POLITECNICO DI TORINO

III Facoltà di Ingegneria

Corso di Laurea in Ingegneria dell'Informazione

Tesi di Laurea Specialistica

UHF Communication System Development for Nanosatellite

Supervisor:
Prof. CLAUDO SANSOE'
Prof. DANTE DEL CORSO
Prof. LEONARDO REYNERI

Candidate:
Lv Shuai

September 2010

Acknowledgement

During the last 10 months, while working on this thesis, I had the opportunity to work in the Nanosatellite group in Politecnico di Torino with many talented people and now I had the chance to acknowledge them.

First of all, I should thank my supervisors, Prof. Sansoe' Claudio, Prof. Del Corso Dante and Prof. Reyneri Leonardo, who introduced me to this project and gave me so many guidances. Also, special thanks should go to Doc. Stefano Speretta, Doc. Maurizio Tranchero and other kind doctors. They gave me so many practical instructions.

In addition, I should thank my companions, Qiu Longjia, Yu Jian, Zhao Yingjie and Duan Wenjie, who ever gave me suggestions and encouragements when I encountered problems.

Torino, September 2th, 2010

Lv Shuai

Summary

Avionics for satellites is a market which is continuously growing in these years. The cost reduction enables many institutions to develop their own satellites. To evaluate the feasibility of COTS components in space projects, some departments of Politecnico di Torino developed a nanosatellite named PiCPoT. In order to greatly reduce cost for further, the group began to develop a true modular satellite—AraMiS, which allows a number of missions to share the same design.

A primary mission requirement of any satellite is the ability to exchange information with a ground based command station. Similar as PiCPot, AraMiS has two radio-frequency communication subsystems. One subsystem works in UHF (437MHz) band, the other one works in S (2.4GHz) band. The two communication subsystems are independent and their functions are interchangeable. Both channels implement a half-duplex protocol, sharing the same frequency for downlink and uplink.

The duty of this thesis is to develop the UHF communication subsystem. To get the compatibility with amateur radios, this communication subsystem needs realize AX.25 communication protocol. It consists of a micro-controller performing a TNC, a transceiver, a power amplifier and an antenna. The micro-controller is TI MSP430F149, which is low cost, low power and easy to operate. The transceiver is CC1020, which is true narrowband, low voltage and power and easy to configure. And it only needs a few external passive components. Still choose to use RFMD RF2175 as the power amplifier, which can provides an +34dBm output power, while using a helical antenna.

This paper copes with all the details to develop and verify the UHF communication subsystem. Chapter 1 is an introduction of this final project, addressing an general idea of the whole development process. In chapter 2, a brief description of a space radiation environment, which largely affects the satellites'

normal functionalites, is presented. From chapter 3 to chapter 5, development details are exactly described. In chapter 3, detail the hardware development. After evaluating several exiting solutions, our choice is addressed, including components selections, circuits realizations and interface descriptions. Software development is shown in chapter 4. The AX.25 protocol is firstly introduced, which involves how to transmit and receive packets based on this protocol. Thereafter, softwares performing hardware modules' functionalities and their relations to exchange information are interpreted in details. Since both hardware and software are designed, realization and test processes are demonstrated in chapter 5. The PCB realization procedures are addressed and experiments for test are established, while results are also reported. In chapter 6, a conclusion is given to conclude this final project.

Contents

List of Figures and Tables

List of Acronyms

AraMiS	Modular architecture for Satellites
SME	Small Medium Enterprise
COTS	Commercial Off The Shelf
UHF	Ultra High Frequency
LEO	Low Earth Orbiting
MOS	Metal-Oxide Semiconductor
SEE	Single Event Effects
SEU	Single Event Upset
BJT	Bipolar Junction Transistors
PCB	Printed Circuit Board
PA	Power Amplifier
MCU	Micro-Control Unit
OBC	On-Board Computer
USART	Universal synchronous and asynchronous receiver/transmitter
ISM	Industrial, Scientific and Medical
SRD	Short Range Device
PLL	Phase Locked Loop
SPI	Serial Peripheral Interface
NRZ	Non-Return to Zero
DCE	Data Communication Equipment
DTE	Data Terminating Equipment
TNC	Terminal Node Controller
HDLC	High-Level Data Link Control
FCS	Frame Check Sequence
CRC	Cyclic Redundancy Check

LFSR	Linear Feedback Shifted Register
NRZI	Non-Return to Zero Inverse
BER	Bit Error Rate
JTAG	Joint Test Action Group
KISS	Keep It Simple, Stupid

Chapter 1
Introduction

In current days, the industrial and academic interest in space and space-related activities is rapidly growing. A cost effective access to space would open a wide range of new opportunities and markets, especially for SMEs (Small Medium Enterprise) and Universities.

After developed PICPOT---a small satellite built with low cost Commercial Off The Shelf (COTS) components, electronic department in Politecnico di Torino wants to design a nanosatellite with true modular architecture (particularly in electronic subsystems) to go beyond the CubSat concept.

The project is aimed at:

- proving the feasibility of low-cost satellites using COTS (Commercial Off The Shelf) devices;
- developing a flight model of flexible and reliable nanosatellite with less than 25,000 Euros;
- training students in the field of avionics space systems;
- developing expertise in the field of low-cost avionic systems, both internally (university staff) and externally (graduated students will bring their expertise in their future work activity);
- gathering expertise and resources which were available inside our university around a common high-tech project.

The main idea of AraMiS (acronym for Modular Architecture for Satellites in Italian) project is the development of distributed and intercommunicating on-board units, built with COTS components, in order to increase fault tolerance and allow a graceful performance degradation, while keeping the costs at acceptable levels. The satellite can use as many basic modules as needed to perform the tasks of the mission. Since the same module design is used in several satellites, the AraMiS architecture

achieves an effective cost sharing between different missions [1]. Figure 1 below shows the model of AraMiS.

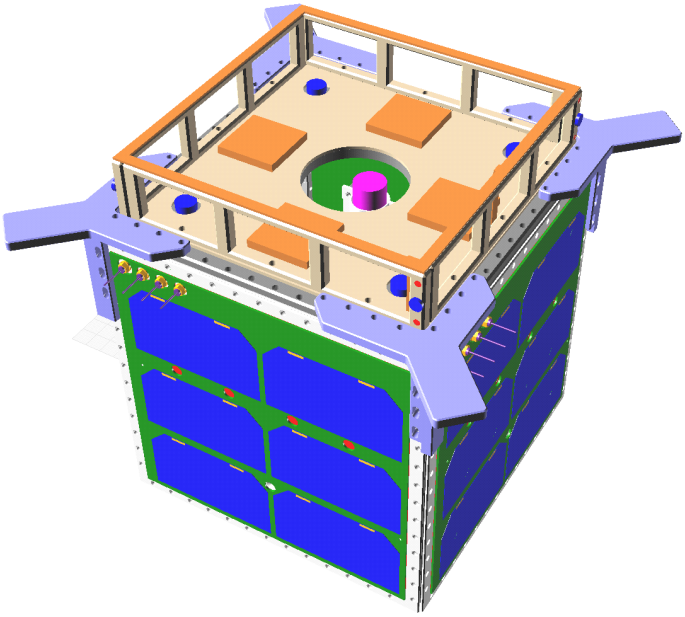

Figure 1.1 Model of AraMiS

AraMiS has two types tiles outside: power management and telecommunication (figure 2).

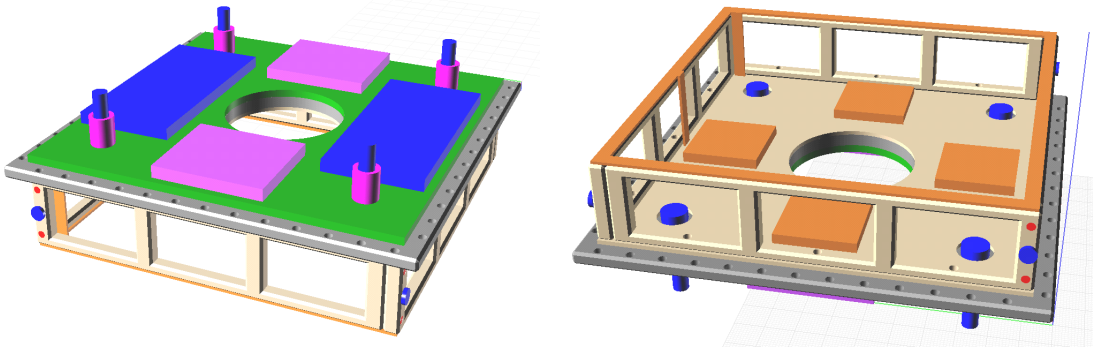

Figure 1.2 Model of telecommunication tile

The telecommunication tile is mostly composed of: 1) a microcontroller-based programmable transceiver; 2) a modem; 3) a power amplifier (for transmission) and low-noise amplifiers (for reception); 4) an antenna system.

In order to achieve fault tolerance, two different channels are used, in the bands allocated by ITU for satellite communications. The first channel lays in the UHF 437MHz band, and the second in the SHF 2.4GHz band. The data contents of the two

links are equivalent, thus providing two interchangeable possibilities to communicate with the satellite. To reduce occupied bandwidth, both channels implement an half-duplex protocol, sharing the same frequency for downlink and uplink.

To avoid the computational overhead of some of the operations required by AX.25 (scrambling and bit-stuffing), the transceiver of S-Band link uses a modulation scheme which is not directly compatible with amateur stations while the UHF downlink is designed to be compatible with the amateur G3RUH packet radio standard which uses the UI frame defined in the AX.25 standard, following the subset for APRS.

The task of this final project is to develop the UHF communication subsystem, both hardware and software. In the following chapters, this thesis will cope with most procedures of the UHF communication subsystem development process. In chapter 2, the space radiation effect, which may induce failures of electronic components, will be presented including radiation source, effect types and mitigation methods. In chapter 3, the whole hardware structure will be described in details. In chapter 4, we focus on software part, describing how to transmit and receive AX.25 protocol packets. In chapter 5, the test process is addressed. Three experiments were designed to test all functionalities both in hardware and software. In chapter 6, the conclusion is presented.

Chapter 2
Space Radiation Effects

Since we choose to use COTS components for low cost aspiration, these components are not space application dedicated and radiation robust. So we need be aware of the space radiation environment (in particular, Low Earth Orbiting) and the damage degree induced by the space radiation effects to our system. In this chapter, we discuss the space radiation environment, how it affects electronic devices and methods to mitigate these effects.

2.1 Space Radiation Environment [2]

Satellites operate in conditions that are much different from terrestrial weather. The space environment, just as any environment on Earth, contains phenomena that are potentially hazardous to humans and technological systems; however, many of these hazards involve plasmas and higher-energy electrons and ions that are relatively uncommon within Earth's atmosphere. These hazards exist in broad spatial regions that change with time. Typical satellite orbits cross many of these regions and spend a variable amount of time in each.

The space environment is populated with electrons and ionized atoms (ions). The unit of kinetic energy for these particles is the electron volt. At high energies (millions of electron volts), these particles have sufficient energy to ionize atoms in materials through which they propagate. At lower energies (below thousands of electron volts) their effects range from charge accumulation on surfaces to material degradation.

The space environment changes with time, often in unpredictable and undiscovered ways, making it a challenge to completely assess the hazards in any orbit.

Interplanetary Space

The sun and most planets in the solar system generate magnetic fields. The space outside the local effects of planetary magnetic fields contains its own population of particles. Several satellites near Earth continuously monitor the intensity of the particles and electromagnetic fields in interplanetary space. These and other space probes have shown that the radiation environment in the solar system is highly variable, but the consistent locations of intense radiation are the planetary magnetospheres.

For instance, Earth's magnetosphere is a teardrop-shaped cavity formed by the interaction of the solar wind with earth's magnetic field (figure 2.1).

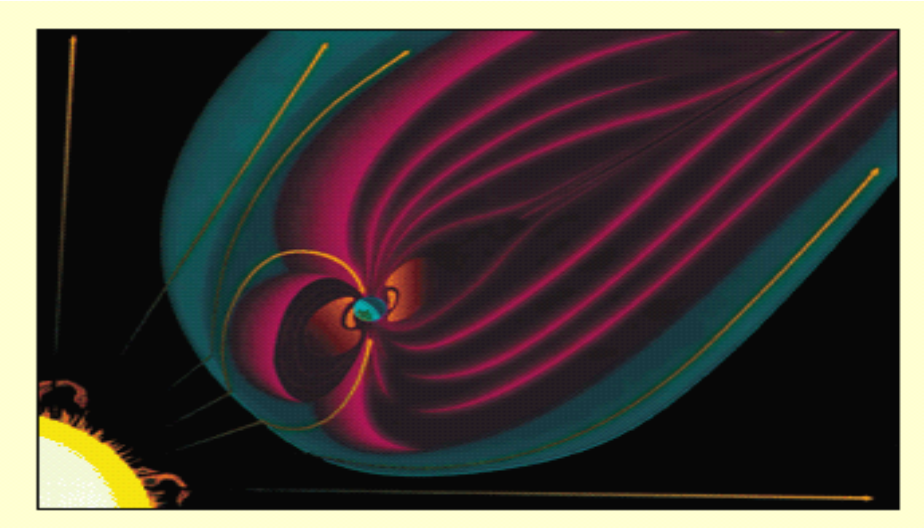

Figure 2.1 Earth's teardrop-shaped magnetosphere

Solar Wind

Most of the particles in interplanetary space are in the form of a hot, ionized gas called the solar wind; it flows radially from the sun with a speed at Earth that varies from about 300 to 1000 kilometers per second, representing a mass loss of about 10^{14} kilograms per day.

Solar Energetic Particles

Many highly variable sources produce interplanetary particles with energies

typically between 10 thousand and 100 million electron volts. These energetic particles originate in acceleration processes in the solar atmosphere, sometimes close to the sun and sometimes beyond Earth's orbit. The transient nature of these particle populations is directly linked to the sun's activity.

Galactic Cosmic Rays

Galactic cosmic rays are the highest-energy particles in the solar system and they originate somewhere outside the solar system.

Earth's Magnetosphere

Earth's magnetic field establishes a volume of space within which the magnetic field dominates charged particle motion. Close to Earth, the magnetic field is roughly a magnetic dipole that is tilted 11.5 degrees from the rotational axis and offset from the center of the planet. For most purposes, the dipole approximation is poor, and there are more sophisticated models that account for the steady changes of the central field as well as the dynamic outer boundaries.

The magnetosphere contains a mixture of plasmas with incredibly diverse sources. Some populations of charged particles are trapped within the magnetosphere while others vary on many time scales. The magnetosphere has its own weather, with complex processes of particle transport and acceleration during geomagnetic storms that contribute to surface charging and internal charging of spacecraft.

Stable trapping of particles occurs, given the right combination of particle charge, energy, and magnetic field strength. As these particles are trapped on time scales ranging from days to years, they execute their gyration, bounce, and drift motions around Earth, resulting in spatial zones of trapped radiation known as the Van Allen belts (figure 2.2).

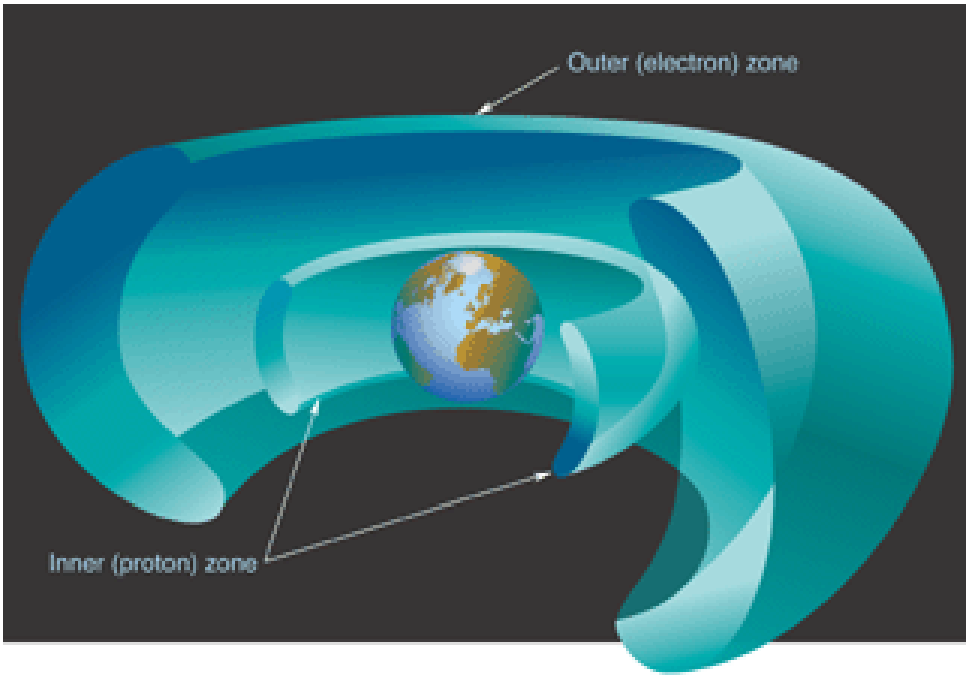

Figure 2.2 Diagram of Van Allen radiation belts

Our satellite will operate in Low Earth Orbiting (LEO), which lies in the inter zone of the Van Allen radiation belts. The Van Allen radiation belts and typical satellite orbits are shown in figure 2.3.

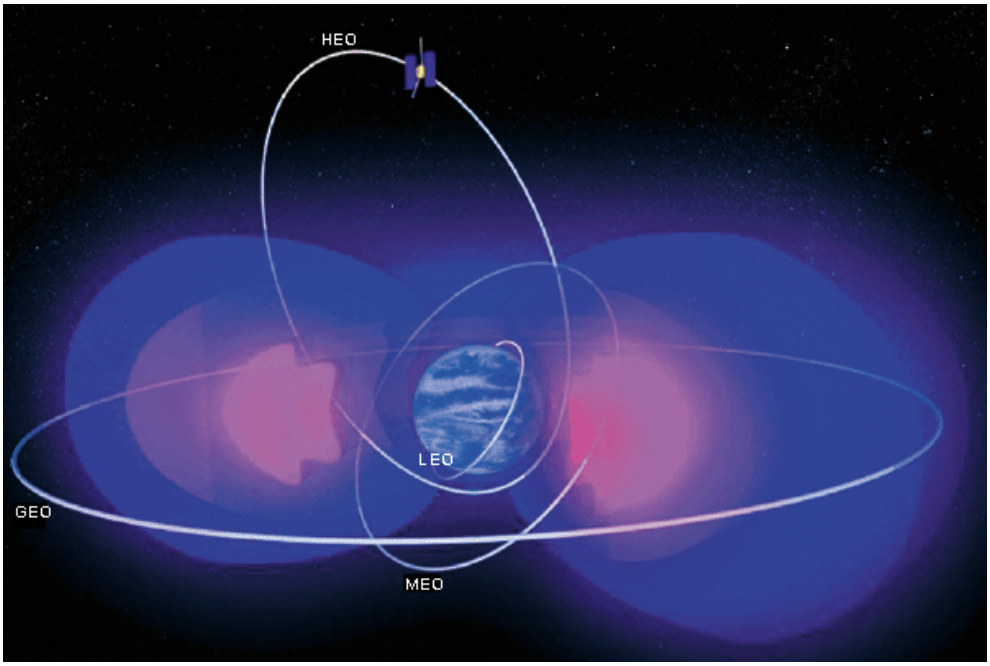

Figure 2.3 The Van Allen radiation belts and typical satellite orbits

2.2 Space Radiation Effect

The general radiation effects and their origins are shown in table 1.

