	Points to data that we calculate CRC from	uint8
length	The length of the output buffer	uint16

4.4 Integration

4.4.1 Introduction

After implementing each protocol separately SSP and AX-25 we will integrate them together to build a full communication subsystem between the ground and the cube satellite

We began by making a common buffer to transfer the data between the two protocols, for transmission we have the buffer ***SSP_to_Control_Buffer*** and for receiving we have the buffer ***data*** both of which contain AX.25 info field (i.e SSP Frame)

Subsequently we add four functions ***fillBuffer***, ***getdata***, ***ax_ssp_framing***, ***ssp_ax_deframing***, and so these functions now form the **application layer**, as shown in the Figure below.

ax_ssp_framing: This function is responsible for building SSP frame as an information field of AX.25 its similar to ***ssp_build_frame*** function we discussed before but with a little difference which is that this function builds the frame again after deconstructing it in the de-framing layer of SSP to be sent in AX.25 frame not in SSP.

ssp_ax_deframing: This function is responsible for the deconstructing of the SSP frame, its similar to ***ssp_deframing*** function but in this layer, we deconstruct it as an information field of AX.25 not as a SSP frame.

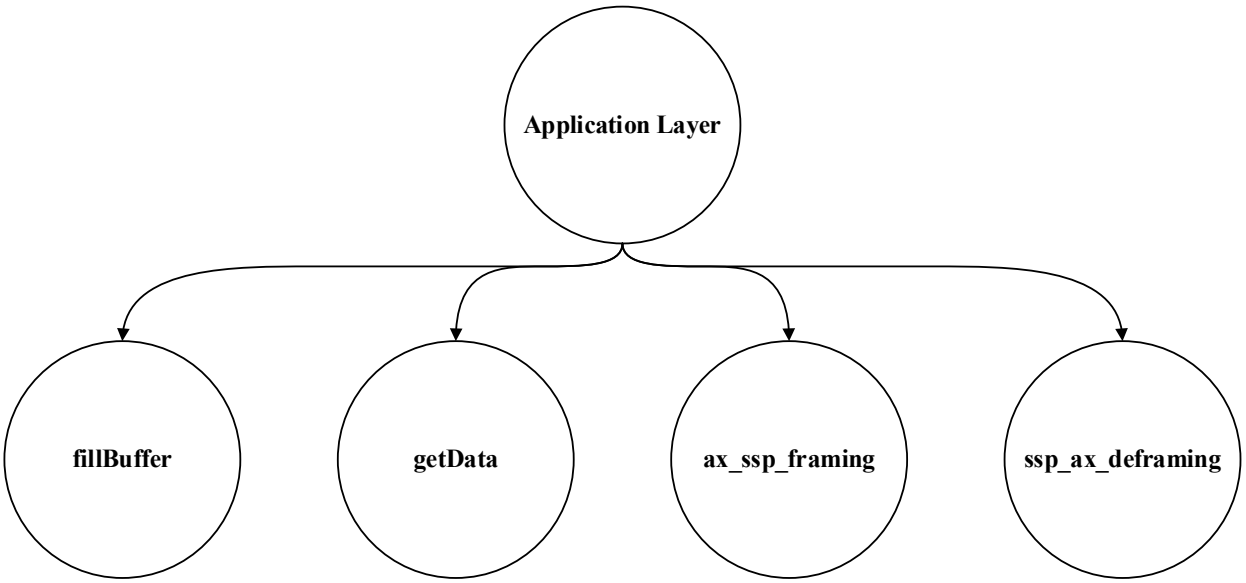

Figure 47 Application Layer Functions

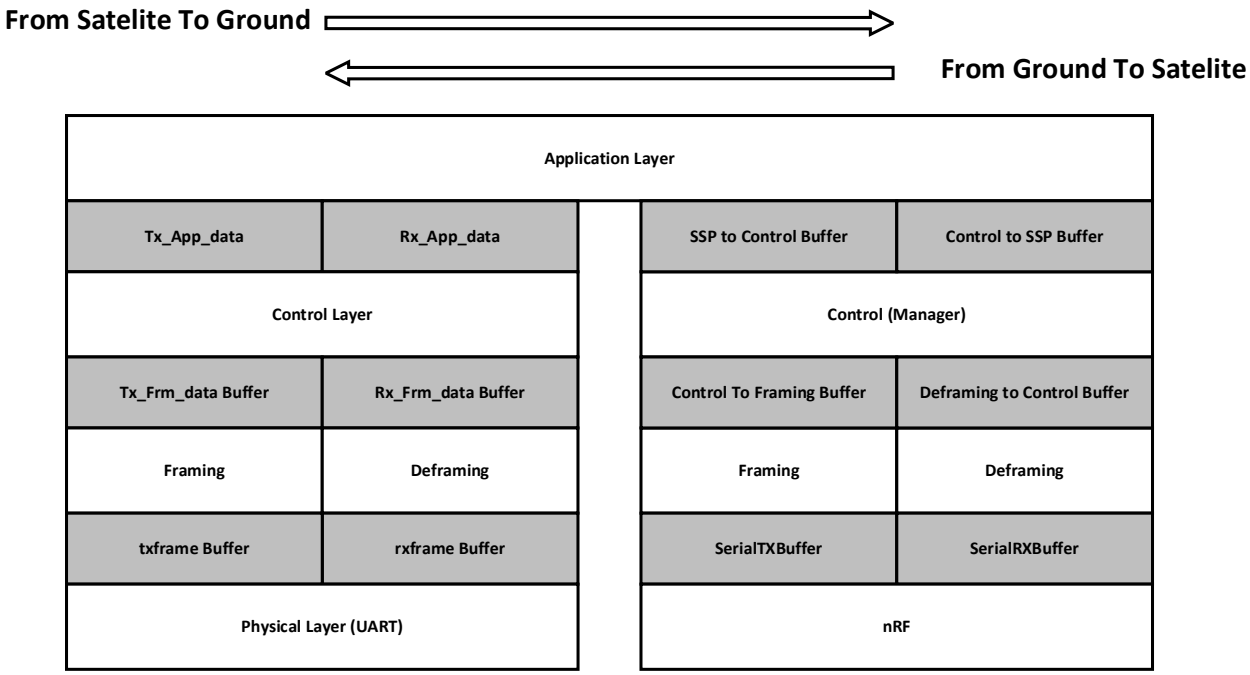

Figure 48 Full integrated protocols stack

4.4.2 Flags

Table 28 Integration Flags

Flag	**Check in the main code**	**Type**
checkcontrol	SSP control layer will not be executed until its EMPTY.	uint8
flag_controltossp	Getdata function will not be executed until its FULL.	uint8
txflag	SSP building frame function will not be executed until its EMPTY.	uint8
layerflag	AX.25 fill buffer function will not be executed until its EMPTY.	uint8
rxflag	SSP de-framing function will not be executed until its FULL.	uint8
deframeflag	SSP de-framing function will not be executed until its EMPTY.	uint8
flag_Control_to_Framing	Indicates whether the Control to Framing Buffer is full or not, the AX manager function won't execute unless it's empty and the Build Frame won't execute unless it's full	uint8
flag_SerialTXBuffer	Indicates whether the SerialTXBuffer Buffer is full or not, the Build Frame won't execute unless it's empty and the PrintSerialTXBuffer won't execute unless it's full	uint8
flag_SerialRXBuffer	Indicates whether the SerialRXBuffer Buffer is full or not, the readFrameFromSerial won't execute unless it's empty and the AX25 Deframe won't execute unless it's full	uint8
flag_Deframing_to_Control	Indicates whether the Deframing_to_Control Buffer is full or not, the AX25 Deframe won't execute unless it's empty and the AX25 Manager won't execute unless it's full	uint8

4.5 Overall System Flow Chart

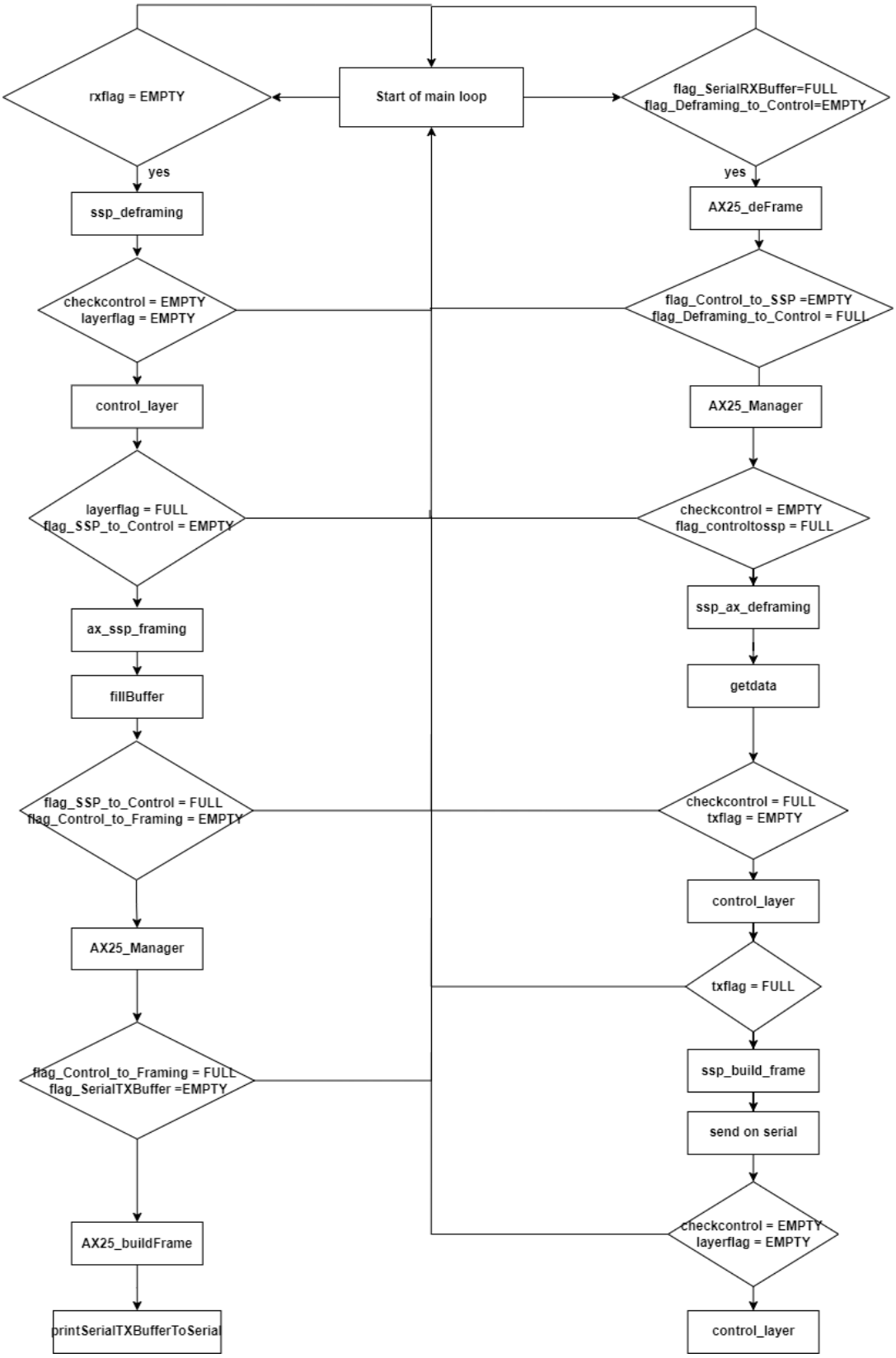

Figure 49 Overall System Flow Chart

4.6 Protocol Functions' Flow Chart

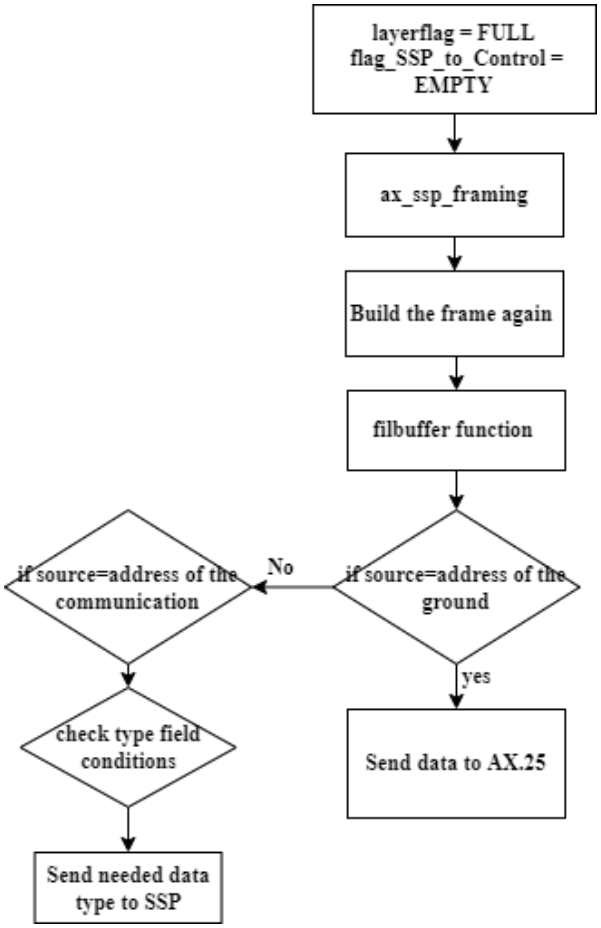

Figure 50 fillBuffer Function Flowchart

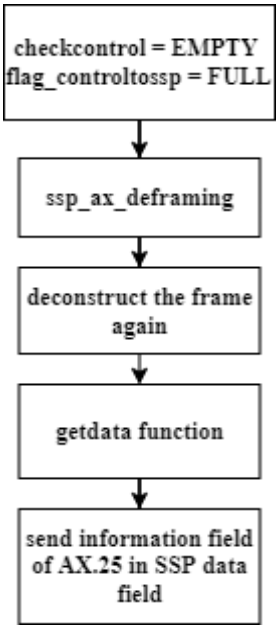

Figure 51 getData Function Flowchart

4.6.1 Functions

Table 29 fillBuffer Function

Function Name	fillBuffer(uint16 *tx_ax_length, uint8 *layerflag, uint8 dest_to_framing, uint8 type_to_framing, uint8 *dataflag, uint8 *data, uint16 *data_length, uint8 *checkcontrol, uint8 *Tx_App_desti, uint8 *Tx_App_type, uint8 *src_to_framing)	
Function Return	void	
Function Location	Application Layer	
Function Description	Fill information field in AX.25 or SSP according to the received frame type	
Description for each argument		
Name	**Usage**	**Type**
tx_ax_length	Input variable to store the full built frame size in it.	uint16
layerflag	Flag to check if this function is FULL and busy or EMPTY and ready to be executed.	uint8
dest_to_framing	Input variable of destination address that was passed out from the above layer be checked with the source of the function.	uint8
type_to_framing	Input variable of the type of the frame that was passed out from the above layer to be checked.	uint8
dataflag	Flag to check if its FULL and there is a data to be sent.	uint8
data	Output buffer of the SSP data field to fill it with the data needed according to the type we receive.	uint8
data_length	Output variable of the size of the frame to be sent.	uint16

checkcontrol	Flag to check if the control layer is FULL and busy or EMPTY and ready to be executed	uint8
Tx_App_desti	Output variable of the destination to store the destination in it to be passed out to the control layer.	uint8
Tx_App_type	Output variable of the type to store the type in it to be passed out to the control layer.	uint8
src_to_framing	Input variable of the source of the received frame to respond to this source with the data needed.	uint8

Table 30 getdata Function

Function Name	getdata(uint8 *data, uint16 *data_length, uint8 *dataflag, uint8 ax_type, uint8 ax_src, uint8 *Tx_App_type, uint8 *Tx_App_desti)	
Function Return	void	
Function Location	Application Layer	
Function Description	Fill data field in SSP.	
Description for each argument		
Name	**Usage**	**Type**
data	Output buffer of the SSP data field to fill it with the data received from AX.25.	uint16
data_length	Output variable of the size of the frame to be sent.	uint8
dataflag	Flag to check if this function is FULL and there is a data to be sent or not.	uint8
ax_type	Input variable of the type of the frame that was passed out from the above layer.	uint8

ax_src	Input variable of the source of the frame that was passed out from the above layer.	uint8
Tx_App_type	Output variable of the type to store the type in it to be passed out to the control layer.	uint8
Tx_App_desti	Output variable of the destination to store the destination in it to be passed out to the control layer.	uint16

Table 31 PrintSerialTXBuffer Function

Function Name	printSerialTXBufferToSerial()	
Function Return	void	
Function Location	Physical Layer	
Function Description	Responsible for writing the AX.25 Frame to Serial and/or to the nRF module to be transmitted	
Description for each argument		
Name	**Usage**	**Type**
SerialTXBuffer	This contains the full AX.25 Frame	uint8[]
Note: this is not a direct argument to the function but rather a global array.		

Table 32 readFrameFromSerial Function

Function Name	readFrameFromSerial()	
Function Return	void	
Function Location	Physical Layer	
Function Description	Responsible for reading the received frame directly from the nRF module	
Description for each argument		
Name	**Usage**	**Type**
SerialRXBuffer	This stores the AX.25 frame read from the nRF module	uint8[]

Note: this is not a direct argument to the function but rather a global array.

Table 33 receive frame here Function

Function Name	receive_frame_here()	
Function Return	void	
Function Location	Physical Layer	
Function Description	Responsible for writing the data received from Serial to rxframe buffer	
Description for each argument		
Name	**Usage**	**Type**
rxframe	This contains the full SSP Frame	uint8[]
Note: this is not a direct argument to the function but rather a global array.		

5 Chapter 5 Testing

5.1 Introduction

This chapter presents the test methodologies of the communication subsystem and the test applications developed using LABVIEW and how these applications are used to test the software designed. LabView was chosen to develop the test applications for its advantage of being scalable, easier control of the input parameters, parallel processing and GUI development. For testing the satellite communication system, the test methodologies are designed as a test bench with user interaction at both the satellite side node and the ground station node, these tests will evaluate the following:

- Test the communication between the satellite communication side node and ground side node
- Test the communication between the satellite communication subsystem and the OBC
- Performance and functionality of the implementation
- Ensuring the proper program flow.
- Verifying proper decoding and encoding of AX.25 UI frames and SSP frames

This chapter also describes each testing phases that we proceeded in order to design the system and how we implemented them. The system was designed according to the SDLC V-Model where the next phase starts only after completion of the previous phase which implies each development activity a corresponding testing activity.

The results from the simulation are also presented in this chapter which verifies the program flow and the functionality of the system.

5.2 Testing design

Recalling the full system's block diagram but from the perspective of testing, we need to test the communication between the OBC and the satellite communication system (node 1) using the internal SSP protocol for physical communication and then the communication between the ground station using the AX.25 protocol for wireless communication (node 2) , thus we need to breakdown the system into smaller units to be tested ,then integrating them to test communication of the internal modules within the system ,finally testing the overall system.

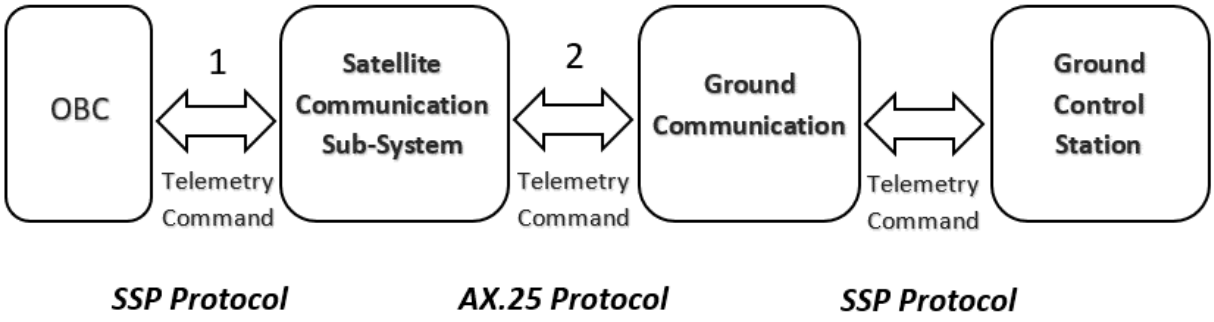

Figure 52 Satellite block diagram

5.2.1 Testing phases

Our approach is designing the system according to the SDLC V-Model as it is a highly-disciplined model that is easy to manage where each phase has a specific deliverable and a review process and for proactive defect tracking whereas bugs are found at early stages, thus enhancing the probability of building an error-free and good quality product.

Under the V-Model, the corresponding testing phase of the development phase is planned in parallel. So, there are Verification phases on one side of the 'V' and Validation phases on the other side. The Coding Phase joins the two sides of the V-Model.

The following illustration depicts the different phases in a V-Model of the SDLC.

