Design and Implementation of LEO Satellite Transceiver

A Project Report

submitted by

ANSHUMAN GAURAV

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

May 2013

THESIS CERTIFICATE

This is to certify that the thesis titled **Design and Implementation of LEO Satellite Transceiver**, submitted by **Anshuman Gaurav**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. David Koilpillai
Project Guide
Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Dr. Devendra Jalihal
Project Guide
Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 20th May 2013

ACKNOWLEDGEMENTS

I would like to express my greatest gratitude for the people who have helped and supported me throughout my project. I am grateful to my guides, Dr. David Koilpillai and Dr. Devendra Jalihal, for their continuous support. This project would not have been a success without their valuable suggestions and insights. Despite their busy schedule they always managed time to discuss the issues and guide the project in a proper course. I would like to thank Akshay Gulati, Systems Engineer of IITM Sat project, for his time and again feedback on our work. He kept us updated with the developments in this field by other Amateur Radio Satellite groups.

Test bench designed by Jagdish M Mevada was of immense help. It was used to verify the performance of my design when it is subjected to various channel impairments. His test bench helped me in debugging my codes and making my design more robust.

Last but not the least, I express my sincere thanks to all the staff members of different labs who have been more than helpful in testing the performance of my design. They were always ready to discuss about circuit design and equipped me with proper tools to perform experiments and measurements.

ABSTRACT

KEYWORDS: LEO; COM; AX.25; CRC; Interrupt.

This thesis explains the design and implementation of on-board communication system (COM) for a nano-satellite orbiting in Low Earth Orbit (LEO). Starting from simulations of individual communication sub-systems we gradually moved towards its practical realization. Final product designed here will be implemented in IITM-Sat. IITM-Sat is a nano-satellite which is being designed by the students of Indian Institute of Technology, Madras. It will collect data about electrons and protons in the Earth's upper-ionosphere throughout its mission life of one year. This satellite, weighing less than 15 kg, will be placed in a sun-synchronous LEO orbit at an altitude of 600-800 km. Satellites in this orbit will have a Line-of-Sight contact with Ground Station 6-8 times per day. Data collected by the satellite over one complete revolution around the Earth will be stored in the on-board memory and it has to be transmitted to the Ground Station during its visibility period which lasts for around 5 to 15 minutes.

This thesis aims to cover each and every details regarding individual sub-system design. All the modules have been described separately and with a new perspective. Without going into the detailed description of internal structures of these modules, this thesis aims to present their simplified version and tries to fill in the gap between what is given in their data-sheet and how we have implemented it in our design. Separate chapters deals with the hardware and software part.

Modified version of AX.25 protocol is being used for information exchange between the Ground station and Satellite. Keeping the basic handshaking mechanism unchanged, some of the fields inside AX.25 frames were modified for our application with the aim of reducing unnecessary overheads. Separate section has been dedicated for calculating CRC which is one of the several fields inside AX.25 frame.

Effort has been put into organising the complete software in a way that is easy to understand. Complete code has been divided into four blocks. Each block is associated

with one of the sub-system of overall COM system. Each block has been explained using flow chart. Different blocks of code communicates using only few common parameters and this helps in keeping an easy track of program flow.

COM is completely interrupt driven in order to save power when COM is not in use. These interrupts can be given by COM sub-systems or can be software generated. Since there are several interrupts that controls the overall service provided by COM, there is a separate sub-section on Interrupt Management. It describes the Interrupt flow and justifies the Interrupt priorities selected for different sub-systems of COM.

Finally this thesis ends with describing the various test bench that we established to verify the performance of our system. A virtual Ground Station was established to issue telecommands over uplink and receive telemetry information over downlink. USRP (Universal Software defined Radio) was used to create the distorted signal as expected for an actual satellite channel.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

CRLB	Cramer-Rao Lower Bound
SPI	Serial Peripheral Interface
PLL	Phase Locked Loop
AFC	Automatic Frequency Control
RSSI	Received Signal Strength Indication
ISR	Interrupt Service Routine
USRP	Universal Software defined Radio
FSK	Frequency Shift Keying
GMSK	Gaussian Minimum Shift Keying
PA	Power Amplifier

CHAPTER 1

INTRODUCTION

1.1 Project Overview

The Goal of this project is to design the Communication System for a LEO Satellite. Right from the selection of components to the design of hardware and software architecture, this project aims to develop the actual product that would be placed on-board satellite. Apart from this, communication protocol has to be decided and implemented. Error detection codes should be implemented to account for the noisy channel. Separate system has to be designed which can perform System Reset in case of satellite malfunctioning.

This thesis work will be implemented on a nano-satellite that will be placed in a sun-synchronous LEO orbit at an altitude of 900 km. Such satellite will cross over any given point on earth every day at approximately the same local time. It will have a line-of-sight contact with the Ground Station for 6-8 times per day and the visibility period for each pass will vary from 5 to 15 minutes. Data collected over one complete revolution around earth will be stored in a SD card and all of this data has to be sent over downlink during the span of visibility period.

Real time working prototype was designed using evaluation boards and their performance was tested using USRP (Universal Software Radio Peripheral). Upon verification of the system performance, schematic of complete system was designed to be fabricated on a single PCB.

1.2 Design Specification

Uplink:

- FSK Modulation
- Data rate: 1200bps

- Bandwidth: 10KHz
- Center Frequency: 145MHz (HAM Band)

Downlink:

- GMSK Modulation
- BT = 0.3
- Data rate: 19.2kbps
- Bandwidth: 20kHz
- Center Frequency: 435MHz

Occupied bandwidth is defined by article 1.153 of the ITU Radio Regulations (ITU RR) as

> The width of a frequency band such that, below and above the upper frequency limits, the mean powers emitted are each equal to a specified percentage β/2 of the total mean power of a given emission

where β is taken to be 1%. For β = 1%, this is often referred to as the 99% power containment bandwidth.

1.3 Approach to the problem

We started off by studying the major impairments associated with a satellite channel. Doppler Shift being a major issue, we studied the Doppler pattern that satellite will encounter. Simulations were performed in Matlab to obtain a plot of Doppler Shift vs. Time during a visibility period where Doppler Effect will be extreme. This data was later implemented in USRP to create a Doppler corrupted data and which were used as an input to our design. After narrowing into FSK modulation for uplink, we simulated various FSK Modems on Matlab to get the feel of basic working of FSK modem. Algorithms described in [1] and [12] were implemented and results were produced as shown in these papers. Later, these algorithms were used as a basis to design our own FSK demodulator after taking Doppler into account.

As we know that when we calculate N point FFT of any signal, it actually represents N equal spaced samples of the actual spectrum. If we transmit FSK signals from Ground Station such that the frequency of sinusoid, corresponding to bit-0 and bit-1 lie exactly at one of the FFT sampling point, then using above algorithms we can construct an optimal performance receiver. But things become complicated due to Doppler Effect. Doppler shift is observed in the signal exchanged among Satellite and Ground Station due to net radial velocity between the two. This causes the spectral peaks to shift to those points which are not sampled by FFT. This causes Spectral Leakage and hence the system performance degrades.

According to Rife [11], the Maximum Likelihood estimator of the signal frequency is given by the argument of peak of the periodogram. He argued that this estimator can be implemented by a peak search of the amplitude spectrum obtained by FFT, followed by an interpolation on the peak in order to locate the true frequency. Rife also showed that the Standard Deviation of the estimated frequency from the true frequency is given by

$$\sigma = \frac{f_s}{2\pi}\sqrt{\frac{6}{N(N^2-1)SNR}} \tag{1.1}$$

Hence it establishes Cramer-Rao Lower bound (CRLB) for the estimate. [1] describes two algorithms to estimate this frequency corresponding to the peak of spectral amplitude, i.e. Quinn and Dichotomous. Both these algorithms were implemented on Matlab and the results were confirmed by simulations. Dichotomous Algorithm gives a better estimate of the true frequency than Quinn's algorithm but at the cost of increased computation. Finally a "Guided Search of the Periodogram" Algorithm was introduced which combines the above two algorithm and gives better performance at reduced complexity.

Above simulations were done with the view of implementing them on a DSP based processor after down-converting the FSK signal received by Satellite. At the same time we found a more power efficient way of demodulating FSK signal using a microcontroller and an FSK transceiver. ADF7020-1 FSK transceiver by Analog Devices supports our required data rate and using its internal PLL we can perform Doppler correction with no additional complexity. Performing Doppler correction in DSP processor will require lots of data processing, which in turn consumes power. ADF7020-1 can downconvert and demodulate FSK signal by consuming less than 58mW power. MSP430 microcontroller was used to collect data from ADF7020-1 and performs all other work of data interpretation and management. Having fixed our approach to the problem, we started looking for the best available GMSK modules in the market. We are now using CMX7164, which is a GMSK baseband modulator and CMX991, which is an I/Q up-converter. Few more components i.e. SD Card, LO and Power Amplifier (PA) were integrated into a single system which are described later in this thesis.