Table 2.1 Types of radiation effects and the corresponding origins[3]

Types of radiation effect	Origin
Total Ionizing Dose (TID)	Trapped protons and electrons; Solar protons
Single Event Effects (SEE)	Trapped and solar protons; Heavier ions from Galactic cosmic ray and solar events; Neutrons
Displacement Damage	Protons and electrons
Spacecraft Charging	Surface for plasma; Deep dielectric for high energy electrons

Total Ionizing Dose

When incident radiation enters a semiconductor solid material such as silicon, an electron−hole pair may be created if an electron in the valence band is excited across the band gap into the material's conduction band [4]. Electron−hole pairs generated in the gate oxide of a metal-oxide semiconductor (MOS) device such as a transistor are quickly separated by the electric field within the space charge region (figure 2.4). The electrons quickly drift away while the lower-mobility holes drift slowly in the opposite direction. Digital microcircuits are affected because trapped charge can shift MOS transistor threshold voltage, which is a key device parameter. Other influences may be leakage current, timing skew and function failures.

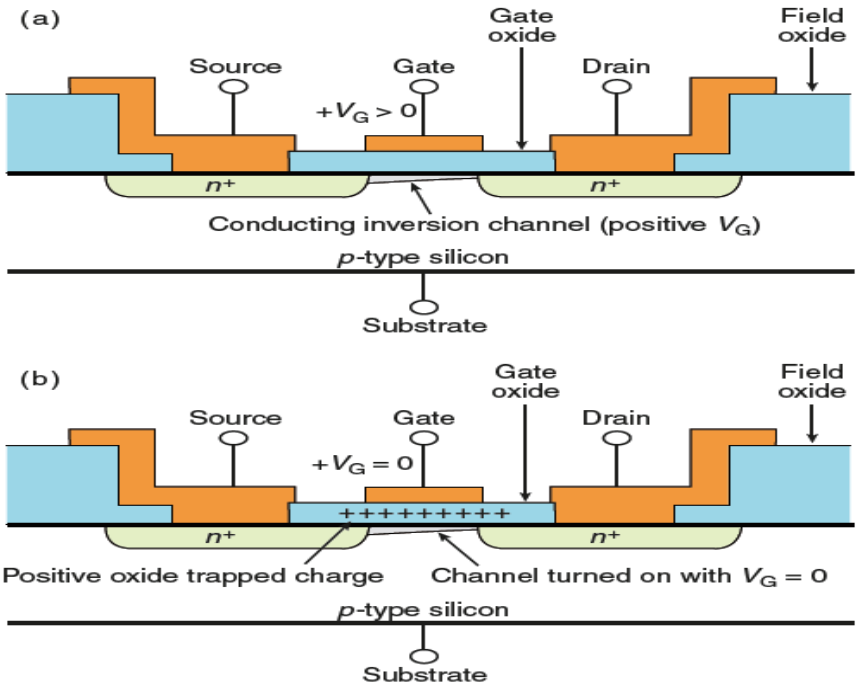

Figure 2.4 Schematic of an n-channel MOSFET illustrating the basic effect of total ionization induced charging of the gate oxide. Normal operation (a) and post irradiation (b) show the residual trapped positive charge (holes) that produces a negative threshold voltage shift

Single Event Effects

If the amount of charge collected at a junction exceeds a threshold, then an SEE can be initiated. An SEE can be destructive or nondestructive. Destructive effects result in catastrophic device failure. Nondestructive effects result in loss of data and/or control. SEEs are generated through several mechanisms. The basic SEE mechanism occurs when a charged particle travels through the device and loses energy by ionizing the device material. Other physical charge generation mechanisms include elastic and inelastic nuclear reactions.

The charge generated by this single strike is collected, producing spurious voltage on a “sensitive” node that causes a circuit-level effect (figure 2.5).

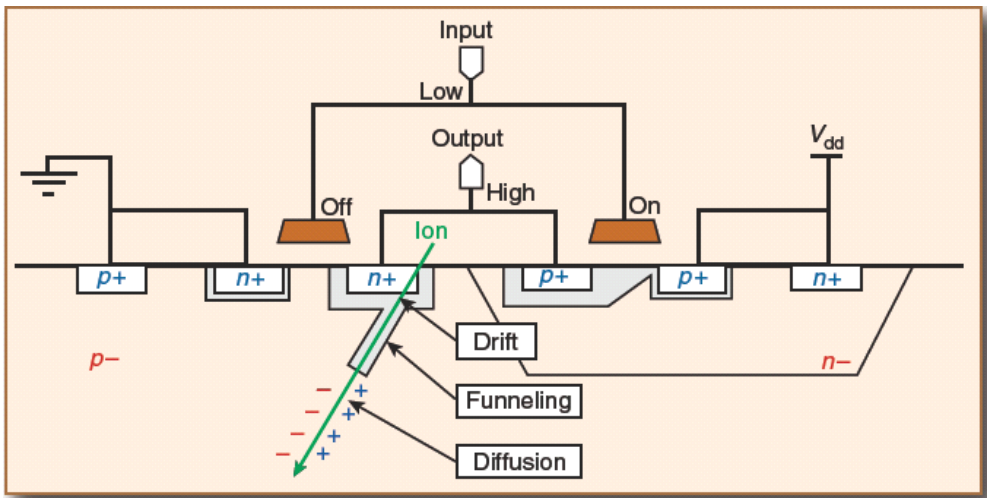

Figure 2.5 Schematic of a heavy ion strike on the cross-section of a bulk CMOS memory cell

A typical nondestructive case is Single Event Upset (SEU). It is the change of state of a bistable element, typically a flip-flop or other memory cell, caused by the impact of an energetic heavy ion or proton. Single Event Latch-up is a typical destructive case. Integrated circuits fabricated with complementary MOS (CMOS) fabrication processes are very widely used in space electronics. These chips inherently include parasitic bipolar junction transistors (BJTs) formed by closely located CMOS structures that under normal conditions form the integrated circuits n-channel and p-channel transistors (figure 2.6).

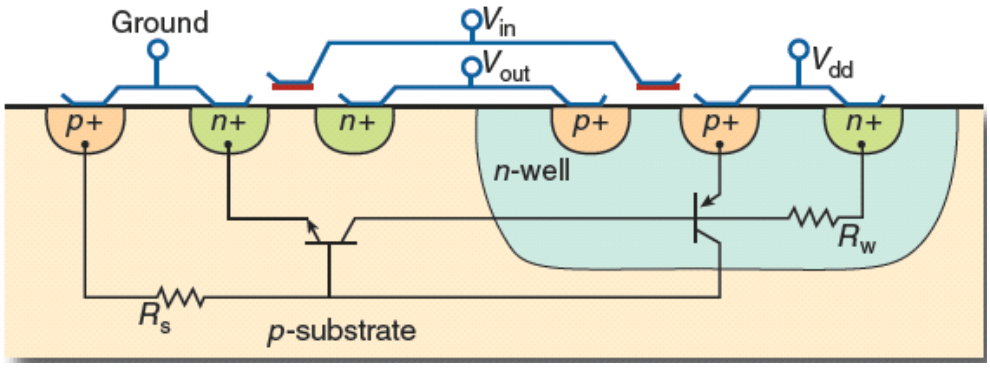

Figure 2.6 Bulk CMOS inverter architecture cross-section showing the parasitic bipolar SCR structure that forms, making it susceptible to SEL

Displacement Damage

Radiation particle such as electrons, protons and neutrons scatter off lattice ions, locally deforming material structure (figure 2.7).

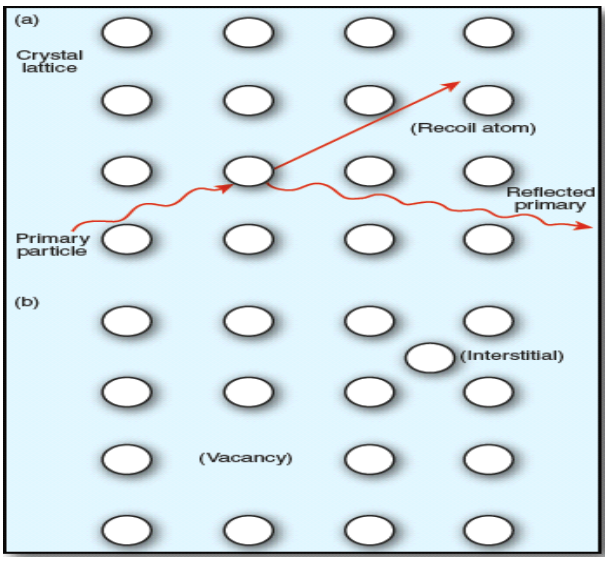

Figure 2.7 schematic of atomic displacement damage in crystal solid

The amount of displacement damage is dependent on the incident particle type, incident particle energy and target material.

2.3 Mitigation Methods

Since we know space radiation effects are very harmful for electronic systems of spacecrafts, we must apply some mitigation methods. Basically, what we can do are as follows:

- Limit current or turn off circuits with excessive current consumption
- Turn off devices when not in use
- Part de-rating and increase operating margin
- Shielding
- Change operating schedule in response to space weather
- Radiation-hard parts selection
- Redundancy
- Error detection and correction
- Memory scrubbing

Chapter 3
Hardware Development

This chapter discusses the hardware development of UHF communication subsystem for AraMiS. After evaluating the existed solutions, our choice is presented, describing components selection, system circuits and interfaces.

3.1 Existing Solutions [5]

To date, based on the type of hardware used, communication subsystem of nanosatellites can be categorized into three classes: 1) COTS devices; 2) modified handhelds; 3) custom hardware.

- COTS devices

Directly purchasing a COTS space-rated transceiver is one choice, which simplifies design of the subsystem. Most of the protocols and modulations are proprietary and device specific, requiring an identical radio at the command ground station. However, this kind device is usually expensive, heavy and big for nanosatellite.

- Modified handhelds

With this approach, handheld amateur radios are modified to be a communication subsystem. Amplifier, transceiver and even TNC sometimes are functionally integrated in one circuit board, which largely simplifies the design process. But it is hard to fit this kind system which weights a lot and has a large size into a small space. The power consumption is also an important issue for less ability to disable individual devices.

- Custom-built

In some projects, people decide to build the whole subsystem out of individual components. It is hard to get a satisfied performance due to the inherent difficulties of

RF board design and time consuming to test, yet it has the most flexibility. Another obvious advantage is lower power consumption since it can easily enable and disable individual components. The table (Table 3.1) below shows a summary of different communication subsystems of some nanosatellite projects.

Table 3.1 Summary of communication subsystem

Project	Transceiver	Frequency (MHz)	TNC	Protocl	Baud rate Modulation
AAU1	Wood & Douglas SX450	437.475	MX909	AX.25	9600 Baud GMSK
DTUsat-1	RFMD RF2905	437.475		AX.25	2400 Baud FSK
CO-57	Nishi RF Lab	436.845	PIC16C622	AX.25	1200 Baud AFSK
UWE-1		473.505	Integrated	AX.25	1200/9600 Baud AFSK
CAPE1	CC1020	435.245	PIC16LF452	AX.25	9600 Baud FSK
MAST	MicrohardMHX-2400	2400	Integrated	Proprietary	15 kbps

3.2 Our Solution

To get the compatibility with amateur radios, the UHF communication subsystem should be capable of transmitting and receiving AX.25 formatted packets. For the power, cost and weight issues, we cannot just purchase a space-rated transceiver or modify a handheld. Moreover, for the aim at teaching, we choose to build a custom system using individual COTS devices, which will be integrated on one small PCB. The diagram below (figure 3.1) shows the final designed structure.

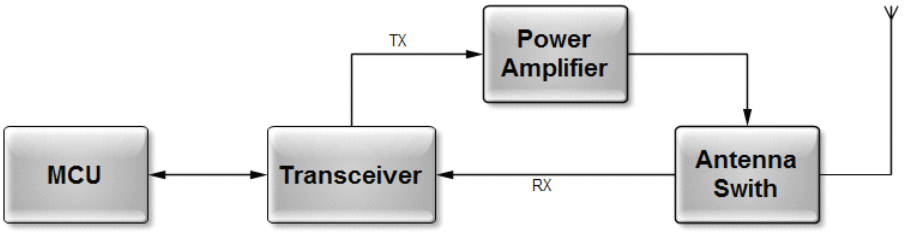

Figure 3.1 Diagram of UHF communication subsystem

This subsystem consists of four parts, MCU (Micro-Control Unit), Transceiver, PA (power amplifier), and antenna. All the devices are selected from COTS components. The MCU performs as a real TNC through software to transmit and receive AX.25 protocol packets and it is controlled by OBC (On-Board Computer). To save bandwidth, the subsystem works in half-duplex mode. The power amplifier is disabled to save power when wording in receiving mode.

3.3 Components, Circuits and Interfaces

This section describes the components selections of the communication subsystem in details. And their interfaces are also presented.

3.3.1 Components selection

Micro-controller Unit (MCU): MSP430F149

The MSP430 is a mixed-signal microcontroller family from Texas Instruments. Built around a 16-bit CPU, the MSP430 is designed for low cost, and specifically, low power consumption embedded applications. The electric current drawn in idle mode can be less than 1 micro amp. TI provides robust design support for the MSP430 microcontroller including technical documents, training, tools, and software, which decreases design time.

The MSP430x1xx Series is the basic generation without an embedded LCD controller. These Flash or ROM based Ultra-Low Power MCUs offer 8 MIPS, 1.8–3.6 V operation, up to 60 KB Flash, and a wide range of high-performance analog and

intelligent digital peripherals. Its functional block diagram is shown below (figure 3.2).

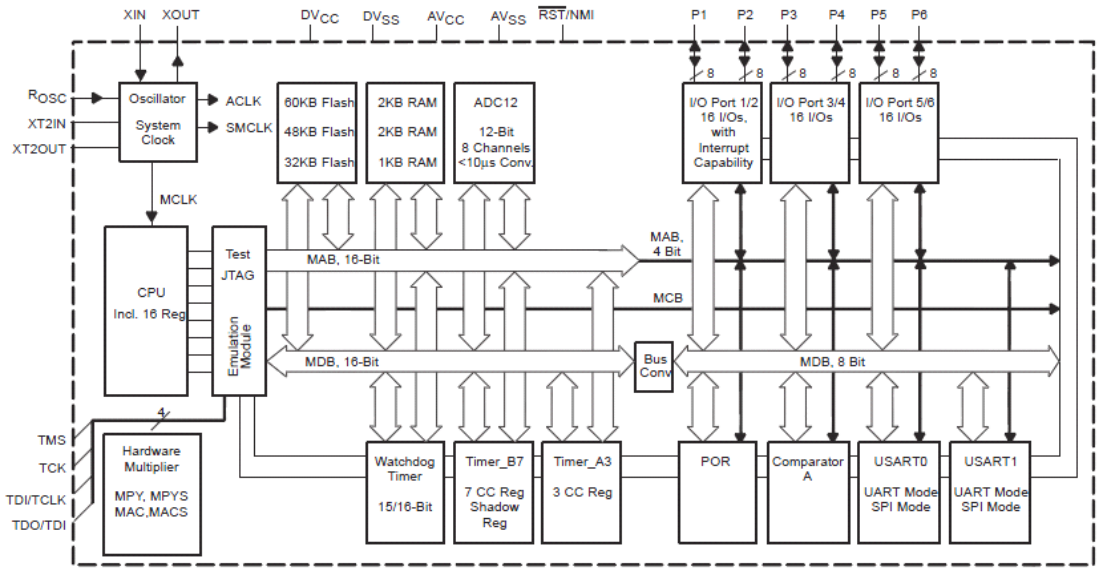

Figure 3.2 Functional block diagram of MSP430x14x[6]

The MSP430149 has two built-in 16-bit timers, a fast 12-bit A/D converter, two universal serial synchronous/asynchronous communication interfaces (USART), 48 I/O pins, 2 KB RAM and 60 KB Flash. The flash can be easily programmed and erased through JTAG interface, which make debugging convenient.

Transceiver: Chipcon CC1020

CC1020 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 402, 424, 426, 429, 433, 447, 449, 469, 868 and 915 MHz, but can easily be programmed for multi-channel operation at other frequencies in the 402 - 470 and 804 - 940 MHz range. What's more, it is very suitable for narrowband application with 12.5 KHz or 25 KHz channel spacing [7].

In typical applications, CC1020 is complied with a micro-controller and only a few external passive components. The interfaces between CC1020 and the needed micro-controller are very simple (figure 3.3).

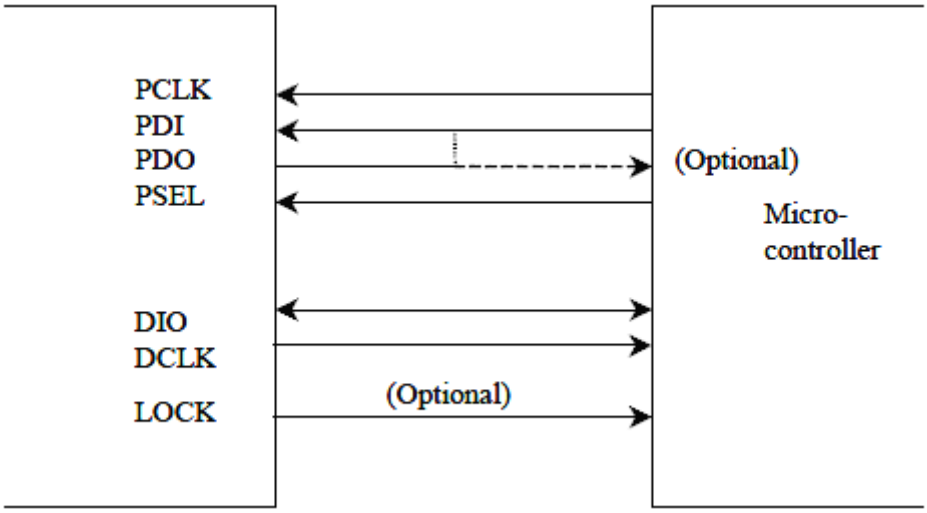

Figure 3.3 Interface between CC1020 and micro-controller

The main operating parameters including the component values needed for the input/output matching circuit, the PLL (Phase Locked Loop) loop filter and the LC filter can be easily generated based the user's selections by a software program SmartRF® Studio (figure 3.4) provided by Chipcon. And these parameters can be programmed through a serial bus interfacing to a micro-controller.

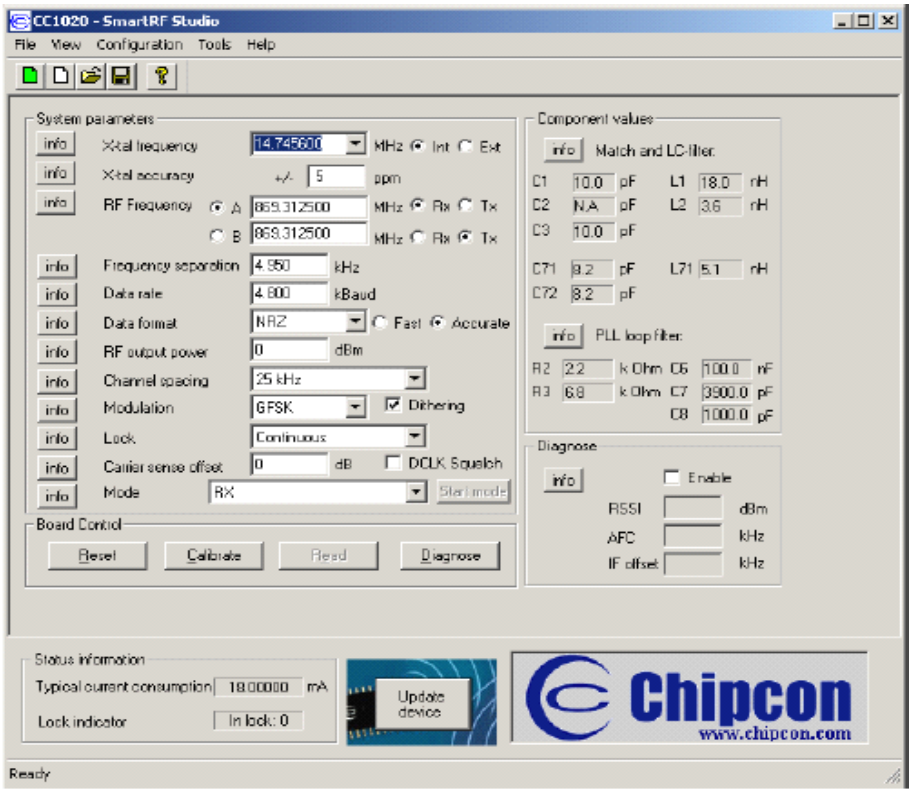

Figure 3.4 SmartRF® Studio user interface

Power Amplifier: RFMD RF2175

This power amplifier has been tested in PiCPot project. By now, we have on reason to change it.