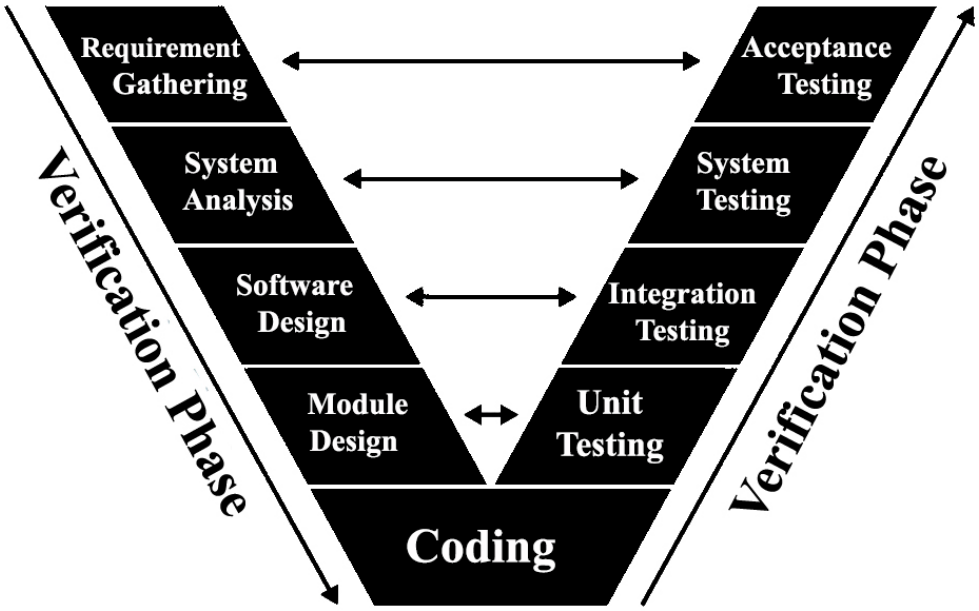

Figure 53 V model

5.2.1.1 Unit testing

Unit Test Plans are executed to eliminate bugs at code or unit level at an early stage before integrating all the units.

5.2.1.2 Integration testing

After completion of unit testing, Integration testing is performed. Integration testing is performed on the architecture design phase. This test verifies the communication of modules among themselves.

5.2.1.3 System testing

System testing is directly associated with the system design phase as it tests the complete application with its functionality, inter dependency, and communication. It tests the functional and non-functional requirements of the developed application.

5.3 Testing implementation

The testing implementation activities are assigned to three main processes in the V-model which are: unit testing, integration testing and system testing as in the

system breakdown structure. For each phase to implement we will need the following:

- LabVIEW interface
- Developing LabVIEW test application which may include one or more sub-VI
- Interactive graphical user interface requiring scalability and easy to navigate through to satisfy the desired requirements.

5.3.1 Integration test implementation

5.3.1.1 SSP testing applications

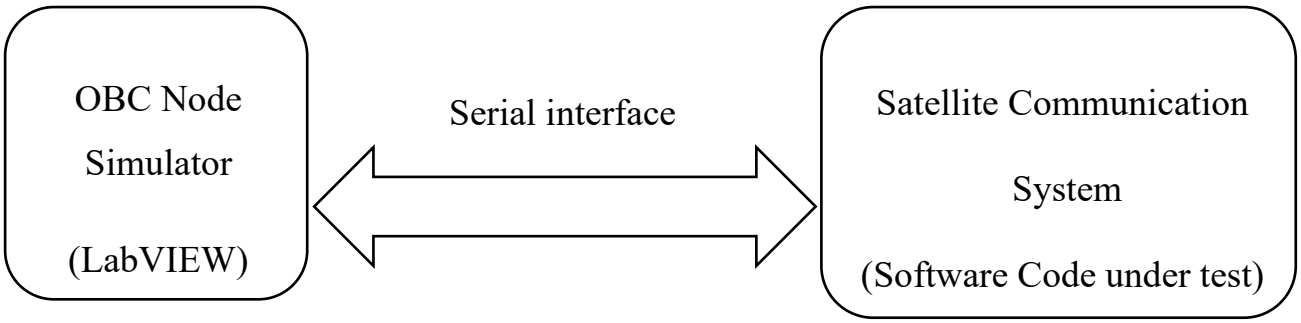

Figure 54 SSP application

This test application communicates with the Arduino over serial interface, its function is to integrate all the unit that tests a particular function inside the protocol such as: (CRC, Frame Addressing, Replacing special characters in the data field and Flags) in one test application. This application is an OBC node simulator where it generates a sequence of SSP frames with different commands to be sent and receives SSP frames back with ACK or NACK.

We have two modes for operations to test:

- **Test Application 1**: The Satellite communication system is in the receiving mode and the LabVIEW is OBC node simulator (transmitter) the block diagram is shown in fig(53)
- **Test Application 2:** The Satellite Communication system is in the transmitting mode and the LabVIEW is OBC node simulator (receiver) the block diagram is shown in fig(54)

we have developed test cases for the expected scenarios and accordingly we can detect which unit generates the error, as in the following table:

Table 34 SSP test cases

Test Type	Case	Expected Outcome
Integration test (Test Application 1)	Testing the receiver's response when sending correct frames	The receiver should send ACK Frame
Integration Test (Test Application 1)	Testing the receiver's response when sending SSP frame with wrong CRC value	1. The receiver should Send NACK Frame 2. The Transmitter should send the same frame again 3 times then discard it and send a new frame 3. The receiver then should send ACK Frame
Integration test (Test Application 2)	Testing the transmitter's response when sending ACK Frames	The transmitter should send (N) new SSP frames
Integration test (Test Application 2)	Testing the transmitter's response when sending 3 successive NACK Frames	The Transmitter should send the same frame again 3 times then discard it and send a new frame

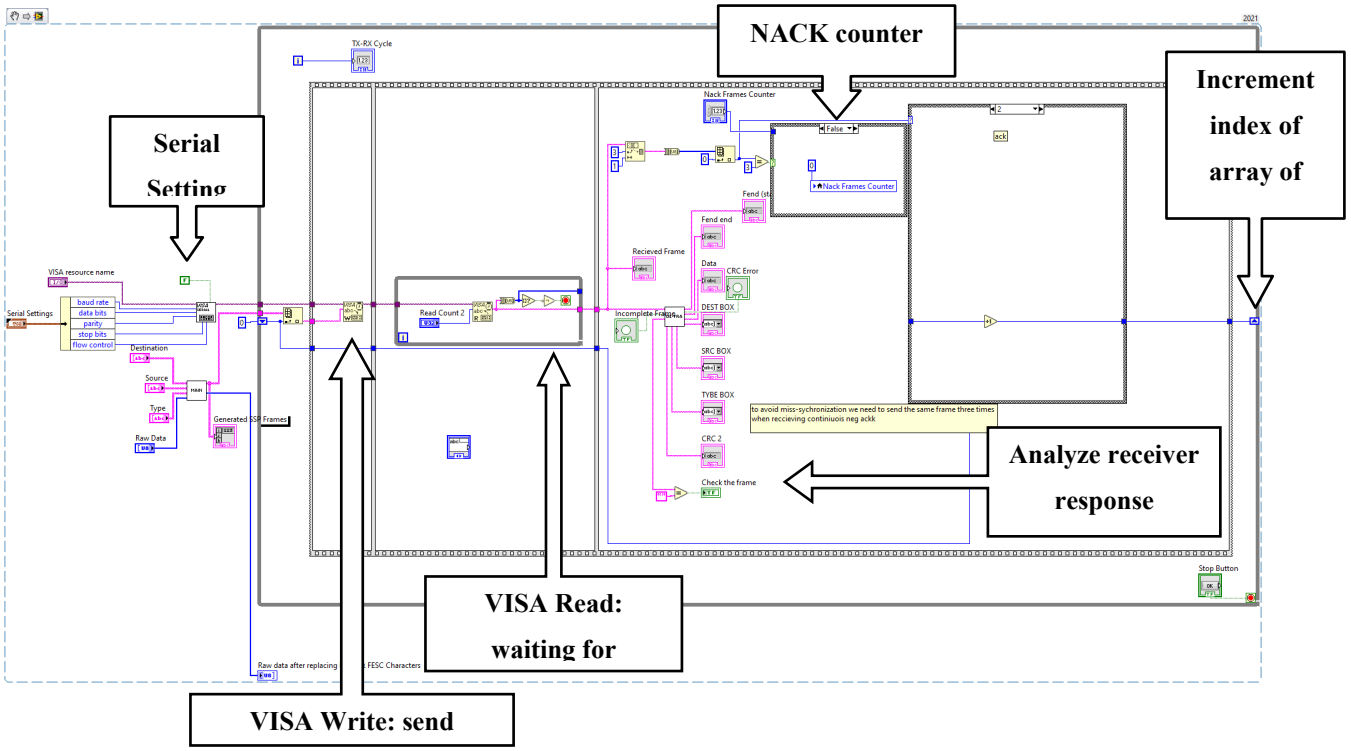

Figure 55 SSP test application 1 (receiving mode) block diagram

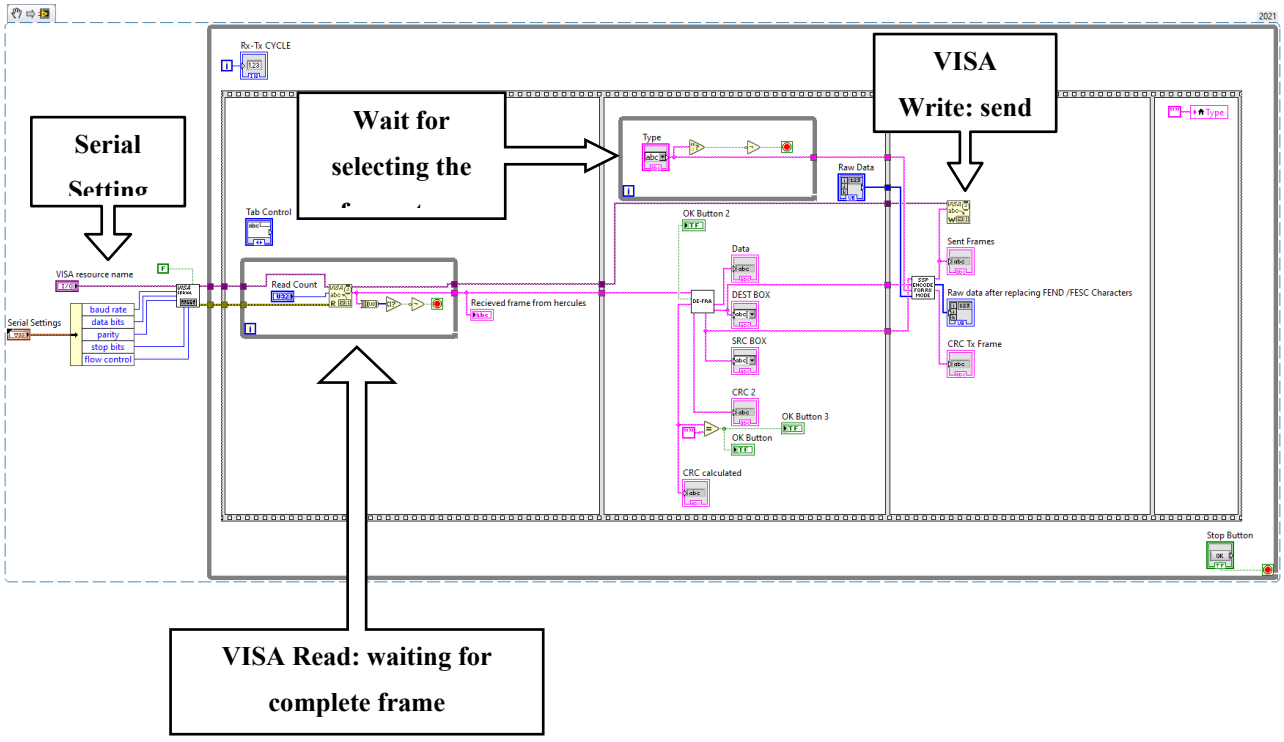

Figure 56 SSP test application 1 (transmitting mode) block diagram

AX.25 Testing applications

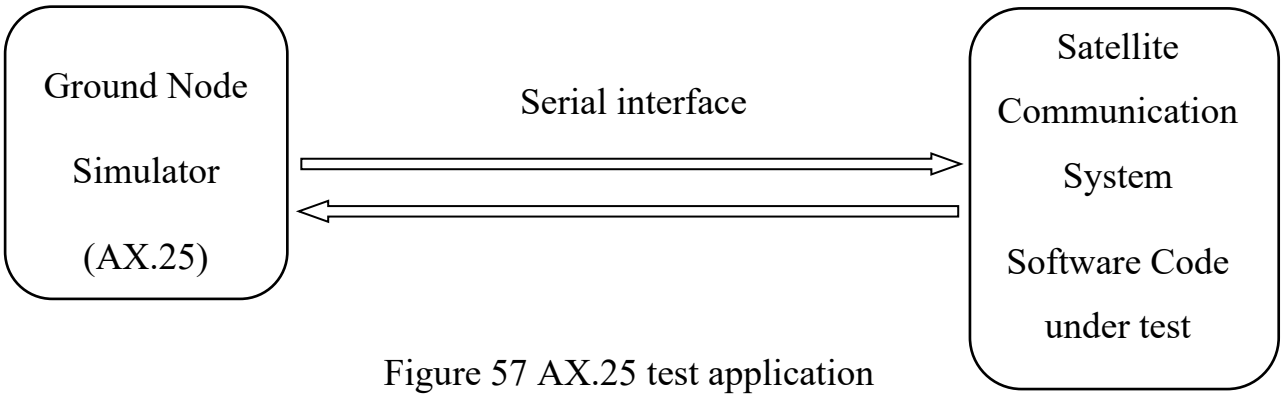

Figure 57 AX.25 test application

This testing applications communicates with the Arduino over serial interface, its function is to integrate all the unit that tests a particular function inside the protocol such as: (CRC, Frame Addressing, padding, frame sequence numbering N(S) and Flags) in one test application. This application is a ground node simulator where it generates a sequence of AX.25frames with different commands to be sent and receives AX.25 frames back with REJ or RR.

We have two modes for operations to test:

- **Test Application 3**: The Satellite communication system is in the receiving mode and the LabView is ground node simulator (transmitter) the block diagram is shown in fig(56)
- **Test Application 4:** The Satellite Communication system is in the transmitting mode and the LabView is ground node simulator (receiver) the block diagram is shown in fig(57)

we have developed test cases for the expected scenarios and accordingly we can detect which unit generates the error, as in the following table:

Table 35 AX.25 test cases

Test Type	Case	Expected Outcome
Integration test (Test Application 3)	Testing the receiver's response when sending (N) AX.25 frames with correct Sequence	The receiver should send (N) AX.25 Frames of type RR (Receiver Ready) and increment the N(s) bits each time
Integration Test (Test Application 3)	Testing the receiver's response when sending (N) AX.25 frames containing 1 frame with wrong Sequence	1. The receiver should send AX.25 frame of type REJ (Reject Frame) and not increment the N(S) bits 2. The Transmitter should send the same frame again 3 times then discard it and send a new frame 3. The receiver should send ACK
Integration test (Test Application 4)	Testing the transmitter's response when sending (N) RR Frames	The transmitter should send (N) new AX.25 frames and increment the n(s) bits each frame
Integration test (Test Application 4)	Testing the transmitter's response when sending 3 successive REJ Frames	The transmitter should send the same frame again 3 times then discard it and send a new frame

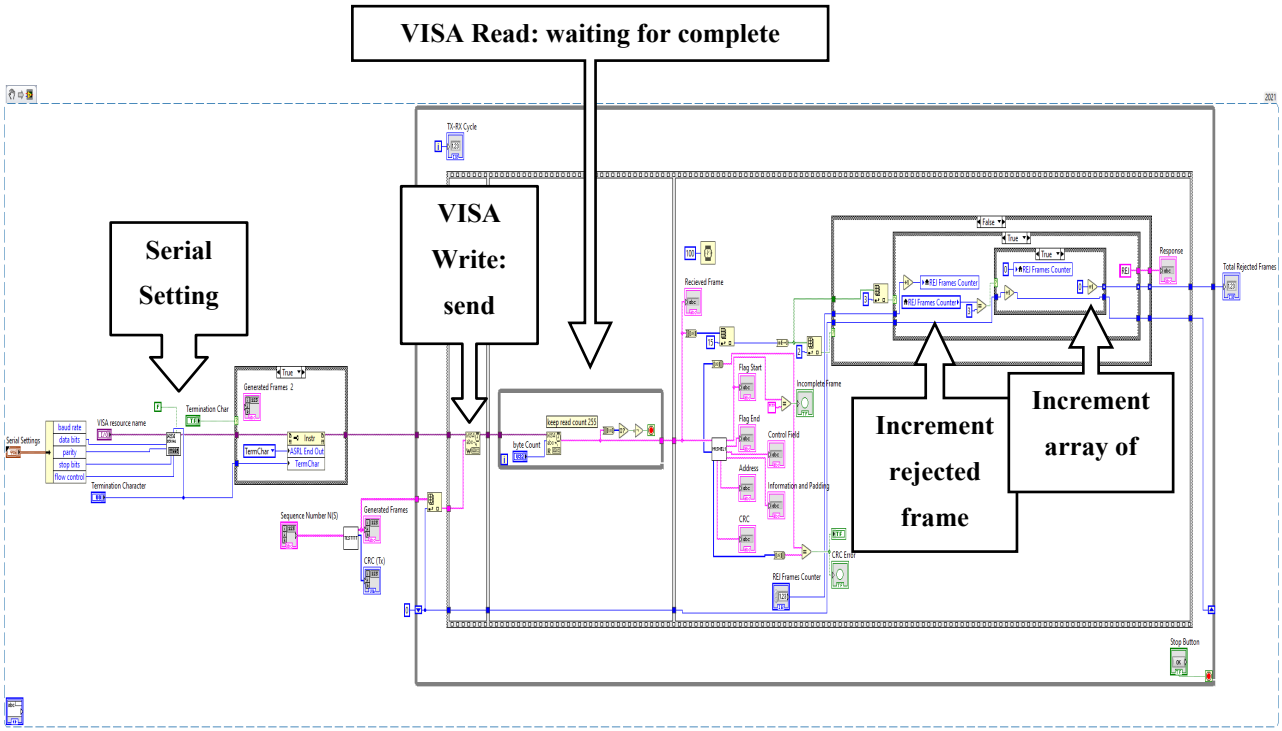

Figure 58 AX.25 Test application 3 (receiving mode) block diagram

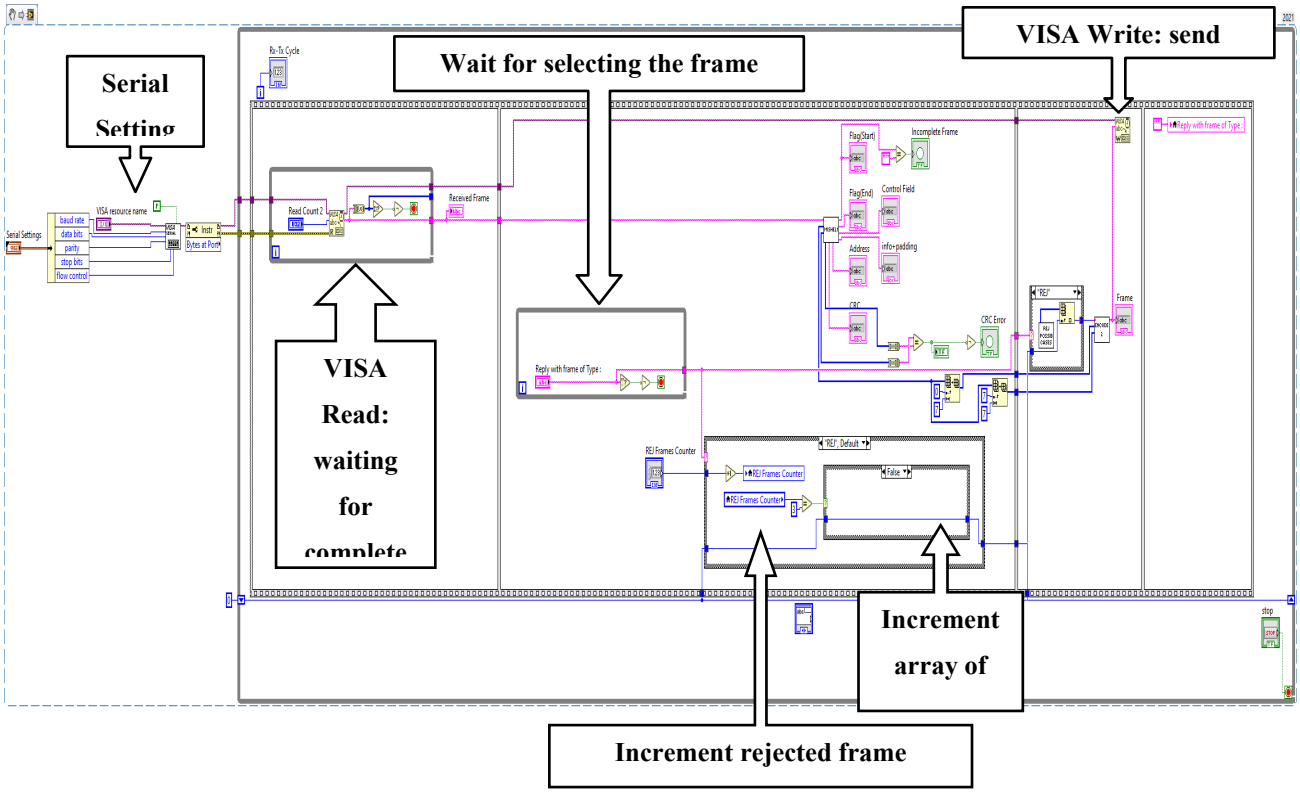

Figure 59 AX.25 Test application 4 (transmitting mode) block diagram

5.3.2 System test implementation

The test applications we developed on LabView consists of two applications one for the satellite side node, and the other for the ground station node. Together they form a framework for testing and evaluating the communication protocol between the communication sub-system and a satellite ground station as well as providing a graphical user interface.