1.4 Types of Impairments and their effects

1.4.1 Doppler Effect

LEO satellites revolve around earth at about 8 km/s. Since the radial velocity itself changes with time, there is a continuous change in Doppler shift during a visibility period. As a worst case, LEO satellites operating at 145 MHz will experience a variation in Doppler shift from 3kHz to -3kHz in single pass. This pattern of Doppler shift as a function of time is shown in figure 1.1.

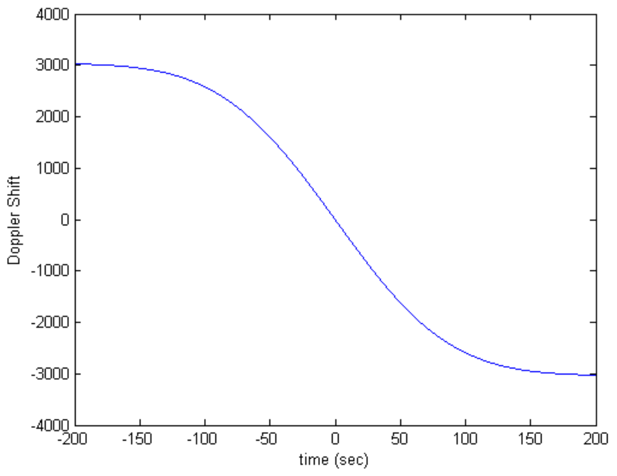


Figure 1.1: Doppler Shift vs. Time

Doppler Shift causes shift in the center frequency of transmitted signal. Doppler corrupted FSK signal is shown in figure 1.2.

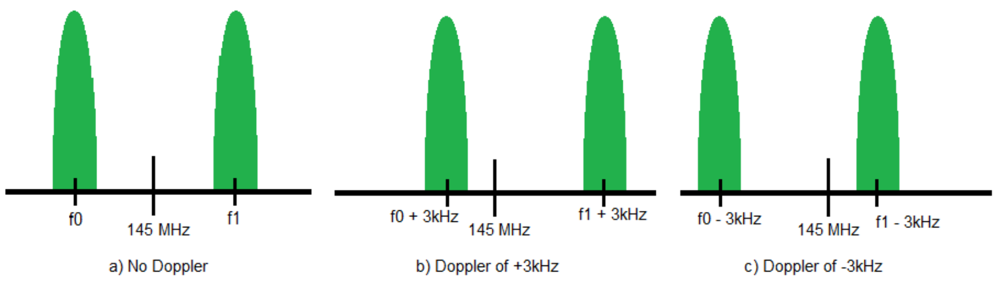


Figure 1.2: Spectrum of Doppler corrupted FSK signal

Thus the receive filter can be designed in two ways: firstly, we can keep the receive filter bandwidth to be wide enough to allow the Doppler shifted signals to pass through it as shown in figure 1.3(a), or we can perform Doppler correction and then adjust the filter bandwidth to be just enough to allow the main lobe to pass as shown in figure 1.3(b). In terms of receiver's performance later one will give less BER since former is allowing more noise to pass through it. We took the second approach for designing uplink receiver. First approach is not even feasible for our design because the Doppler shift is in the order of $f_1 - f_0$.

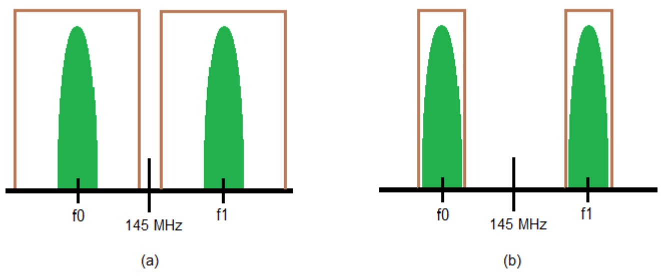


Figure 1.3: Receive filter design (a) Bandwidth is large enough to accommodate Doppler corrupted signal (b) Bandwidth is just enough to allow the main lobes to pass through it

1.4.2 Carrier Leakage

Carrier Leakage is the presence of an un-modulated Carrier within the signal's bandwidth, whose amplitude is independent of the signal's amplitude. DC Offset and direct leakage from LO to output are the two main source of carrier leakage.

Effect of DC offset in input signal is shown in figure 1.4. Just because of the presence of DC offset in input there is an un-modulated carrier present in output signal. Apart from this, even direct leakage of LO can happen because of the non-linearity of multiplier itself. Since in direct up-conversion transmission the LO frequency is same

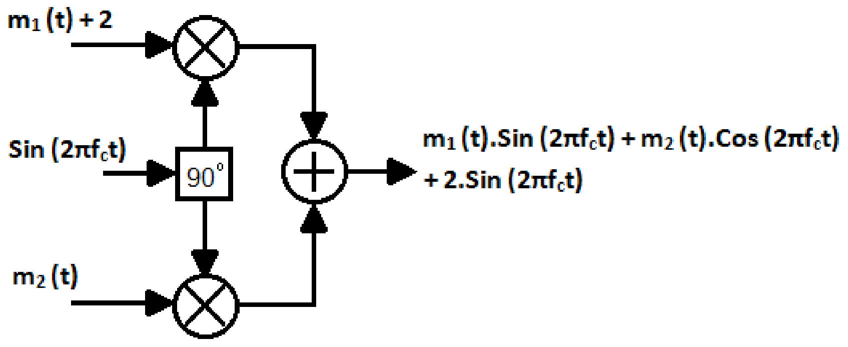


Figure 1.4: Carrier Leakage due to DC offset

as the carrier frequency, even BPF cannot filter out the leaked LO since it lies within signal bandwidth. Carrier leakage can destroy the constellation and raise Error Vector magnitude (EVM). This is where the advantage of heterodyne transmitter comes in.

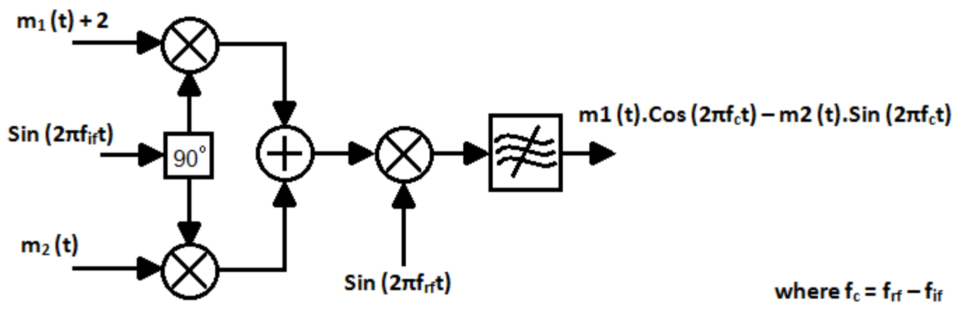


Figure 1.5: Heterodyne up-conversion suppresses Carrier Leakage

As shown in figure 1.5, heterodyne transmitter has two Local Oscillators i.e. IF LO (corresponding to f_{if}) and RF LO (corresponding to f_{rf}). Since Carrier frequency is equal to none of the LO (i.e. Carrier frequency can be either $|f_{rf} - f_{if}|$ or $|f_{rf} + f_{if}|$), leaked signals from these LO can be easily filtered out by BPF (see figure 1.6). As we can easily verify, heterodyne transmitter is even resistant to DC offset. Thus it produces a more legitimate signal.

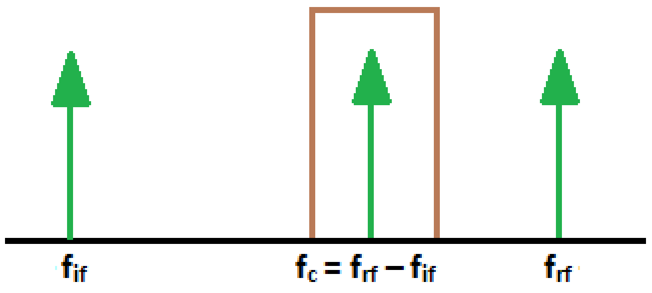


Figure 1.6: BPF centered at Carrier frequency

CMX991 supports both Direct up-conversion and Heterodyne transmission. Initially when we were using CMX991 evaluation board for Direct up-conversion of GMSK signal, we observed distortion in GMSK spectrum and there were issues in synchronizing it when it was fed to USRP. It was also observed that input signals (I and Q) of CMX991 were having DC offset due to some extra components on the evaluation board. These unnecessary components were removed and following this a significant improvement was observed in the received signal. Later we switched over to Heterodyne transmission mode of CMX991.

CHAPTER 2

COM Architecture

On-board satellite there are several systems which co-ordinates with each other to make the whole satellite work. Figure 2.1 presents the network topology over which we are working. HEPD (High Energy Particle Detector) keeps count of the number of particles in ionosphere, ADCS (Attitude Determination and Control System) tries to keep the on-board antenna pointing towards Earth, EPS (Electrical Power System) manages power distribution and finally COM (Communication System) is responsible for communicating with ground station. Our project is to design the COM system for satellite and this whole thesis is dedicated to it.