3.3.2 Circuits Realization

Power Supply

To power the entire subsystem steadily, we use a low current three-terminal adjustable positive voltage regulator (LM317), which generates a constant output voltage. The application circuit (figure 3.5) is very simple and only requires two external resistors to set the output voltage.

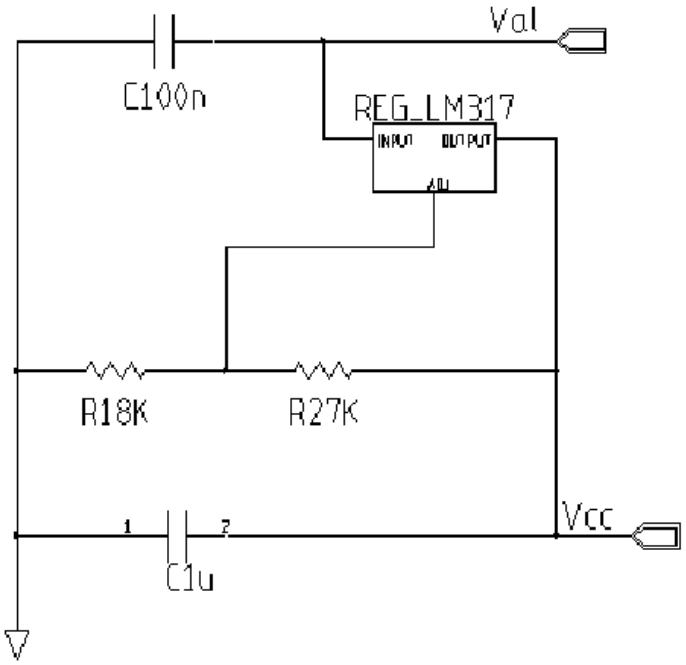

Figure 3.5 LM317 application circuit

The approximation equation to calculate is

$$V_{out} = 1.25 \times (1 + \frac{R_2}{R_1})$$

Here, R2 and R1 are 27 KΩ and 18 KΩ, respectively. Thus the output voltage is set to 3.125 V approximately, which is suitable to power the MCU and tranceiver. In practice, the value is 3.27 V.

Connections between MSP430 and CC1020

These connections can be divided into two parts, configuration interface and signal interface as shown in Figure 3.6.

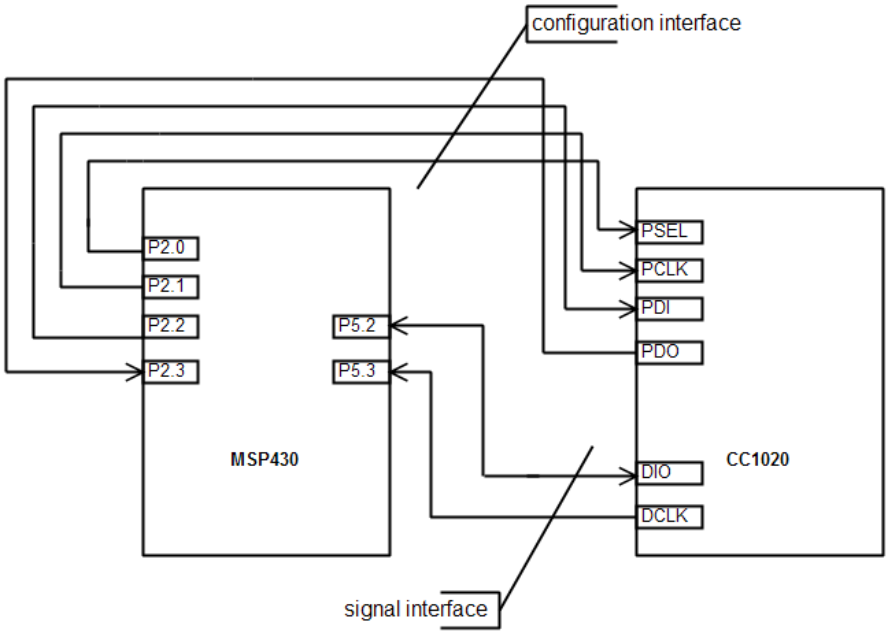

Figure 3.6 Connections between MSP430F149 and CC1020

The configuration interface performs as a 4-wire SPI bus. When configuring, the MCU is the master and CC1020 is its slave. In CC1020, there are 8-bit configuration registers, each addressed by a 7-bit address. A Read/Write bit initiates a read or write operation.

During each cycle, 16 bits are sent in series following this order: Address (7 bits), Read/Write (1 bits) and Data (8 bits). During the write cycle, the 16 bits are sent on the PDI line, and PDI should be configured as output by the MCU. During the read cycle, the Address bits and Read bit are sent by the MCU firstly on the PDI line, then CC1020 outputs the corresponding Data 8 bits to the MCU through the PDO line and PDO should be configured as input by the MCU. Between each read or write cycle, PSEL must be set high. The write and read operations are illustrated in Figure 3.7 and Figure 3.8, respectively.

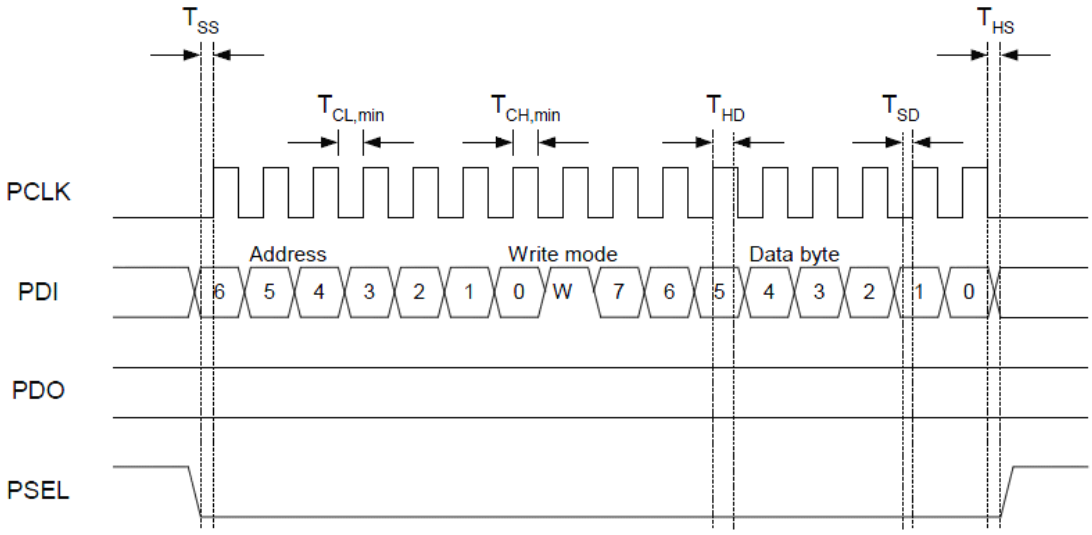

Figure 3.7 Configuration registers write operation

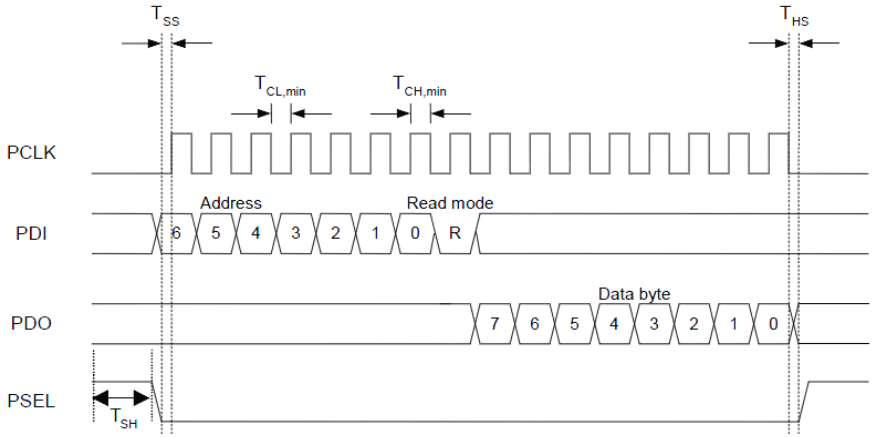

Figure 3.8 Configuration registers read operation

The signal interface has 2 wires. Here, CC1020 acts as the master and the MCU is its slave. CC1020 can be configured for three different data formats: synchronous NRZ (Non-Return to Zero) mode, synchronous Manchester encoding mode and transparent asynchronous UART mode. We choose to use the synchronous NRZ mode data format. Both in transmit mode and receive mode, CC1020 provides a clock (9600 bps for us) at DCLK and data at DIO should be always clocked at the rising edge of DCLK. DIO is used as data input of CC1020 in transmit mode and data output of CC1020 in receive mode. The whole timing diagrams are shown in Figure 3.9, demonstrating both transmit and receive processes.

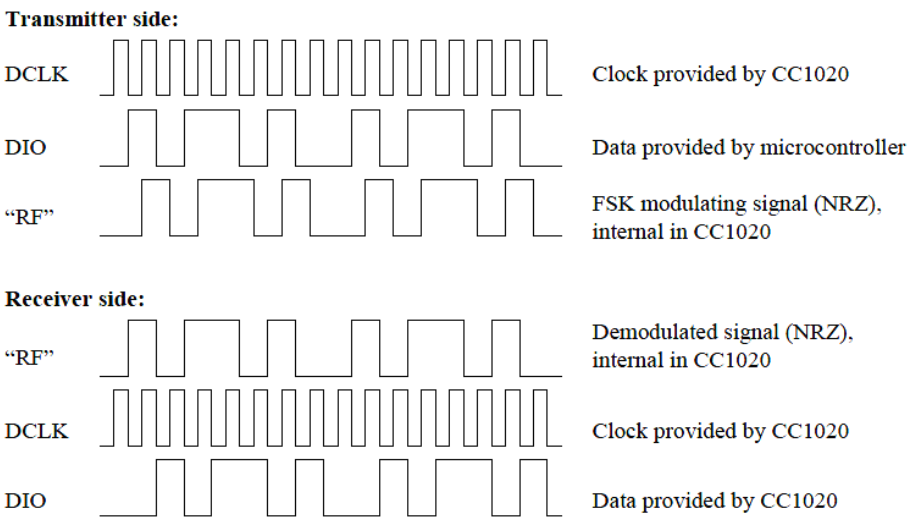

Figure 3.9 Synchronous NRZ mode Transmit & Receive processes

The clock frequency is decided by CC1020 external oscillator and several related registers. In our case, we use a 14.7456 MHz (optimum value referring to datasheet) oscillator and enable CLOCK A register. According to the equation

$$BaudRate = \frac{f_{xosc}}{8 \bullet (REF\_DIV + 1) \bullet DIV1 \bullet DIV2},$$

we set REF_DIV, MCLK_DIV1 and MCLK_DIV2 of CLOCK A to 1 (001), 48(110), and 2 (01) to get 9600 baud. That means setting register CLOCK A to 0x39.

CC1020 application circuit

To manage CC1020 to work properly, a few external passive components with fitting values should be mounted besides correct configurations. The values of these components can be easily calculated with the help of SmartRF® Studio. The application is shown below (figure 3.10).

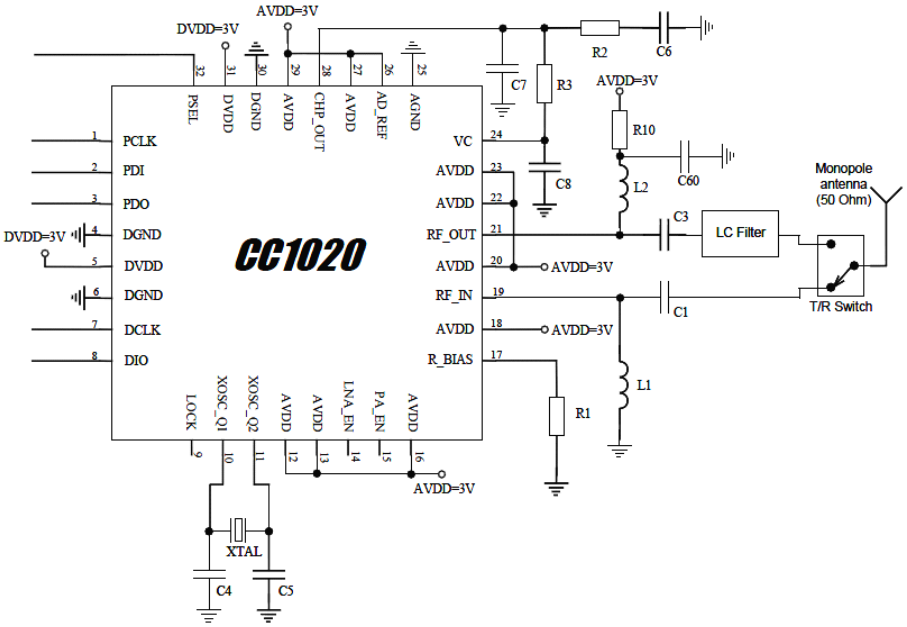

Figure 3.10 CC1020 application circuit

The circuit is realized identically except omitting the LC Filter part next to T/R Switch. In this schematic, L_1 and C1 is the input match for the receiver; L_1 is also a DC choke for biasing;L_2 and C_3 are used to match the transmitter to 50 Ω; The PLL loop filter consists of two resistors (R_2 and R_3) and three capacitors (C_6-C_8); An external crystal with two loading capacitors (C4 and C5) is used for the crystal oscillator. A very important part which is not shown is power supply decoupling and filtering capacitors. To get optimum performance for narrowband applications, the values and positions of these capacitors closely follow the CC1020EMX reference design, which can be found in TI website.

Antenna Switch

This section of the subsystem includes switch and antenna two parts. Figure 3.11 shows the switch circuit. The switch is actually a relay. Its control part is a MOSFET switch. The control signal is from the MCU.

The relay has four terminals. The VCC is used to power it. The NC and NO are used to connect RFIN and RFOUT pins of CC1020 standing for receive and transmit channel. The COM terminal connects the antenna.

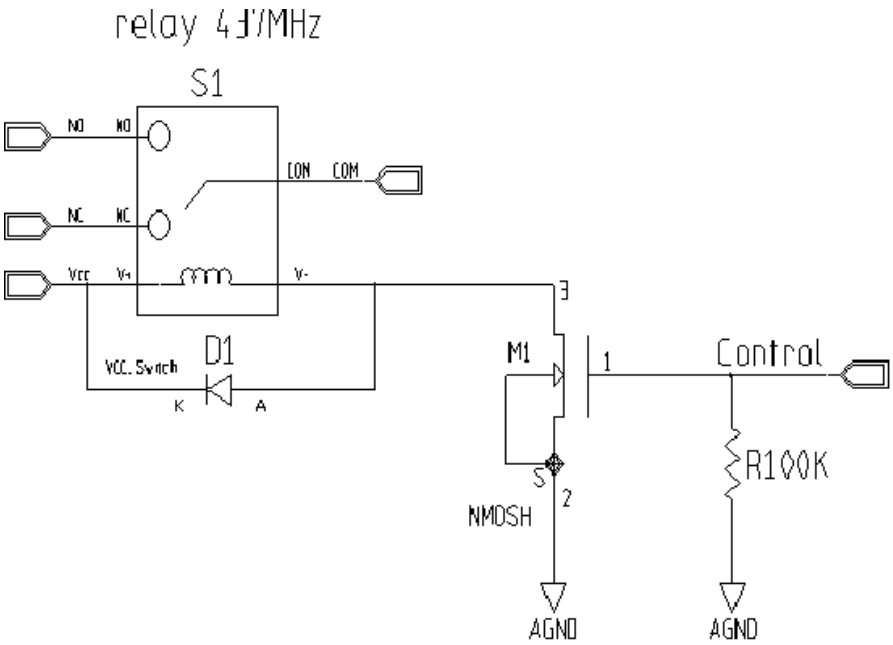

Figure 3.11 Switch circuit

3.3.3 Interfaces

MCU (MSP430F149) and OBC

The MSP430F149 exchanges data with OBC with a serial bus. On the micro-controller side, its module USART0 is used. USARTs of MSP430F149 can be configured to work in UART (Universal Asynchronous Receiver/Transimitter) mode, SPI (Serial Peripheral Interface) mode, I^2C (Inter-Integrated Circuit) mode or just basic digital I/O terminals[8]. In this case, USART0 is working in UART mode. In the later test procedure, the MCU is connected to PC with a RS232 bus.

The UART block diagram is shown in the figure below (figure 3.5). In UART mode, the USART0 connects the MSP430 to an external system via two external pins, URXD and UTXD. And this mode is selected when SYNC bit is cleared also as shown in the same figure.

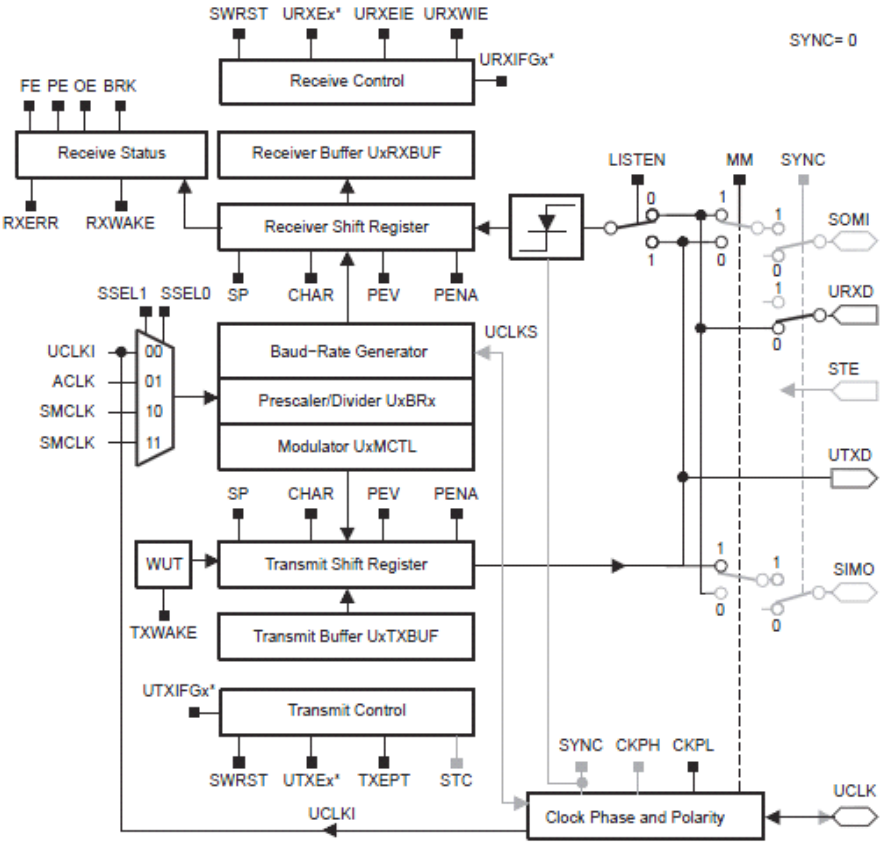

Figure 3.12 USART block diagram (UART mode)

The UART character format, shown in Figure 3.6, consists of a start bit, seven or eight data bits, an even/odd/no parity bit, an address bit (address-bit mode), and one or two stop bits. The bit period is defined by the selected clock source and setup of the baud rate registers. Timing for each character is based on the selected baud rate of the USART.