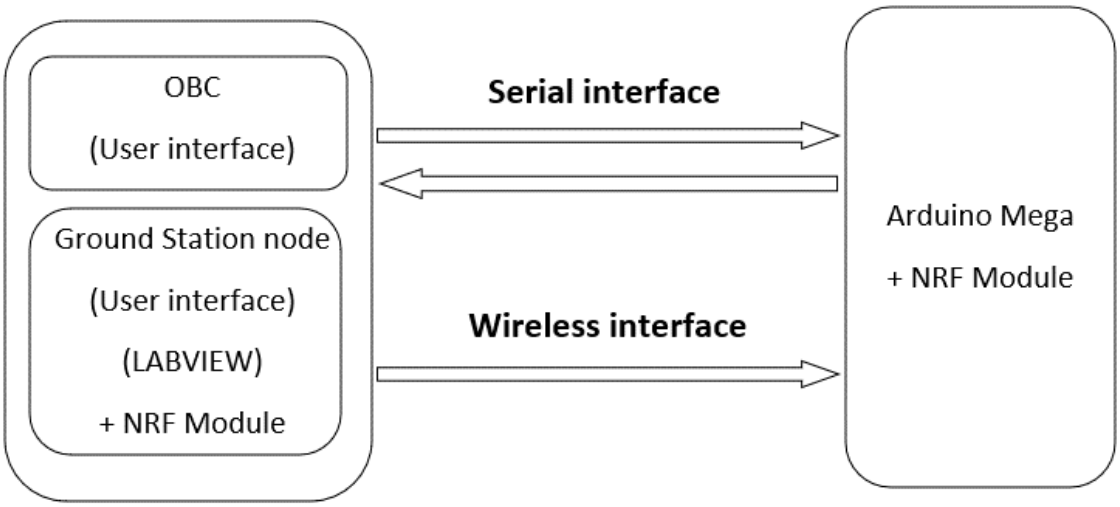

Figure 60 system test block diagram

5.3.2.1 The satellite communication side test application 5

This test application communicates with the Arduino MEGA over serial interface, its function is to test the satellite communication system side node by sending SSP frames from the OBC node simulator to the communication system and the success criteria is when the communication system sends the same SSP frame encapsulated inside an AX.25 frame.

Table 36 communication system test case

Test Type	Case	Expected Outcome
System test (Test Application 5)	Testing the receiver's response when sending SSP frame	The receiver should send the same SSP frame encapsulated in AX.25 frame

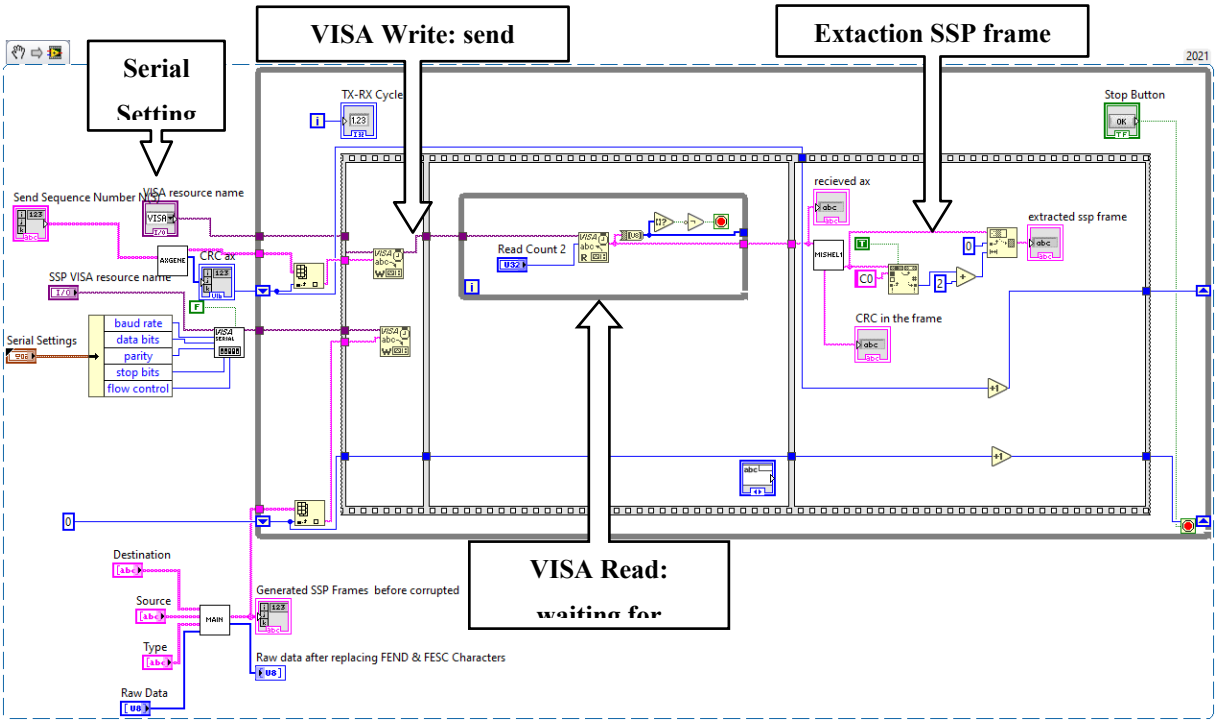

Figure 61 Satellite Communication system's test application LabVIEW block diagram

5.3.2.2 The Ground Control Station (GCS) side test application 6

This test application communicates with the Arduino over serial interface, its function is to emulate the ground station side node by sending and receiving AX.25 frames, where SSP frames are encapsulated inside them.

Table 37 GCS simulation test cases

Test Type	Case	Expected Outcome
System test (Test Application 6)	Testing the receiver's response when sending (N) AX.25 frames with correct Sequence	The receiver should send (N) AX.25 Frames of type RR (Receiver Ready) and increment the N(s) bits each time
system Test (Test Application 6)	Testing the receiver's response when sending	1. The receiver should send AX.25 frame of type REJ (Reject Frame) and not

	(N) AX.25 frames containing 1 frame with wrong Sequence	increment the N(S) bits 2. The Transmitter should send the same frame again 3 times then discard it and send a new frame 3. The receiver should send ACK
System test (Test Application 6)	Testing the transmitter's response when sending (N) RR Frames	The transmitter should send (N) new AX.25 frames and increment the n(s) bits each frame
System test (Test Application 6)	Testing the transmitter's response when sending 3 successive REJ Frames	The transmitter should send the same frame again 3 times then discard it and send a new frame
System test (Test Application 6)	Testing communications system response when ground station send I frame	The communication system should send ACK Frame
System test (Test Application 6)	Communication system sends I frames to ground	1. The ground sends ACK Frame 2. The communication system should send a new I frame 3. The ground Sends NACK frame 4. The communication system shouldn't send new I frames

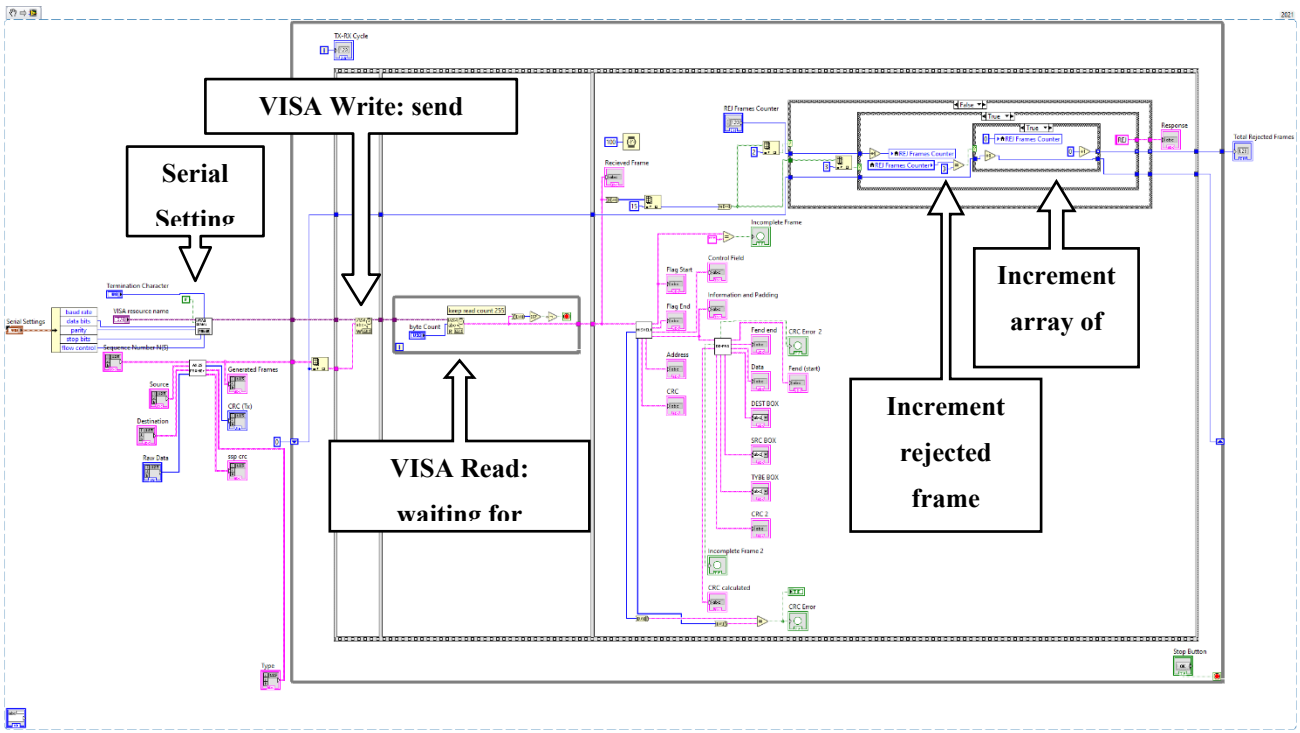

Figure 62 Ground Station Test Application LabVIEW block diagram

5.3.3 ARDUINO MEGA & NRF MODULES

It was necessary to have a wireless interface for the LabView in order to test the AX.25 protocol, it was possible to use the National Instruments (NI) I2C/SPI interface device to support sending and receiving over NRF modules but it was not available. Alternatively, we used an additional Arduino mega with the second Arduino mega that is programmed on it the desired code for testing and they both communicate wireless over NRF modules.

5.4 Results

By utilizing the test methodologies with the test applications, it has been verified that the main program for the communication sub-systems performs according to the specified program flow.

5.4.1 INTEGRATION TESTS RESULTS

5.4.1.1 AX.25 Testing results

The following figure represents the front panel view, this test verifies that the communication system transmits ax.25 frames whenever it receives RR (receiver ready) frame from the lab view application and that the transmitted frames are correctly encoded.

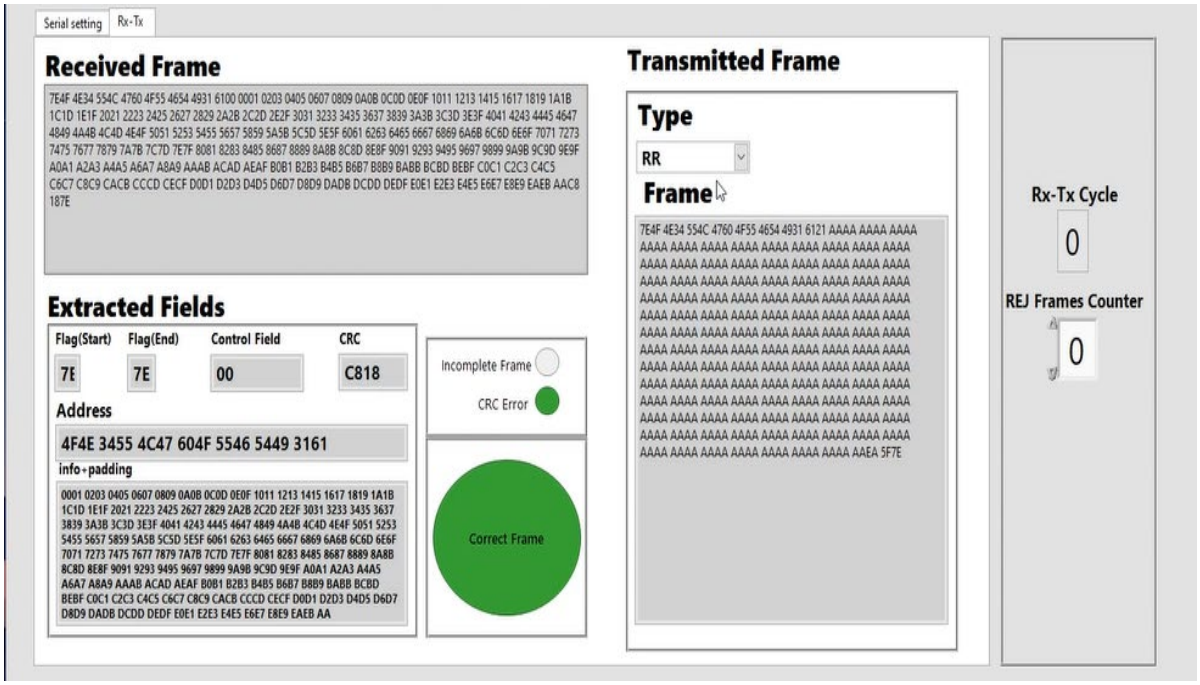

Figure 63 Front panel view from testing the communication system in the transmitting mode

Figure 64 Front panel view from testing the communication system in the receiving mode

The test in figure 62 verifies that the communication system sends the same frame again 3 times then discard it and send a new frame whenever it receives a REJ type frame.

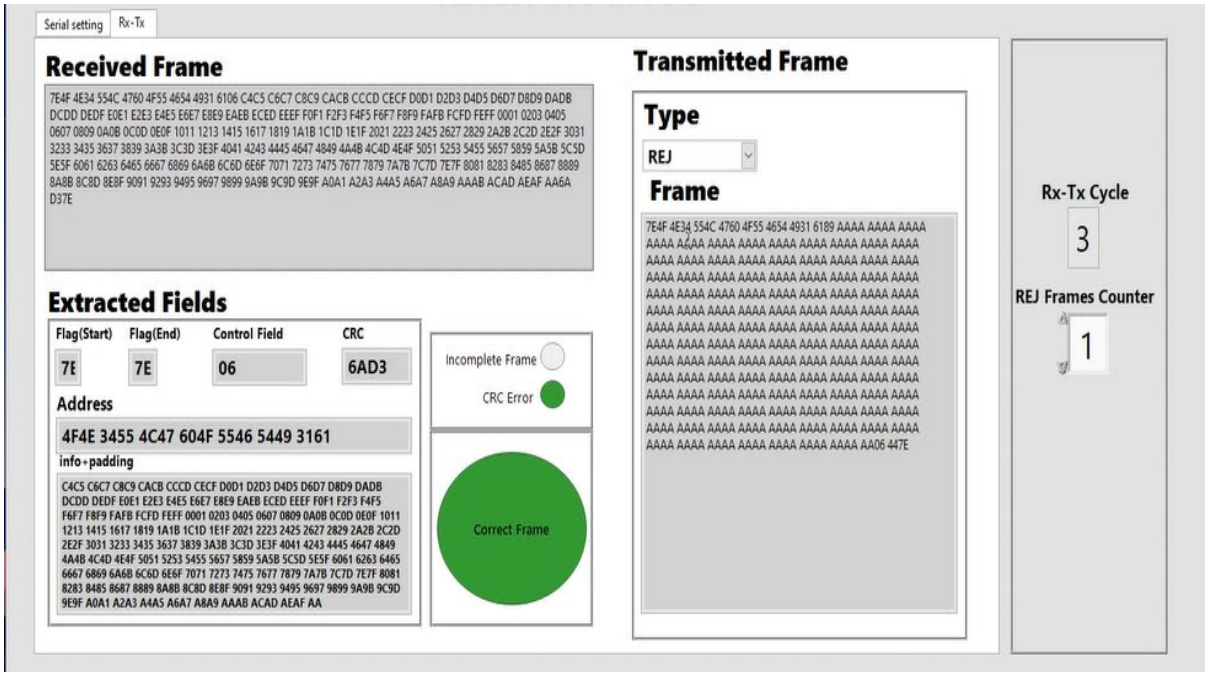

figure 65 Front panel view from testing the communication system in the transmitting mode (frame rejection case)

The test in figure 63 verifies that the communication system sends RR Frames when it receives AX.25 frames and increment the N(s) bits each frame.

The test in figure 64 has verifies that the communication system sends AX.25 frame of type REJ (Reject Frame) it doesn't increment the N(S) bits.

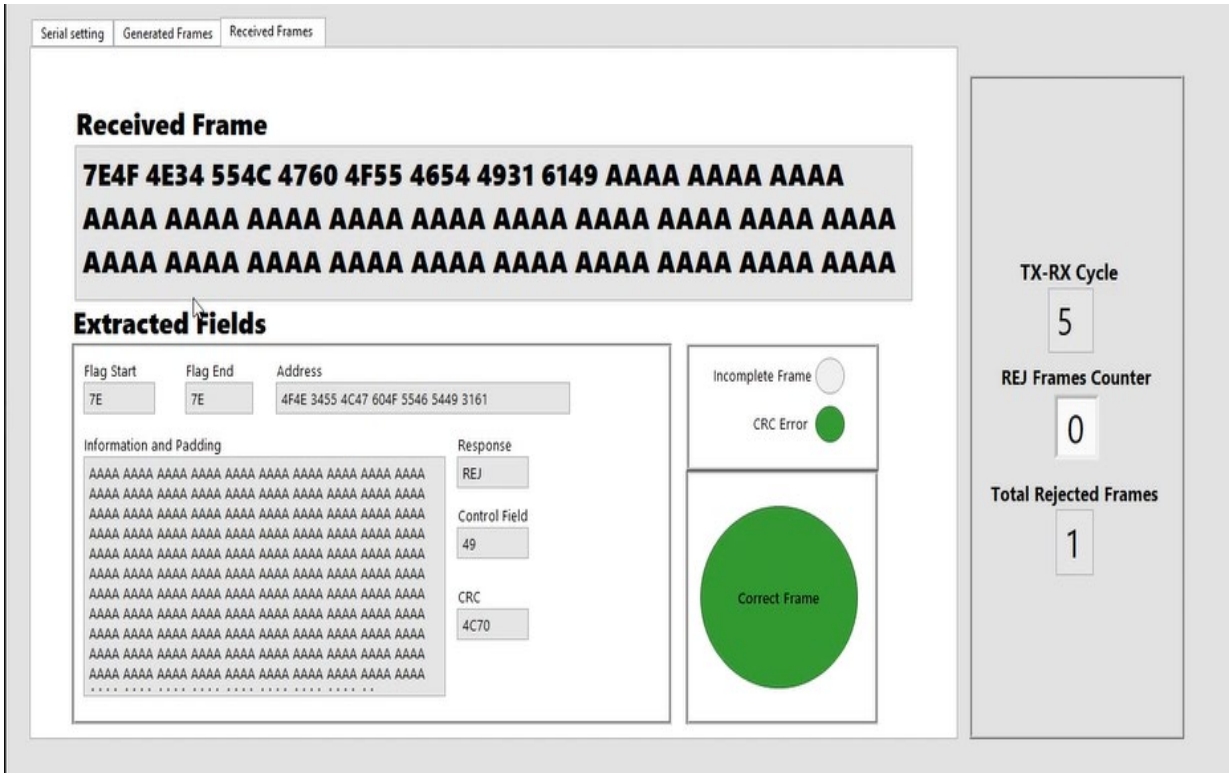

figure 66 front panel view from testing the communication system in the receiving mode (frame rejection case)

After we reached the previous results and comparing them to the expected results, we could verify that the communication system sends and receives correct AX.25 frames, we started to proceed in the system tests designed for the ground station.