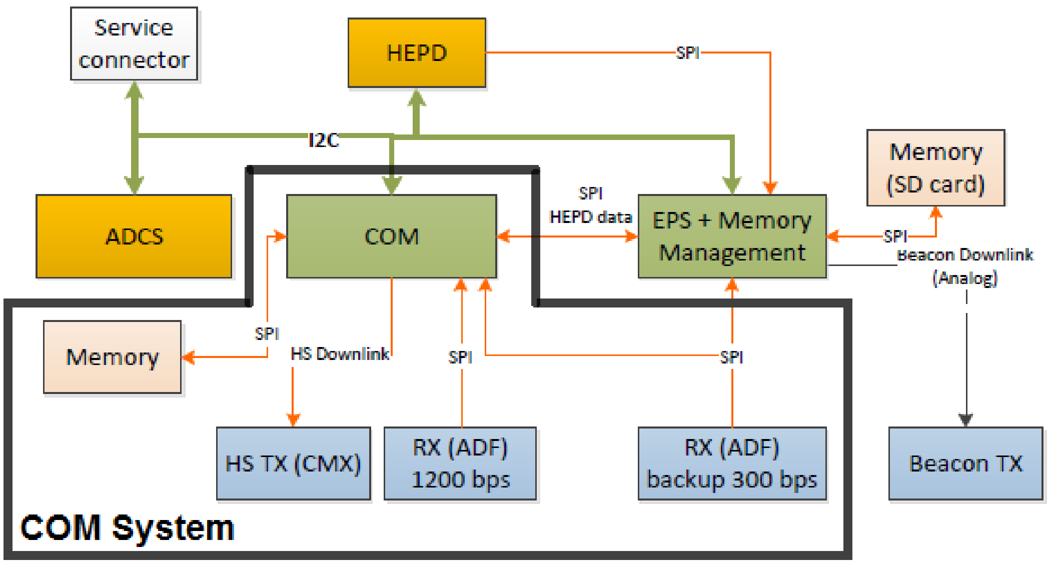


Figure 2.1: Network Topology of on-board System

2.1 Architecture overview

This chapter describes the hardware design of on-board communication system. On-board satellite there will be four microcontrollers and out of which only one will be assigned with the task of communication related activities. We will refer to this particular microcontroller as COM. Figure 2.2 shows the basic block diagram version of our design. Individual blocks in this figure are described in a separate section.

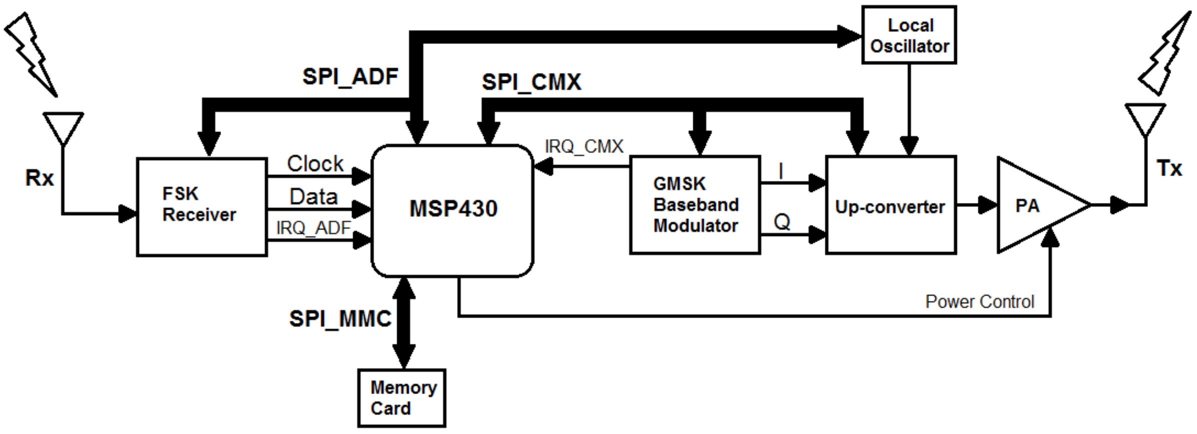


Figure 2.2: COM Architecture

2.2 Overview of Selected Components

2.2.1 FSK Receiver (ADF7020-1)

ADF7020-1 is a FSK/ASK transceiver manufactured by Analog Devices. Though it supports many more modulation schemes, we will program it to receive FSK signals. Some of its desirable features are

- It supports FSK data rate from 150 bps to 200 kbps. Hence we can use it for two purposes. Firstly, as a Telecommand receiver and Secondly, in a System-Reset Arrangement. System-Reset arrangement is discussed in details in Section 2.4.
- It can operate from 80 MHz to 325 MHz. Thus it works in our frequency range of interest i.e. 145 MHz.
- Transceiver RF frequency, Filter Bandwidths, Frequency Deviation, Modulation Schemes can be programmed using SPI interface. Details regarding SPI interface are given in Section 2.3.
- It allows trading off Sensitivity and Selectivity of the receiver with Current consumption, depending on the application.
- Automatic Frequency Control (AFC) loop allows the PLL to compensate for frequency error in the incoming signal. This allows us to counter Doppler.
- Provides the Temperature reading, Battery voltage level and (Received Signal Strength Indication) RSSI signal.

ADF7020-1 downconverts the received FSK signal, corrects doppler, decodes it and gives the decoded data and its corresponding clock signal at its external pin. This Data and Clock can be fed to the SPI of microcontroller in Slave mode.

Details regarding working of individual section of ADF7020-1 in receive mode can be found in its datasheet [3]. This section will describe the way it has been implemented in our design and will stress some of the key points of datasheet. From a laymanŠs point of view ADF7020-1 is being used in the way shown in figure 2.3.

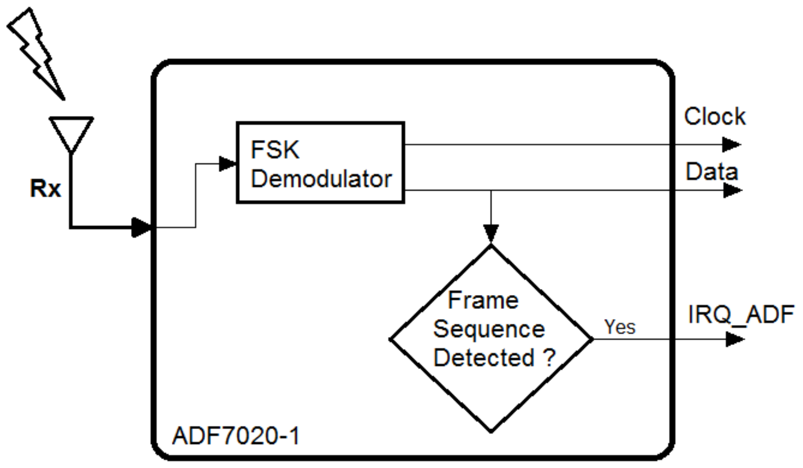


Figure 2.3: A rough overview on the working of ADF7020-1

ADF7020-1 supports the feature of Automatic Sync Word Recognition. Sync Word can be preprogrammed into Register 5 of ADF7020-1. In receive mode, this Sync Word is compared with the incoming bit streams. As soon a Sync Word is detected in the incoming data stream, an Interrupt is generated. This interrupt signal is generated by asserting an external pin INT/LOCK (refer pin diagram of ADF7020-1), which is used by MSP430 as an indication to start capturing data in its SPI Receive Buffer (UCB0RXBUF). INT/LOCK is automatically de-asserted by ADF7020-1 after nine data clock cycles.

C functions and variables which are defined exclusively for ADF7020-1 is present in a separate header **ADF.h**. Complete code is presented in Appendix A.2.

2.2.2 GMSK Baseband Modulator (CMX7164)

CMX7164 is a GMSK baseband modulator. Its basic block diagram in Transmit mode is shown in following figure.

Details regarding working of individual blocks can be found in its datasheet [6]. This section will just give a brief overview and clarify few points which have not been

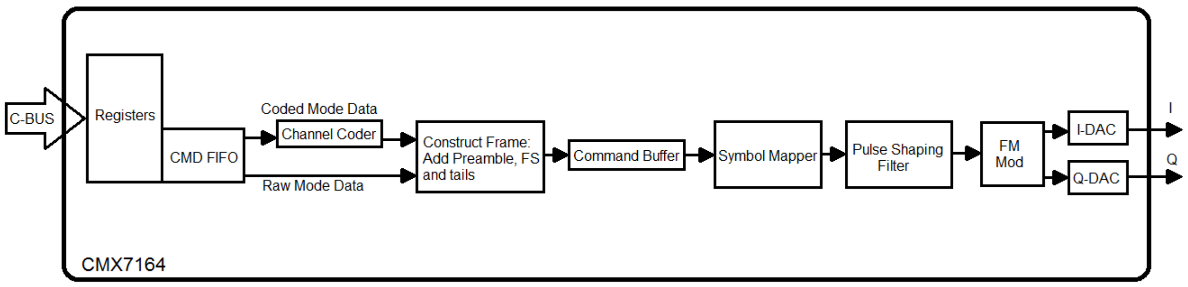


Figure 2.4: Block diagram of CMX7164 in Transmit mode

mentioned explicitly.