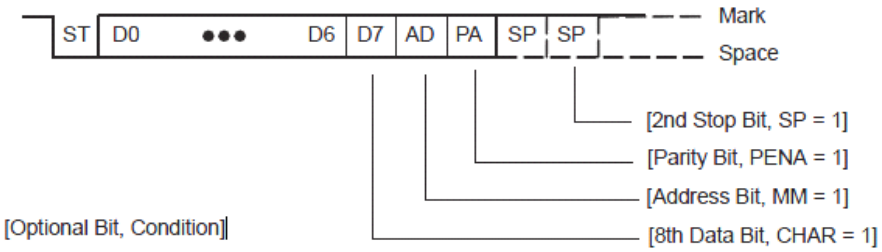

Figure3.13 Date format of UART mode

To select baud rate in UART mode, what we need to do is to configure the 16-bit register storing an integer N. The register is actually the combination of two 8-bit registers, U0BR0 storing 8 LSB (Least Significant Bit) and U0BR1 storing 8 MSB

(Most Significant Bit).

$$N = [\frac{BRCLK}{BaudRate}]$$

where BRCLK the frequency of the crystal oscillator of the micro-controller and [] means just taking the integral part.

For instance, if BRCLK is 4 MHz and baud rate is 9600 bps, N should be 416. And 416 is 0x01A0, thus U0BR0 should be configured to be 0xA0 and U0BR0 should be configured to be 0x01. When BRCLK is 8 MHz, we should write 0x41 and 0x03 to U0BR0 and U0BR1, respectively.

To manage the UART0 module to work properly, besides U0BR0 and U0BR1, there are other registers must be written into right values to implement selected functions. Table 3.2 shows these kind registers and their proper values for our case. Our main specification is 9600 baud; UART mode and no parity, 8 bits length and 1 stop bit data format.

Table 3.2 USART0 control and status registers

Register	Short Form	Configuration Value
Pin selection	P3SEL	0x30
USART control	U0CTL	0x10
Transmit control	U0TCTL	0x39
Modulation control	U0MCTL	0x00
Baud rate control 0	U0BR0	0x03
Baud rate control 1	U0BR1	0x41
SFR module enable	ME1	0xC0

MCU and CC1020

As referred before, interfaces between can be divided into two parts, configuration interface and signal interface. Connection details can be referred to 3.3.2. Here, we just give the cons and pros of SPI bus.

The advantages are:

- Full duplex communication
- Higher throughput than I²C or SMBus
- Complete protocol flexibility for the bits transferred
- Not limited to 8-bit words
- Arbitrary choice of message size, content, and purpose
- Extremely simple hardware interfacing
- Typically lower power requirements than I²C or SMBus due to less circuitry (including pullups)
- No arbitration or associated failure modes
- Slaves use the master's clock, and don't need precision oscillators
- Slaves don't need a unique address -- unlike I²C or GPIB or SCSI
- Wires in board layouts or connectors, much less than parallel interfaces
- At most one "unique" bus signal per device (chip select); all others are shared
- Signals are unidirectional allowing for easy Galvanic isolation

The disadvantages are:

- Requires more pins on IC packages than I²C, even in the "3-Wire" variant
- No in-band addressing; out-of-band chip select signals are required on shared buses
- No hardware flow control (but master can delay the next clock edge to slow the transfer rate)
- No hardware slave acknowledgment (the master could be "talking" to nothing and not know it)
- Only handles short distances compared to RS-232, RS-485, or CAN-bus
- Supports only one master device
- No error-checking protocol is defined
- Generally prone to noise spikes causing faulty communication
- Without a formal standard, validating conformance is not possible

The following section only describe signal interface for further. The micro-controller uses P5.2 and P5.3 to communicate with DCLK and DIO of CC1020. Port 5 is actually the USART1 module of MSP430F149. And USART1 is configured

differently in different communication subsystem working modes (transmit or receive).

1) Transmit Mode

As mentioned before, USARTs of MSP430 can operate in four modes. When the whole subsystem is working in transmit mode, USART1 operates in SPI mode (so SYNC is set high) and performs as a slave while CC1020 works as an external master.

The configuration is shown in Figure 3.14.

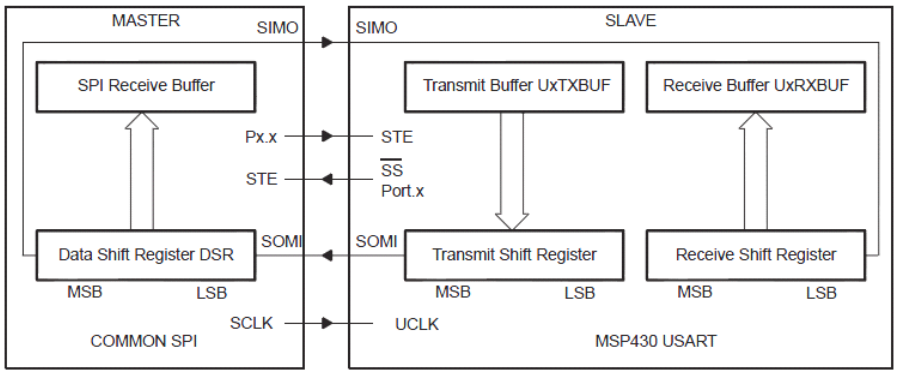

Figure 3.14 USART1 Slave and SPI mode

UCLK is used as the input for the SPI clock and must be supplied by the external master CC1020. The data-transfer rate is determined by this clock and not by the internal baud rate generator of USART1, thus there is no need to configure registers U1BR0 and U1BR1. Typical applications are 3 or 4 wires connections, but here only SOMI and UCLK are needed for we just have one slave and always enabling transmit or receive operations between the MCU and CC1020.

The polarity and phase of UCLK are independently configured via the CKPL and CKPH control bits of the USART. Timing for each case is shown in Figure 3.15.

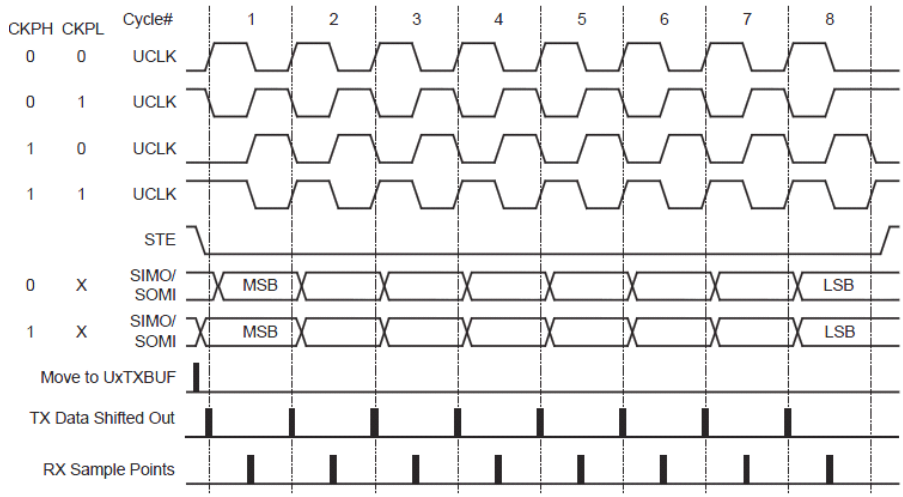

Figure 3.15 USART SPI timing

We configure CKPH and CKPL to be 0 and 1 to let TX data shifted out at the falling edge. Still, we need to configure control and status registers properly (Table 3.3).

Table 3.3 UART1 control and status registers

Register	Short Form	Configuration Value
Pin selection	P5SEL	0x0E
USART control	U1CTL	0x14
Transmit control	U1TCTL	0x83
Modulation control	U1MCTL	0x00
SFR module enable	ME2	0x10

2) Receive mode

When the subsystem is working in receive mode, the module function of Port 5 is disabled which means P5.0-P5.7 performs as normal I/O pins. This can be easily done by configuring ME2 as 0x00. Both UCLK and SOMI should be configured as input.

Since Port 5 performs as digital I/O, registers P5IN, P5OUT, P5DIR and P5SEL are useful. Each bit in P5IN register reflects the value of input signal of the corresponding I/O pin. Bit 0 means that input is low and bit 1 means that input is high.

Each bit in P5OUT is the output signal on the corresponding pin when the pin is configured as output direction. Each pin's direction is decided by the corresponding bit stored in P5DIR. Bit 0 means input direction and bit 1 means output direction. Because port pins are often multiplexed with other peripheral module functions, PxSELs are used to enable or disable I/O function of each pin. Bit 0 means enabling I/O function and bit 1 stands for peripheral module function.

Here, we only use P5.2 (UCLK) and P5.3 (SOMI) of MSP430F149. It is receive mode. So, both UCLK and SOMI should be configured as I/O function and input direction. Through the software, write both P5SEL and P5DIR as 0x00, respectively. At the same time, ME2 is set to 0x00 to disable module function and U1CTL is also reset by software to disable SPI.

Unused I/O pins should be configured as I/O function, output direction, and left unconnected on the board to reduce power consumption.

Chapter 4
Software Development

This chapter presents the details of software development. Since the UHF communication is dedicated to be compatible with amateur radios, AX.25 protocol and G3RUH packet radio standard (a standard way to scramble and descramble amateur radio packets) are addressed first of all. Then, modules realized by software and their relations are discussed. And the final section provides descriptions of functions of all modules.

4.1 AX.25 protocol and G3RUH standard

To design a UHF communication subsystem compatible with amateur radios so that every amateur radio over the world can communicate with AraMiS, the software must be capable of transmit and receive AX.25 protocol packets. The radio we are using is PK96 and it is based on the G3RUH packet radio standard. Thus the transmit and receive methods are coherent to this standard.

4.1.1 AX.25 protocol

AX.25 is a data link layer protocol derived from the X.25 protocol suite and designed for use by amateur radio operators. It is used extensively on amateur packet radio networks. The AX.25 version 2.2 Link-Layer Protocol provides this service, independent of the existence of any upper layer.

Most link-layer protocols assume that one primary (or master) device (generally called a Data Communication Equipment, or DCE), is connected to one or more secondary (or slave) device(s) (usually called a Data Terminating Equipment, or DTE). This type of unbalanced operation is not practical in a shared RF amateur radio environment. Instead, AX.25 assumes that both ends of the link are of the same class,

thereby eliminating the two different classes of devices. In this protocol specification, the phrase Terminal Node Controller (TNC) refers to the balanced type of device found in amateur packet radio [9].

Link layer packet radio transmissions are sent in small blocks of data, called frames. There are three general types of AX.25 frames: 1) Information frame (I frame); 2) Supervisory frame (S frame); and 3) Unnumbered frame (U frame). Each frame is made up of several smaller groups, called fields. Figures 4.1 and 4.2 illustrate the three basic types of frames.

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

Figure 4.1 U and S frame construction

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

Figure 4.2 Information frame construction

In the two figures, FCS is Frame Check Sequence field and PID is Protocol Identifier field. All fields except the Frame Check Sequence (FCS) are transmitted low-order bit first. FCS is transmitted bit 15 first. The following section describes each field in details.

Flag Field

To avoid overruns and data losses, the flag field is needed to distinguish every frame. It is one octet long. Because the flag delimits frames, it occurs at both the beginning and end of each frame. Two frames may share one flag, which would denote the end of the first frame and the start of the next frame. A flag consists of a zero followed by six ones followed by another zero, or 01111110 (7E hex). As a result of bit stuffing, this sequence is not allowed to occur anywhere else inside a complete frame.

Flags are sent over and over again when no data are transmitting. For instance, when you set the TXdelay on your TNC to some value, it sends flags (7E's) over and

over again for that period. These flags provide the receiver with a clear indication of when one packet has ended and the next is beginning.

Address Field

The address field identifies both the source of the frame and its destination. Optionally, it also consists of two Data Link Layer repeater sub-fields. Each sub-field consists of an amateur callsign and a Secondary Station Identifier (SSID). The call-sign is made up of upper-case alpha and numeric ASCII characters only. The SSID is a four-bit integer that uniquely identifies multiple stations using the same amateur call-sign.

The HDLC address field is extended beyond one octet by assigning the least-significant bit of each octet to be an "extension bit". The extension bit of each octet is set to "0" to indicate the next octet contains more address information, or to "1", to indicate that this is the last octet of the HDLC address field. To make room for this extension bit, the amateur radio call- sign information is shifted one bit left. Reference 9 section 3.12 details the address field encoding.

Control Field

The control field identifies with one or two octets in length the type of frame being passed and controls several attributes of the Data Link Layer connection. This field in AX.25 are modeled after the ISO HDLC balanced operation control fields.

Figures 4.3 and 4.4 illustrate the basic format of the control field associated with each of AX.25 three types of frames. The control field can be one or two octets long and may use sequence numbers to maintain link integrity. These sequence numbers may be three-bit (modulo 8) or seven-bit (modulo 128) integers.

Control Field 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

Figure 4.3 Control field formats (modulo 8)

Control Field Type	Control-Field Bits															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I Frame	N(R)							P	N(S)							0
S Frame	N(R)							P/F	0	0	0	0	S	S	0	1

Figure 4.4 Control field formats (modulo 128)

PID Field

The Protocol Identifier (PID) field appears in information frames (I and UI) only. It identifies which kind of Layer 3 (Network Layer) protocol, if any, is in use. The PID itself is not included as part of the octet count of the information field. The encoding of the PID is as follows (Figure 4.5):

HEX	MSB LSB	Translation
**	yy01yyyy	AX.25 layer 3 implemented.
**	yy10yyyy	AX.25 layer 3 implemented.
0x01	00000001	ISO 8208/CCITT X.25 PLP
0x06	00000110	Compressed TCP/IP packet. Van Jacobson (RFC 1144)
0x07	00000111	Uncompressed TCP/IP packet. Van Jacobson (RFC 1144)
0x08	00001000	Segmentation fragment
0xC3	11000011	TEXNET datagram protocol
0xC4	11000100	Link Quality Protocol
0xCA	11001010	Appletalk
0xCB	11001011	Appletalk ARP
0xCC	11001100	ARPA Internet Protocol
0xCD	11001101	ARPA Address resolution
0xCE	11001110	FlexNet
0xCF	11001111	NET/ROM
0xF0	11110000	No layer 3 protocol implemented.
0xFF	11111111	Escape character. Next octet contains more Level 3 protocol information.
Escape character. Next octet contains more Level 3 protocol information.	00001000	

Figure 4.5 PID definition[1]

Information Field

This field is where the users' data locates. The Information field is allowed only in these five types of frames: the I frame, the UI frame, the XID frame, the TEST frame and the FRMR frame. The default length of this field is 256 octets. Any information in the Information field is passed along the link transparently, except for the zero-bit insertion (Bit Stuffing) necessary to prevent flags from accidentally appearing in the Information field.

[1] An “Y” indicates all combinations used.

Frame Check Sequence

The Frame-Check Sequence (FCS) is a sixteen-bit number calculated by both the sender and the receiver of a frame. It is computed over the Address, Control, and Information fields. It provides a method by which the receiver can detect errors that may have been induced during the transmission of the frame, such as lost bits, flipped bits, and extraneous bits.

In practical applications, FCS is more preferred to be CRC (Cyclic Redundancy Check).

4.1.2 HDLC encoding Polynomial scrambling/descrambling

The main concepts addressed in this sub-section are HDLC encoding and LFSR (Linear Feedback Shifted Register) scrambling and descrambling which is called G3RUH standard named by James miller.

4.1.2.1 Transmit AX.25 packets

Before the data is sent out, it is handled by these procedures: NRZ encoding, bit stuffing, polynomial scrambling and frame delimiting.

NRZI encoding

In NRZI (Non-Return to Zero Inverse) encoding, a zero is transmitted by a change in the output, while a one is sent by no change in the output. This is illustrated in Figure 4.6.

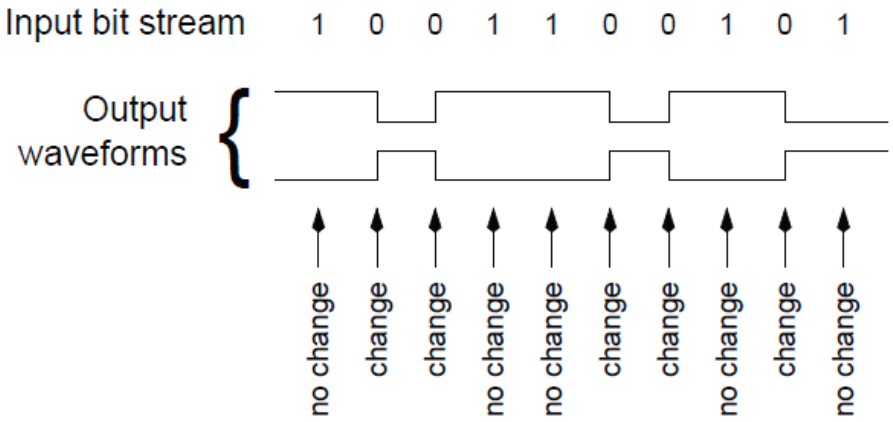

Figure 4.6 NRZI encoding

Bit stuffing

There is an obvious weakness of NRZI encoding that if there are too many one bits in a row, a DC component is needed for transmission. And this is not reliable for reception to recovery the timing of each bit. The solution of usual stream-oriented data transmission schemes is bit stuffing. That is if too many bits of one kind appear in a row, insert a bit of the other kind on transmission, and remove it on reception. The process is shown in Figure 4.7. In this standard, only maximum five consecutive one are allowed in a row. When the sixth one occurs, insert a zero before the one bit on transmission and remove the zero on reception.

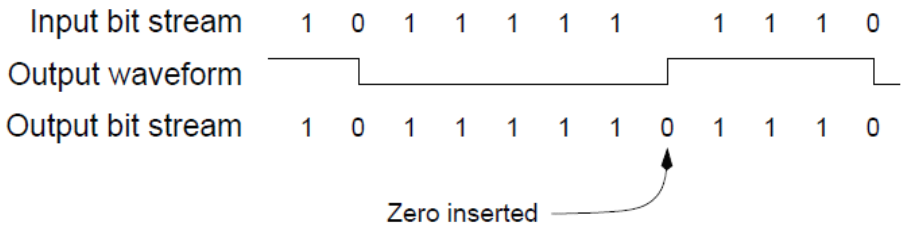

Figure 4.7 bit stuffing process

Frame delimiting

HDLC indicates the beginning and end of a frame with a bit pattern (preferred to call flag) that is not permitted in user data: the octet 0x7e. And this pattern is not the subject of bit stuffing.

Polynomial scrambling

Polynomial scrambling is a typical modem function to handle transmitted data. The process of scrambling the data enhances its transmission in several ways:

- An increased density of transitions further eases timing recovery.
- An increased density of transitions further reduces the low frequency bandwidth requirements of the system.
- The pseudo-random nature of the scrambled data renders the transmitted spectrum noise-like, with no spectral lines that could interfere with other

services in shared spectrum allocations.

The standard 9600 baud modem is designed by James Miller and use polynomial scrambling. Its polynomial is $1+X^{12}+X^{17}$. And this design is named by his amateur radio call-sign G3RUH. The scrambling process using LFSR is shown in the figure below (Figure 4.8).