5.4.1.2 SSP Testing results

The test in figure 65 represents the front panel view, this test verifies that the communication system sends new SSP frames when sending ACK Frames

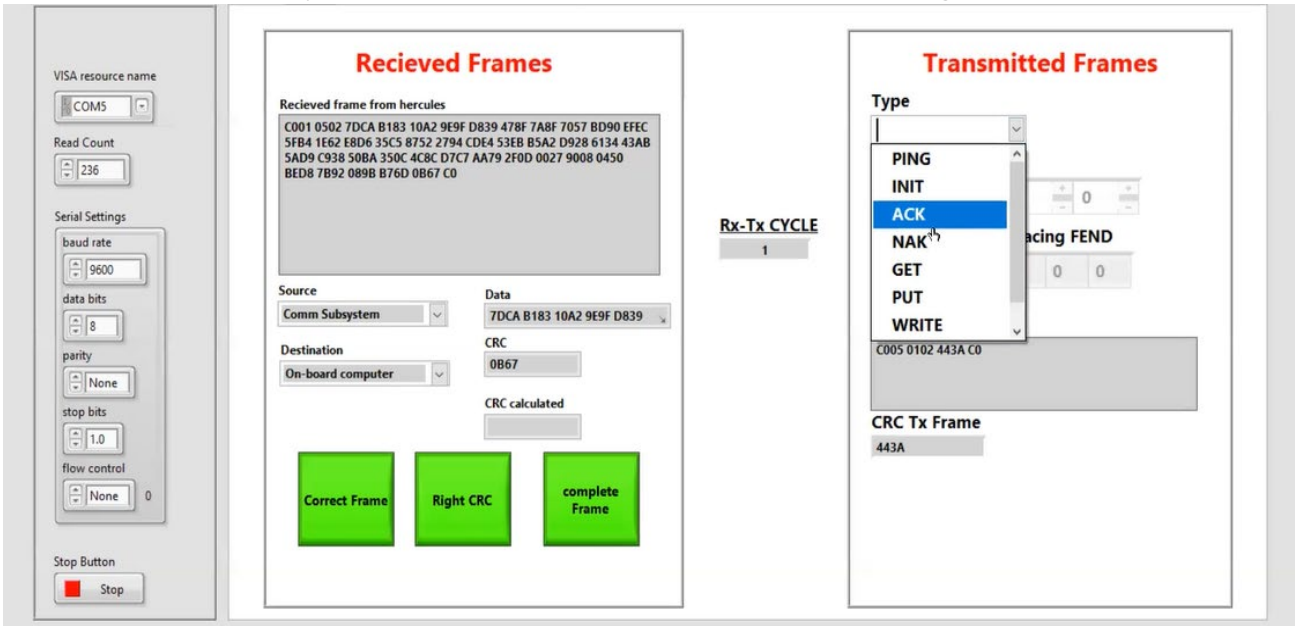

figure 67 Front panel view from testing the communication system in the transmitting mode

figure 68 Front panel view from testing the communication system in the transmitting mode (NACK case)

The test represented in figure 66 represents the front panel view of the LABVIEW and the serial monitor, this test verifies that the communication system sends the same frame again 3 times then discard it and send a new frame when the LabVIEW sends 3 successive NACK frames.

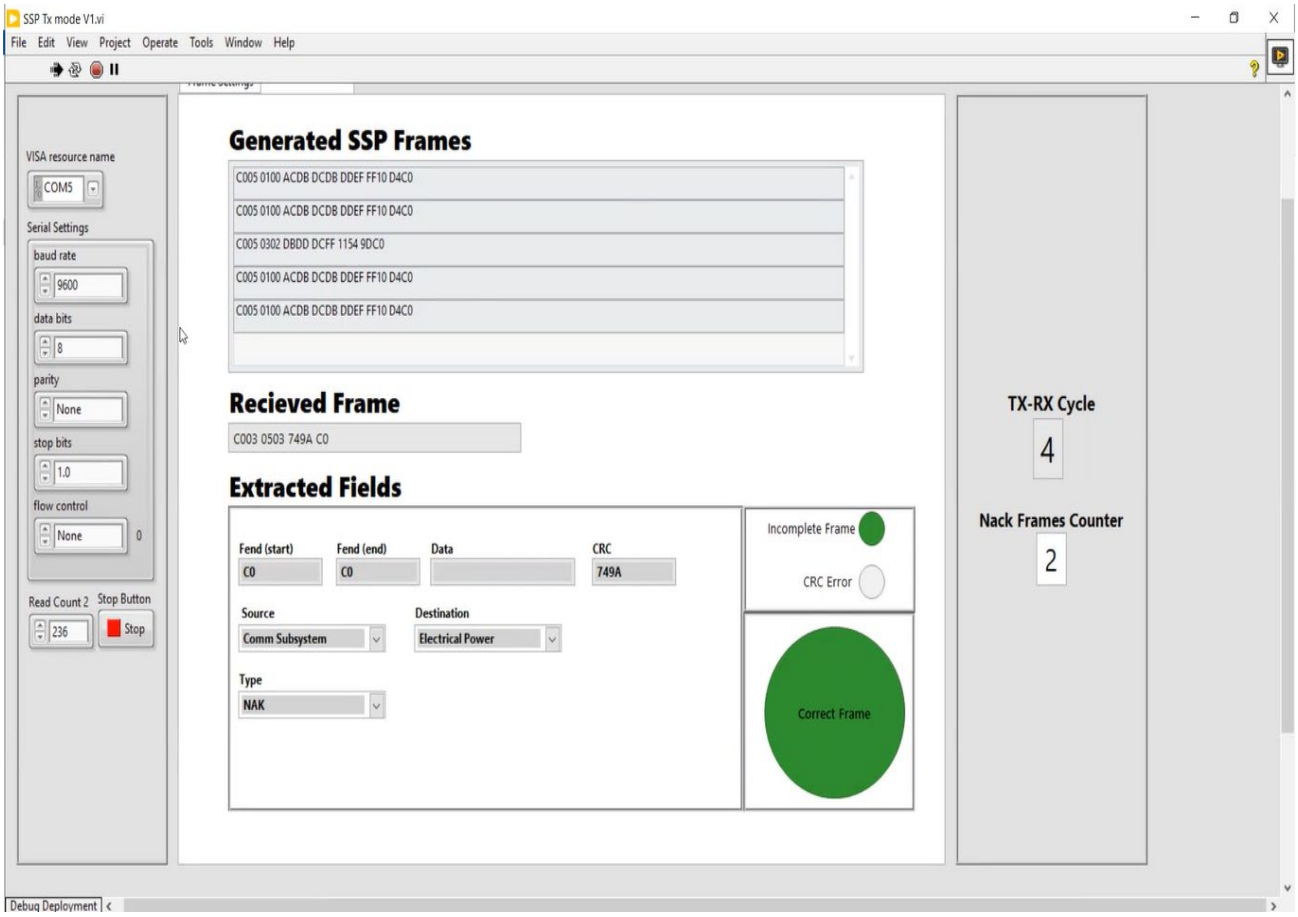

figure 69 Front panel view from testing the communication system in the receiving mode (NACK case)

The test represented in figure 67 represents the front panel view of the LABVIEW this test verifies that the communication system sends NACK frames when sending from LABVIEW SSP frame with wrong CRC value.

After we reached the previous results and comparing them to the expected results, we could verify that the communication system sends and receives correct SSP frames, we started to proceed in the system tests.

5.4.2 System test results

The test represented in figure 68 represents the front panel view of the communicaton system test application front panel , this test verifies that the comunication system communicates with the OBC by receiving SSP frames and encapsulates them inside AX.25.

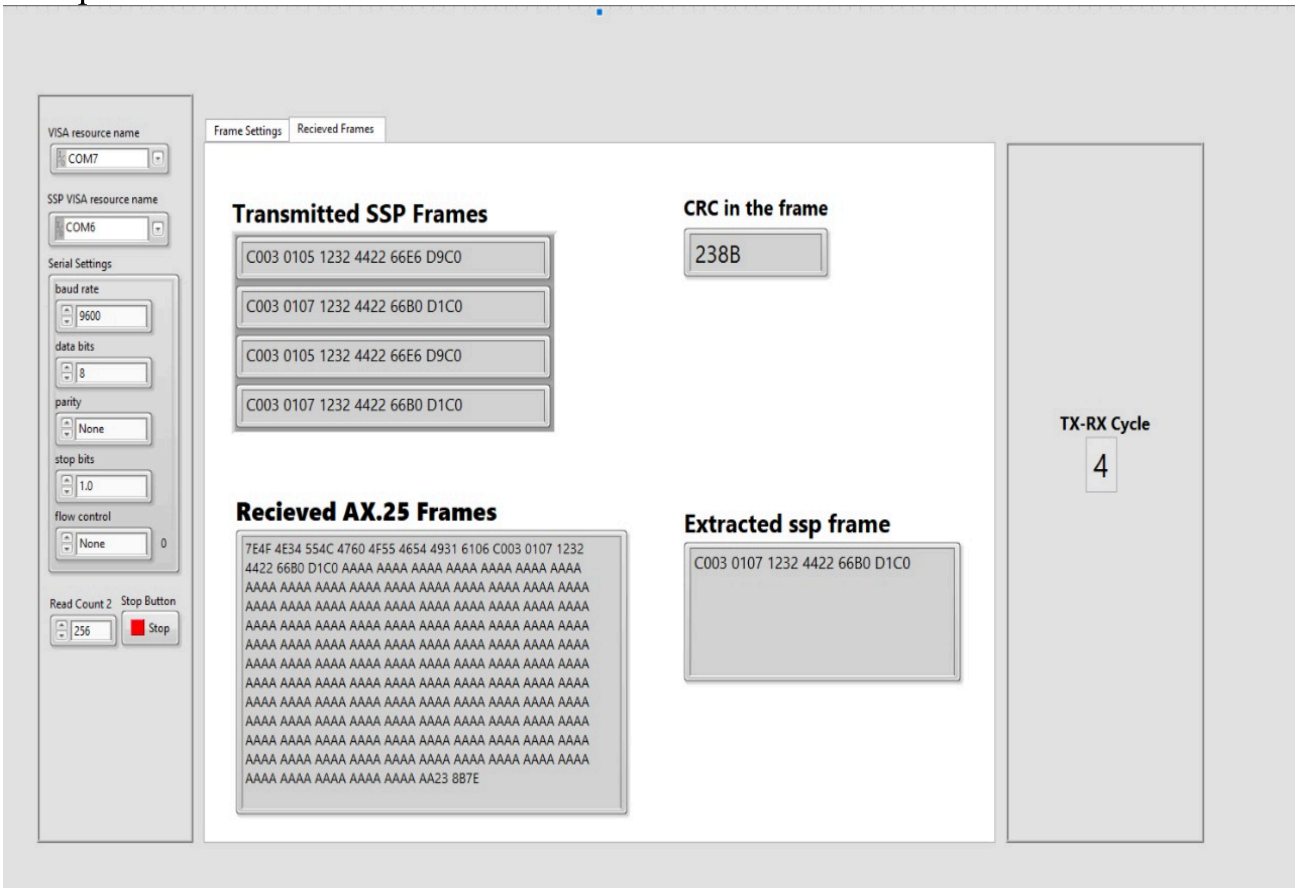

Figure 70 Satellite Communication system’s test application front panel

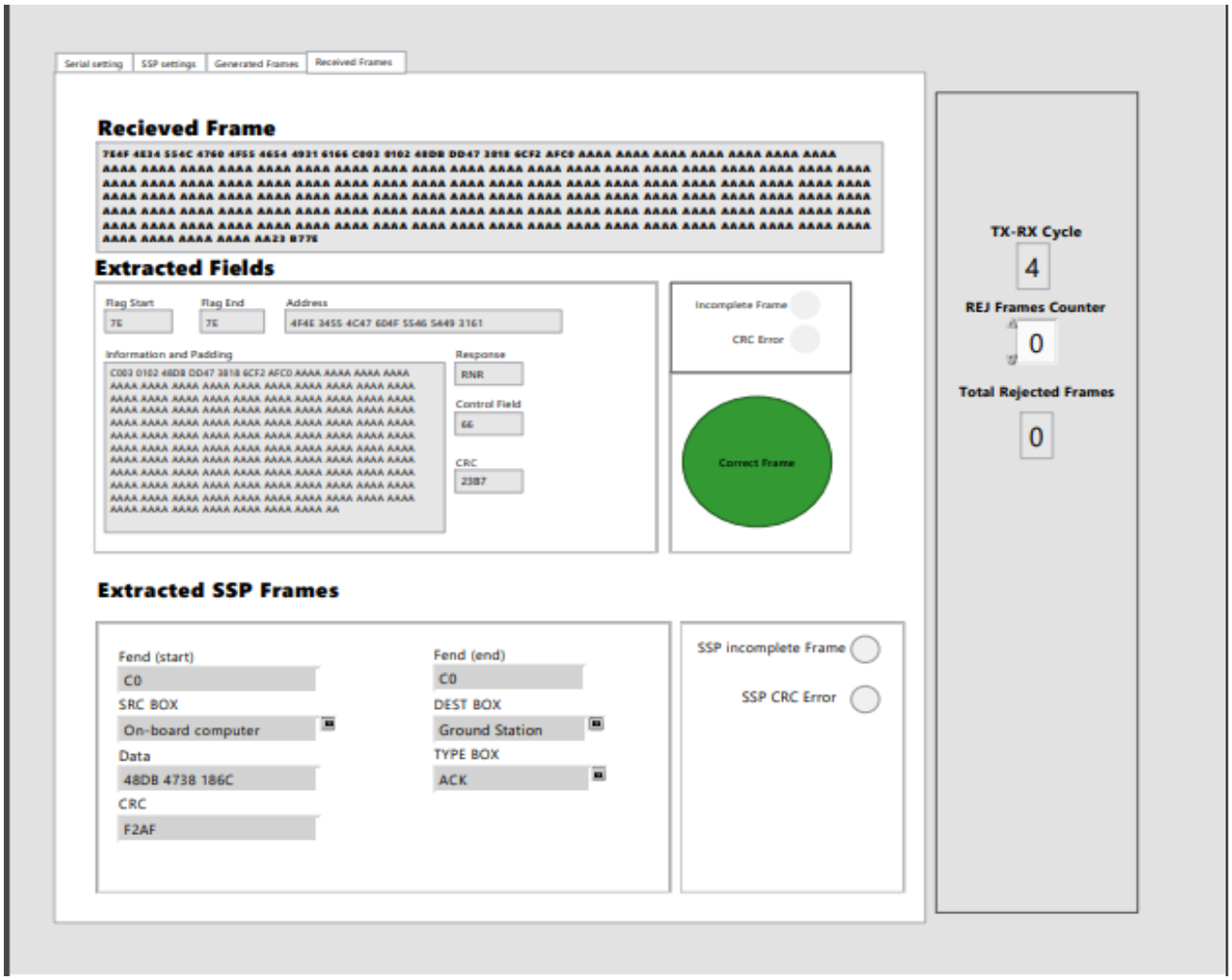

Figure 71 Ground Station Test Application front view panel

Finally, after integrating all the units we tested the overall system, the test in figure 69 verifies that the communication system is able to communicate with the ground station by sending SSP frames encapsulated inside AX.25 frames and vice versa.

6 Chapter 6 Conclusion

In this thesis we have described the process of designing, implementing and testing of a communication system (Software and Hardware) for CubeSat, we have built the communication protocols using the C Programming Language and tested the software implementation using LabVIEW as a GUI, through the 3 testing levels of SDLC-V Model (Unit, Integration and System Testing).

finally, we implemented the complete software on custom built hardware (4-Layer PCB), we have created software that is able to encode and decode frames for the data link layer frames for the communication protocols, the performance and constraints of this implementation have been shown with link budgets and practical tests of the communication protocol.

Since this thesis is from the first few in the satellite communication subsystem projects, we believe that this documentation is important for the future generations who will take on this project, they can continue their work from where we stopped and they can use the test applications we built in order to test the protocols.

Our software implementation has many separated layers which allows for modification of each layer on its own without affecting the other layers as much, also the hardware can be changed and modified without having to change too much in terms of source code.

7 References

2021. [online] Available at: <https://www.researchgate.net/publication/314203397_Design_and_implementation_of_communication_subsystem_for_ISRASAT1_Cube_Satellite> [Accessed 26 December 2021].

2021. Lecture Notes for Satellite, Microprocessor Course and EGSA Summer Training Material.

2021. *Satellite Communications Systems Engineering Atmospheric Effects, Satellite Link Design and System Performance*. [online] Available at: <https://pce-fet.com/common/library/books/31/711_%5BLouis_J._Ippolito_Jr.%5D_Satellite_Communications_S(b-ok.org).pdf> [Accessed 26 December 2021].

2021. *What is a CubeSat*. [online] Available at: <https://www.asc-csa.gc.ca/eng/satellites/cubesat/what-is-a-cubesat.asp>.

Grønstad, M., 2021. *Implementation of a communication protocol for CubeSTAR*. 1st ed. UNIVERSITY OF OSLO.

Hillman Curtis: China PCB Manufacturer and Assembly. 2021. *All about Printed circuit boards Manufacturing*. [online] Available at: <https://hillmancurtis.com/printed-circuit-boards-manufacturing/>.

Inpe.br. 2021. [online] Available at: <http://www.inpe.br/nordeste/conasat/arquivos/projetos/CP2/CP2-COM-Communication_Subsystem.pdf>.

Mouser.com. 2021. *ATmega Datasheet*. [online] Available at: <https://www.mouser.com/pdfdocs/gravitech_atmega328_datasheet.pdf> [Accessed 26 December 2021].

Noe, C., 2004. *Design and Implementation of the Communications Subsystem for the Cal Poly CP2 Cubesat Project*. 1st ed. San Luis Obispo: California Polytechnic State University.

Sparkfun.com. 2021. [online] Available at: <https://www.sparkfun.com/datasheets/Components/SMD/nRF24L01Pluss_Preliminary_Product_Specification_v1_0.pdf> [Accessed 26 December 2021].

SSP Documentation from the Egyptian Space Agency, 2021. SSP Documentation from the Egyptian Space Agency.

Tapr.org. 2021. *AX.25 Documentation*. [online] Available at: <https://www.tapr.org/pdf/AX25.2.2.pdf> [Accessed 26 December 2021].

Ti.com. 2021. *CC1101 Datasheet*. [online] Available at: <https://www.ti.com/lit/ds/symlink/cc1101.pdf> [Accessed 26 December 2021].

User, S., 2021. *Communication subsystem*. [online] Cubesat.org.ua. Available at: <http://www.cubesat.org.ua/en/nano-satellites/polyitan-1/com>.

Web.mst.edu. 2021. [online] Available at: <https://web.mst.edu/~kosbar/Previous_Student_Papers/SamplePaper30.pdf>.

8 Appendix A Link Budget

- Orbit type: a near-circular Sun-synchronous
- Satellite altitude is from 600 km to 700 km
- Uplink frequency band 430-440 MHz
- Downlink frequency band 430-440 MHz
- The modulation type shall be GMSK
- Telemetry data volume about 1200bps
- Command data volume about 1200bps
- shall provide reception of the data by the data reception station with a bit error rate of no more than 1·10-4
- GCS Noise Température, T = 203 °K
- Boltzmann's constant k = 228.6 dBW/K.Hz
- Total Lousses = 198.38 dB
- GCS G/T = 17.5 dB
- SAT G/T = 19 dB
- GCS Antenna Gain = 30 dB
- Sat Antenna Gain = 0.09 dB
- Link budget safety margin Δ = 3 dB

Uplink Budget Calculation

$$\boldsymbol{GCS\ Transmitted\ P_t = \frac{P_c}{N_o} + L_{total} - \mathrm{K} - \mathrm{SAT}\ Rx\frac{G}{T} -}$$

$$\boldsymbol{GCS\ Transmitting\ Antenna\ Gain(G_{OBE})}$$

$$\boldsymbol{\frac{P_c}{N_o} = \frac{E_b}{N_o} + 10log(Dr) + \Delta}\ \textbf{, from GMSK BER Curve,}\ \boldsymbol{\frac{E_b}{N_o}\ is\ approx.}\ 11\ \mathrm{dB}$$

$$\boldsymbol{\frac{P_c}{N_o} = 11 + 10log(1200) + 3}\ \textbf{, => 44.8 dB}$$

$$\boldsymbol{GCS\ Transmitted\ P_r = 44.8 + 198.38 - 228.6 - 19 - 30,\ => -34.42\ dB}$$