C-BUS is a protocol, just like SPI, using which microcontrollers can talk to CMX7164. Before we are able to use CMX7164 as a GMSK baseband modulator, we must load a proper Function Image (FI) onto it. It is provided by CML microcircuits in the form of C Header file. Detailed method of loading the FI into CMX7164 is given in its Datasheet. CMX7164 supports multiple modulation schemes and by loading a proper Function Image we define the operational capabilities of this device.

Continuous stream of data is transferred by COM to CMX7164 using Streaming C-BUS (Refer datasheet of CMX7164 for details regarding Streaming C-Bus). Before we start transferring data from MSP430 to CMX7164 via Streaming C-Bus, it is necessary to program Modem Command FIFO Control Byte (0x4A) and specify exactly how many continuous data bytes will be transmitted at a time via streaming C-BUS. At max 15 bytes can be transferred in a single Streaming C-BUS transaction. To repeat, Modem Command FIFO Control Byte (0x4A) must be written for each C-BUS transaction.

As shown in Figure 2.5, CMX7164 has two internal buffers- **CMD FIFO** and **Command Buffer**. We can transmit data from COM to CMX7164 at a very high speed (up to 10 MHz). These data are first buffered in CMD FIFO and then Channel Coding is done on it (optional), followed by Framing. This Framed data is finally stored in a Command Buffer. CMD FIFO can store 128 data bytes whereas Command Buffer size is 255 bytes. Data will remain buffered in Command Buffer until CMX7164 is instructed to start Transmission.

This architecture allows COM to send bulk of data to CMX7164 at a very high speed and then concentrate on some other job. Various types of interrupts are supported by CMX7164 which can interrupt the COM before CMX7164 runs out of data in Com-

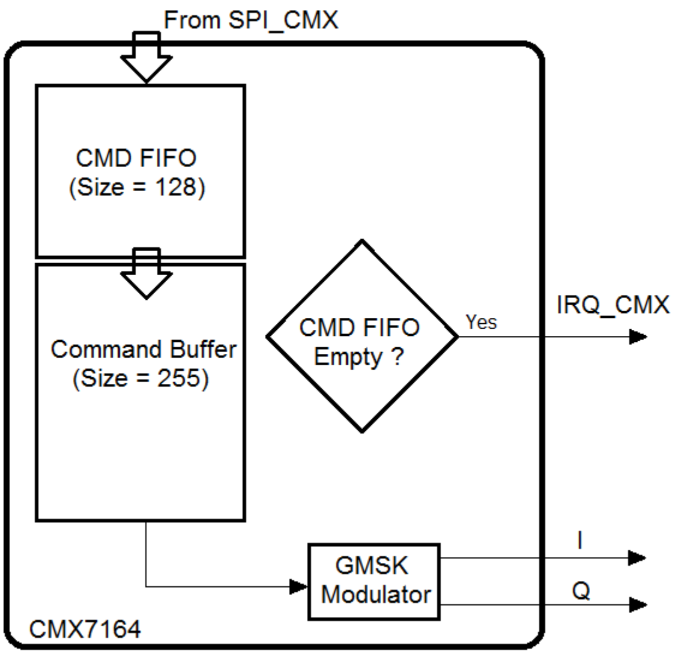


Figure 2.5: A rough overview on working of CMX7164

mand Buffer. In our application, we are generating an interrupt as soon the CMD FIFO is empty. Downlink transmission is occurring at just 19.2 kbps, so, by considering that CMD FIFO is empty and Command Buffer is full, transmitting 15 bytes in a single C-Bus transaction will generate another interrupt at an interval of $\frac{15\times8}{19200} = 6.25m$sec. Since we are using microcontroller at 16 MHz, it has lot of clock cycles before it need to refill CMD FIFO. At the C-BUS speed of 2 MHz it will take roughly $\frac{17\times8}{2\times10^6} = 0.68\mu$sec to transfer 15 bytes of data to CMX7164 (including the time it takes to write in Control Word).

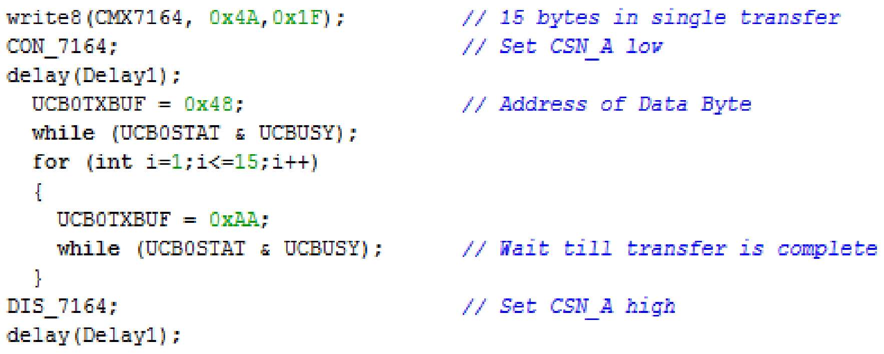
Lastly, the GMSK Baseband output given by CMX7164 is in differential form. I-channel output is divided among I+ and I-, similarly Q-channel is divided into Q+ and Q-. Advantage of using Differential Signalling is that it reduces external interference. Since external interference tend to affect both wires together, and information is sent only by the difference between the wires, the technique improves resistance to electromagnetic noise compared with use of only one wire and an un-paired reference (ground).

C functions and variables which are defined exclusively for CMX modules are present in a separate header **CML.h**. Complete code is presented in Appendix A.3. It defines following functions:

1. **char** read8(**int** reg, **char** add)

2. **unsigned int** read16(**int** reg, **char** add)
3. **void** write8(**int** reg, **char** add, **char** val)
4. **void** write16(**int** reg, **char** add, **unsigned int** val)
5. **int** FI_Load(**void**);
6. **void** CMX7164_initialize(**void**);
7. **void** CMX991_initialize(**void**);

Value of **reg** can be 1 (for CMX7164) or 2 (for CMX991). This is done to use the same function for both CMX7164 and CMX991. **read8()** reads 8-bit register from the address **add** and **read16()** reads 16-bit register from **add**. **write8()** writes 1 byte data (**val**) into the register with address **add** and write16() write 16-bit data (**val**) into the register with address **add**. FI_Load() loads the Function Image into CMX7164 whereas CMX7164_initialize() and CMX991_initialize() initializes the two modules before we start using it. Wherever we need to use 16-bit data, data-type for that variable was defined as *unsigned int* (IAR Compiler defines its size as 2-byte). Rather than sending one byte of data at a time using **write8()**, we can send at max 15 bytes of data at a time using streaming C-BUS. Figure 2.6 shows the C-code to transfer 15 byte of data into CMX7164 using streaming C-BUS.



```
write8(CMX7164, 0x4A,0x1F);          // 15 bytes in single transfer
CON_7164;                            // Set CSN_A low
delay(Delay1);
  UCB0TXBUF = 0x48;                  // Address of Data Byte
  while (UCB0STAT & UCBUSY);
  for (int i=1;i<=15;i++)
  {
    UCB0TXBUF = 0xAA;
    while (UCB0STAT & UCBUSY);       // Wait till transfer is complete
  }
DIS_7164;                            // Set CSN_A high
delay(Delay1);
```

Figure 2.6: Using Streaming C-BUS to write 15 bytes in a single transaction

2.2.3 RF Quadrature Transmitter(CMX991)

CMX991 is a RF Quadrature Transceiver. We are using it as a Transmitter which up-converts the complex baseband signals (I and Q) generated by CMX7164. It is basically a heterodyne transmitter and the overview of its working is shown in figure 2.7, only difference being that the signals are in Differential form.

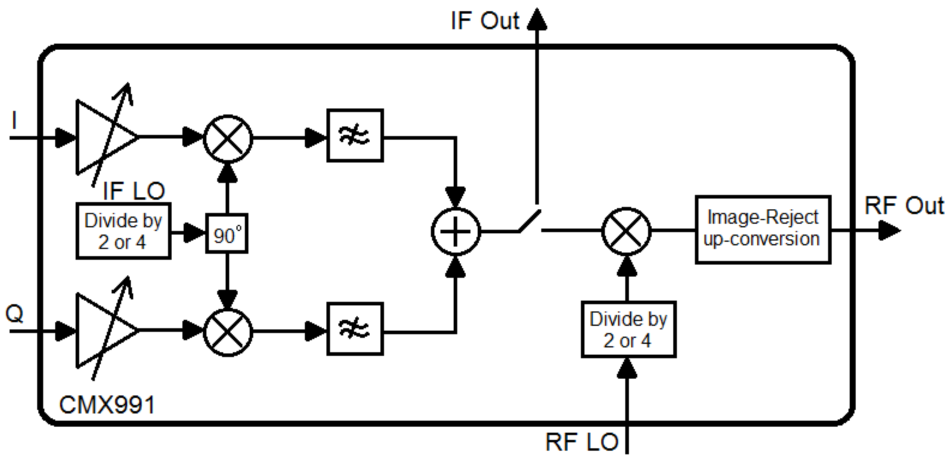


Figure 2.7: Block diagram of CMX991 in transmit mode

CMX991 has inbuilt IF PLL and IF VCO subsystems but it needs an external Loop Filter to generate IF LO signal. This IF LO signal is used to modulate the I/Q input which are finally up-converted and then transmitted after image-rejection. The I/Q inputs are the GMSK Baseband Differential signals generated by CMX7164. As represented in figure, it is possible to use IF OUT directly by setting Bit_0 = 1 in Tx Control Register (0x14). But direct up-conversion of the baseband signal gives rise to carrier leakage and as explained in section 1.4.2, carrier leakage can destroy the constellation and raise Error Vector magnitude (EVM). Hence we will use heterodyne architecture for transmission. For this we need two Local Oscillators (LO) i.e. IF LO and RF LO. Though IF LO can be developed internally, it does not provide an internal RF LO. RF LO must be applied from an external source and for which we are using ADF4351 frequency synthesizer from Analog Devices.