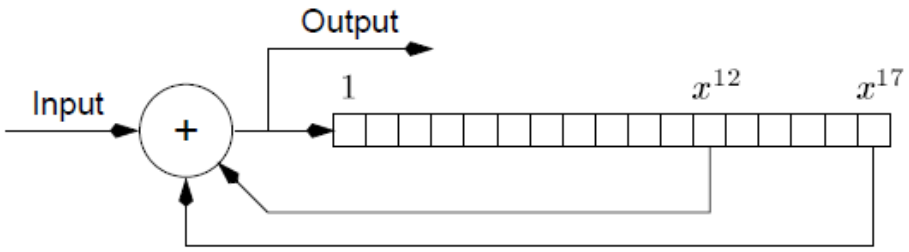

Figure 4.8 polynomial scrambler in G3RUH modem

The original implementation was built with discreet logic shift registers and exclusive-OR gates. Modern implementations use programmable logic or perform these operations in software.

4.1.2.2 Receive AX.25 packets

On reception, it is merely the reverse process of transmission: descrambling, flags eliminating, skipping bit stuffing zero and recovery from NRZI encoding.

The scrambling and descrambling work in the same way. The scrambling divides the bit sequence by the polynomial, while the descrambling multiplies the same polynomial illustrated in Figure 4.9.

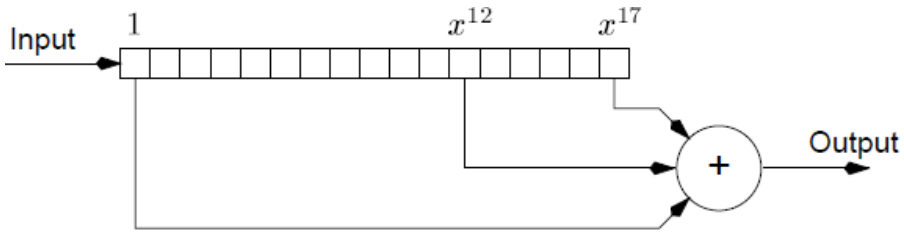

Figure 4.9 polynomial descrambler in the G3RUH modem

4.2 Software modules

The communication software is implemented in the C language with IAR Embedded Workbench provided by TI Company. The source codes are structured as a set of functionality modules required by the communication subsystem. Table 4.1 gives a brief description of each module.

Table 4.1 software module brief description

File	Description
main.c	Implement the communication controller main processing loop
CC1020.c	Implement the CC1020 transceiver interfaces
AX25.c	Implement the TNC functionality in software
uart.c	Implement the UART functionality in software
SPI.c	Implement the SPI bus interface
timer.c	Implement a timer functionality

The relations between each software module are shown in the block diagram below (Figure 4.10).

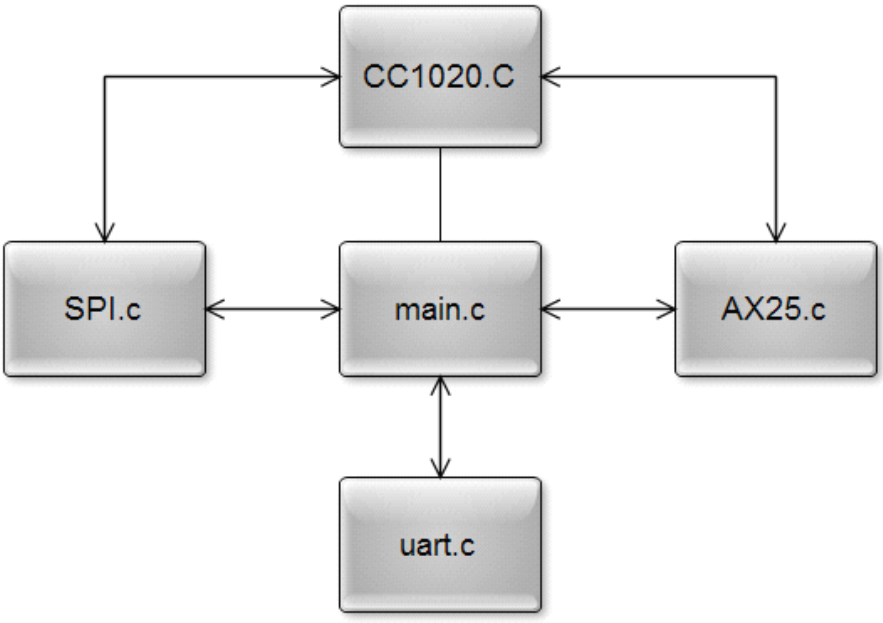

Figure 4.10 relations between each software module

4.3 Functionality description

The section provides the details of every functionality file of the software design. The full source codes are available in Appendix A except CC1020 software module.

4.3.1 Main communication Control (main.c)

The behavior of the UHF communication subsystem is very simple. The entire loop process is illustrated in Figure 4.11.

After initialization, the system calls functions in uart.c to receive commands from the OBC. There will be three states to handle. If it is a transmit command, configure the whole system in transmit mode, especially the transceiver with functions in CC1020.c. The transmission process calls functions from AX25.c which performs as a real TNC. After that, go back to the beginning of the whole loop. Provided that a receive command, configure the system in receive mode and begin to receive packets. There are three cases to end the reception: timeout, Non-valid CRC and full valid packet. Then, go back to the beginning of the loop. If the command is a order to generate a carrier, configure properly and send a carrier in some time and then go back to the loop beginning. If no valid command is got from the OBC, the software will keep listening and all other parts like transceiver and power amplifier are in power-down mode to save power.

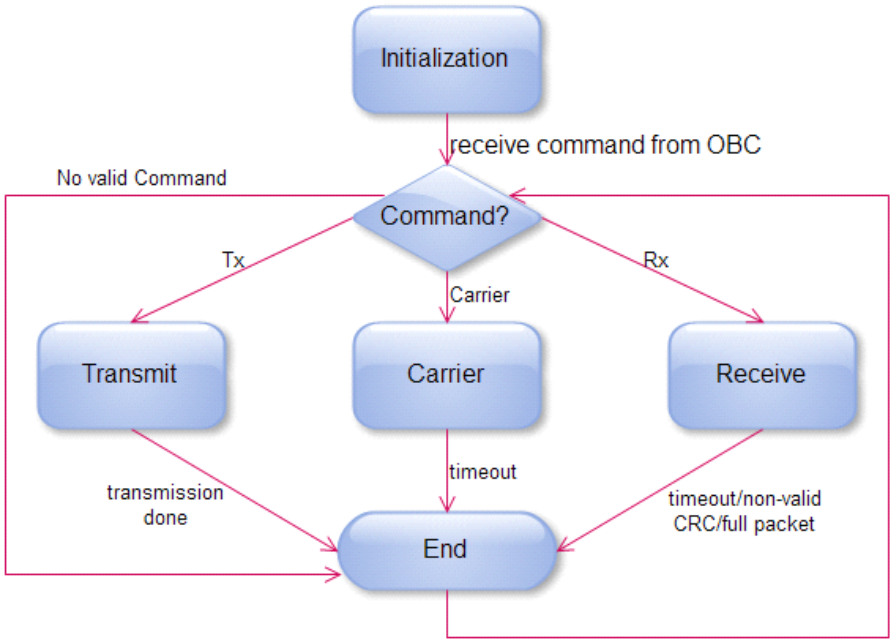

Figure 4.11 main.c loop behaviour

After whatever transmission, generating carrier or reception, the system will disable the transceiver right now to save power. Before this, bus terminals' module functions are also disabled.

4.3.2 Transceiver interface (CC1020.c)

The CC1020 is a highly programmable device. Using its programming bus (SPI), the carrier frequency and power consumption levels can be set, the PLL can be re-calibrated, and switching between transmit and receive modes is easy to be done. This programmability is achieved through the 8-bit registers built in the CC1020 itself. These registers control every aspect of the operations of the CC1020 and are fully programmable though the programming bus.

The CC1020.c file includes all functions to configure or read status from the control registers. The main functions are as the following:

- void CC1020_SetReg(char registro, char dato)

  The basic function to configure one certain register of CC1020.

- char CC1020_ReadReg(char registro)

  The basic function to read value from a certain register of CC1020.

- void CC1020_Init(void)

Configure the interface pins (PSEL, PDI, PDO, PCLK) in order to proceed the working configurations of CC1020.

- char CC1020_Reset(void)

Configure the registers of CC1020 with default values.

- void CC1020_WakeUpToTX(char txanalog)

Configure the register ANALOG to wake up the CC1020 to transmit data by switching on the quartz, bias generator and frequency synthesizer. Before starting bias generator, the quartz needs 2-5 ms to get stabilization. Otherwise, functionality failure will be induced. Another 150 ms should be set between the power generator and the bias of the synthesizer to hang up the PLL.

- char CC1020_Calibrate(char pa_power)

Ensure the PLL working properly while PA should be off.

- char CC1020_SetupTX(char txanalog, char pa_power)

Configure the registers ANALOG and PA Power which control the PLL and output power respectively. After calling this function, the system is ready to transmit.

- void CC1020_SetupPD(void)

After transmission, disable the internal power amplifier.

- char CC1020_Config_X[2](void)

Configure the registers of CC1020 to control frequency used, desired output power, baud rate, modulation type and other parameters of the subsystem.

- void CC1020_WakeUpToRX(char RXANALOG)

Configure the register ANALOG to wake up the CC1020 to receive data by switching on the quartz, bias generator and frequency synthesizer. Before starting bias generator, the quartz needs 2-5 ms to get stabilization. Otherwise, functionality failure will be induced. Another 150 ms should be set between the power generator and the bias of the synthesizer to hang up the PLL.

- char CC1020_SetUpToRX(char RXANALOG, char PA_POWER)

[2] X stands for Carrier, Tx and Rx three cases.

Configure the registers ANALOG and PA Power which control the PLL and output power respectively. After calling this function, the system is ready to receive.

These functionalities are just indicated by their names. Most functionality like initialization and configuration of transmit or receive is highly abstracted. But the ability to read or write an individual register is also available.

4.3.3 Software TNC (AX25.c)

This file realizes the functionality of a real TNC based on the G3RUH standard. There are two main functions, void AX25_SendPacket(unsigned char * packet, unsigned int packet_len) and unsigned int AX25_ReceivePacket(unsigned char *data), which are used in transmit mode and receive mode separately. Every bit both in transmission and reception are handled as AX.25 protocol, HDLC encoding and G3RUH standard require. The transmission and reception process are shown correspondingly in Figure 4.12 and Figure 4.13.

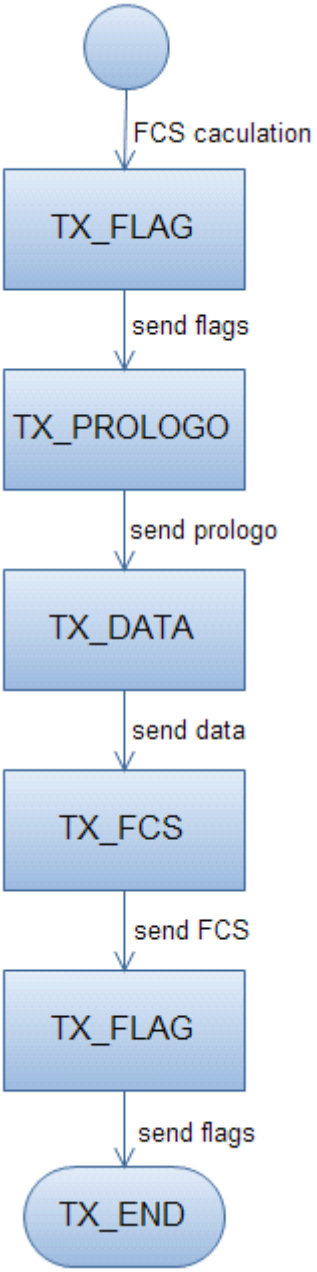

Figure 4.12 software TNC transmission process

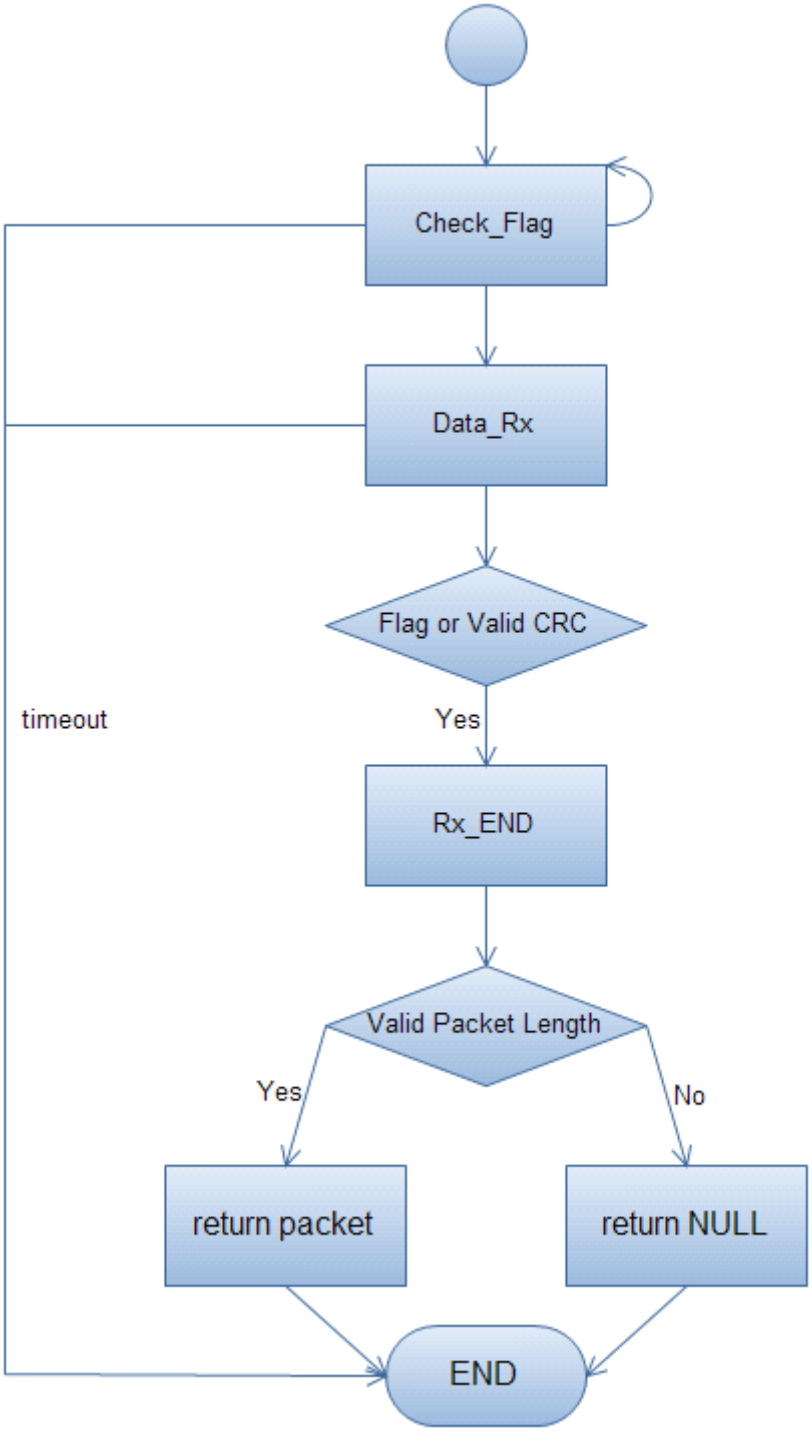

Figure 4.13 software TNC reception process

One thing should be referred is the FCS calculation. A Frame Check Sequence (FCS) refers to the extra checksum characters added to a frame in a communication protocol for error detection and correction. All frames and the bits, bytes, and fields contained within them, are susceptible to errors from a variety of sources. The FCS

field contains a number that is calculated by the source node based on the data in the frame. This number is added to the end of a frame that is sent. When the destination node receives the frame the FCS number is recalculated and compared with the FCS number included in the frame. If the two numbers are different, an error is assumed, the frame is discarded. The sending host computes a checksum on the entire frame and appends this as a trailer to the data. The receiving host computes the checksum on the frame using the same algorithm, and compares it to the received FCS. This way it can detect whether any data was lost or altered in transit. The FCS is transmitted in such a way that the receiver computes a running sum over the entire frame, including the trailing FCS, and expects to see a fixed result when it is correct.

The Frame Check Sequence can use a number of different methods; however these are the most popular:

- CRC – Cyclic redundancy Check – Polynomial calculations are performed on the data
- Two Dimensional Parity – Uses a parity bit to make sure the data has not been corrupted.
- Checksum – Sums the data to arrive at a total.

Most people prefer to use the CRC method. The figure below shows the hardware (Figure 4.14) in a real TNC (PK96) with which to generate FCS of a packet.

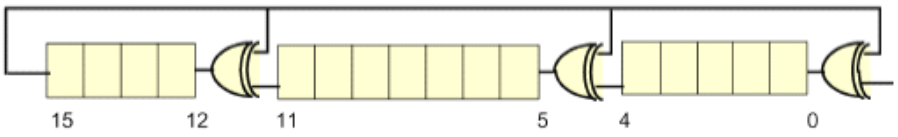

Figure 4.14 CRC calculation hardware of PK96

In our software, we use the bye-wise CRC-16 polynomial method. This method is a byte-wise CRC calculation which can handle 8 bits at once and it is almost four times faster than a bit-oriented calculation[10]. And the drawback is more memory needed. The implemented polynomial is $X^{16}+X^{12}+X^5+X^0$ as well and the look-up table is shown in Table 4.2.

Table 4.2 look-up table for CRC calculation

0x0000	0x1189	0x2312	0x329b	0x4624	0x57ad	0x6536	0x74bf
0x8c48	0x9dc1	0xaf5a	0xbed3	0xca6c	0xdbe5	0xe97e	0xf8f7
0x1081	0x0108	0x3393	0x221a	0x56a5	0x472c	0x75b7	0x643e
0x9cc9	0x8d40	0xbfdb	0xae52	0xdaed	0xcb64	0xf9ff	0xe876
0x2102	0x308b	0x0210	0x1399	0x6726	0x76af	0x4434	0x55bd
0xad4a	0xbcc3	0x8e58	0x9fd1	0xeb6e	0xfae7	0xc87c	0xd9f5
0x3183	0x200a	0x1291	0x0318	0x77a7	0x662e	0x54b5	0x453c
0xbdcb	0xac42	0x9ed9	0x8f50	0xfbef	0xea66	0xd8fd	0xc974
0x4204	0x538d	0x6116	0x709f	0x0420	0x15a9	0x2732	0x36bb
0xce4c	0xdfc5	0xed5e	0xfcd7	0x8868	0x99e1	0xab7a	0xbaf3
0x5285	0x430c	0x7197	0x601e	0x14a1	0x0528	0x37b3	0x263a
0xdecd	0xcf44	0xfddf	0xec56	0x98e9	0x8960	0xbbfb	0xaa72
0x6306	0x728f	0x4014	0x519d	0x2522	0x34ab	0x0630	0x17b9
0xef4e	0xfec7	0xcc5c	0xddd5	0xa96a	0xb8e3	0x8a78	0x9bf1
0x7387	0x620e	0x5095	0x411c	0x35a3	0x242a	0x16b1	0x0738
0xffcf	0xee46	0xdcdd	0xcd54	0xb9eb	0xa862	0x9af9	0x8b70
0x8408	0x9581	0xa71a	0xb693	0xc22c	0xd3a5	0xe13e	0xf0b7
0x0840	0x19c9	0x2b52	0x3adb	0x4e64	0x5fed	0x6d76	0x7cff
0x9489	0x8500	0xb79b	0xa612	0xd2ad	0xc324	0xf1bf	0xe036
0x18c1	0x0948	0x3bd3	0x2a5a	0x5ee5	0x4f6c	0x7df7	0x6c7e
0xa50a	0xb483	0x8618	0x9791	0xe32e	0xf2a7	0xc03c	0xd1b5
0x2942	0x38cb	0x0a50	0x1bd9	0x6f66	0x7eef	0x4c74	0x5dfd
0xb58b	0xa402	0x9699	0x8710	0xf3af	0xe226	0xd0bd	0xc134
0x39c3	0x284a	0x1ad1	0x0b58	0x7fe7	0x6e6e	0x5cf5	0x4d7c
0xc60c	0xd785	0xe51e	0xf497	0x8028	0x91a1	0xa33a	0xb2b3
0x4a44	0x5bcd	0x6956	0x78df	0x0c60	0x1de9	0x2f72	0x3efb
0xd68d	0xc704	0xf59f	0xe416	0x90a9	0x8120	0xb3bb	0xa232
0x5ac5	0x4b4c	0x79d7	0x685e	0x1ce1	0x0d68	0x3ff3	0x2e7a
0xe70e	0xf687	0xc41c	0xd595	0xa12a	0xb0a3	0x8238	0x93b1
0x6b46	0x7acf	0x4854	0x59dd	0x2d62	0x3ceb	0x0e70	0x1ff9
0xf78f	0xe606	0xd49d	0xc514	0xb1ab	0xa022	0x92b9	0x8330
0x7bc7	0x6a4e	0x58d5	0x495c	0x3de3	0x2c6a	0x1ef1	0x0f78

The calculation procedures are as the following:

1. Initial the FCS with 0xFFFF;
2. Exclusive-OR the new input byte with the least significant byte of FCS, and use the result as the index to get values from the look-up table;
3. FCS shifts 8 bits to the right
4. Exclusive-OR FCS with the new value got from the look-up table;

5. Repeat the steps from 1 to 4 for all data bytes.

4.3.4 Timer (timer.c)

In this file, the function void TIMER_SetupTimer_ms(short volatile *semaforo) realizes a timer functionality with the ms-wise counter. void TIMER_Wait_ms(short volatile semaforo) and void TIMER_Wait_us(short volatile semaforo) are used to be waiting cycles whose time units are ms and us separately.