Downlink Budget Calculation

$$\textbf{Satellite Transmitted } P_t = \frac{P_c}{N_o} + L_{total} - \text{K} - \text{GCS } Rx\frac{G}{T} -$$

$$\textbf{Satellite Transmitting Antenna Gain}(G_{OBE})$$

$$\frac{P_c}{N_o} = 11 + 10log(1200) + 3 \text{ , => } \textit{44.8 dB}$$

Satellite Transmitted P_t= 44.8 + ***198.38*** – 228.6 – 17.5-0.09,

=> -3.01 dB

Pt in Absolute value is **500 m W,** as mentioned in the EUTS Specs

9 Appendix B Software Code

```
/*-------------------------------------------------------------------------*

 * AX25_CRC.h

 *-------------------------------------------------------------------------*/

#ifndef AX25_CRC_H

#define AX25_CRC_H

#include "std_types.h"


uint16 computeCRC(uint8 *, uint16 *);

void AX25_putCRC(uint8 *, uint16 *);


#endif /* AX25_CRC_H */
```

```
/*-------------------------------------------------------------------------*
 * AX25_CRC.c
 * FCS operations of the AX.25 frame.
 *-------------------------------------------------------------------------*/

#include "AX25_CRC.h"

/*-------------------------------------------------------------------------*
 * AX.25 FCS calculation (bitwise method).
 * CRC-16-CCITT G(x) = x16 + x12 + x5 + 1.
 * Polynom = 0x1021.
 *
 * PARAMETERS:
 * *buffer       pointer of the frame buffer.
 * *OpArrSize    it stores the index of the last inserted element in the array to keep track of size
 *
 * RETURN:
 * the CRC with a final XORed operation.
 *-------------------------------------------------------------------------*/

uint16 computeCRC(uint8 *data_p, uint16 *length) {
        unsigned char x;
```

```
unsigned short crc = 0xFFFF;
uint16 len;
len = *length;
uint8 data_copy;
while (len--) {
        data_copy = *data_p;
        /*  reverse the bits in each 8-bit byte going in */
        data_copy = (data_copy & 0x55555555) << 1
                        | (data_copy & 0xAAAAAAAA) >> 1;
        data_copy = (data_copy & 0x33333333) << 2
                        | (data_copy & 0xCCCCCCCC) >> 2;
        data_copy = (data_copy & 0x0F0F0F0F) << 4
                        | (data_copy & 0xF0F0F0F0) >> 4;

        x = crc >> 8 ^ data_copy;
        data_p++;
        x ^= x >> 4;
        crc = (crc << 8) ^ ((unsigned short) (x << 12))
                        ^ ((unsigned short) (x << 5)) ^ ((unsigned short) x);
}
/*reverse the 16-bit CRC*/
crc = (crc & 0x55555555) << 1 | (crc & 0xAAAAAAAA) >> 1;
crc = (crc & 0x33333333) << 2 | (crc & 0xCCCCCCCC) >> 2;
```

```
	crc = (crc & 0x0F0F0F0F) << 4 | (crc & 0xF0F0F0F0) >> 4;
	crc = (crc & 0x00FF00FF) << 8 | (crc & 0xFF00FF00) >> 8;

	return crc;
}
/*---------------------------------------------------------------------*
 * AX.25 FCS positioning.
 * Put the FCS in the right place in the frame. The FCS is sent MSB first
 * so we prepare the 15th bit of the CRC to be sent first.
 *
 * PARAMETERS:
 * *buffer       pointer of the frame buffer.
 * *OpArrSize    it stores the index of the last inserted element in the array to keep track of size
 *---------------------------------------------------------------------*/
void AX25_putCRC(uint8 *buffer, uint16 *OpArrSize) {
	uint16 crc;

	/* FCS calculation. */
	crc = computeCRC(buffer, OpArrSize);

	/* Put the FCS in the right place with the 15th bit to be sent first. */
	buffer[*OpArrSize] = ((crc >> 8) & 0xff);
```

```
        *OpArrSize = *OpArrSize + 1;
        buffer[*OpArrSize] = (crc & 0xff);
        *OpArrSize = *OpArrSize + 1;
}
```

```
/*-------------------------------------------------------------------*
 * AX25.h
 *-------------------------------------------------------------------*/

#ifndef AX25_H_
#define AX25_H_

/*************************************************
 *                 includes                      *
 *************************************************/
#include "std_types.h"

/*************************************************
 *                 definitions                   *
 *************************************************/
#define FLAG_LEN 1

#define AX25_FRAME_MAX_SIZE  256  // Fixed frame size

#define ADDR_LEN 14         // Address length in bytes
#define ADDR_OFFSET 1       // offset where address start (offset = 1, since flag is 1 byte)
```

```
#define CNTRL_OFFSET ADDR_LEN+ADDR_OFFSET

#define CNTRL_LEN 1


#define INFO_OFFSET CNTRL_LEN+CNTRL_OFFSET

#define SSP_FRAME_MAX_SIZE       236  // Max number of bytes for Info field. (256 is the default value).

#define INFO_MAX_LEN  SSP_FRAME_MAX_SIZE+1    // Even in the case of max SSP frame size there will be a byte of padding


#define PADDING_LEN 5

#define PADDING_OFFSET SSP_FRAME_MAX_SIZE + ADDR_LEN + FLAG_LEN + CNTRL_LEN


#define FCS_OFFSET 253



#define FRAME_SIZE_WITHOUT_INFO_AND_PADDING 19


#define FULL 1U

#define EMPTY 0U


#define SIZE_SSP_to_Control_Buffer 229 //236
```

```
/*********************************************
 *                global variables          *
 *********************************************/
typedef enum{
  I,S,U
}frameType;

typedef enum{
  RR, RNR, REJ, SREJ, SABME = 0x6C, DISC = 0x40, DM = 0x04, UA = 0x60,
UI = 0, TEST = 0xE0
}frameSecondaryType;

typedef enum{
  CLEAR,SET
}flagMode;

typedef enum{
  idle, TX, RX, start_TX, start_RX
}controlLayerMode;

typedef enum{
  REJECT, ACCEPT
}Status;
```

```
/*********************************************
 *            Function Prototypes            *
 *********************************************/
void AX25_buildFrame(uint8 *, uint8 *, uint16 * , uint8 *, uint8, uint8);
void AX25_deFrame(uint8*, uint16, uint8);
uint8 AX25_getControl(frameType frameType, frameSecondaryType
secondaryType, uint8 NS, uint8 NR, uint8 pollFinal);
void AX25_Manager(uint8 *);
void AX25_getInfo(uint8 * info);
void fillBuffer(uint16 *tx_ax_length,uint8 *layerflag,uint8 dest_to_framing,uint8
type_to_framing,uint8 *dataflag,uint8 *data,uint16 *data_length,uint8
*checkcontrol,uint8 *Tx_App_desti,uint8 *Tx_App_type,uint8 *src_to_framing);

void AX25_buildFrame_TEST(uint8 *buffer, uint8 *info, uint8 *ADDR,
   uint8 control, uint8 infoSize);

#endif /* AX25_H_ */
```

```
/*

================================================================
==================

 AX25.c

================================================================
================

 */

#include <stdio.h>
#include <stdlib.h>
#include "ax25.h"
#include "AX25_CRC.h"
#include "Arduino.h"

#define DEBUG
uint8 g_pollFinal = 0;
uint8 g_Received_NR = 0;

uint8 g_received_address[ADDR_LEN], Control_To_SSP[236],
                g_control_recived[CNTRL_LEN],
g_padding_recived[PADDING_LEN];
extern uint8 g_info_reciver[SSP_FRAME_MAX_SIZE];
```

```
extern uint8 addr[ADDR_LEN], ax_ssp_frame[dt];

uint8 flag_RX_crc; /* used in the Manager function to check whether CRC is correct or not */
uint8 flag_info = EMPTY;

uint8 flag_Status = ACCEPT; /* used in Manager function to determine if the I-frame we sent was accepted or rejected by the Receiver  */
uint8 rejCounter = 0;

extern uint8 flag_SSP_to_Control;
extern uint8 flag_Control_to_Framing;
extern uint8 flag_Control_to_SSP;
extern uint8 flag_Deframing_to_Control;
extern uint8 flag_controltossp;
extern uint8 info[SSP_FRAME_MAX_SIZE];
extern uint8 SSP_to_Control_Buffer[SIZE_SSP_to_Control_Buffer];

extern uint8 g_infoSize;
extern uint8 flag_SerialTXBuffer;
extern uint8 flag_SerialRXBuffer;
extern uint8 flag_next_frame;
/* -------------------- TX Functions --------------------*/
```

```
/*
 * Description: computes control byte given the following parameters
 * parameters:
 * frameType: give frame type (I, S, U)
 * secondaryType: RR, RNR, REJ, SREJ, SABME, DISC, DM, UA , UI, TEST
 *
 */
uint8 AX25_getControl(frameType frameType, frameSecondaryType secondaryType,
		uint8 NS, uint8 NR, uint8 pollFinal) {
	uint8 control = 0;
	switch (frameType) {
	case I:
		control = (control & 0x1F) | ((NR << 5) & 0xE0); /* insert N(R) into control field */
		control = (control & 0xF1) | ((NS << 1) & 0x0E); /* insert N(S) into control field */
		control = (control & 0xEF) | ((pollFinal << 4) & 0x10); /* insert P into control field */
		control &= ~(1 << 0); /* insert 0 in rightmost bit */
		break;
	case S:
```

```
            control = (control & 0x1F) | ((NR << 5) & 0xE0); /* insert N(R) into
control field */

            control = (control & 0xEF) | ((pollFinal << 4) & 0x10); /* insert P/F
into control field */

            control = (control & 0xFC) | 0x01; /* insert 01 in the two rightmost
bits */

            control = (control & 0xF3) | ((secondaryType << 2) & 0x0C); /* insert
S bits into their place */

            break;

        case U:

            control = (control & 0xEF) | ((pollFinal << 4) & 0x10); /* insert P/F
into control field */

            control = (control & 0xFC) | 0x03; /* insert 11 in the two rightmost
bits */

            control = (control & 0x13) | (secondaryType & 0xEC); /* insert M
bits into their proper location */

            break;

        }

        return control;

}


/*

 * TESTING function

 * Description: function fills in info array
```

```
 * parameters:
 *  *info: pointer to the global info array
 */
void fillBuffer(uint16 *tx_ax_length, uint8 *layerflag, uint8 dest_to_framing,
                uint8 type_to_framing, uint8 *dataflag, uint8 *data,
                uint16 *data_length, uint8 *checkcontrol, uint8 *Tx_App_desti,
                uint8 *Tx_App_type, uint8 *src_to_framing) {
#ifdef SSP_DEBUG
//      Serial1.print("dkhl hena");
#endif
        uint8 i;
        uint8 source1 = 0x05;
        uint8 source2 = 0x03;
        uint8 arr[] = { 0x54, 0x26, 0xc0, 0x34, 0x7 };

#if 0
        static uint8 Data = 0;

        //currently fill buffers this way
        if (flag_Status == REJECT) {
                // stay at same value since we want to send the same data in the frame
        } else {
                //increment to send next frame with different data as normal
```

```
            for (int i = 0; i < size; i++) {
                buffer[i] = Data++;
            }
        }
#endif
        if (dest_to_framing == source2) {
            for (i = 0; i < *tx_ax_length; i++) {
                //Serial1.print("source 03");
                SSP_to_Control_Buffer[i] = ax_ssp_frame[i];

                //      Serial1.print(SSP_to_Control_Buffer[i], HEX);

            }
            //SIZE_SSP_to_Control_Buffer=Rx_length;

            g_infoSize = *tx_ax_length;
            //*ax_ssp_flag=EMPTY;
            *layerflag = EMPTY;
            flag_SSP_to_Control = FULL;
            //flag_next_frame=FULL;
        }
```

```
        if (dest_to_framing == source1&&
*dataflag==EMPTY&&*checkcontrol==EMPTY) {
                Serial1.print("el type");
                Serial1.print(type_to_framing, HEX);
                if (type_to_framing == 0x04) {
                        Serial1.print("\nGET\n");
                        uint8 i;

                        for (i = 0; i < 5; i++) {
                                data[i] = arr[i];
                        }
                        *dataflag = FULL;
                        *data_length = 5;
                        *Tx_App_type = type_to_framing;
                        *Tx_App_desti = *src_to_framing;
                        *checkcontrol = FULL;
                        *layerflag = EMPTY;
//flag_next_frame=FULL;
                } else if (type_to_framing == 0x00) {
                        Serial1.print("\nPING\n");
                        uint8 i;

                        for (i = 0; i < 5; i++) {
```

```
                data[i] = arr[i];
            }
            *dataflag = FULL;
            *data_length = 5;
            *Tx_App_type = type_to_framing;
            *Tx_App_desti = *src_to_framing;
            *checkcontrol = FULL;
            *layerflag = EMPTY;
//flag_next_frame=FULL;
        } else if (type_to_framing == 0x06) {
            Serial1.print("\nREAD\n");
            uint8 i;

            for (i = 0; i < 5; i++) {
                data[i] = arr[i];
            }
            *dataflag = FULL;
            *data_length = 5;
            *Tx_App_type = type_to_framing;
            *Tx_App_desti = *src_to_framing;
            *checkcontrol = FULL;
            *layerflag = EMPTY;
//flag_next_frame=FULL;
```

```
        } else if (type_to_framing == 0x07) {
                Serial1.print("\nWRITE\n");

                uint8 i;

                for (i = 0; i < 5; i++) {
                        data[i] = arr[i];
                }
                *dataflag = FULL;
                *data_length = 5;
                *Tx_App_type = type_to_framing;
                Serial1.print(*Tx_App_type, HEX);
                *Tx_App_desti = *src_to_framing;
                *checkcontrol = FULL;
                *layerflag = EMPTY;
//flag_next_frame=FULL;
        }

        else {
                Serial1.print("\n GWA el else\n");

                *dataflag = EMPTY;
                *checkcontrol = EMPTY;
```

```
                flag_next_frame = EMPTY;
                *layerflag = EMPTY;
                flag_SSP_to_Control = EMPTY;
                flag_controltossp = EMPTY;
            }
        }
}

/*
 * Description: increments the given State Variable (VR or VS) and overflows if reach value of 7.
 * parameters:
 *      *stateVar: pointer to either VR or VS.
 *
 */

void incrementStateVar(uint8 *stateVar) {
        if (*stateVar < 7) {
            *stateVar = *stateVar + 1;
        } else {
            *stateVar = 0;
        }
}
```

```
/*
 * Description: this function acts as Control Layer
 * parameters:
 *
 */

void AX25_Manager(uint8 *a_control) {
        uint8 control;
        uint8 prev_state;
        uint8 i;
//  uint8 SSP_to_Control_Buffer_Copy[SIZE_SSP_to_Control_Buffer];
//  uint8 SSP_to_Control_Buffer[SIZE_SSP_to_Control_Buffer];
        uint8 Deframing_To_Control_Buffer[256];
        uint8 Deframing_To_Control_Buffer_Copy[256];
        static uint8 state = idle;
        uint8 pollfinal = 0;
        static uint8 VS = 0; /* holds the number of the frame to be sent */
        static uint8 VR = 0; /* holds the number of the frame that is expected to be
received */
        uint8 NS;
        uint8 NR;
        uint8 Received_NS;
```

```
uint8 PollFinal;
uint8 Received_PollFinal;
uint8 Sbits;
uint8 Received_Sbits;
uint8 Mbits;
uint8 Received_Mbits;
uint8 received_control;
uint8 Address_Copy[ADDR_LEN];
uint8 myAddress[ADDR_LEN] = { 'O', 'N', '4', 'U', 'L', 'G', 0x60, 'O', 'U',
                'F', 'T', 'I', '1', 0x61 };

uint8 notMyAddress = CLEAR;

//uint8 flag_Status = ACCEPT;
uint8 g_Recieved_NR_1;
static uint8 flag_NS_VR; /* flag to indicate if received NS equals VR */

uint8 flag_RxFrameType; /* used when node acts as RX, indicates type of
received frame (I or S) *//*todo: hana */

/* used to determine timeout for AX Frames */
static uint32 AX_startTimeout;
static uint32 AX_endTimeout;
```

```
        switch (state) {
        case idle:
                /*---------------------- TX part ----------------------*/
                if ((flag_SSP_to_Control == FULL && flag_Control_to_Framing ==
EMPTY)) {
#ifdef AX_DEBUG
                        Serial.print("\nIdle tx part\n");
#endif
                        /*TODO: check from Dr. if we should do this copy or not */

                        for (i = 0; i < SIZE_SSP_to_Control_Buffer; i++) {
                                info[i] = SSP_to_Control_Buffer[i];
                        }

                        flag_SSP_to_Control = EMPTY;
                        flag_Control_to_Framing = FULL;
                        NS = VS;
                        NR = VR;
                        *a_control = AX25_getControl(I, RR, NS, NR, pollfinal);
                        state = TX;
                        AX_startTimeout = millis();
                }
```

```
		/*---------------------- RX part ----------------------*/
		if ((flag_Control_to_SSP == EMPTY &&
flag_Deframing_to_Control == FULL)) {

#ifdef AX_DEBUG
			Serial.print("\nIdle rx part\n");
#endif

			/* Checks if received address is ours */
			for (i = 0; i < ADDR_LEN; i++) {
				if (g_received_address[i] != myAddress[i]) {
					notMyAddress = SET; /* set the flag to show the
address is not ours */
#ifdef AX_DEBUG

					Serial.println("\n Address is NOT ours \n");
#endif
					flag_Deframing_to_Control = EMPTY; /* clears
Buffer in case address is not ours */
					break; /* breaks as soon as it finds a difference in
the address */
				}
			}
```

```
                /* check if type is I-frame */
                received_control = g_control_recived[0]; /* copy the received
cntrol byte */
                if ((received_control & 0x01) == 0) {

                        /* indicate that we received an I-Frame so that after idle
we go to RX state */
                        flag_RxFrameType = I;

                        /* get subfield values from the control byte */
                        g_Received_NR = (received_control & 0xE0) >> 5;
                        Received_NS = (received_control & 0x0E) >> 1;
                        PollFinal = (received_control & 0x10) >> 4;

#ifdef AX_DEBUG
                        Serial.print("\n Received NS = ");
                        Serial.print(Received_NS);
                        Serial.print("\n VR = ");
                        Serial.print(VR);
#endif

                        /* check if Received NS equals V(R) */
```

```
                    if (Received_NS == VR) {
                        flag_NS_VR = SET; /* set flag to indicate that
received NS == VR */

                    } else {

                        /*TODO: check from Dr. make send REJ */
                        flag_NS_VR = CLEAR;
                        state = RX; /* sets the state to RX in order to send
REJ in this case */
                        // flag_Deframing_to_Control = EMPTY; /*
Discards Frame */
                    }
                } else if ((received_control & 0x03) == 0x01) {

                    /*todo: check this entire else if from DR. */
                    /* indicate that we received an S-Frame so that we stay in
idle mode */
                    flag_RxFrameType = S;

                    /* get subfield values from the control byte */
                    g_Received_NR = (received_control & 0xE0) >> 5;
                    Received_PollFinal = (received_control & 0x10) >> 4;
                    Received_Sbits = (received_control & 0x0C) >> 2;
```

```
                        /* reset to 0 when > 7 */
                        g_Recieved_NR_1 = g_Received_NR - 1;
                        if (g_Recieved_NR_1 > 7) {
                                g_Recieved_NR_1 = 7;
                        }

                        /* checks if the received frame has correct order and is of
ACK type (RR or RNR) */
                        if ((g_Recieved_NR_1) == VS
                                        && (Received_Sbits == RR ||
Received_Sbits == RNR)) {
                                //incrementStateVar(&VR);  /*todo:check from dr
*/
                        }
                        flag_Deframing_to_Control = EMPTY;
                }
#ifdef AX_DEBUG

                Serial.print("\n CHECK FLAGS\n");
                Serial.print(notMyAddress);
                Serial.print(flag_NS_VR);
                Serial.print(flag_controltossp);
```

```
                Serial.print("\n CHECK FLAGS\n");
#endif

                if (notMyAddress
                        != SET&& flag_NS_VR == SET&&
flag_controltossp==EMPTY) { /*continues if the address is ours and the number
of frame is what is expected*/
                    /* copy array to upper layer */
                    Serial1.print("\n gwa management layer ezzat\n");

                    // Serial1.print(ax_rx_length,HEX);
                    for (i = 0; i < SSP_FRAME_MAX_SIZE; i++) {
                        Control_To_SSP[i] = g_info_reciver[i];
                        //Serial1.print(Control_To_SSP[i], HEX);

                    }
                    if (flag_RxFrameType == I) {
                        flag_controltossp = FULL; /*todo: check if it
should be done here only or below aswell (before state = idle line) */

                        state = RX; /* sets the state to RX */
                    } else if (flag_RxFrameType == S) {
                        state = idle;
```

```
                        }
                    }
                }
                break;

        case TX:
                if (flag_Deframing_to_Control == FULL) {
#ifdef AX_DEBUG
                    Serial.print("\n TX State \n");
#endif

                    received_control = g_control_recived[0];

                    /* check which type I or S or U */
                    if ((received_control & 0x01) == 0) {

                        /* type is I frame */
#ifdef AX_DEBUG
                        Serial.print("\nI frame\n");
#endif
                        g_Received_NR = (received_control & 0xE0) >> 5;
                        Received_NS = (received_control & 0x0E) >> 1;
```

```
                Received_PollFinal = (received_control & 0x10) >> 4; /*
TODO: check if it should be Poll or PollFinal */

                /*TODO: send info field to upper layer*/

                /*TODO: check from Dr since we already copy I frame
in RX part above by few lines */

                /* copy array to upper layer */

                for (i = 0; i < SSP_FRAME_MAX_SIZE; i++) {

                        Control_To_SSP[i] = g_info_reciver[i];

                }

        } else if ((received_control & 0x03) == 1) {

                /* type is S frame */

#ifdef AX_DEBUG

                Serial.print("\nS frame\n");

#endif

                g_Received_NR = (received_control & 0xE0) >> 5;

                Received_PollFinal = (received_control & 0x10) >> 4;

                Received_Sbits = (received_control & 0x0C) >> 2;

                /* in case the received NR is 0 this means that we ack
frames up to 7 from previous window */
```

```
                    g_Recieved_NR_1 = g_Received_NR - 1;
                    if (g_Recieved_NR_1 > 7) {
                        g_Recieved_NR_1 = 7;
                    }

//#ifdef DEBUG
//      Serial1.print(g_Recieved_NR_1);
//      Serial1.print(VS);
//      Serial1.print(Received_Sbits);
//#endif
                    if ((g_Recieved_NR_1) == VS
                            && (Received_Sbits == RR ||
Received_Sbits == RNR)) { /* check if frame was received properly or not by
other side */
                        flag_Status = ACCEPT; /* this means that the
frame sent was accepted */
                        rejCounter = 0; /* reset REJ counter */

                        /*TODO: check from Dr. */
                        /* original line was @ line 154 */
                        /* moved this here so that a new frame is made
only when an RR is received otherwise if we receive REJ we will send the same
frame */
                        //  flag_SSP_to_Control = EMPTY;
```

```
#ifdef AX_DEBUG
                                    Serial.print("\n The Frame we sent was Accepted
\n");
#endif

                                    /* make values of VS range from 0 --> 7 only */
                                    incrementStateVar(&VS);

                                    state = idle;