Considering that IF LO frequency is f_{if} and RF LO frequency is f_{rf}, output will be present at both $f_{rf} - f_{if}$ and $f_{rf} + f_{if}$. Work of Image-Reject up-converter is to remove one of this carrier. Which one will be removed is decided by Bit_2 of Tx Mode Register (0x15). If Bit_2 = 0 then $f_c = f_{rf} - f_{if}$ whereas if Bit_2 = 1 then $f_c = f_{rf} + f_{if}$.

Just like CMX7164, even CMX991 needs C-BUS serial interface for transfer of control and status information between its internal registers and an external host (COM). Internal registers needs to be written only once after reset and before we start using it. Since this C-BUS connection with host won't be used after initialization is over, we have merged it with the C-BUS connection of CMX7164. Thus, same C-BUS wires connects host (MSP430), CMX7164 and CMX991 except CSN. C-BUS includes a CSN line which has to be asserted before data transfer. By using different CSN for CMX7164 and CMX991 we can keep the information meant for CMX991 separate from CMX7164's information. Further details regarding register description, device architecture and C-BUS interfacing can be found in its datasheet [7].

2.2.4 Memory Card

Transcend 2GB microSD card is used to store the data collected by Satellite payload. Data was stored in raw format. Different SD card can has its own specification of maximum data rate. Our data rate over this SPI interface is 4 MHz. Transcend 2GB microSD card can support up to 25 MHz in data transfer mode. Details regarding basic interface with SD cards can be found in [9]. SD Card supports multiple mode of operation (i.e. SD mode and SPI mode). SPI being simple, we will use microSD card in SPI mode to communicate with COM.

Normally, card initialization starts by setting SPI Clock to 400 kHz and sending some basic commands. If card responds positively, then by reading TRAN_SPEED field in CSD register (Refer [2]) we can know the maximum data transfer rate supported by that card. Using this information COM can adjust its SPI rate to maximum possible value. This is required for compatibility across different Cards. In our Satellite application, since the memory card wont be changed once the satellite is gone, we can skip this step and from the very beginning we can program SPI speed to the maximum possible value.

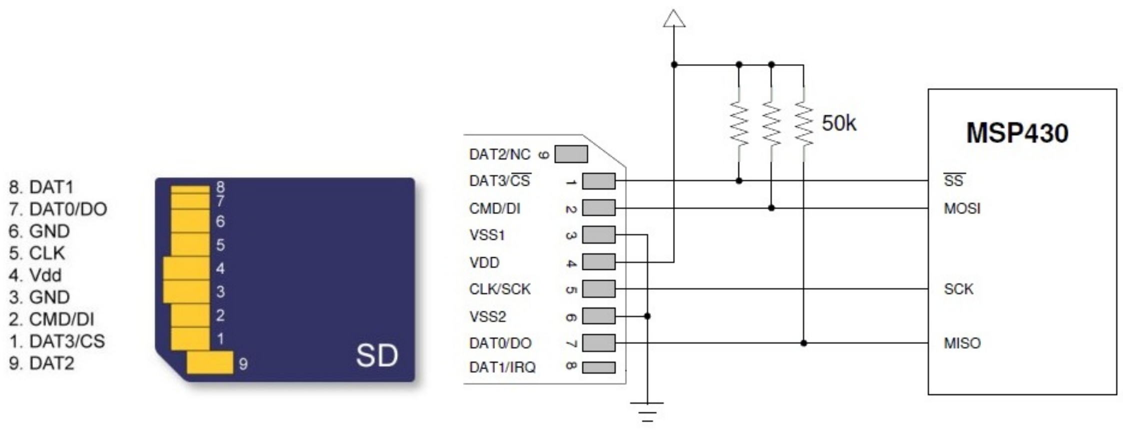


Figure 2.8: Pin configuration and Circuit diagram of Memory Card

Following steps have to be followed to place SD Card in SPI Mode

- After power-on, wait for about 1ms and then issue at-least 74 clocks before any attempt is made to communicate with the card. This allows the card to initialize any internal state registers before card initialization proceeds.
- Next, the card is Reset by issuing the command CMD0 while holding the SS pin low. This both resets the card and instructs it to enter SPI mode. By default CRC is enabled, hence SD card will perform CRC check for each command that has been transferred. While the CRC, in general, is ignored in SPI mode, the very CMD0 command must be followed by a valid CRC, since the card enters SPI mode only after the issue of CMD0. The CRC byte for a CMD0 command with a zero argument is a constant 0x95 and can be hard coded into microcontroller.
- The card initiates the initialization process when a CMD1 is received. To detect end of the initialization process, the host controller must send CMD1 and check the response until end of the initialization. When the card is initialized successfully, In Idle State bit in the R1 response is cleared (i.e. R1 response changes from 0x01 to 0x00). The initialization process can take hundreds of milliseconds, so that this is a consideration to determine the time out value. After the In Idle State bit is cleared, generic read/write commands will be accepted.
- Finally, Block Size is programmed to 512 bytes.

Data can be read only from the starting address of each Block. Under the header file SD_Card.h there are following four functions which perform all SD Card related transactions:

1. **int** SD_init(**void**);
2. **unsigned char** SD_sendCommand(**unsigned char** cmd, **unsigned long int** arg, **char** crc);
3. **unsigned char** SD_readSingleBlock(**unsigned long int** startBlock);
4. **unsigned char** SD_writeSingleBlock(**unsigned long int** startBlock);

SD_init() function initializes SD Card in the way described above. **SD_sendCommand()** sends command to SD Card. Here **arg** represents an argument associated with a particular command and **crc** represents 8-bit CRC of the command. As described earlier only CMD0 needs a valid CRC, for other commands we can put it as 0. **SD_readSingleBlock()** function reads 512 bytes from a block whose index number is **startBlock**. Similarly, **SD_writeSingleBlock()** function writes 512 bytes of data to the block with index number **startBlock**.

C functions and variables which are defined exclusively for CMX modules are present in a separate header **SD_Card.h**. Complete code is presented in Appendix A.4.

2.2.5 COM (MSP430)

COM is the name given to that microcontroller which manages all communication related activities of Satellite with Ground Station. Considering the large amount of constant data that has to be placed on-board, we selected MSP430F2619 microcontroller. It offers 120 kB of Code memory and 4 kB of RAM. Code memory stores all constant data in our code and the code itself. We needed Code memory to be large enough to accommodate code along with following constant data:

- Function Image (FI). Size of FI itself is 52 kB.
- Look-up table to calculate CRC. Size of this look-up table is 512 bytes.