4.3.5 Data interface (uart.c)

Functions used to communicate with OBC are defined in this file.

- void UART_Init(void);

  Function that enables the module USART0 of the micro-controller. Character attributes and baud rate are defined.

- void UART_SendByte (unsigned char data);

  Function that allows to transmit a byte to USART.

- unsigned char UART_ReceiveByte (void);

  Function that allows USART to receive a byte.

- void printUART (unsigned char *message);

  Function whose purpose is to print the whole message through the RS232 serial bus.

- unsigned char CRC(unsigned char *pDato, unsigned short dim);

  Function which is used to calculate CRC of transmitted or received packets.

- unsigned char UART_ReceivePacket(unsigned char *pDato);

  Function that allows USART to receive a packet which can be more than one byte.

- unsigned char uart_SendPacket(unsigned char *pDato);

  Function that allows to transmit more than one byte data to USART.

4.3.6 Configuration interface (SPI.c)

Functions used for USART0 in SPI mode which performs as a SPI bus to configure the registers of CC1020 are defined in this file.

- void SPI_Init(void)

  Configure the port of USART0 to work in SPI mode. The character attributes, baud rate and other parameters are defined.

- void SPI_Disable(void)

  Function that disables the module functionality of USART0 and enables the digital I/O functions of the corresponding port.

- void SPI_SendByte (unsigned char data)

  Function that transmits character from RXTXData buffer.

- unsigned char SPI_ReceiveByte (void)

  Function that receive character from RXTXData buffer.

- void SPI_ResetBit (void)

  Reset bit transmission counter.

- void SPI_SendBit (unsigned char data)

  Function that transmits character from RXTXData buffer bit by bit.

Chapter 5
Realization and Test

This chapter addresses the PCB realization of the communication subsystem and experiments established to test all functionalities.

5.1 PCB realization

To design and realize a PCB, the main following procedures can be followed with the help of the software Mentor Graphics:

- Create a components library with Library Manager;
- Draw and verify schematics with Design Capture;
- Produce PCB layout and optimize connections with Expedition PCB;
- Generate Gerber files used for manufactures.

The symbols and cells of each component are defined in the overall library of the AraMis project, serving for the schematics and layouts design.

Figure 5.1 shows the final PCB layout of the UHF communication subsystem. From this layout, we can easily find the 32-pin CC1020 and below the transceiver, it is the 64-pin micro-controller MSP430F149. The LM317 is on the left side of the MCU and the position to place the antenna is above th transceiver, which has 5 terminals to solder.

Figure 5.2 shows the PCB in kind, which is a realization of a former version layout. And we built two prototypes of the communication subsystem with this kind board.

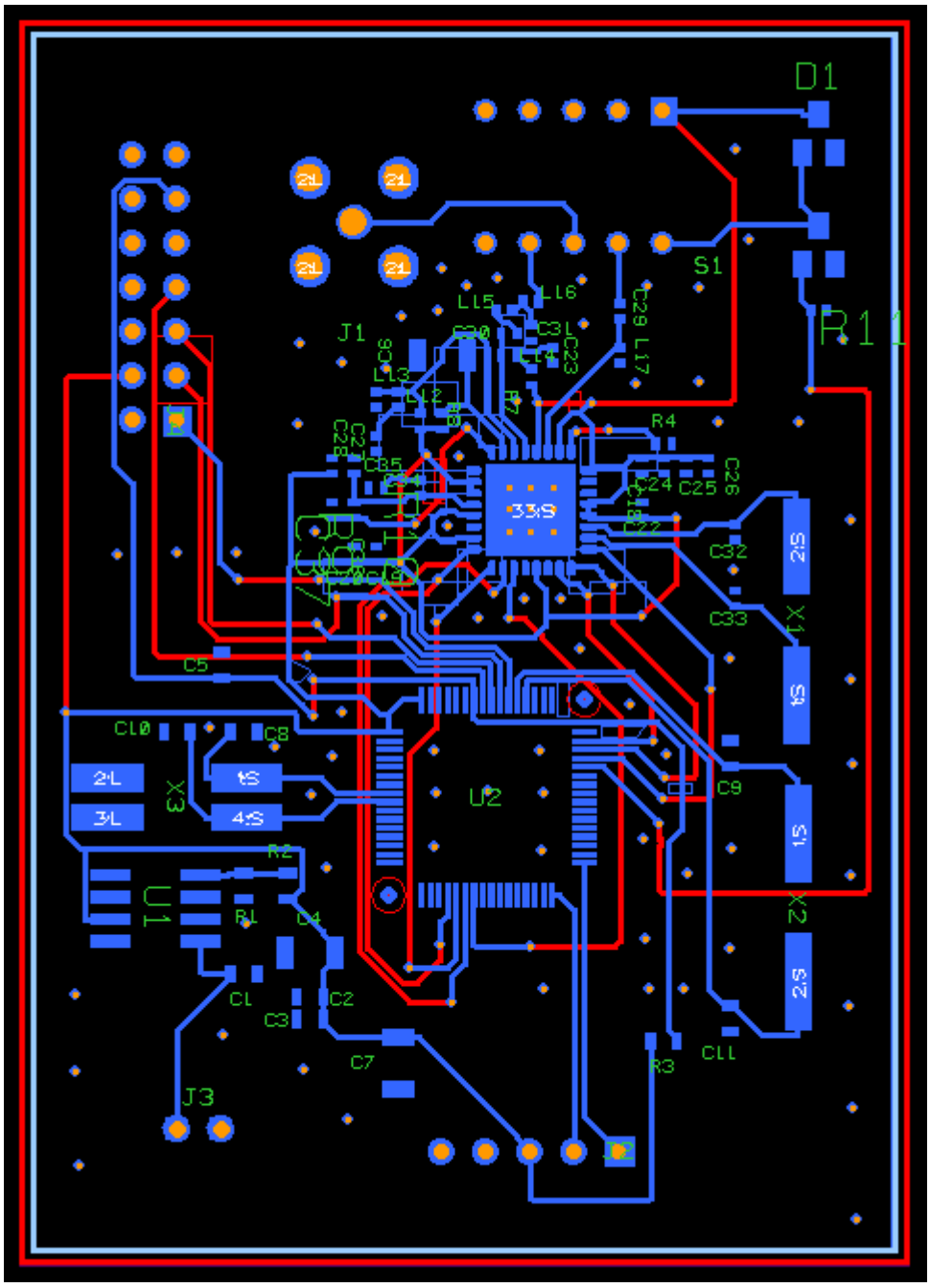

Figure 5.1 PCB layout

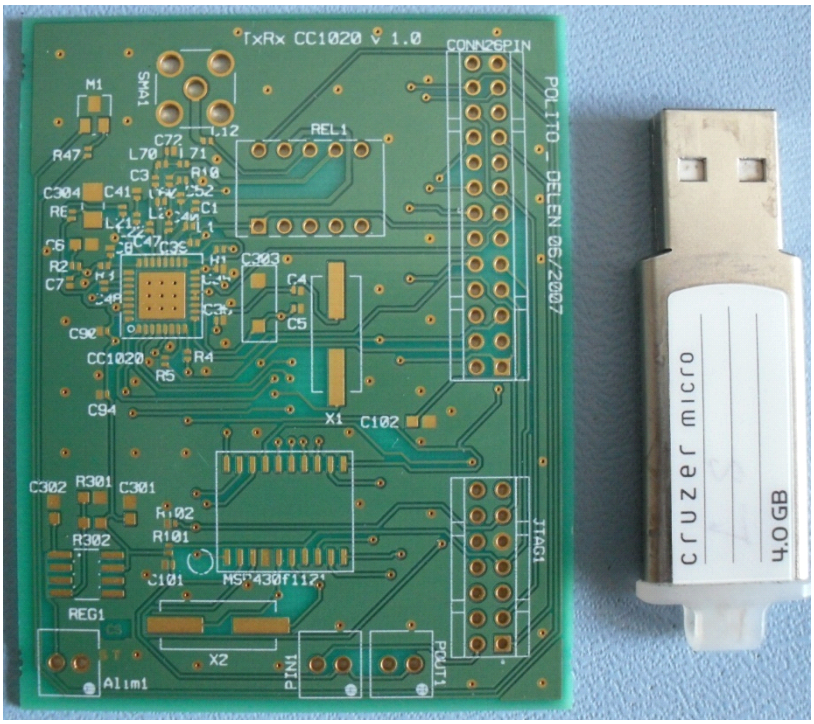

Figure 5.3 a realization of layout

5.2 Experiments for test

To evaluate the functionalities of the whole system in both hardware and software, we have designed and completed some experiments. These equipment and software are used in the test:

Hardware	
PC	DC power supply (GPC-3030D)
Oscilloscope (ADS7102C)	Radio (YAESU FT-847)
TNC (PK96)	2 evaluation boards
RS233 cable and USB adaptor	Spectrum Analyzer
Soldering equipments	

Software	
IAR Embedded Workbench	RealTerm
Control panel software	

A brief description is presented in the following table (Table 5.1).

Table 5.1 brief description of experiments

Experiment	Descriptions
# 1	Transparent data transmission and reception to test hardware
# 2	Predefined AX.25 packet transmission and reception to test the software TNC
# 3	Communication with amateur radio

The section below gives the details of these three test experiments.

Experiment # 1

The goal of this experiment is to test the fundamental hardware functionality that if the subsystem can transmit and receive properly. The communication is done between the two evaluation boards which both are the prototypes of the communication subsystem, built with the same components. The data is transparent, which means that it is not encoded on transmission so that there is no need to decode it.

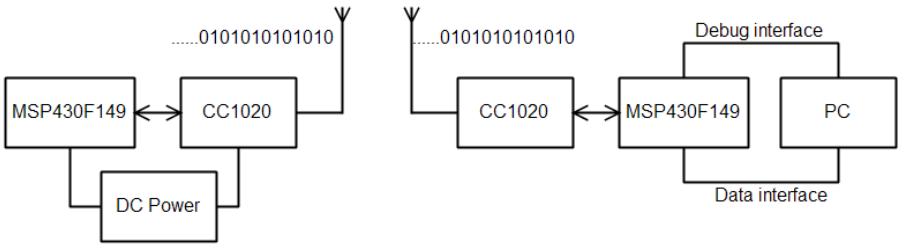

Figure 5.2 experiment 1 construction

Figure 5.2 shows the construction of this experiment. The debug interface is a JTAG port to program the micro-controller with the IAR Embedded Workbench software and the data interface is a RS232 serial bus through which the commands from PC are sent to the micro-controller and all information can be sent back using the USART1 module functionality.

One board runs automatically and is not controlled manually. It keeps transmit a predefined sequence bits. The other board is controlled by the software to receive this

packet. The first part of the transmitted packet are "010101010101.....", which are recommended to use for the transceiver CC1020 to sense the carrier. This kind bit pattern can improve the performance of BER (Bit Error Rate) and it is proved by the experiment. If one board transmit packets including this kind preambles and other parts, after sensing the preambles which means the receiver has got synchronization, the other begins to store what it receives and the bit error is zero in the short distance. Otherwise, if we use preambles which are similar as "0111110111....." (many same bit in a row), the bit error will be increased largely.

Actually, before other tests, the communication subsystem works in the right frequency should be ensured. And the Spectrum Analyzer can detect the broadcasting signal. The figure below (Figure 5.3) displays the detecting spectrum, which proves the transmitter uses the right 437 MHz frequency[3].

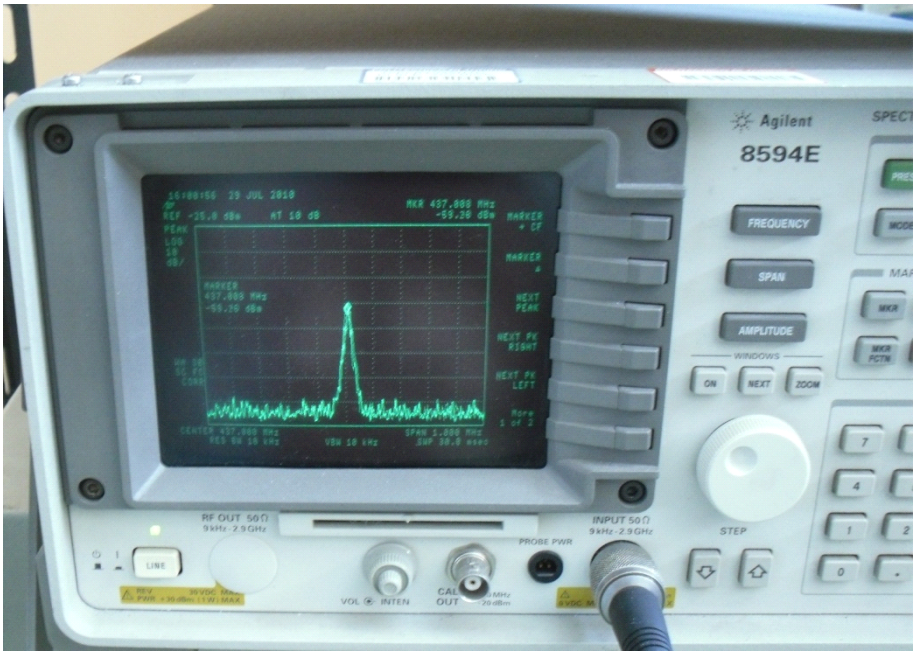

Figure 5.3 spectrum analyzer

Through this experiment, from the hardware point of view, we can verify that at least the subsystem can transmit and receive data properly. From the software point of view, the functions, except those that realize a real TNC functionalities (encoding, decoding and FCS calculation of AX.25 protocol packets), also work.

[3] Precisely, the actual frequency is 437.008 MHz.

Experiment # 2

The main goal of this experiment is to test the AX25.c software module. The construction is the same as experiment 1. A predefined AX.25 protocol packet is transmitted by one evaluation board over and over again. The MCU of the sending board performs the same transmission functionality of a TNC and the MCU of the receiving board performs the same reception functionality of a TNC. The experiment has two steps. The first step is to test the reception function and the second step is to test the transmission function.

In the first step, the test packet we use is: "~ p r o v a . j ~ ~", and after decoding, the packet transmitted by CC1020 is: "FE F1 6E 90 A0 BC A5 6A FA E4 2D CE 34 FA AB 29 19 FB ED BD D3 FD 28 D2 E7 B9 F6 BD 6E 76 AE AF 43 5D 6A 86 E3 D3 B2 2C 05 28 AE E0 47 8A A5 91 F5 29 56 FF 3A 72 C4 EB D2 36 34 86 AC D7 65 E3 12 3E 54 04 94 B5 FF FB FE BC EA 6E 2D 2B 3A D8 91 C4 A5 B6 58 F4 6D C2". The received packet should be the former one. Fortunately, the result is what we expect, thus the reception function works. Based on this fact, we can test the TNC transmission function now. The transmission packet includes starting FLAG, PROLOGO, and a number of 0xAA whose quantities can be controlled, FCS and ending FLAG. And the receiving board can get the same packet, which demonstrates the TNC reception function works as well.

In the receiving process, before every bit is sampled by the I/O pin of the micro-controller, descrambling calculations are done and CRC is re-calculated after every byte by the software. Since the baud rate is fixed, to avoid missing the coming bits, we should be ensuring that the calculation time is much less than the period of the data rate. To know the time the two calculations needs, we set P1.0 of the micro-controller to 1 before calculations and set it to 0 after that. The test result is that the descrambling processes for every bit needs 10 us and the CRC calculation for every byte needs 5 us under the condition of 8 MHz crystal oscillator for the

micro-controller. The period of the data is 100 us, thus it is stable enough. And these data determine the minimum frequency of the crystal oscillator. We can calculate the minimum value in this way: $Fxosc_{min}$ = 8 MHz/(baud period/15 us). Baud rate is 9600 bps, so the minimum value required is 1.152 MHz.

Experiment # 3

The former two experiments have tested all functionalities of the software. All functions work properly when data is exchanged between the two evaluation boards, each of which is the prototype of the UHF communication subsystem. But the ultimate goal of the subsystem design is that it can communicate with amateur radios. The experiment is assigned to test the ability in this aspect.

The experiment diagram is shown in Figure 5.4.

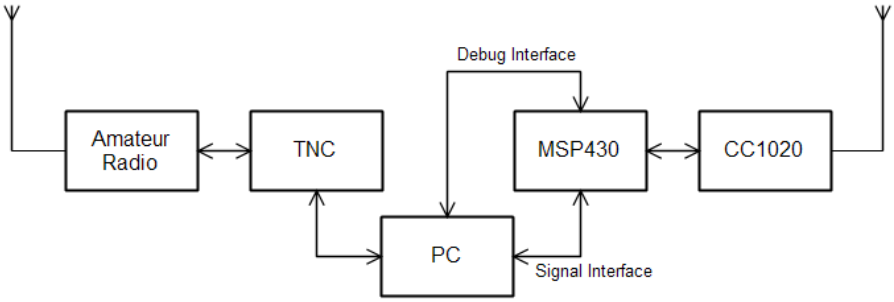

Figure 5.4 experiment 3 construction

Three software are needed, IAR Embedded Workbench, Realterm and the control panel software, shown in Figure5.5.