                                } else {

                                    flag_Status = REJECT;

                                    if (rejCounter == 2) {
#ifdef AX_DEBUG
                                        Serial.print("\n Reject Counter Reached 3 so
frame is skipped \n");
#endif
                                        flag_Status = ACCEPT; /* this means that
the frame sent was skipped but treat it as accepted*/
                                        flag_SSP_to_Control = EMPTY;
                                        incrementStateVar(&VS);
                                        state = idle;
```

```
                                        rejCounter = 0;
                                    } else {
                                        rejCounter++;

#ifdef AX_DEBUG
                                        Serial.print("\nThe Frame we sent was
Rejected\n");
                                        Serial.print("\n rejCounter: ");
                                        Serial.print(rejCounter);
                                        Serial.print("\n");
#endif
                                    }
                                    state = idle;
                                }

                                flag_next_frame = FULL;
                            } else {

                                /* type is U frame */
#ifdef AX_DEBUG
   Serial.print("\nU frame\n");
#endif
                                Received_Mbits = (received_control & 0xEC) >> 2;
```

```
                Received_PollFinal = (received_control & 0x10) >> 4;
            }
            flag_Deframing_to_Control = EMPTY;
        } else {

            AX_endTimeout = millis();
            if ((AX_endTimeout - AX_startTimeout) > 2000) {

                /* Serial Message */
#ifdef AX_DEBUG
                Serial.print("\nAX Timeout\n");
#endif
                /*clear flags*/
                flag_Deframing_to_Control = EMPTY;

                /*we then clear the timing variables since they are
static*/
                AX_endTimeout = 0;
                AX_startTimeout = 0;

                /* change state of protocol */
                state = idle;
            }
```

```
            }
            break;

        case RX:
#ifdef AX_DEBUG
            Serial.print("\n RX State \n");
#endif
            flag_Deframing_to_Control = EMPTY; /* clears Buffer after copying data in it */

            /* Generate Required Control Byte */
            NS = VS;
            NR = VR + 1;
            incrementStateVar(&VS);

            /* check on CRC flag (in de-frame function) if True make RR if False make REJ */
            if (flag_RX_crc == SET && flag_NS_VR == SET) {
#ifdef AX_DEBUG
                Serial.print("\n Accept the received frame \n");
#endif
                //*a_control = AX25_getControl(S, RR, NS, g_Received_NR, pollfinal);
```

```
                *a_control = AX25_getControl(S, RR, NS, NR, pollfinal);
                incrementStateVar(&VR); /*(TODO: check from DR.)
increments VR if I-frame is accepted */
                flag_NS_VR = CLEAR;
            } else {

#ifdef AX_DEBUG
                Serial.print("\n Reject the received frame \n");
#endif
                *a_control = AX25_getControl(S, REJ, NS, NR, pollfinal);
            }

            /*---------------------------------------*/

            g_infoSize = 0; /* set info field size to 0 in S frame */

            /* Fill address array */
            for (i = 0; i < ADDR_LEN; i++) {
                addr[i] = myAddress[i];
            }

            flag_Control_to_Framing = FULL;
```

```
            state = idle;

            break;

        default:
            break;
        }

}

void AX25_buildFrame(uint8 *buffer, uint8 *a_info_ptr, uint16 *frameSize,
            uint8 *ADDR, uint8 control, uint8 infoSize) {
        uint16 i;

        /* Put flags at the right place in the buffer. */
        buffer[0] = 0x7E;

        /* Add the address in the buffer. */
        for (i = 1; i < ADDR_LEN + ADDR_OFFSET; i++) {
            buffer[i] = ADDR[i - 1];
        }
        /* Add the control byte */
        for (; i < CNTRL_OFFSET + CNTRL_LEN; i++) {
```

```
            buffer[i] = control;
        }
        /* Add the info field in the buffer. */
        for (; i < infoSize + INFO_OFFSET; i++) {
            buffer[i] = *a_info_ptr;
            a_info_ptr++;
        }
        for (; i < FCS_OFFSET; i++) {
            buffer[i] = 0xaa;
        }

        /* Calculation and insertion of the FCS in the buffer. */
        AX25_putCRC(buffer, &i);

        buffer[i] = 0x7E;
        *frameSize = i + 1;

        flag_Control_to_Framing = EMPTY;
        flag_SerialTXBuffer = FULL;
}

/*
 * Description: takes whole frame and splits it into fields (address, control, info).
```

```
*       and also checks on the frame flags (namely 0x7E) and the CRC
*
* Parameters:
*       buffer: the buffer that contains the full frame which will be de-framed.
*       frameSize: the size of the frame.
*       infoSize: the size of the info field in that frame.
*
* Notes:
*       Sets the De-framing to Control flag
*       controls the flag_RX_crc
*
*/


void AX25_deFrame(uint8 *buffer, uint16 frameSize, uint8 infoSize) {
        uint8 newbuffer[AX25_FRAME_MAX_SIZE]; // this was set to frameSize, i changed it to AX25_FRAME_MAX_SIZE to test it.
        uint16 crc;
        uint8 *ptrz;
        uint16 i = 0;
        uint16 j;
        flag_RX_crc = CLEAR; /* initially assume CRC is wrong, if CRC is correct it will SET the flag. otherwise it will remain CLEAR */
```

```
        ptrz = (uint8*) &crc;
        for (; i < AX25_FRAME_MAX_SIZE; i++) {
                newbuffer[i] = buffer[i];
        }

        if (newbuffer[0] == 0x7E) {
                for (i = 1, j = 0; i < ADDR_LEN + ADDR_OFFSET; i++, j++) {
                        g_received_address[j] = newbuffer[i];
                }
                for (j = 0; i < CNTRL_LEN + CNTRL_OFFSET; i++, j++) {
                        g_control_recived[j] = newbuffer[i];
                }
                //Serial1.print("\n el size \n");
                for (j = 0; i < infoSize + INFO_OFFSET; i++, j++) {
                        g_info_reciver[j] = newbuffer[i];

                        //Serial1.print(g_info_reciver[j],HEX);
                } //Serial1.print("\n \n");
                for (j = 0; i < FCS_OFFSET; i++, j++) {
//      g_padding_recived[j] = newbuffer[i];
                }
                flag_Deframing_to_Control = FULL;
                flag_SerialRXBuffer = EMPTY;
```

```
                crc = computeCRC(newbuffer, &i);
                ptrz++;
                if (*ptrz == newbuffer[i]) {
                        i++;
                        ptrz--;
                        if (*ptrz == newbuffer[i]) {
                                flag_RX_crc = SET;

//      i++;
//      printf("\n**received frame**\n");
//      if (newbuffer[i] == 0x7E) {
//        printf("flag=: %x", newbuffer[i]);
//        printf("\n address:\t");
//        for (i = 0; i < ADDR_LEN; i++) {
//          printf("%x", g_received_address[i]);
//        }
//        printf("\n control byte\t");
//        for (i = 0; i < CNTRL_LEN; i++) {
//          printf("control[%d]=%x\t", i, g_control_recived[i]);
//        }
//
//        printf("\n info \n");
//        for (i = 0; i < infoSize; i++) {
```

```
//            printf("%x", g_info_reciver[i]);
//          }
//          printf("\n padding: \t");
//          for (i = 0; i < INFO_MAX_LEN - infoSize; i++) {
//            printf("%x", g_padding_recived[i]);
//          }
//          printf("\nFCS\n");
//          printf("\n CRC = %x\n", crc);
//          printf("\n flag = %x \n",
//              newbuffer[AX25_FRAME_MAX_SIZE - 1]);
//        }
                        }
                }
        }

        else { /* prevent from continuously going to de-frame */
                flag_SerialRXBuffer = EMPTY;
        }
}

#if 0
void AX25_prepareIFrame(TX_FRAME *frame, uint8 control) {
```

```
frame[0] = 0x7E;
uint8 SSID_OctetDest = 0, SSID_OctetSource = 0;
/*-------------------------------------------------------------------------*
 * AX.25 TX header : < Address  | Control >
 *                   < 14 bytes |  1 byte >
 *
 *   - Frame address data from page 10 in documentation
 *   - Address : Destination = NJ7P  (+ 2 spaces), SSID
 *   - Address : Source = N7LEM (+ 1 space) , SSID
 *-------------------------------------------------------------------------*/

static uint8 AX25_txAddressField[ADDR_L] = { 'N', 'J', '7', 'P', ' ', ' ',
    SSID_OctetDest, 'N', '7', 'L', 'E', 'M', ' ', SSID_OctetSource };

/*******************************
 *    config SSID subfield     *
 *******************************/
uint8 SSIDSource = 0xf;
uint8 SSIDDest = 0xe;

SSID_OctetSource |= (1 << 0); /* set X bit */
SSID_OctetSource |= ((SSIDSource & 0x0F) << 1); /* insert SSID into the SSID
octet */
```

```
  SSID_OctetDest |= (SSIDDest << 1); /* insert SSID into the SSID octet */


  for (uint16 i = 0; i < ADDR_L; i++) {
    frame->address[i] = AX25_txAddressField[i];
  }


  /*****************************
   *    config control field
   *****************************/
  IframeControlField();
  SframeControlField();
  UframeControlField();


}
void printTxFrame(uint8 * tx_ptr, uint16 size) {
  for (uint16 i = 0; i < size; i++) {
    printf("%d", tx_ptr[i]);
  }
}


IframeControlField(TX_FRAME *frame) {
```

```
  frame->control = (frame->control & 0x1F) | ((NR << 5) & 0xE0); /* insert N(R)
into control field */

  frame->control = (frame->control & 0xF1) | ((NS << 1) & 0x0E); /* insert N(S)
into control field */

  frame->control &= ~(1 << 0); /* put zero in BIT0 of control field as in page 3 I
frame */

  frame->control &= ~(1 << 4); /* clear P bit as it's not used as stated in section 6.2
*/


  NR++;

  NS++;

  if (NR > 7) {

    NR = 0;

  }

  if (NS > 7) {

    NS = 0;

  }

}


SframeControlField(TX_FRAME *frame) {

  frame->control = 1; /* to initially make two LSB = 01 */

  frame->control = (frame->control & 0x1F) | ((NR << 5) & 0xE0); /* insert N(R)
into control field */
```

```
  if (RRFrame) {
    SSBits = 0;
  } else if (RNR) {
    SSBits = 1;
  } else if (REJ) {
    SSBits = 2;
  } else if (SREJ) {
    SSBits = 3;
  }
  frame->control = (frame->control & 0xF3) | ((SSBits << 2) & 0x0C); /* insert SS
Bits into control field */

}

UframeControlField() {

#endif
```

```
/*-------------------------------------------------------------------------*
 * gradAXandSSP.c
 *-------------------------------------------------------------------------*/

#include "Arduino.h"
#include <stdlib.h>
#include <stdio.h>

/* settings */

//#define AX_DEBUG
#define SSP_DEBUG

#define AX_NRF // note en kan feh moshkela bt5ly el run bty2 awy lma knt bn3ml comment lel line dh

/* note: do not un-comment both at the same time */
#define AX_WRITE_FRAME_TO_LABVIEW // use this line to write frame on serial (readable for LabVIEW)
//#define AX_WRITE_FRAME_TO_HERCULES // use this line to write frame on serial (readable for Hercules)

/* settings */
```

```
#include "ax25.h"
#include "ssp.h"
//ay 7aga
/* keep this line when in Arduino is in RX mode, otherwise, comment it out. */
//#define RX_M
/* keep this line when Debugging, otherwise, comment it out. */
#define DEBUG

/* Include nRF Libraries */
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

//create an RF24 object
RF24 radio(9, 8);  // CE, CSN

//address through which two modules communicate.
const byte address[6] = "00001";

uint8 SerialTXBuffer[AX25_FRAME_MAX_SIZE];
uint8 SerialRXBuffer[AX25_FRAME_MAX_SIZE];
uint8 info[SSP_FRAME_MAX_SIZE]; /* this is in Control to Framing part */
```

```
uint8 SSP_to_Control_Buffer[SIZE_SSP_to_Control_Buffer];
uint8 addr[ADDR_LEN] = { 'O', 'N', '4', 'U', 'L', 'G', 0x60, 'O', 'U', 'F', 'T',
                'I', '1', 0x61 };
uint8 g_info_reciver[SSP_FRAME_MAX_SIZE];
extern uint8 Control_To_SSP[236];
uint8 flag_SSP_to_Control = EMPTY;
uint8 flag_Control_to_Framing = EMPTY;
uint8 flag_Control_to_SSP = EMPTY;
uint8 flag_Deframing_to_Control = EMPTY;
uint8 flag_SerialTXBuffer = EMPTY;
uint8 flag_SerialRXBuffer = EMPTY;
uint8 flag_next_frame = FULL;
extern uint8 flag_controltossp = EMPTY;
uint8 g_infoSize = 236;  //temp set as 236

uint8 txframe[dt];
uint8 data[information];
uint8 data2[information];
uint8 rxframe[dt];
uint8 Rx_data[information];
uint8 layerdata[information];
uint8 ax_rx_data[information];
//uint8 Rx_App_data[information];
```

```
uint8 ax_ssp_frame[dt];
uint8 type2;

uint8 checkcontrol = EMPTY;
uint16 data_length;
uint8 dataflag = EMPTY;
uint8 rxflag = EMPTY;
uint16 ax_rx_length = 0;
uint8 typee;
void receive_frame_here() {
        if (rxflag == EMPTY) {
                if (Serial1.available() > 0) {
                        Serial1.readBytes(rxframe, 236);
                        serial_flush_buffer();
                        rxflag = FULL;

#ifdef SSP_DEBUG
                        Serial1.println("\n Received frame\n");
#endif
                        Serial1.flush();
                }
        }
}
```

```
void serial_flush_buffer() {
	while (Serial1.read() >= 0)
		; // do nothing
}
#if 0
void printSerialTXBufferToSerial() {
	if (flag_SerialTXBuffer == FULL) {
		for (int i = 0; i < AX25_FRAME_MAX_SIZE; ++i) {
			Serial.print(SerialTXBuffer[i], HEX);


			//  Serial.print(SerialTXBuffer[i], HEX);
			Serial.flush();
		}
		flag_SerialTXBuffer = EMPTY;
		//  Serial.print("\n\n");
	}
}

void readFrameFromSerial() {
	uint8 flag_flagAndDestMatchSerialRXBuffer = SET; /* init value as set */
	uint8 flagAndDestAddress[8] = { 0x7e, 'O', 'N', '4', 'U', 'L', 'G', 0x60 };
```

```
        if (Serial.available() && flag_SerialRXBuffer == EMPTY) {
                g_infoSize = SSP_FRAME_MAX_SIZE;

#ifdef DEBUG
                Serial.print("\n waiting for data \n");
#endif

                Serial.readBytes(SerialRXBuffer, 8);
                for (uint8 i = 0; i < 8; i++) {
                        if (SerialRXBuffer[i] != flagAndDestAddress[i]) {
                                flag_flagAndDestMatchSerialRXBuffer = CLEAR;
                        }
                }

//#ifdef DEBUG
//    Serial1.println(SerialRXBuffer[0]);
//#endif
                if (flag_flagAndDestMatchSerialRXBuffer == SET) {
                        for (uint8 i = 1; i < 32; i++) {
                                Serial.readBytes(SerialRXBuffer + (8 * i), 8);
                        }
```

```
                //Serial.readBytes(SerialRXBuffer,
AX25_FRAME_MAX_SIZE);


                //Serial.flush();
                delay(100);
                flag_SerialRXBuffer = FULL;


#ifdef DEBUG
                Serial.print("\n Received Frame\n");
#endif
                Serial.flush();


                /* prints the frame received from serial on serial monitor */


                for (int i = 0; i < 30; ++i) {


#ifdef DEBUG
                        //  Serial.print(SerialRXBuffer[i], HEX);
                        Serial.flush();
#endif


                }
```

```
#ifdef DEBUG
                        Serial.print("\n\n");
#endif

                        for (int i = 230; i < 256; ++i) {

#ifdef DEBUG
                                // Serial.print(SerialRXBuffer[i], HEX);
                                Serial.flush();
#endif
                        }

#ifdef DEBUG
                        Serial.print("\n\n");
#endif

                }
        }
}
#endif

/*
 * Description: prints the data stored in SerialTXBuffer to Serial or nRF
```

```
 * flags: flag_SerialTXBuffer flag.
 */
void printSerialTXBufferToSerial() {
        if (flag_SerialTXBuffer == FULL) {


                radio.stopListening();
                for (int i = 0; i < AX25_FRAME_MAX_SIZE; ++i) {
#ifdef AX_NRF
                        radio.write(&SerialTXBuffer[i], sizeof(uint8)); /* writes Serial RX Buffer to nRF 1 byte at a time */
#endif
                        // Note: the serial.write and print lines below are deprecated after using nRF Module
#ifdef AX_WRITE_FRAME_TO_LABVIEW
                        Serial.write(SerialTXBuffer[i]); // used to write serially to LabVIEW


#else
#ifdef AX_WRITE_FRAME_TO_HERCULES
                        Serial.print(SerialTXBuffer[i], HEX); // to display array as string in HEX format (mostly used for debugging)
#endif
#endif
                        //Serial.flush();
```

```
            }
            flag_SerialTXBuffer = EMPTY;
            radio.startListening();
        }
}

void readFrameFromSerial() {
        uint8 flag_flagAndDestMatchSerialRXBuffer = SET; /* init value as set (do not change init value, changing it will cause unwanted behavior)*/
        uint8 flagAndDestAddress[8] = { 0x7e, 'O', 'N', '4', 'U', 'L', 'G', 0x60 };

#ifdef SerialWire

        if (Serial.available() && flag_SerialRXBuffer == EMPTY) {
            g_infoSize = SSP_FRAME_MAX_SIZE;

#ifdef DEBUG
            Serial1.print("\n waiting for data \n");
#endif

            Serial.readBytes(SerialRXBuffer, 8);
            for (uint8 i = 0; i < 8; i++) {
                if (SerialRXBuffer[i] != flagAndDestAddress[i]) {
```

```
                        flag_flagAndDestMatchSerialRXBuffer = CLEAR;
                }
        }

//#ifdef DEBUG
//              Serial1.println(SerialRXBuffer[0]);
//#endif
        if (flag_flagAndDestMatchSerialRXBuffer == SET) {
                for (uint8 i = 1; i < 32; i++) {
                        Serial.readBytes(SerialRXBuffer + (8 * i), 8);
                }

                //Serial.readBytes(SerialRXBuffer,
AX25_FRAME_MAX_SIZE);

                //Serial.flush();
                delay(100);
                flag_SerialRXBuffer = FULL;

#ifdef DEBUG
                Serial1.print("\n Received Frame\n");
#endif
                Serial.flush();
```

```
                /* prints the frame received from serial on serial monitor */

                for (int i = 0; i < 30; ++i) {

#ifdef DEBUG
                        Serial1.print(SerialRXBuffer[i], HEX);
                        Serial.flush();
#endif

                }

#ifdef DEBUG
                Serial1.print("\n\n");
#endif

                for (int i = 230; i < 256; ++i) {

#ifdef DEBUG
                        Serial1.print(SerialRXBuffer[i], HEX);
                        Serial.flush();
#endif

                }
```

```
#ifdef DEBUG
                    Serial1.print("\n\n");
#endif


            }
        }
#endif


#ifndef SerialWire
        /* code to read from nrf  */


        if (radio.available() && flag_SerialRXBuffer == EMPTY) {


#ifdef AX_DEBUG
            Serial.print("\nreceived Frame from nrf\n");
#endif


            g_infoSize = SSP_FRAME_MAX_SIZE;


            //Read the data if available in buffer
            unsigned char text[1] = { 0 };
            for (int j = 0; j < 8; j++) {
```

```
        while (!(radio.available()))
                ;
        radio.read(&text, sizeof(text));
        SerialRXBuffer[j] = text[0];
}

for (uint8 i = 0; i < 8; i++) {
        if (SerialRXBuffer[i] != flagAndDestAddress[i]) {
                flag_flagAndDestMatchSerialRXBuffer = CLEAR;
        }
}

if (flag_flagAndDestMatchSerialRXBuffer == SET) {

        //Read the data if available in buffer
        unsigned char text_2[1] = { 0 };
        for (int j = 8; j < 256; j++) {
                while (!(radio.available()))
                        ;
                radio.read(&text_2, sizeof(text_2));
                SerialRXBuffer[j] = text_2[0];
        }
```

```
//                      delay(100);
                        flag_SerialRXBuffer = FULL;
                        Serial.flush();
                }
#ifdef AX_DEBUG
                for (int i = 0; i < 256; i++) {
                        Serial.print(SerialRXBuffer[i], HEX);
                }
#endif
        }
#endif
}

void setup() {
        Serial.begin(9600);
        Serial1.begin(9600);
        radio.begin();
        //set the address
        radio.openWritingPipe(address);
        radio.openReadingPipe(0, address);
        //Set module as receiver
        radio.startListening();
```

```
        radio.setPayloadSize(1);


//  if (checkcontrol == EMPTY) {
//    getdata(data, &data_length, &dataflag);
//    checkcontrol = FULL;
//  }
//#ifndef RX_M
        // if (flag_SSP_to_Control == EMPTY) {
        //   fillBuffer(SSP_to_Control_Buffer, SIZE_SSP_to_Control_Buffer);
        //   flag_SSP_to_Control = FULL;
//  }
//#endif
}

void loop() {

        static uint8 txflag = EMPTY;
        uint8 i;
        static uint8 crcflag = EMPTY;
        uint8 desti;
        uint8 srce;
        uint8 ax_src;
        uint8 ax_type;
```

```
uint8 desti2;
static uint16 tx_size = 0;
uint8 adddest;
uint8 addsrc;
static uint8 type = 0;
static uint16 Rx_length = 0;
static uint8 layerflag = EMPTY;
static uint8 deframeflag = EMPTY;
static uint8 framingflag = EMPTY;
uint8 dest_to_framing;
uint8 src_to_framing;
uint8 type_to_framing;
//uint8 ax_dest;
//uint8 ax_src;
//static uint8 ax_type = 0;
//static uint8 deframe_ax_flag = EMPTY;
static uint16 tx_ax_length = 0;
static uint8 ax_ssp_flag = EMPTY;
//static uint8 deframetoframeflag = EMPTY;

uint8 control;
uint16 frameSize = 0;
```

```
//        delay(100);


//delay(1000);
        //Serial1.print(flag_controltossp, HEX);


//if(checkcontrol == EMPTY &&
deframe_ax_flag==EMPTY&&flag_Deframing_to_Control == FULL)


        if (checkcontrol == EMPTY && flag_controltossp == FULL) {
                //dataflag = EMPTY;


                ssp_ax_deframing(Control_To_SSP, ax_rx_data, &ax_rx_length,
&ax_type,
                                &ax_src);


                //Serial1.print("\n size w dkhal hena \n");
                //Serial1.print(ax_rx_length, HEX);
                //Serial1.print("\n \n");
                //for (i = 0; i < ax_rx_length; i++) {
                //        data[i] = ax_rx_data[i];
                //Serial1.print(g_info_reciver[i], HEX);
                //        }
                //flag_controltossp=EMPTY;
```

```
            //      deframe_ax_flag=EMPTY;
            //      data_length = ax_rx_length;
            //      dataflag = FULL;
            getdata(data, &data_length, &dataflag, ax_type, ax_src, &typee,
&desti);

            checkcontrol = FULL;
        }

        if ((checkcontrol == FULL && txflag == EMPTY)
                || (checkcontrol == EMPTY && layerflag == EMPTY)) {
            control_layer(data, data_length, &desti, &srce, &typee, &type2,
data2,
                    &desti2, &type, Rx_data, &adddest, &Rx_length,
&dataflag,
                    &deframeflag, &txflag, layerdata, crcflag, &tx_size,
&addsrc,
                    &layerflag, &checkcontrol, &dest_to_framing,
&src_to_framing,
                    &type_to_framing);

            //      layerflag = EMPTY;

        }
```

```
        if (txflag == FULL) {
                ssp_build_frame(txframe, data2, desti2, srce, type2, tx_size, &txflag);
        }

        receive_frame_here();

        if (rxflag == FULL && deframeflag == EMPTY) {
                //Serial1.print("DEFRAME");
                ssp_deframing(rxframe, &adddest, &addsrc, &type, Rx_data,
&Rx_length,
                                &rxflag, &crcflag, &deframeflag);