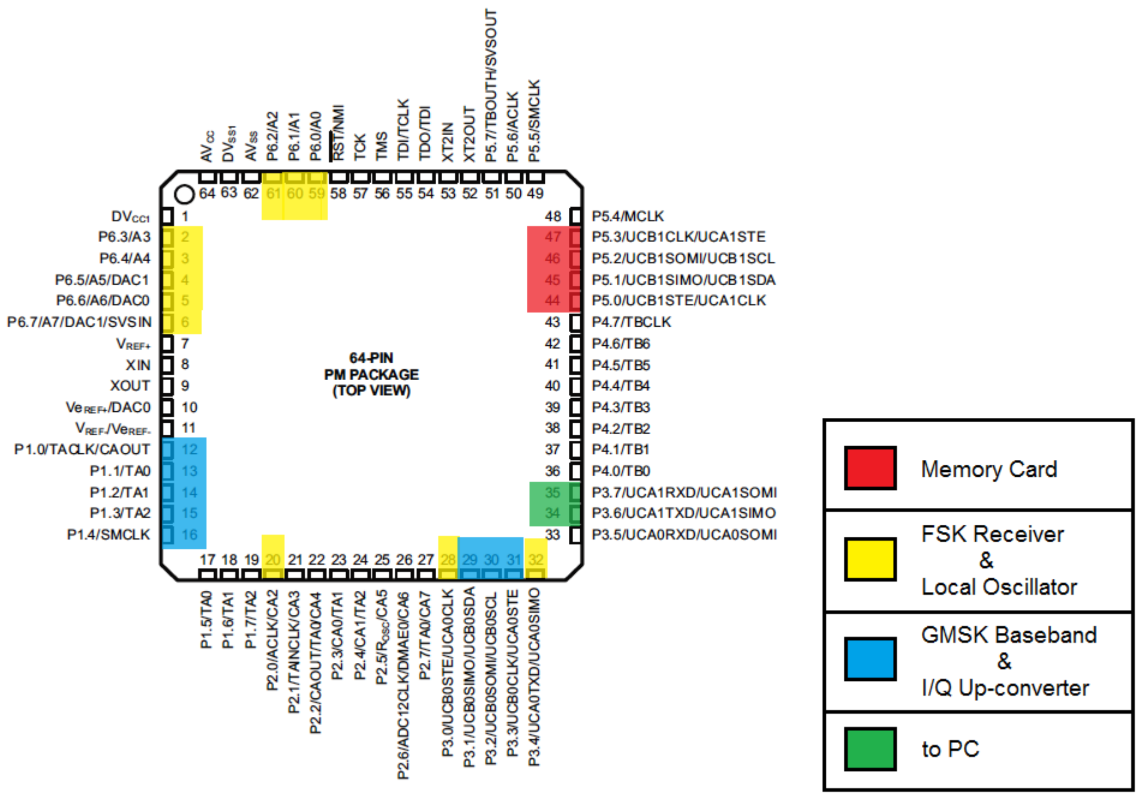


Figure 2.9: MSP430 Connection with other Modules

It provides four USCI modules (USCI_A0, USCI_B0, USCI_A1 and USCI_A2) which support multiple serial communication modes. As shown in figure 2.9, USCI_A0, USCI_B0 and USCI_B1 were used in SPI mode (described in next section) to interface with ADF7020-1, CMX modules and Memory Card respectively. Both the CMX modules i.e. CMX7164 and CMX991 share a common SPI except Chip-Select (CSN). As shown in main figure, CSN_A goes to CMX7164 and CSN_B goes to CMX991. Fourth

USCI module namely USCI_A1 was used in UART mode to communicate with PC. Received tele-commands can be directly observed in PC Hyperterminal using UART. This helped us a lot in debugging our system. Finally, Bit-banging was implemented on pins 2, 3, 4 and 5 to program the internal registers of ADF7020-1. Details regarding individual pin connections are given in Appendix A.1. Full details regarding MSP430 features are provided in [5].

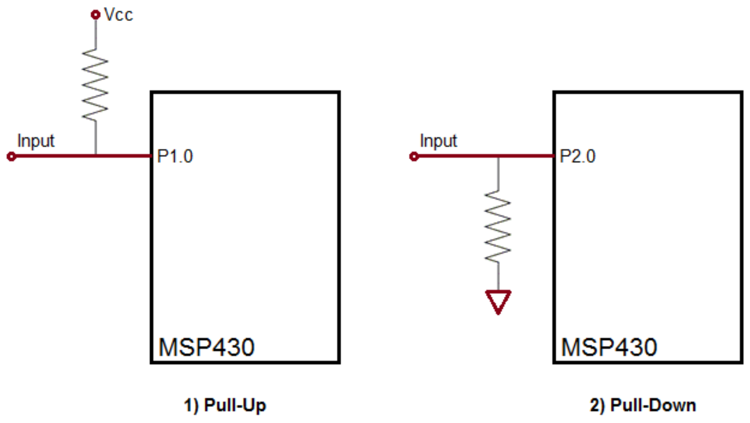


Figure 2.10: Pull-up and Pull-down resistor

Another desirable feature of this microcontroller is that it provides a provision for Pull-up/down its I/O pins. The basic function of a Pull-up/down resistor is to ensure that any floating input will be allocated a default value. If nothing is connected to an input pin, the value of the input is considered to be floating. Most gates will float towards a high state but this is a very weak condition, and any electrical noise could cause the input to go low. Function of Pull-up and Pull-down circuit is same i.e. to create a default value for a floating input, but Pull-up pulls the line high whereas Pull-down pulls it low. Reason for its importance is that floating input becomes very sensitive to its immediate surrounding and picks up a random signal of 1Šs and 0Šs even when we place our fingers close to the pin. Figure 2.10 shows how external resistor is used to pull an input pin of any module. P1.0 of MSP430 is connected to CMX7164 interrupt whereas P2.0 is connected to ADF7020-1 interrupt. CMX7164 interrupt is active-low, so P1.0 is Pulled-up. Interrupt from ADF7020-1 is active-high, so P2.0 is Pulled-low. Having an internal Pull-up/down resistor in microcontroller spares us from including an extra resistor in our design. Price that we pay by enabling this feature is in terms of increased response time. High speed transactions cannot be performed by enabling internal Pull.

2.2.6 Local Oscillator (ADF4351)

ADF4351 is a frequency synthesizer from Analog Devices. It is being used as an RF LO for CMX991. According to the datasheet of CMX991, RF LO input frequency has to be reduced by a factor of 2 or 4. Following figure is taken from CMX991 datasheet which highlights this point. External RF LO is connected across pin TXLOP and pin TXLON.

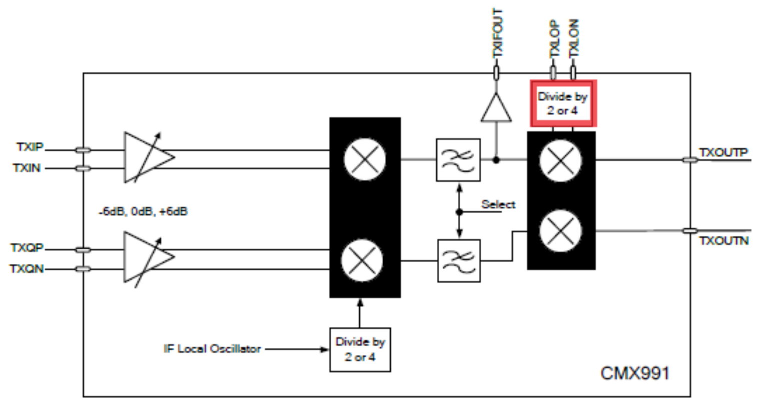


Figure 2.11: Internal structure of CMX991

In our application CMX991 has been configured for $f_c = f_{rf} - f_{if}$ and $f_{if} = 90MHz$. So to get an output at $435MHz$, f_{rf} must be equal to $525MHz$. Since we are dividing RF LO input by 2, ADF4351 must generate $525 \times 2 = 1050MHz$ signal. Refer to its datasheet [4] for further details.

2.2.7 Power Amplifier (RF5110)

Power of GMSK modulated output from RF Quadrature transmitter is 0 dBm. As per our link budget we need to transmit GMSK signal with 1 Watt (i.e. 30 dBm) of power. Thus we need an amplifier with 30 dB of gain. For this we are using power amplifier RF5110 from RF Micro Devices. As per the graphs given in RF5110 datasheet [8], we can get 31 dBm of gain at 450 MHz with an efficiency of 49% and $V_{cc} = 2.8V$. For 32 dBm of gain we have to use $V_{cc} = 3.3V$ and in this case efficiency is 47%.

Control over output power can be achieved by controlling V_{APC} pin of RF5110. Control range varies from approximately 1.0 V for -10 dBm to 2.6 V for +33 dBm RF output power.

2.3 SPI Interface

SPI stands for "Serial Peripheral Interface" and it is widely used with embedded systems because it is a simple and efficient interface. MSP supports two types of SPI: 3-pin SPI and 4-pin SPI. We used 3-pin SPI for all SPI interface. Its three signal wires hold a clock (CLK), a "Master Out, Slave In" (MOSI) data line, and a "Master In, Slave Out" (MISO) data line. Apart from these three pins, one additional pin is required which is called Chip Select or Slave Select. ADF7020-1 manual calls it as SLE (Load Enable). Use of this line is clearly visible in the Serial Interface Timing Diagram provided in the datasheet of ADF7020-1, CMX7164, CMX991 and SD Card. Serial data is transmitted and received by devices using a clock provided by the Master.

SPI is a full duplex protocol i.e. for each bit shifted out the MOSI line (one per clock) another is shifted in on the MISO line. As soon a byte of data is written into UCxTXBUF, Master will start transmitting it over MOSI line and at the same time it will sample MISO line. When the transmission UCxTXBUF is over, we can extract information from UCxRXBUF since transmission and reception takes place simultaneously. Following figure is taken from MSP430 datasheet which shows SPI in action. Original Image has been modified to include a Chip Select (CS) pin for our application.