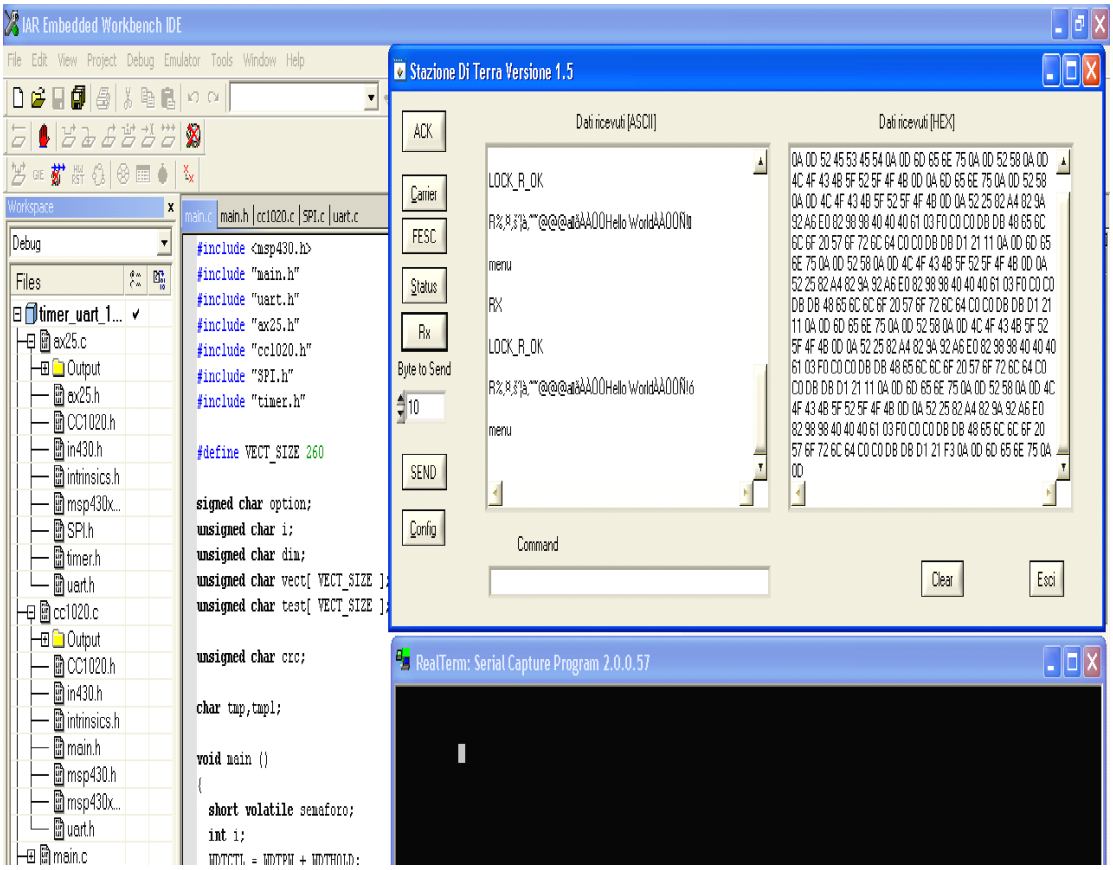

Figure 5.5 windows of software

The Realterm software controls the TNC in KISS (Keep It Simple, Stupid) mode. Then, users can manage the TNC to transmit or receive packets after proper configurations. The KISS is a simple Host-to-TNC communication protocol [11]. Asynchronous frame format is used to delimit frames. Each frame is both preceded and followed by a special FEND (Frame End) character, analogous to an HDLC flag. No CRC or checksum is provided. In addition, no RS-232C handshaking signals are employed. The special characters are in table 5.2. The reason for both preceding and ending frames with FENDs is to improve performance when there is noise on the asynchronous line. The FEND at the beginning of a frame serves to "flush out" any accumulated garbage into a separate frame (which will be discarded by the upper layer protocol) instead of sticking it on the front of an otherwise good frame. As with back-to-back flags in HDLC, two FEND characters in a row should not be interpreted as delimiting an empty frame.

Table 5.2 special characters in KISS protocol

Abbreviation	Description	Hex value
FEND	Frame End	C0
FESC	Frame Escape	DB
TFEND	Transposed Frame End	DC
TFESC	Transposed Frame Escape	DD

Experiment #3 can also be divided into two steps. Similarly, the first step is transmission test. The transmission packet includes starting FLAG, PROLOGO, and a number of 0xAA whose quantities can be controlled, FCS and ending FLAG. The software Realterm communicates with the TNC and it can show what the received data by the radio after the TNC decoding (Figure 5.6), which can demonstrate that amateur radios can receive data from the UHF communication subsystem satisfactorily.

The second step is obviously reception test. The software Realterm controls the KISS mode TNC to transmit the packet in Hex "C0 00 82 A4 82 9A 92 A6 E0 82 98 98 40 40 40 61 03 F0 DB DC DB DC DB DD DB DD 48 65 6C 6C 6F 20 57 6F 72 6C 64 DB DC DB DC DB DD DB DD D1 21 C0" which includes some KISS special characters, AraMiS PROLOGO and "Hello World!" information. Figure 5.7 shows what the communication subsystem receives, which can demonstrate that amateur radios are able to transmit data to the UHF communication subsystem effectively.

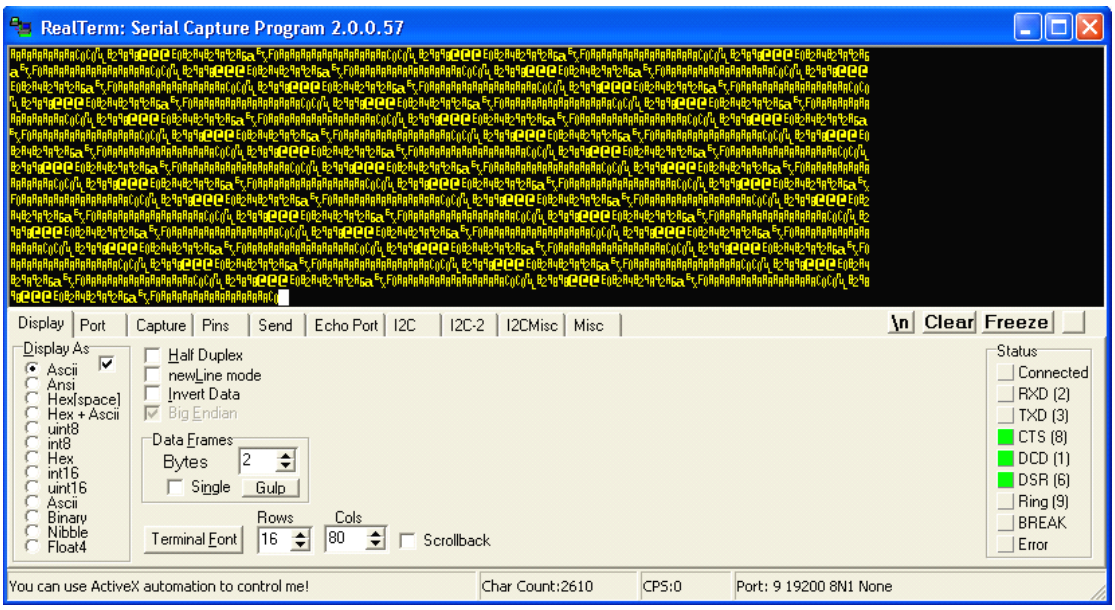

Figure 5.5 amateur radio reception

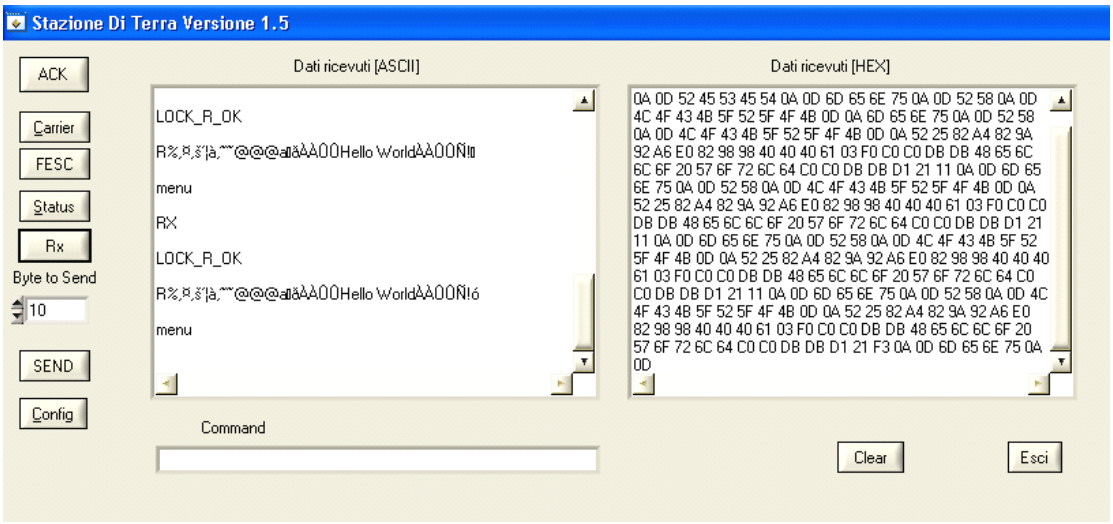

Figure 5.6 amateur radio transmission

Test results of this experiment shows that the UHF communication subsystem is able to communication with amateur radios with quite a good performance. Both the hardware and the software are well developed.

Chapter 6
Conclusion

In this final project, a fully amateur radio compatible UHF communication subsystem is well developed. And its design exploits small size, low power and COTS components which are both relatively cheap and power efficient. In addition, the programmable low power micro-controller implements the functionality of a real TNC through its software which largely reduces the size and weight of the communication subsystem. This communication subsystem is very suitable for nanosatellite application.

Thanks to the narrowband capability of the CC1020, the subsystem can communicate with the amateur radio YAESU FT-847 with a quite good performance. And the required data rate is 9600 bps which is pretty high for narrowband operation. Fortunately, we got a satisfactory result at last.

The only problem left is radiation effect to the communication subsystem. A test under a radiation environment should be done to ensure the reliability in the future work.

Through this practice, the graduation candidate has improved basic practical engineering skills, grasped the process of PCB realization and been familiar with software programming and debugging in the C language. What's more, the ability to use theoretical knowledge to instruct practice is highly enhanced.

Appendix A Source Codes

main.h

```
#ifndef MAIN_H
#define MAIN_H
//#define FOSC 8000000                    //use 8MHz oscillator
#define FOSC 4000000                      //use 4MHz oscillator
#define TX_CMD              'T'
#define RX_CMD              'r'
#define PORTANTE_CMD        'p'
#define STATUS_CMD          's'
#define ECO_CMD              e'
#define CONFIG_CMD          'c'
#define ON                   1
#define OFF                  0
#define ACK                 'A'
#define NACK_WRONG_COMMAND 'n'
#define NACK_WRONG_CRC      'N'
#endif
```

main.c

```
#include <msp430.h>
#include "main.h"
#include "uart.h"
#include "ax25.h"
#include "cc1020.h"
#include "SPI.h"
#include "timer.h"

#define VECT_SIZE 260
signed char option;
unsigned char i;
unsigned char dim;
unsigned char vect[ VECT_SIZE ];
unsigned char test[ VECT_SIZE ];
unsigned char crc;
char tmp,tmp1;

void main ()
{
   short volatile semaforo;
```

```
    int i;
    WDTCTL = WDTPW + WDTHOLD;                   // Stop watchdog timer
    BCSCTL1 &= ~XT2OFF;                         // XT2on

    do
    {
      IFG1 &= ~OFIFG;                           // Clear OSCFault flag bit
      for (i = 0xFF; i > 0; i--);               // Time for flag to set
    }
    while ((IFG1 & OFIFG));                     // OSCFault flag still set?
    BCSCTL2 |= SELM_2 + SELS;                   // MCLK = SMCLK = XT2 (safe)
    UART_Init();
    CC1020_Init();
    CC1020_SetupPD();
    _BIS_SR(GIE);
    printUART("RESET\n\r");

    while (1)
    {
      printUART("menu\n\r");
      crc = UART_ReceivePacket(vect);
      if (crc != 0)
      {
        switch (vect[0])
        {
        case PORTANTE_CMD:
          printUART("PORTANTE\n\r");
          SPI_Init();
          CC1020_Config_Carrier();
          CC1020_WakeUpToTX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3);
          CC1020_SetupTX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3,
POWER_10_DBM);
          semaforo = 5000;
          TIMER_SetupTimer_ms(&semaforo);
          while (semaforo)
            SPI_SendByte(0xFF);
          printUART("carrier\n\r");
          SPI_Disable();
          CC1020_SetupPD();
          break;
        case TX_CMD:
            printUART("TX\n\r");
            SPI_Init();//USART1 in SPI mode
            CC1020_Config_Tx();
```

```
                CC1020_WakeUpToTX(LO_DC|PA_BOOST| DIV_BUFF_CURRENT_3);
                CC1020_SetupTX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3,
POWER_10_DBM);
                AX25_SendPacket(vect+2,vect[1]);
                SPI_Disable();
                CC1020_SetupPD();
              break;
            case RX_CMD:
              /*
              //config port 1
              P1SEL = 0x00;
              P1DIR = 0xFF;
              P1OUT = 0x00;
              */
              printUART("RX\n\r");
              SPI_Init();
              CC1020_Config_Rx();
              CC1020_WakeUpToRX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3);
              CC1020_SetUpToRX(LO_DC | PA_BOOST | DIV_BUFF_CURRENT_3,
0x00);
              SPI_Disable();  //disable USART module and its SPI mode
              unsigned int rec_bytes  = AX25_ReceivePacket(&(test[2]));
              if ( rec_bytes != 0 )
              {
                test[0]='R';
                test[1]=rec_bytes;
                uart_SendPacket(test);
                printUART("\n\r");
                CC1020_SetupPD();
              }
              break;
            default:
              printUART("NO\n\r");
            }
          }
          else
          {
            printUART("NO CRC\n\r");
          }
        }
      }
```

ax25.h

```
#ifndef APRS_H
#define APRS_H

#define PPPINITFCS          0xffff    /* Initial FCS value */
#define INITFCS() fcs.u=PPPINITFCS
#define CPLFCS() fcs.u ^=0xffff

#define TX_DELAY          80
#define FLAG              0x7E
#define MAX_AX25_ONES          5
#define MAX_PACKET_SIZE          255
#define MIN_FLAG_PREAMBLE         20
#define MIN_PACKET_LENGTH         17

void AX25_SendPacket(unsigned char * packet, unsigned int packet_len);
unsigned int AX25_ReceivePacket(unsigned char *data);
char RxByte(void);
#endif
```