//Serial1.print("\n hana\n");
        }

        /* Sends next frame */
        //Serial1.print(layerflag,HEX);
        //delay(2000);
//if(deframetoframeflag==FULL){
//        ax_ssp_framing(ax_ssp_frame,layerdata, desti, srce,typee,Rx_length,
&tx_ax_length,&ax_ssp_flag);
//        deframetoframeflag = EMPTY;
//}
```

```
//delay(1000);
//        Serial1.print(flag_next_frame,HEX);
        if (flag_next_frame == FULL && layerflag == FULL
                        && flag_SSP_to_Control == EMPTY) {
        //        Serial1.print("DKHAL EL FUNCTION ASASN ");
                ax_ssp_framing(ax_ssp_frame, layerdata, &dest_to_framing,
                                &src_to_framing, &type_to_framing, Rx_length,
&tx_ax_length);
                fillBuffer(&tx_ax_length, &layerflag, dest_to_framing,
type_to_framing,
                                &dataflag, data, &data_length, &checkcontrol, &desti,
&typee,
                                &src_to_framing);
//Serial1.print("\n el data el mafrod tro7\n\n");
//Serial1.print(tx_ax_length);
//Serial1.print("\n\n\n");
//Serial1.print("\n check gded\n\n");

// for (i = 0;i < Rx_length; i++) {

                //                Serial1.print(layerdata[i], HEX);

                //                }
```

```
            //Serial1.print("\n\n\n");

            /*for (i = 0; i < tx_ax_length; i++) {
             SSP_to_Control_Buffer[i] = ax_ssp_frame[i];

             Serial1.print(SSP_to_Control_Buffer[i], HEX);

             }
             //SIZE_SSP_to_Control_Buffer=Rx_length;

             g_infoSize = tx_ax_length;
             ax_ssp_flag=EMPTY;
             layerflag=EMPTY;
             flag_SSP_to_Control = FULL;*/

            flag_next_frame = EMPTY;
        }

        /* Calls the manager function */
        if ((flag_SSP_to_Control == FULL && flag_Control_to_Framing ==
EMPTY)
                    || (flag_Control_to_SSP == EMPTY
```

```
                                && flag_Deframing_to_Control == FULL)) {

#ifdef AX_DEBUG
            Serial.print("\nManagement\n");
#endif
            AX25_Manager(&control);
        }

        /* Builds Frame after receiving fields */
        if (flag_Control_to_Framing == FULL && flag_SerialTXBuffer ==
EMPTY) {

#ifdef AX_DEBUG
            Serial.print("\nBuild Frame\n");
#endif
            AX25_buildFrame(SerialTXBuffer, info, &frameSize, addr, control,
                        g_infoSize);
        }

        /* Prints Serial TX buffer */
        printSerialTXBufferToSerial();

        /* Gets frame from serial */
```

```
        readFrameFromSerial();
        //serialFlush();

        /* Calls the de-framing function */
        if (flag_Deframing_to_Control == EMPTY && flag_SerialRXBuffer ==
FULL) {

#ifdef AX_DEBUG
                Serial.print("\nDeframe\n");
#endif
                AX25_deFrame(SerialRXBuffer, frameSize, g_infoSize);
        }

}
```

```
/*-------------------------------------------------------------------------*
 * SSP.h
 *-------------------------------------------------------------------------*/
#ifndef SSP_H_
#define SSP_H_

#define fend 0
#define dt 236
#define dest 1
#define src 2
#define typ 3
#define header 8
#define information 229
//#define FULL 1
//#define EMPTY 0
#define tx 1
#define rx 2
//#define idle 0
typedef unsigned char        uint8;
typedef unsigned short       uint16;
```

```
/* Set a certain bit in any register */
#define SET_BIT(REG,BIT) (REG|=(1<<BIT))

/* Clear a certain bit in any register */
#define CLEAR_BIT(REG,BIT) (REG&=(~(1<<BIT)))

/* Toggle a certain bit in any register */
#define TOGGLE_BIT(REG,BIT) (REG^=(1<<BIT))

/* Rotate right the register value with specific number of rotates */
#define ROR(REG,num) ( REG= (REG>>num) | (REG<<(8-num)) )

/* Rotate left the register value with specific number of rotates */
#define ROL(REG,num) ( REG= (REG<<num) | (REG>>(8-num)) )

/* Check if a specific bit is set in any register and return true if yes */
#define BIT_IS_SET(REG,BIT) ( REG & (1<<BIT) )

/* Check if a specific bit is cleared in any register and return true if yes */
#define BIT_IS_CLEAR(REG,BIT) ( !(REG & (1<<BIT)) )
```

```
unsigned short compute_crc16( unsigned char* data_p, unsigned char length);

void ssp_build_frame(uint8 *txframe, uint8 *data , uint8 desti, uint8 srce, uint8
typee, uint16 tx_size, uint8 *txflag);

void getdata(uint8 *data, uint16 *data_length, uint8 *dataflag,uint8 ax_type,uint8
ax_src, uint8 *Tx_App_type, uint8 *Tx_App_desti);

void ssp_deframing(uint8 *rxframe, uint8* adddest, uint8* addsrc, uint8* type,
uint8 *datta, uint16 *size3, uint8 *rxflag, uint8 *crcflag, uint8 *deframeflag);

void control_layer(uint8 *Tx_App_data, uint16 data_length, uint8 *Tx_App_desti,
uint8 *Tx_Frm_srce, uint8 *Tx_App_type, uint8 *Tx_Frm_type, uint8
*Tx_Frm_data, uint8 *Tx_Frm_desti, uint8 *Rx_Frm_type, uint8 *Rx_Frm_data,
uint8 *Rx_Frm_dest, uint16 *Rx_length, uint8 *dataflag, uint8 *deframeflag,
uint8 *txflag, uint8 *Rx_App_data, uint8 crcflag, uint16 *tx_size, uint8
*Rx_Frm_src, uint8 *layerflag, uint8 *checkcontrol,uint8 *dest_to_framing,uint8
*src_to_framing,uint8 *type_to_framing);

void ssp_ax_deframing(uint8 *Control_To_SSP,uint8 *ax_rx_data, uint16
*ax_rx_length,uint8 *ax_type,uint8 *ax_src);

void ax_ssp_framing(uint8 *ax_ssp_frame, uint8 *Rx_App_data, uint8 *desti,
uint8 *srce,uint8 *typee, uint16 Rx_length, uint16 *tx_ax_length);

#endif
```

```
/*-------------------------------------------------------------------------*
 * SSP.c
 *-------------------------------------------------------------------------*/

#include "ssp.h"
#include "Arduino.h"
#include "ax25.h"

#if 0

//check here since these varibales are not externeed nor declared.

extern uint16 ax_rx_length;
extern uint8 flag_controltossp;
extern uint8 flag_next_frame;
extern uint8 flag_SSP_to_Control;
//extern uint16 ax_rx_data;
#endif

unsigned short compute_crc16(unsigned char *data_p, unsigned char length) {
        unsigned char x;
        unsigned short crc = 0xFFFF;
```

```
        while (length--) {
                /*  reverse the bits in each 8-bit byte going in */
                *data_p = (*data_p & 0x55555555) << 1 | (*data_p &
0xAAAAAAAA) >> 1;
                *data_p = (*data_p & 0x33333333) << 2 | (*data_p &
0xCCCCCCCC) >> 2;
                *data_p = (*data_p & 0x0F0F0F0F) << 4 | (*data_p & 0xF0F0F0F0)
>> 4;

                x = crc >> 8 ^ *data_p++;
                x ^= x >> 4;
                crc = (crc << 8) ^ ((unsigned short) (x << 12))
                                ^ ((unsigned short) (x << 5)) ^ ((unsigned short) x);
        }
        /*reverse the 16-bit CRC*/
        crc = (crc & 0x55555555) << 1 | (crc & 0xAAAAAAAA) >> 1;
        crc = (crc & 0x33333333) << 2 | (crc & 0xCCCCCCCC) >> 2;
        crc = (crc & 0x0F0F0F0F) << 4 | (crc & 0xF0F0F0F0) >> 4;
        crc = (crc & 0x00FF00FF) << 8 | (crc & 0xFF00FF00) >> 8;

        return crc;
}
```

```
void getdata(uint8 *data, uint16 *data_length, uint8 *dataflag, uint8 ax_type,
        uint8 ax_src, uint8 *Tx_App_type, uint8 *Tx_App_desti) {
    *dataflag = EMPTY;
    uint8 i;

    for (i = 0; i < ax_rx_length; i++) {
        data[i] = ax_rx_data[i];
    }
    flag_controltossp = EMPTY;
    //*deframe_ax_flag=EMPTY;
    *data_length = ax_rx_length;
    *Tx_App_type = ax_type;
    *Tx_App_desti = ax_src;
    Serial1.print(*Tx_App_type, HEX);
    *dataflag = FULL;
}

void control_layer(uint8 *Tx_App_data, uint16 data_length, uint8 *Tx_App_desti,
        uint8 *Tx_Frm_srce, uint8 *Tx_App_type, uint8 *Tx_Frm_type,
        uint8 *Tx_Frm_data, uint8 *Tx_Frm_desti, uint8 *Rx_Frm_type,
        uint8 *Rx_Frm_data, uint8 *Rx_Frm_dest, uint16 *Rx_length,
        uint8 *dataflag, uint8 *deframeflag, uint8 *txflag, uint8
*Rx_App_data,
```

```
        uint8 crcflag, uint16 *tx_size, uint8 *Rx_Frm_src, uint8 *layerflag,
        uint8 *checkcontrol, uint8 *dest_to_framing, uint8 *src_to_framing,
        uint8 *type_to_framing) {

    static uint8 controlflag = idle;
    static uint8 counter = 0;
    uint8 source1 = 0x05;
    uint8 source2 = 0x03;
    static uint32 time_out=0;
    static uint32 current_time=0;
    uint8 i;
    //Serial1.println("\n Sending  Data \n");

    // Serial1.print("control");
    if (controlflag == idle) {
        if (*dataflag == FULL && *txflag == EMPTY) {
#ifdef SSP_DEBUG
            Serial1.println("\n Sending  Data \n");
            Serial1.println("\n Starting TX Mode \n");
            Serial1.flush();
#endif

            controlflag = tx;
```

```
                *Tx_Frm_srce = source1;


                for (i = 0; i < data_length; i++) {
                        Tx_Frm_data[i] = Tx_App_data[i];
                }


                *tx_size = data_length;
                *Tx_Frm_desti = *Tx_App_desti;
                *Tx_Frm_type = *Tx_App_type;
                //Serial1.print(*Tx_App_type,HEX);
                *txflag = FULL;
                *dataflag = EMPTY;
        time_out=millis();
            } else if (*deframeflag == FULL && *layerflag == EMPTY) {


                if (*Rx_Frm_dest == source1 || *Rx_Frm_dest == source2) {


#ifdef SSP_DEBUG
                        Serial1.print("\n Received Data \n");
            Serial1.println("\n Starting RX Mode \n");
#endif
                        Serial1.flush();
                        controlflag = rx;
```

```
                *layerflag = FULL;
                *deframeflag = EMPTY;
                flag_next_frame = FULL;
                //*dataflag=EMPTY;
                // *deframetoframeflag=FULL;
                *dest_to_framing = *Rx_Frm_dest;
                //Serial1.print(*dest_to_framing, HEX);
                *src_to_framing = *Rx_Frm_src;
                *type_to_framing = *Rx_Frm_type;
                for (i = 0; i < *Rx_length; i++) {
                        Rx_App_data[i] = Rx_Frm_data[i];

                }

        } else {
                //Serial1.print("\nhelp\n");

                *deframeflag = EMPTY;
        }
    }

}
```

```
else if (controlflag == tx) {
        //Serial1.println("\n tx tania \n");
        //Serial1.println("\n flag deframe\n");
        //Serial1.println("\n deframeflag \n");
        if (*deframeflag == FULL) {

            if (*Rx_Frm_dest == source1 && *Rx_Frm_type == 0x02) {
                Serial1.println("\n Respond with an ACK \n");
                Serial1.flush();
                controlflag = idle;
                *deframeflag = EMPTY;
                *checkcontrol = EMPTY;
                // *dataflag = EMPTY;
                counter = 0;
            }

            else if ((*Rx_Frm_dest == source1)
                    && (*Rx_Frm_type == 0x03 || *Rx_Frm_type ==
0x13
                            || *Rx_Frm_type == 0x23)) {
                Serial1.println("\n Response with NACK \n");
                Serial1.println("\n Sending Data Again \n");
                Serial1.flush();
```

```
        for (i = 0; i < data_length; i++) {
                Tx_Frm_data[i] = Tx_App_data[i];
        }
        *tx_size = data_length;
        *txflag = FULL;
        *deframeflag = EMPTY;
        *checkcontrol = FULL;
        counter++;
        if (counter == 3) {
                Serial1.println("\n NACK Counter= 3 \n");
                Serial1.flush();
                controlflag = idle;
                counter = 0;
                *txflag = EMPTY;
                *checkcontrol = EMPTY;
        }

        }

else if (*Rx_Frm_dest != source1) {
        *deframeflag = EMPTY;

}
```

```
            }


        else {
                    //Serial1.println("\nalooo EL OLA \n");
                    current_time=millis();
                    if((current_time-time_out)>3000){
                        Serial1.println("\n SSP Timeout! \n");
                            *deframeflag = EMPTY;
                                *checkcontrol = EMPTY;
                                controlflag = idle;
                            flag_SSP_to_Control = EMPTY;
                            flag_controltossp=EMPTY;
                            current_time=0;
                        time_out=0;
                                                        }
        }
        }else if (controlflag == rx) {



            if (crcflag == EMPTY) {
#ifdef SSP_DEBUG
```

```
                Serial1.println("\n Correct CRC \n");
#endif
                Serial1.flush();
                *Tx_Frm_srce = *Rx_Frm_dest;
                *tx_size = 0;

                *Tx_Frm_desti = *Rx_Frm_src;
                *Tx_Frm_type = 0x02;
                *txflag = FULL;
                controlflag = idle;
                //*layerflag = FULL;
            } else if (crcflag == FULL) {
                Serial1.println("\n Wrong CRC \n");
                Serial1.flush();
                *Tx_Frm_srce = *Rx_Frm_dest;
                *tx_size = 0;
                *Tx_Frm_desti = *Rx_Frm_src;
                *Tx_Frm_type = 0x03;
                *txflag = FULL;
                *layerflag=EMPTY;
                controlflag = idle;
            }
        }
```

```
        //Serial1.println("el time");
        //Serial1.println(time_out);
        //Serial1.println("dlwaaty");
        //Serial1.println(current_time);
}

void ssp_build_frame(uint8 *txframe, uint8 *data, uint8 desti, uint8 srce,
                uint8 typee, uint16 tx_size, uint8 *txflag) {

        uint16 p, k;

        txframe[fend] = 0xc0;

        txframe[dest] = desti;

        txframe[src] = srce;

        txframe[typ] = typee;

        uint8 f, d, count = 0, w = 0, count2 = 0, arr[dt];
        int temp = 0, temp2 = 0;
        uint16 crc, crc0, crc1;
        for (k = 0; k < tx_size; k++) {
```

```
        if (data[k] == 0xc0) {

                count++;

        }
}
for (k = 0; k < tx_size; k++) {

        if (data[k] == 0xdb) {

                count2++;

        }
}
temp = tx_size + count;
temp2 = tx_size + count2;

for (k = 0; k <= temp; k++) {

        if (data[k] == 0xc0) {

                data[k] = 0xdb;
```

```
                for (f = temp; f >= (k + 1); f--) {

                        data[f] = data[f - 1];

                }
                data[k + 1] = 0xdc;

        } else if (data[k] == 0xdb) {
                data[k] = 0xdb;
                for (f = temp2; f >= (k + 1); f--) {
                        data[f] = data[f - 1];

                }

                data[k + 1] = 0xdd;

        }
}

w = temp + count2 + 4;

for (p = 4; p < (w); p++) {
```

```
        txframe[p] = data[p - 4];


}
for (d = 1; d < w; d++) {
        arr[d - 1] = txframe[d];


}
crc = compute_crc16(arr, (w - 1));
crc0 = (crc & 0x00ff);
crc1 = ((crc & 0xff00) >> 8);
txframe[w] = crc0;                  // crc0
txframe[(w += 1)] = crc1;               //crc1
txframe[(w + 1)] = 0xc0;


uint8 countttt = 1, j, i;
for (j = 1; j < dt; j++) {


        if (txframe[j] == 0xc0) {
                countttt++;
                break;


        } else {
```

```
                counttttt++;

            }

        }

        for (i = 0; i < counttttt; i++) {
            Serial1.print(txframe[i], HEX);
            Serial1.flush();
        }

        *txflag = EMPTY;
}

void ssp_deframing(uint8 *rxframe, uint8 *adddest, uint8 *addsrc, uint8 *type,
            uint8 *Rx_data, uint16 *length, uint8 *rxflag, uint8 *crcflag,
            uint8 *deframeflag) {
        uint16 i, j, d, size2, size = 1, crc, size3;
        uint8 count = 0, k, y = 0, arr[dt], datta[information + 4];

        *adddest = rxframe[dest];
        *addsrc = rxframe[src];
```

```
*type = rxframe[typ];

for (j = 1; j < dt; j++) {

        if (rxframe[j] == 0xc0) {
                size++;
                break;

        } else {

                size++;

        }

}

for (d = 1; d < size - 1; d++) {
        arr[y] = rxframe[d];

        y++;
}

crc = compute_crc16(arr, y);
```

```
size2 = size - 3;


if (rxframe[fend] == 0xc0 && crc == 0x00) {
        *adddest = rxframe[dest];
        *addsrc = rxframe[src];


        for (i = 4; i < (size - 3); i++) {


                datta[i] = rxframe[i];
        }
        for (i = 4; i < (size - 3); i++) {
                if ((datta[i] == 0xdb) && (datta[i + 1] == 0xdc)) {


                        size2--;


                }
        }


        for (i = 4; i < (size - 3); i++) {
                if ((datta[i] == 0xdb) && (datta[i + 1] == 0xdd)) {


                        count++;
```

```
        }
}

for (i = 4; i < (size - 3); i++) {
        if ((datta[i] == 0xdb) && (datta[i + 1] == 0xdc)) {

                datta[i] = 0xc0;
                for (k = i + 1; k < size - 3; k++) {
                        datta[k] = datta[k + 1];
                }
        }
}

for (i = 4; i < (size - 3); i++) {
        if ((datta[i] == 0xdb) && (datta[i + 1] == 0xdd)) {

                datta[i] = 0xdb;
                for (k = i + 1; k < size - 3; k++) {

                        datta[k] = datta[k + 1];

                }
```

```
                }
        }
        size3 = size2 - count;
        *length = size3 - 4;

        for (i = 0; i < (*length); i++) {
                Rx_data[i] = datta[i + 4];
        }
        *crcflag = EMPTY;

    } else {
        *crcflag = FULL;
    }

    *rxflag = EMPTY;
    *deframeflag = FULL;
    //Serial1.print(*length);
}

void ssp_ax_deframing(uint8 *Control_To_SSP, uint8 *ax_rx_data,
        uint16 *ax_rx_length, uint8 *ax_type, uint8 *ax_src) {
    uint16 i, j, d, size2, size = 1, crc, size3;
```

```
uint8 count = 0, k, y = 0, arr[dt], datta[information + 4];
uint8 ax_dest;
//static uint8 ax_type = 0;
//Serial1.print("\n  bshof \n");
//  for (j = 0; j < 235; j++) {
//  Serial1.print(Control_To_SSP[j],HEX);
// }

ax_dest = Control_To_SSP[dest];
*ax_src = Control_To_SSP[src];
*ax_type = Control_To_SSP[typ];

for (j = 1; j < dt - 1; j++) {

        if (Control_To_SSP[j] == 0xc0) {
                size++;
                break;

        } else {

                size++;

        }
```

```
}

// Serial1.print("\ndeframe gdeda\n");
//Serial1.print(size);
for (d = 1; d < size - 1; d++) {
        arr[y] = Control_To_SSP[d];

        y++;
}

crc = compute_crc16(arr, y);

size2 = size - 3;

if (Control_To_SSP[fend] == 0xc0 && crc == 0x00) {
        ax_dest = Control_To_SSP[dest];
        *ax_src = Control_To_SSP[src];

        for (i = 4; i < (size - 3); i++) {

                datta[i] = Control_To_SSP[i];
        }
```

```
for (i = 4; i < (size - 3); i++) {
        if ((datta[i] == 0xdb) && (datta[i + 1] == 0xdc)) {

                size2--;

        }
}

for (i = 4; i < (size - 3); i++) {
        if ((datta[i] == 0xdb) && (datta[i + 1] == 0xdd)) {

                count++;

        }
}

for (i = 4; i < (size - 3); i++) {
        if ((datta[i] == 0xdb) && (datta[i + 1] == 0xdc)) {

                datta[i] = 0xc0;
                for (k = i + 1; k < size - 3; k++) {
                        datta[k] = datta[k + 1];
                }
```

```
            }
        }

        for (i = 4; i < (size - 3); i++) {
            if ((datta[i] == 0xdb) && (datta[i + 1] == 0xdd)) {

                datta[i] = 0xdb;
                for (k = i + 1; k < size - 3; k++) {

                    datta[k] = datta[k + 1];

                }

            }
        }
        size3 = size2 - count;
        *ax_rx_length = size3 - 4;

        for (i = 0; i < (*ax_rx_length); i++) {
            ax_rx_data[i] = datta[i + 4];
        }

}
```

```
        // *deframe_ax_flag = FULL;
        //Serial1.print(*length);
}

void ax_ssp_framing(uint8 *ax_ssp_frame, uint8 *Rx_App_data, uint8 *desti,
                uint8 *srce, uint8 *typee, uint16 Rx_length, uint16 *tx_ax_length) {

        uint16 p, k;
        uint8 n;
//      Serial1.print("\n check gwa el function el gdeda\n\n");

        //for (n = 0; n < Rx_length; n++) {

        //                      Serial1.print(Rx_App_data[n], HEX);
        //              }

        //Serial1.print("\n\n\n");
        ax_ssp_frame[fend] = 0xc0;

        ax_ssp_frame[dest] = *desti;
        //Serial1.print(desti, HEX);
        ax_ssp_frame[src] = *srce;
```

```
ax_ssp_frame[typ] = *typee;

uint8 f, d, count = 0, w = 0, count2 = 0, arr[dt];
int temp = 0, temp2 = 0;
uint16 crc, crc0, crc1;
for (k = 0; k < Rx_length; k++) {

        if (Rx_App_data[k] == 0xc0) {

                count++;

        }
}
for (k = 0; k < Rx_length; k++) {

        if (Rx_App_data[k] == 0xdb) {

                count2++;

        }
}
temp = Rx_length + count;
```

```
temp2 = Rx_length + count2;

for (k = 0; k <= temp; k++) {

        if (Rx_App_data[k] == 0xc0) {

                Rx_App_data[k] = 0xdb;

                for (f = temp; f >= (k + 1); f--) {

                        Rx_App_data[f] = Rx_App_data[f - 1];

                }
                Rx_App_data[k + 1] = 0xdc;

        } else if (Rx_App_data[k] == 0xdb) {
                Rx_App_data[k] = 0xdb;
                for (f = temp2; f >= (k + 1); f--) {
                        Rx_App_data[f] = Rx_App_data[f - 1];

                }

                Rx_App_data[k + 1] = 0xdd;
```

```
        }
}

w = temp + count2 + 4;

for (p = 4; p < (w); p++) {

        ax_ssp_frame[p] = Rx_App_data[p - 4];

}
for (d = 1; d < w; d++) {
        arr[d - 1] = ax_ssp_frame[d];

}
crc = compute_crc16(arr, (w - 1));
crc0 = (crc & 0x00ff);
crc1 = ((crc & 0xff00) >> 8);
ax_ssp_frame[w] = crc0;                 // crc0
ax_ssp_frame[(w += 1)] = crc1;              //crc1
ax_ssp_frame[(w + 1)] = 0xc0;

uint8 countttt = 1, j, i;
```

```
        for (j = 1; j < dt; j++) {

            if (ax_ssp_frame[j] == 0xc0) {
                countttt++;
                break;

            } else {

                countttt++;

            }

        }
        *tx_ax_length = countttt;
        // for (i = 0; i < countttt; i++) {
        //   Serial1.print(ax_ssp_frame[i],HEX);
        //  Serial1.flush();
//  }

//  *ax_ssp_flag = FULL;
}
```

```
/*****************************************************************************
************
 * std_types.h

*****************************************************************************
*************/

#ifndef STD_TYPES_H_
#define STD_TYPES_H_

/* Boolean Data Type */
//typedef unsigned char boolean;

/* Boolean Values */
#ifndef FALSE
#define FALSE       (0u)
#endif
#ifndef TRUE
#define TRUE        (1u)
#endif

#define LOGIC_HIGH        (1u)
#define LOGIC_LOW         (0u)
```

```
#define NULL_PTR    ((void*)0)

typedef unsigned char         uint8;          /*           0 .. 255             */
typedef signed char           sint8;          /*        -128 .. +127            */
typedef unsigned short        uint16;         /*           0 .. 65535           */
typedef signed short          sint16;         /*      -32768 .. +32767          */
typedef unsigned long         uint32;         /*           0 .. 4294967295      */
typedef signed long           sint32;         /* -2147483648 .. +2147483647     */
typedef unsigned long long    uint64;         /*       0 .. 18446744073709551615  */
typedef signed long long      sint64;         /* -9223372036854775808 ..
9223372036854775807 */
typedef float                 float32;
typedef double                float64;


#endif /* STD_TYPE_H_ */
```