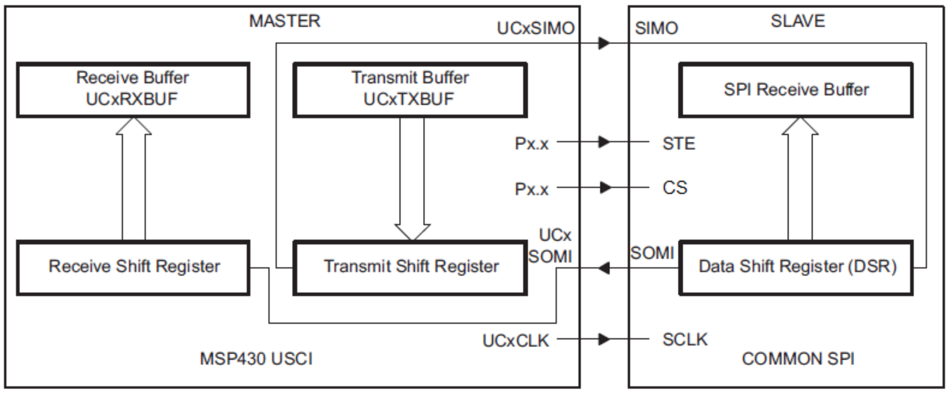


Figure 2.12: SPI Master and external slave

Another important thing about SPI is setting Clock polarity and Phase. Each SPI enabled module will have its own choice of Clock polarity and Phase which cannot be changed. Microcontroller should abide with those requirements to communicate with it. Following figure (taken from MSP430 datasheet) shows all four possible combinations of Clock polarity and Phase which can be selected using UCCKPH and UCCKPL bits in SPI Control Registers.

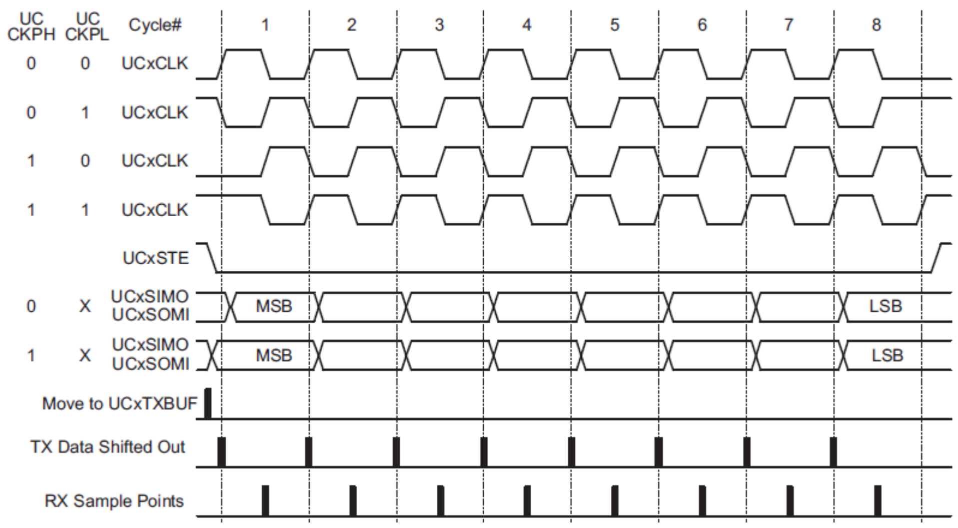


Figure 2.13: SPI timing diagram

Before using ADF7020-1, Memory Card, CMX7164 and CMX991, their internal registers needs to be programmed. All these chips will be in Slave mode and COM will be their Master. Using two different Chip Select lines, CSN_A and CSN_B for CMX7164 and CMX991 respectively, we can use USCI_B0 for both CMX7164 and CMX991. Similarly, we used two different SLE's pin, SLE_A and SLE_B, to share same SPI among ADF7020-1 and ADF4351. USCI_B1 module was reserved for Memory Card whereas bit-banging was used on Port-6 I/O pins of MSP430 to imitate them as an SPI port. ADF7020-1 and ADF4351 are programmed using the bit-banged I/O pins as shown in figure 2.9.

Apart from an SPI which can program internal registers, ADF7020-1 needs another SPI on MSP430 which can accept its Data and Clock signal. We saw earlier that ADF7020-1 (an FSK Demodulator) demodulates RF signal and gives Data and Clock signal at its output. Since, only Master can generate Clock, USCI_A0 SPI module of MSP430 was put in Slave mode to sample the Data line at positive Clock edges.

2.4 System Reset Arrangement

It is recommended to have a separate mechanism to reset the system in case emergency arises. Considering that satellite starts malfunctioning and the transmitted signal spectrum starts leaking outside our allocated band, then we should be able to stop it immediately. We should be capable of doing this by sending a Reset Command using a link which is separate from the normal communication uplink.

Looking into the advantages of ADF7020-1, we have decided to use it as a receiver for Reset Command also. We will send Reset Command using FSK modulation and at a very low data rate of 300 bps. Hence its two main lobes will occupy net 1200 Hz of bandwidth. We have selected a separate band within the allocated bandwidth of 10 kHz for reset signalling. Figure 2.14 shows the complete architecture along with the System reset arrangement and the uplink spectrum.

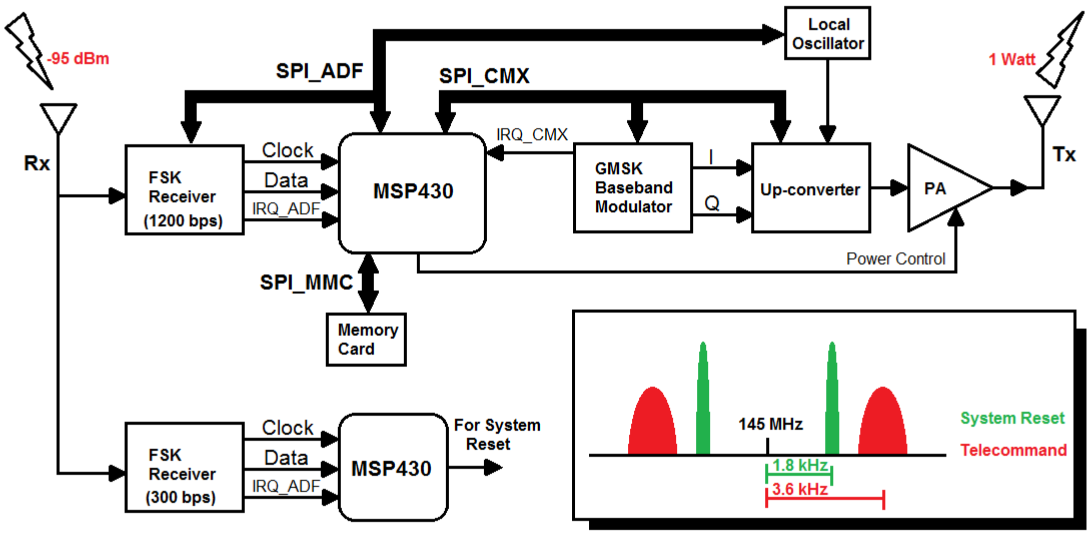


Figure 2.14: COM architecture along with System Reset arrangement

Moreover, it was observed by other HAM users that their satellite were unable to listen to the telecommands once they start transmitting. So it is good to switch off the transmitter automatically after some time. Recognizing that the maximum visibility period is 10 minutes, we have implemented a full system reset, 10 minutes after the last telecommand that was received. It was implemented using a Watchdog timer. Implementation of Watchdog timer is discussed in section 4.2.5. Performing system reset after 10 minutes of receiving last telecommand will makes sure that satellite will atleast receive telecommand in the next pass. Apart from this, Watchdog timer will keep giving reset signal at every 10 minutes when the satellite is not communicating with Ground Station.

CHAPTER 3

Communication Protocol

3.1 AX.25 Protocol

It is a data link layer protocol derived from the X.25 protocol which is an ITU-T standard protocol for packet switched wide area network (WAN) communication. It is designed for use by amateur radio operators. AX.25 is used extensively on amateur packet radio networks. It occupies the first and second layer of OSI reference model, and is responsible for transferring data (encapsulated in packets) between nodes and detecting errors introduced by the communication channel.

We are using AX.25 protocol because of its simplicity, compatibility and widespread use among satellites network over amateur band. The information to be transmitted are sent in small blocks of data, called frames. Each frame is made up of several smaller groups, called fields. Figure 3.1 shows 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

U and S frame structure

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

Information frame structure

Figure 3.1: Three types of AX.25 frame

An Information (I) frame is used to send data using flow control, an Unnumbered(U) frame can be used to send data without any flow control and Supervisory (S) frame is used to control the connection (example- to establish connection, terminate connection or to send flow control request). Further details and handshaking information can be found in AX.25 document. Calculation of FCS is described in next section. CRC is one of the popular FCS algorithm and for AX.25 encoding it is recommended to use CRC CCITT-16 polynomial (i.e. $x^{16} + x^{12} + x^5 + 1$).

While transmitting information over downlink, data stored in Memory card was divided by COM into blocks of 512 bytes over which AX.25 encoding was implemented. Finally this information was transmitted over downlink using Unnumbered(U) frame. Since U-frame do not provide any feedback facility, the frames lost can not be recovered.