ax25.c

```
#include "ax25.h"
#include "uart.h"
#include "SPI.h"
#include "CC1020.h"
#include "timer.h"

int    totalbytes;
char ONEScount = 0;
char lastBit = 0;
char nextBit;
char isFlag;
unsigned int total_byte;
unsigned char preamble = 0x00;
#define INTSCRAMBLER
//look-up table used in Byte-wise CRC caculation
const unsigned short fcstab[256] = {
   0x0000, 0x1189, 0x2312, 0x329b, 0x4624, 0x57ad, 0x6536, 0x74bf,
   0x8c48, 0x9dc1, 0xaf5a, 0xbed3, 0xca6c, 0xdbe5, 0xe97e, 0xf8f7,
   0x1081, 0x0108, 0x3393, 0x221a, 0x56a5, 0x472c, 0x75b7, 0x643e,
   0x9cc9, 0x8d40, 0xbfdb, 0xae52, 0xdaed, 0xcb64, 0xf9ff, 0xe876,
   0x2102, 0x308b, 0x0210, 0x1399, 0x6726, 0x76af, 0x4434, 0x55bd,
   0xad4a, 0xbcc3, 0x8e58, 0x9fd1, 0xeb6e, 0xfae7, 0xc87c, 0xd9f5,
   0x3183, 0x200a, 0x1291, 0x0318, 0x77a7, 0x662e, 0x54b5, 0x453c,
```

```
    0xbdcb, 0xac42, 0x9ed9, 0x8f50, 0xfbef, 0xea66, 0xd8fd, 0xc974,
    0x4204, 0x538d, 0x6116, 0x709f, 0x0420, 0x15a9, 0x2732, 0x36bb,
    0xce4c, 0xdfc5, 0xed5e, 0xfcd7, 0x8868, 0x99e1, 0xab7a, 0xbaf3,
    0x5285, 0x430c, 0x7197, 0x601e, 0x14a1, 0x0528, 0x37b3, 0x263a,
    0xdecd, 0xcf44, 0xfddf, 0xec56, 0x98e9, 0x8960, 0xbbfb, 0xaa72,
    0x6306, 0x728f, 0x4014, 0x519d, 0x2522, 0x34ab, 0x0630, 0x17b9,
    0xef4e, 0xfec7, 0xcc5c, 0xddd5, 0xa96a, 0xb8e3, 0x8a78, 0x9bf1,
    0x7387, 0x620e, 0x5095, 0x411c, 0x35a3, 0x242a, 0x16b1, 0x0738,
    0xffcf, 0xee46, 0xdcdd, 0xcd54, 0xb9eb, 0xa862, 0x9af9, 0x8b70,
    0x8408, 0x9581, 0xa71a, 0xb693, 0xc22c, 0xd3a5, 0xe13e, 0xf0b7,
    0x0840, 0x19c9, 0x2b52, 0x3adb, 0x4e64, 0x5fed, 0x6d76, 0x7cff,
    0x9489, 0x8500, 0xb79b, 0xa612, 0xd2ad, 0xc324, 0xf1bf, 0xe036,
    0x18c1, 0x0948, 0x3bd3, 0x2a5a, 0x5ee5, 0x4f6c, 0x7df7, 0x6c7e,
    0xa50a, 0xb483, 0x8618, 0x9791, 0xe32e, 0xf2a7, 0xc03c, 0xd1b5,
    0x2942, 0x38cb, 0x0a50, 0x1bd9, 0x6f66, 0x7eef, 0x4c74, 0x5dfd,
    0xb58b, 0xa402, 0x9699, 0x8710, 0xf3af, 0xe226, 0xd0bd, 0xc134,
    0x39c3, 0x284a, 0x1ad1, 0x0b58, 0x7fe7, 0x6e6e, 0x5cf5, 0x4d7c,
    0xc60c, 0xd785, 0xe51e, 0xf497, 0x8028, 0x91a1, 0xa33a, 0xb2b3,
    0x4a44, 0x5bcd, 0x6956, 0x78df, 0x0c60, 0x1de9, 0x2f72, 0x3efb,
    0xd68d, 0xc704, 0xf59f, 0xe416, 0x90a9, 0x8120, 0xb3bb, 0xa232,
    0x5ac5, 0x4b4c, 0x79d7, 0x685e, 0x1ce1, 0x0d68, 0x3ff3, 0x2e7a,
    0xe70e, 0xf687, 0xc41c, 0xd595, 0xa12a, 0xb0a3, 0x8238, 0x93b1,
    0x6b46, 0x7acf, 0x4854, 0x59dd, 0x2d62, 0x3ceb, 0x0e70, 0x1ff9,
    0xf78f, 0xe606, 0xd49d, 0xc514, 0xb1ab, 0xa022, 0x92b9, 0x8330,
    0x7bc7, 0x6a4e, 0x58d5, 0x495c, 0x3de3, 0x2c6a, 0x1ef1, 0x0f78
};
#define PROLOGO_LEN 16
const char prologo[PROLOGO_LEN] = {
    'A' << 1, 'L' << 1, 'L' << 1, ' ' << 1, ' ' << 1, ' ' << 1, 0xE0,
    'A' << 1, 'R' << 1, 'A' << 1, 'M' << 1, 'I' << 1, 'S' << 1, 0x61,
    0x03, 0xF0
};
unsigned char out;
#ifdef INTSCRAMBLER
unsigned short scrambled1, scrambled2;
#else
unsigned char scrambled1, scrambled2, scrambled3;
#endif
union
{
    struct
    {
        unsigned char crc2; //byte piu' significativo
        unsigned char crc1;  //byte meno significativo
```

```
    }b;
    unsigned short u;
}fcs;


void pppfcs(unsigned char *cp, int len)
{
    while (len--)
        fcs.u = (fcs.u >> 8) ^ fcstab[(fcs.u ^ *cp++) & 0xFF];
}

void AX25_ComputeFCS(unsigned char * packet, int packet_len)
{
    INITFCS();
    pppfcs(packet, packet_len);
    CPLFCS();

}

void AX25_SendByte(unsigned char byte, int flag)
{
    char i;
    unsigned int tmp;
    unsigned char tx;
    static unsigned char ones_cnt;
    /* for every bit in byte */
    for (i=0; i<8; i++)
    {
        /* extract LSB and shift */
        tmp = byte & 0x01;
        byte >>= 1;
        if (tmp)
            ones_cnt ++;
        else {
            ones_cnt = 0;
            out = ~out;
        }
#ifdef INTSCRAMBLER
        tx = (out ^ (scrambled1 >> 11) ^ (scrambled2)) & 0x1;
        scrambled2 = (scrambled1 >> 15);
        scrambled1 = (scrambled1 << 1) | tx;
#else
        tx = (out ^ (scrambled2 >> 3) ^ (scrambled3)) & 0x1;
        scrambled3=(((signed char)scrambled2) < 0) ? 1 : 0;
```

```
        scrambled2<<=1;
        scrambled2+=((signed char)scrambled1)<0? 1:0;
        scrambled1 = (scrambled1 << 1) | tx;
#endif
        SPI_SendBit(tx);
        /* if sent bit is 1 increment ones counter */
        /* if we have reached max number
        * of ones allowed by AX.25 proto */
        if (ones_cnt >= MAX_AX25_ONES)
        {
            // ... and we reset ones counter
            ones_cnt = 0;
            // ... we send a stuffing 0 bit...
            out = ~out;
#ifdef INTSCRAMBLER
            tx = (out ^ (scrambled1 >> 11) ^ (scrambled2)) & 0x1;
            scrambled2 = (scrambled1 >> 15);
            scrambled1 = (scrambled1 << 1) | tx;
#else
            tx = (out ^ (scrambled2 >> 3) ^ (scrambled3)) & 0x1;
            scrambled3=((signed char)scrambled2)<0? 1:0;
            scrambled2<<=1;
            scrambled2+=((signed char)scrambled1)<0? 1:0;
            scrambled1 = (scrambled1 << 1) | tx;
#endif
            // send it
            SPI_SendBit(tx);
        }
        if(flag)
            ones_cnt=0;
    }
}

void AX25_SendPacket(unsigned char * packet, unsigned int packet_len)
{
    int i;
    scrambled1 = 0;
    scrambled2 = 0;
#ifndef INTSCRAMBLER
    scrambled3 = 0;
#endif
    out = 0;
    INITFCS();
    pppfcs( (unsigned char *)prologo, PROLOGO_LEN);
```

```
        pppfcs( packet, packet_len);
        CPLFCS();
        SPI_ResetBit();
        /* prologo transmission */
        for (i=0; i<TX_DELAY; i++)
           AX25_SendByte(FLAG, 1);
        /* send prologo */
        for (i=0; i<PROLOGO_LEN; i++)
           AX25_SendByte (prologo[i], 0);
        /* send packet */
        for (i=0; i<packet_len; i++)
           AX25_SendByte (packet[i], 0);
        /* now we send 15:8 bits MSB first */
        AX25_SendByte(fcs.b.crc2, 0);
        /* and now 7:0 MSB first */
        AX25_SendByte(fcs.b.crc1, 0);
        for (i=0; i<2; i++)
           AX25_SendByte(FLAG, 1); /* ones count =-3 removes bit stuffing from Flag
TX */
        AX25_SendByte(0x00, 1);    //wait for the end of transmission (one more byte,
to ensure reception)
     }

     unsigned int AX25_ReceivePacket(unsigned char *data)
     {
        int flag_counter = 0;
        char byte = 0x00;
        char DIN;
        short volatile semaforo;
        semaforo = 5000;
        TIMER_SetupTimer_ms(&semaforo);
        do
        {
           scrambled1 = 0;
           scrambled2 = 0;
           byte = 0;
           do
           {
              ONEScount = 0;
              lastBit = 0;
              totalbytes = 0;
              // Check for flag ------------------------------------------------------------------
              while ( (byte != FLAG) && semaforo )              // Check for flag
              {
```

```
            byte = byte >> 1;    // shift byte 1 bit to the right inserting a 0 as MSB
            while( (P5IN & DCLK) == 0 );    // wait for CC1020 clock to rise and then
            //P1OUT = 0x01;
            DIN = ( (P5IN & DIO) != 0 );          // sample nextBit from CC1020 DIO
            nextBit = (DIN ^ (scrambled1 >> 11) ^ (scrambled2)) & 0x1;
            scrambled2 = (scrambled1 >> 15);
            scrambled1 = (scrambled1 << 1) | DIN;
            if ( nextBit == lastBit )             // if no change, then bit is a 1 using NRZI
            {
               byte = byte | 0x80;// insert a 1 as the MSB               }
            else                                   // otherwise, bit is a 0 using NRZI
            {
               byte &= 0x7f;          // insert a 0 as the MSB
            }
            lastBit = nextBit;
            // change the value of lastBit for next NRZI comparison
            //P1OUT = 0x00;          // one bit descrambing needs 10 us
            while( (P5IN & DCLK) != 0 );         // wait for next clock cycle
         }
         // End flag check ------------------------------------------------------------------
         byte = 0x00;                          // reset byte so it is no longer 0x7e
         totalbytes = 0;
         flag_counter = 1;
         byte = RxByte();
         while ( byte == FLAG )
         {
            byte = RxByte();
            flag_counter++;
         }
      }while ((flag_counter < MIN_FLAG_PREAMBLE) && semaforo);

      INITFCS();          //initial FCS value

      while (( byte != FLAG ) && (totalbytes < MAX_PACKET_SIZE) && semaforo )
      {
         data[totalbytes] = byte;
         totalbytes++;
         //P1OUT = 0x01;
         fcs.u = (fcs.u >> 8) ^ fcstab[(fcs.u ^ data[totalbytes-1]) & 0xFF];
         if ( (fcs.b.crc2 == (data[totalbytes-2]^0xFF)) && (fcs.b.crc1 ==
(data[totalbytes-1]^0xFF)) )
            return totalbytes-2;
         //P1OUT = 0x00;              // one byte CRC caculation needs 5us when
FOSC = 8 MHz
```

```
            byte = RxByte();
          }
        } while ( (totalbytes < MIN_PACKET_LENGTH) && semaforo);
        /*
        AX25_ComputeFCS(data, totalbytes);                    //receiver caculates CRC
        if ( (fcs.b.crc2 == data[totalbytes-2]) && (fcs.b.crc1 == data[totalbytes-1]) )
        //            if  ( (fcs.b.crc2 == (data[totalbytes-2]^0xFF)) && (fcs.b.crc1 ==
(data[totalbytes-1]^0xFF)) )
        return totalbytes-2;
          else
        return 0;
        */
        return 0;
      } // End void RxPacket(void)

      char RxByte(void)
      {
        char byte = 0x00;
        char DIN;
        for ( int len = 0; len < 8; len++ )                    // for all 8 bits
        {
          byte = byte >> 1;                                  // shift the byte over to the right
inserting a 0 for MSB
          while( (P5IN & DCLK) == 0 );        // wait for CC1020 clock to rise and then
          DIN = ( (P5IN & DIO) != 0 );              // sample nextBit from CC1020 DIO
          nextBit = (DIN ^ (scrambled1 >> 11) ^ (scrambled2)) & 0x1;
          scrambled2 = (scrambled1 >> 15);
          scrambled1 = (scrambled1 << 1) | DIN;
          if ( nextBit != lastBit )
          {
            byte &= 0x7f;                                  // zero out MSB
            ONEScount = 0;                                  // reset ONEScount
          }
          else if ( nextBit == lastBit )          // then no change so one in NRZI
          {
            byte = byte | 0x80;                // MSB when shifting right
            ONEScount = ONEScount + 1;
          }
          lastBit = nextBit;
          while( (P5IN & DCLK) != 0 );        // wait for next clock cycle

          //-----------------------------------------------------------------------------------
          // if there have been 5 ones and the next bit is a zero (change in bit stream)
          // then remove the bitstuffed zero.
```

```
            if ( ONEScount >= 5 )
            {
               //still need to right shift byte---lv
               byte = byte >> 1;
               // Check the next bit to see if it is a bitstuffed zero; if it is not then it
               // is probably the flag
               len = len + 1;
               while( (P5IN & DCLK) == 0 );          // wait for CC1020 clock to rise and
then
               //for (int j = 0; j < 10; j++) asm("nop;");
               DIN = ( (P5IN & DIO) != 0 );               // sample nextBit from CC1020 DIO
               nextBit = (DIN ^ (scrambled1 >> 11) ^ (scrambled2)) & 0x1;
               scrambled2 = (scrambled1 >> 15);
               scrambled1 = (scrambled1 << 1) | DIN;
               if (nextBit != lastBit)         // then zero needs to be skipped
               {
                  lastBit = nextBit;           // fix lastBit for NRZI
                  ONEScount = 0;                  // reset ONEScount
                  len--;
                  byte = byte << 1;           //shift back, zero skipped---lv
                  while( (P5IN & DCLK) != 0 );         // wait for next clock cycle
               }
               else                                          // otherwise, flag byte has been
encountered
               {
                  while( (P5IN & DCLK) != 0 );         // wait for next clock cycle
                  ONEScount = 0;                  // Reset ones since we're assuming a flag
has been received
                  len = len + 1;
                  while( (P5IN & DCLK) == 0 );    // wait for CC1020 clock to rise and then
                  DIN = ( (P5IN & DIO) != 0 );           // sample nextBit from CC1020 DIO
                  nextBit = (DIN ^ (scrambled1 >> 11) ^ (scrambled2)) & 0x1;
                  scrambled2 = (scrambled1 >> 15);
                  scrambled1 = (scrambled1 << 1) | DIN;
                  lastBit = nextBit;
                  while( (P5IN & DCLK) != 0 );         // wait for next clock cycle
                  return 0x7e;                  // If we have 6 ones then it is either the flag
                  // or an error.    Either way we trick TxPacket into
                  // thinking it is an end flag by setting it as 0x7e &
               }
            }
         }
         return byte;
      }
```

SPI.h

```
#ifndef SPI_H
#define SPI_H
#define DIO             BIT2
#define DCLK            BIT3
#define LOCK_BIT    BIT1
void SPI_Init(void);
void SPI_Disable(void);
void SPI_SendByte (unsigned char data);
unsigned char SPI_ReceiveByte (void);
void SPI_ResetBit (void);
void SPI_SendBit (unsigned char data);
#endif
```

SPI.c

```
#include "main.h"
#include "uart.h"
#include "SPI.h"
#include <msp430x14x.h>

unsigned char SPI_tmp_value;
signed char SPI_tmp_index;

void SPI_Init(void)
{
  P5SEL |= DIO | DCLK | LOCK_BIT;            // P5.1,2,3 SPI option select
  U1CTL = CHAR + SYNC + SWRST;               // 8-bit, SPI, Slave
  //U1TCTL = STC;                            // Polarity, SMCLK, 3-wire
  U1TCTL = 0x82;                                   //UCLK delayed one half
circle--lv
  U1BR0 = 0x02;                              // SPICLK = SMCLK/2
  U1BR1 = 0x02;
  U1MCTL = 0x00;                             //no modulation---lv
  ME2 |= USPIE1;                             // Module enable
  U1CTL &= ~SWRST;                           // SPI enable SWRST = 0x01
}

void SPI_Disable(void)
{
  P5SEL &= ~(DIO | DCLK | LOCK_BIT);          // P5.1,2,3     I/O function is
selected
```

```
    P5DIR &= ~(DIO | DCLK | LOCK_BIT);          //set SPI pin as input
    ME2 &= ~USPIE1;                             // Module disable
    U1CTL |= SWRST;                             // SPI disable, software reset
enable, usart logic held in reset state.
  }

  // Function Transmits Character from RXTXData Buffer
  void SPI_SendByte (unsigned char data)
  {
    while (!(IFG2 & UTXIFG1));                  // USART1 TX buffer ready?
    TXBUF1 = data;
  }

  // Function Receive Character from RXTXData Buffer
  unsigned char SPI_ReceiveByte ( void )
  {
    unsigned char data;
    while (!(IFG2 & URXIFG1));                  // USART1 RX buffer ready?
    data = RXBUF1;
    return data;
  }

  void SPI_ResetBit (void)
  {
    SPI_tmp_value = 0;
    SPI_tmp_index = 7;
  }

  // Function Transmits Character from RXTXData Buffer
  void SPI_SendBit (unsigned char data)
  {
    SPI_tmp_value += (data & 1) << SPI_tmp_index;
    SPI_tmp_index--;
    if (SPI_tmp_index < 0)
    {
      //UART_SendByte(SPI_tmp_value);
      while (!(IFG2 & UTXIFG1));                // USART1 TX buffer ready?
      TXBUF1 = SPI_tmp_value;
      SPI_tmp_index = 7;
      SPI_tmp_value = 0;
    }
  }
```

uart.h

```
#ifndef UART_H
#define UART_H
#include <msp430x14x.h>
#define DATARATE 9600
#define RXD          32                          // RXD on P3.5
#define TXD          16                          // TXD on P3.4
#define Bitime_5   ((int)((float)FOSC/(1.91*DATARATE)))
// ~ 0.5 bit length + small adjustment
#define Bitime      ((int)(((float)(FOSC))/DATARATE))
// 8.6 us bit length ~ 115942 baud

void UART_SendByte (unsigned char data);
unsigned char UART_ReceiveByte (void);
void UART_Init(void);
void printUART (unsigned char *message);
unsigned char CRC( unsigned char *pDato, unsigned short dim);
unsigned char UART_ReceivePacket(unsigned char *pDato);
unsigned char uart_SendPacket(unsigned char *pDato);
#endif
```

uart.c

```
#include <msp430.h>
#include "main.h"
#include "uart.h"
#include "SPI.h"

unsigned int RXTXData;
unsigned char BitCnt;
unsigned short uart_i;
unsigned char uart_crc;
unsigned char uart_crc1;
unsigned char uart_crc2;
unsigned char CRC( unsigned char *pDato, unsigned short dim)
{
   uart_crc = 0;
   for ( uart_i=0; uart_i<dim ; uart_i++ )
      uart_crc += pDato[uart_i];
   return uart_crc;
}

void printUART (unsigned char *message)
```

```
    {
      int i;
      for (i=0; message[i]!=0; i++) UART_SendByte(message[i]);
    }

    void UART_Init(void)
    {
      P3SEL |= 0x30;                                    // P3.4,5 = USART0
TXD/RXD
      ME1 |= UTXE0 + URXE0;                             // Enabled USART0
TXD/RXD
      UCTL0 |= CHAR;                                    // 8-bit character
      UTCTL0 |= SSEL1 + SSEL0 + URXSE;    // UCLK = SMCLK, start edge detect
      UBR00 = (unsigned char)(FOSC/9600);             // 9600 bps
      UBR10 = (unsigned char)((FOSC/9600) >> 8);// 9600 bps
      UMCTL0 = 0x00;                                    // no modulation
      UCTL0 &= ~SWRST;                        // Initialize USART state machine
    }

    // Function Transmits Character from RXTXData Buffer
    void UART_SendByte (unsigned char data)
    {
      while (!(IFG1 & UTXIFG0));                        // USART0 TX buffer ready?
      TXBUF0 = data;
    }

    // Function Transmits Character from RXTXData Buffer
    unsigned char UART_ReceiveByte ( void )
    {
      unsigned char data;
      while (!(IFG1 & URXIFG0));                        // USART0 RX buffer ready?
      data = RXBUF0;
      return data;
    }

    unsigned char UART_ReceivePacket(unsigned char *pDato)
    {
      pDato[0] = UART_ReceiveByte();
      pDato[1] = UART_ReceiveByte();

      for ( uart_i = 2 ; uart_i < ((short)pDato[1]) + 2 ; uart_i++ )
        pDato[uart_i] = UART_ReceiveByte();
      //*dim = pDato[1];
      pDato[uart_i] = UART_ReceiveByte();
```

```
    uart_crc2 = CRC(pDato, ((short)(pDato[1])) + 2 );
    if (uart_crc2 == pDato[uart_i])
        return 1;
    else
        return 0;
}

unsigned char uart_SendPacket(unsigned char *pDato)
{
    for ( uart_i = 0 ; uart_i < pDato[1] + 2 ; uart_i++ )
        UART_SendByte(pDato[uart_i]);
    uart_crc = CRC(pDato, pDato[1] + 2);
    UART_SendByte(uart_crc);
    return uart_i;
}
```

timer.h

```
#ifndef TIMER_H
#define TIMER_H
void TIMER_SetupTimer_ms(short volatile *semaforo);
void TIMER_Wait_ms(short volatile semaforo);
void TIMER_Wait_us(short volatile semaforo);
#endif
```

timer.c

```
#include <msp430x14x.h>
#include "timer.h"
#include "uart.h"
#include "main.h"
short volatile *contatore;

void TIMER_SetupTimer_ms(short volatile *semaforo)
{
    contatore = semaforo;
    TACTL = 0;                                          //stop the timer
    TAR = 0;                                            //reset timer
    TACCR0 = (short)(FOSC/1000);                        // 1 ms at 4 MhZ
    TACTL = TASSEL1 | MC_1 | TAIE;        //start!
    TACCTL0 = CCIE;                                     //enable interrupt
}

void TIMER_Wait_ms(short volatile semaforo)
```

```
{
   contatore = &semaforo;
   TACTL = 0;                                          //stop the timer
   TAR = 0;                                            //reset timer
   TACCR0 = (short)(FOSC/1000);                              // 1 us at 4 MhZ
   TACTL = TASSEL1 | MC_1 | TAIE;       //start!
   TACCTL0 = CCIE;                                     //enable interrupt
   while(semaforo);
}

void TIMER_Wait_us(short volatile semaforo)
{
   contatore = &semaforo;
   TACTL = 0;                                          //stop the timer
   TAR = 0;                                            //reset timer
   TACCR0 = (short)(FOSC/1000000);                     // 1 us at 4 MhZ
   TACTL = TASSEL1 | MC_1 | TAIE;       //start!
   TACCTL0 = CCIE;                                     //enable interrupt
   while(semaforo);
}

// Timer A0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
   TACTL &= ~TAIFG;                 //reset interrupt flag
   (*contatore)--;
   if ((*contatore) == 0)
   {
      TACCTL0 = 0;
      TACTL = 0;
   }
}
```

Reference

[1]. Stefano Speretta, Leonardo M. Reyneri, Claudio Sanso´e, Maurizio Tranchero, Claudio Passerone, Dante Del Corso,MODULAR ARCHITECTURE FOR SATELLITES

[2]. J. E. Mazur, An Overview of the Space Radiation Environment

[3]. JANET Barth, IEEE NSREC Short cource SESSION 1, available at http://radhome.gsfc.nasa.gov/radhome/papers/slideshow10/SC_NSREC97/sld001.htm.

[4]. Richard H. Maurer, Martin E.Fraeman, Mark N. Martin, and David R. Roth, Harsh Environments: Space Radiation Environment, Effects, and Mitigation.

[5]. Bryan Klofas (KF6ZEO), Jason Anderson (KI6GIV),A Survey of CubeSat Communication System.

[6]. MSP430F149 DATASHEET, available at www.ti.com.

[7]. CC1020 DATASHEET, available at www.ti.com.

[8]. MSP430x1xxx Family User's Guide, available at www.ti.com.

[9]. William A. Beech, NJ7P, Douglas E. Nielsen, N7LEM, Jack Taylor, N7OO,AX.25 Link Access Protocol for Amateur Packet Radio.

[10]. Aram Perez,Wismer & Becker, Byte-wise CRC Calculations.

[11]. Mike Chepponis, K3MC and Phil Karn, KA9Q, The KISS TNC, available at http://www.ka9q.net/papers/kiss.html.