10 Appendix C Hardware

ATmega328p-pu pin configuration

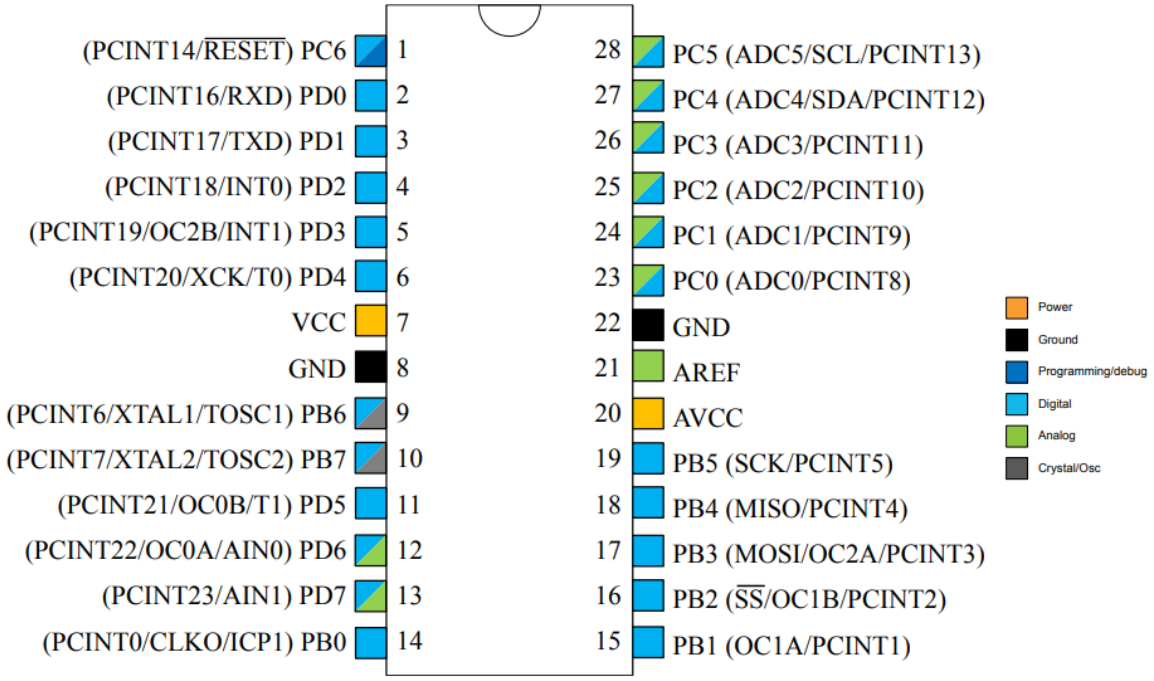

Figure 72 pin configuration of atmega328p-pu

Table 38 pin descriptions of atmega328p-pu

Pin No.	Pin Name	Pin Function	Pin Function Description
1	PC6	Reset	This pin helps to reset the microcontroller.
2	PD0	Digital Pin (RX)	This is the input pin for serial communication
3	PD1	Digital Pin (TX)	This is the output pin for serial communication
4	PD2	Digital Pin	It is used as an external interrupt 0
5	PD3	Digital Pin (PWM)	It is used as an external interrupt 1
6	PD4	Digital Pin	It is used for external counter source Timer0

7	Vcc	Positive Voltage	Positive supply of the system.
8	GND	Ground	The Ground of the system
9	XTAL	Crystal Oscillator	This pin should be connected to one pin of the crystal oscillator to provide an external clock pulse to the chip
10	XTAL	Crystal Oscillator	This pin should also be connected to the other pin of the crystal oscillator to provide an external clock pulse to the chip
11	PD5	Digital Pin (PWM)	Pin 11 is used for external counter source Timer1
12	PD6	Digital Pin (PWM	Positive Analog Comparator i/ps
13	PD7	Digital Pin	Negative Analog Comparator i/ps
14	PB0	Digital Pin	Counter or Timer input source pin
15	PB1	Digital Pin (PWM)	Counter or Timer compares match A.
16	PB2	Digital Pin (PWM)	This pin act as a slave choice i/p.
17	PB3	Digital Pin (PWM)	This pin is used as a master data output and slave data input for the SPI interface.
18	PB4	Digital Pin	This pin act as a master clock input and slave clock output.
19	PB5	Digital Pin	This pin act as a master clock output and slave clock input for SPI.
20	AVcc	Positive Voltage	Positive voltage for ADC (power)

21	AREF	Analog Reference	Analog Reference voltage for ADC (Analog to Digital Converter)
22	GND	Ground	The Ground of the system
23	PC0	Analog Input	Analog input digital value (channel 0)
24	PC1	Analog Input	Analog input digital value (channel 1)
25	PC2	Analog Input	Analog input digital value (channel 2)
26	PC3	Analog Input	Analog input digital value (channel 3)
27	PC4	Analog Input	Analog input digital value (channel 4). This pin can also be used as a serial interface connection for data.
28	PC5	Analog Input	Analog input digital value (channel 5). This pin is also used as a serial interface clock line.

ATmega2560 pin configuration

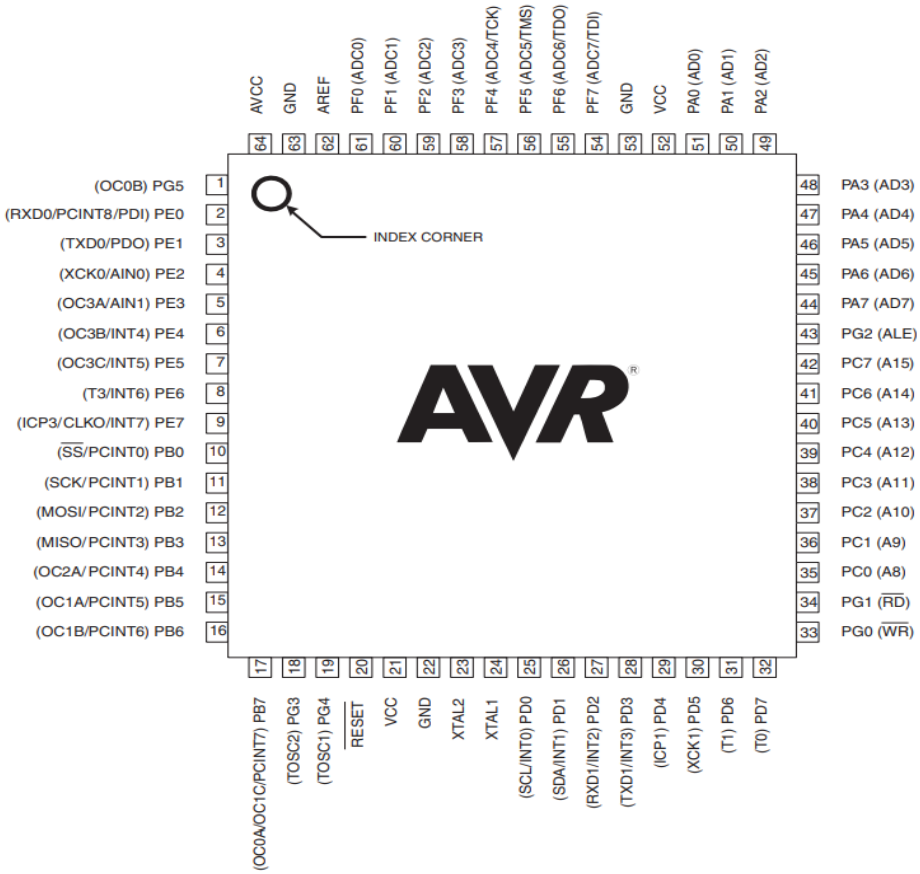

Figure 73 configuration of atmega2560

Table 39 pin descriptions of atmega2560

Pin Number	Pin Name	Mapped Pin Name
1	PG5 (OC0B)	Digital pin 4 (PWM)
2	PE0 (RXD0/PCINT8)	Digital pin 0 (RX0)
3	PE1 (TXD0)	Digital pin 1 (TX0)
4	PE2 (XCK0/AIN0)	-
5	PE3 (OC3A/AIN1)	Digital pin 5 (PWM)
6	PE4 (OC3B/INT4)	Digital pin 2 (PWM)
7	PE5 (OC3C/INT5)	Digital pin 3 (PWM)

8	PE6 (T3/INT6)	-
9	PE7 (CLKO/ICP3/INT7)	-
10	VCC	VCC
11	GND	GND
12	PH0 (RXD2)	Digital pin 17 (RX2)
13	PH1 (TXD2)	Digital pin 16 (TX2)
14	PH2 (XCK2)	-
15	PH3 (OC4A)	Digital pin 6 (PWM)
16	PH4 (OC4B)	Digital pin 7 (PWM)
17	PH5 (OC4C)	Digital pin 8 (PWM)
18	PH6 (OC2B)	Digital pin 9 (PWM)
19	PB0 (SS/PCINT0)	Digital pin 53 (SS)
20	PB1 (SCK/PCINT1)	Digital pin 52 (SCK)
21	PB2 (MOSI/PCINT2)	Digital pin 51 (MOSI)
22	PB3 (MISO/PCINT3)	Digital pin 50 (MISO)
23	PB4 (OC2A/PCINT4)	Digital pin 10 (PWM)
24	PB5 (OC1A/PCINT5)	Digital pin 11 (PWM)
25	PB6 (OC1B/PCINT6)	Digital pin 12 (PWM)
26	PB7 (OC0A/OC1C/PCINT7)	Digital pin 13 (PWM)
27	PH7 (T4)	-

28	PG3 (TOSC2)	-
29	PG4 (TOSC1)	-
30	RESET	RESET
31	VCC	VCC
32	GND	GND
33	XTAL2	XTAL2
34	XTAL1	XTAL1
35	PL0 (ICP4)	Digital pin 49
36	PL1 (ICP5)	Digital pin 48
37	PL2 (T5)	Digital pin 47
38	PL3 (OC5A)	Digital pin 46 (PWM)
39	PL4 (OC5B)	Digital pin 45 (PWM)
40	PL5 (OC5C)	Digital pin 44 (PWM)
41	PL6	Digital pin 43
42	PL7	Digital pin 42
43	PD0 (SCL/INT0)	Digital pin 21 (SCL)
44	PD1 (SDA/INT1)	Digital pin 20 (SDA)
45	PD2 (RXDI/INT2)	Digital pin 19 (RX1)
46	PD3 (TXD1/INT3)	Digital pin 18 (TX1)
47	PD4 (ICP1)	-

48	PD5 (XCK1)	-
49	PD6 (T1)	-
50	PD7 (T0)	Digital pin 38
51	PG0 (WR)	Digital pin 41
52	PG1 (RD)	Digital pin 40
53	PC0 (A8)	Digital pin 37
54	PC1 (A9)	Digital pin 36
55	PC2 (A10)	Digital pin 35
56	PC3 (A11)	Digital pin 34
57	PC4 (A12)	Digital pin 33
58	PC5 (A13)	Digital pin 32
59	PC6 (A14)	Digital pin 31
60	PC7 (A15)	Digital pin 30
61	VCC	VCC
62	GND	GND
63	PJ0 (RXD3/PCINT9)	Digital pin 15 (RX3)
64	PJ1 (TXD3/PCINT10)	Digital pin 14 (TX3)
65	PJ2 (XCK3/PCINT11)	-
66	PJ3 (PCINT12)	-
67	PJ4 (PCINT13)	-

68	PJ5 (PCINT14)	-
69	PJ6 (PCINT 15)	-
70	PG2 (ALE)	Digital pin 39
71	PA7 (AD7)	Digital pin 29
72	PA6 (AD6)	Digital pin 28
73	PA5 (AD5)	Digital pin 27
74	PA4 (AD4)	Digital pin 26
75	PA3 (AD3)	Digital pin 25
76	PA2 (AD2)	Digital pin 24
77	PA1 (AD1)	Digital pin 23
78	PA0 (AD0)	Digital pin 22
79	PJ7	-
80	VCC	VCC
81	GND	GND
82	PK7 (ADC15/PCINT23)	Analog pin 15
83	PK6 (ADC14/PCINT22)	Analog pin 14
84	PK5 (ADC13/PCINT21)	Analog pin 13
85	PK4 (ADC12/PCINT20)	Analog pin 12
86	PK3 (ADC11/PCINT19)	Analog pin 11
87	PK2 (ADC10/PCINT18)	Analog pin 10

88	PK1 (ADC9/PCINT17)	Analog pin 9
89	PK0 (ADC8/PCINT16)	Analog pin 8
90	PF7 (ADC7/TDI)	Analog pin 7
91	PF6 (ADC6/TDO)	Analog pin 6
92	PF5 (ADC5/TMS)	Analog pin 5
93	PF4 (ADC4/TCK)	Analog pin 4
94	PF3 (ADC3)	Analog pin 3
95	PF2 (ADC2)	Analog pin 2
96	PF1 (ADC1)	Analog pin 1
97	PF0 (ADC0)	Analog pin 0
98	AREF	Analog Reference
99	GND	GND
100	AVCC	VCC