3.2 CRC Implementation

In order to verify the authenticity of telecommand, a 16-bit checksum is added at the end of each AX.25 frame. This 16-bit CRC word is generated using CCITT-16 generator polynomial (i.e. $x^{16} + x^{12} + x^5 + 1$). Details regarding CRC error detecting codes can be found in [10]. Kinds of errors that can be detected by using a particular generator polynomial depends upon its structure. By going though the theorems and arguments in [10] we can evaluate the performance of CCITT-16 Code. To summarize them up we can make following points regarding CCITT-16:

- All single bit errors will be detected
- All adjacent double-bit errors will be detected
- All odd number of bit errors will be detected
- All two-bit errors which are separated by $2^{15} - 1 = 32767$ bits (i.e. 4095 bytes) will be detected
- All error bursts of 16 bits or less are detected

A burst-error of length b is defined as any pattern of errors for which the number of symbols between the first and last errors, including these errors, is b. We have used maximum packet size of 523 bytes. Hence all single and double bit errors will be definitely detected. Quoting a line from [13] :

> For randomly distributed errors, it is estimated that the likelihood of CRC-16 not detecting an error is 10^{-14}, which equates to one undetected error every two years of continuous data transmission at a rate of 1.544 Mbps.

A paper on cyclic codes, [10], describes several ways of computing CRC. At first we directly implemented the method described in Fig. 1 of this paper, using generator polynomial $x^{16}+x^{12}+x^5+1$. CRC encoding was done by using bit-wise shift operation as shown in figure 3.2. If we go through [10], it is clearly mentioned that by using this method we will have to enter 16-zeros after the entry of last data bit to get a valid Check sum into the registers. Once these 16 zeros are inserted, we can extract Checksum from Bit-0 to Bit-15 and place them at the end of last data bit. We can see from figure 3.2 that for each bit of data we have to perform a bit-wise shift operation and a modulo-2 addition. This requires large number of computation and consumes lot of clock cycles.

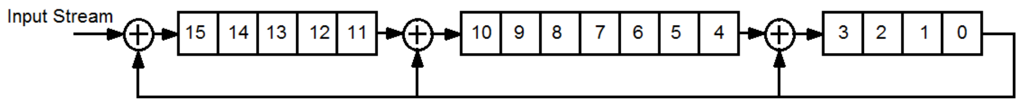
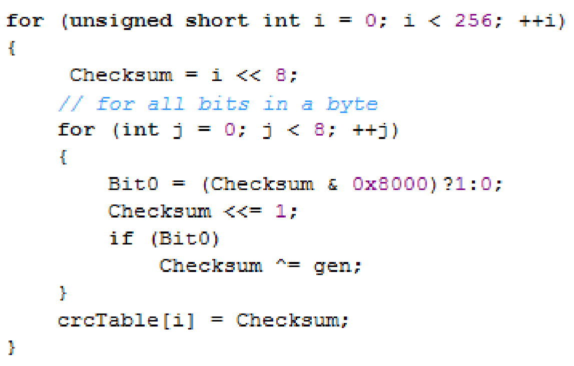


Figure 3.2: CRC calculation using shift registers

We overcome this issue by using a look-up table to compute checksum. Previously we were working one bit at a time, but now, using look-up table we can work with one byte at a time. Looking at the bottom 8 bits of the CRC register and scanning this byte from right to left (Bit-0 to Bit-7). Every time we find a non-zero bit, we XOR the key into the register at a bit offset such that this bit is turned off by the top invisible bit of the key. This operation can modify the remaining bits in the byte we scanned but XORing is like addition, it is Associative and we can do it in any order. So we can XOR together a bunch of shifted keys corresponding to a given byte and store it in a table for later use. C code for calculating this table entries is shown in figure 3.3.

Once the table is pre-calculated, we can use it to evaluate the CRC of any message byte-by-byte. Since the table is constant, we have stored it into the code memory of COM. At each step when we shift one byte out of the register and shift in the next message byte, we use the shifted out byte as an index to our CRC table. The value from the table contains the accumulated XOR of appropriately shifted keys. We XOR this value into the register and continue the process until the whole message is consumed.

As mentioned previously, we will have to enter 16-zeros after the entry of last data bit to get a valid checksum into the registers. When we are implementing byte-wise operation, we will have to insert two bytes of all zeros. In order to avoid inserting these



```
for (unsigned short int i = 0; i < 256; ++i)
{
     Checksum = i << 8;
    // for all bits in a byte
    for (int j = 0; j < 8; ++j)
    {
        Bit0 = (Checksum & 0x8000)?1:0;
        Checksum <<= 1;
        if (Bit0)
            Checksum ^= gen;
    }
    crcTable[i] = Checksum;
}
```

Figure 3.3: C code to calculate CRC Table

extra zeros, the very same paper [10] have explained another method in Fig. 2. We can merge the idea of this method with above byte wise operation and come up with a solution where checksum is present in the registers as soon the last data byte has been entered. We can keep XORing the accumulated keys into the register, but postpone the XORing of the message byte until its time comes to be shifted out of the register. Again, it is possible because of the Associativity property of addition.

CHAPTER 4

Software Architecture

4.1 Architecture Overview

Complete code can be divided into three primary blocks. First is ADF Block (for ADF7020-1), Second is CMX Block (i.e. for CMX7164 and CMX991) and last one is General Block. CMX Block can again be divided into two sub-blocks (CMX-A and CMX-B). Different blocks can communicate using some common parameters as shown in figure 4.1, i.e. ADF Block communicates with CMX-A Block using a single parameter *Command* but both the blocks can alter parameter *Ready*. CMX-A and ADF Block controls General Block using *Ready*. General Block and CMX-B Block communicate using *Empty*.

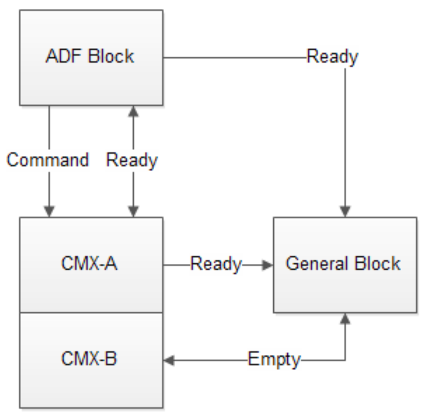


Figure 4.1: Software blocks of COM and interface parameters

ADF Block is that part of code which defines the duty of COM when an interrupt is issued by ADF7020-1, CMX Block defines the duty of COM when an interrupt is issued by CMX7164 and General Block is the set of instructions which are executed inside `main()` function. Role of various parameters are defined as follows:

- *Command* : Each telecommand has been associated with a number. Command is an integer which indicates what tele-command has been received.
- *Ready* : It indicates the present state of COM. At a time COM can be in any one of the three states, namely Idle (Ready=0), Housekeeping (Ready=1) and Data Transfer(Ready=2). State flow diagram for COM is shown in figure 4.2.

- *Empty* : It indicates the status of circular Queue which is used to store data frames to be transmitted (described in section 4.2.1).

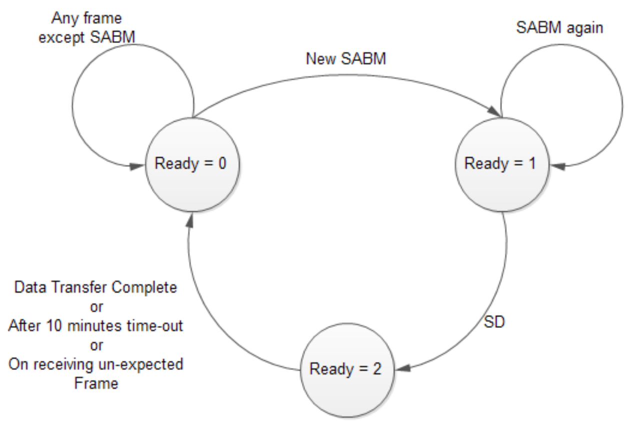

Any frame except SABM
New SABM
SABM again
Ready = 0
Ready = 1
Data Transfer Complete or After 10 minutes time-out or On receiving un-expected Frame
SD
Ready = 2

Figure 4.2: State flow diagram of COM

4.2 Block Interface

4.2.1 General Block

This block defines the set of works which COM will do after Power-on/Reset. Apart from this, it is responsible for reading blocks of data from Memory card. Flow-chart of figure 4.3 summarizes the job of this block.

First thing to do after Power-on/Reset is to initialize the pins of COM (MSP430). Then COM tries to establish connection with ADF7020-1. COM confirms the working state of ADF7020-1 by reading its Silicon Revision Readback Word (refer datasheet of ADF7020-1). If ADF7020-1 returns Silicon Revision Readback Word as 0x2018 then COM verifies its presence and moves on to initializing CMX7164, else it will reset ADF7020-1 and then try to re-initialize it.

Initializing CMX7164 takes some time. As mentioned before, CMX7164 can be used only after the Function-Image has been loaded into it. COM confirms the proper loading of Function-Image (FI) when it receives a valid Product-ID and Function-Image code from CMX7164. If somehow FI fails to load in first attempt, COM resets it and tries to load it again.

When connection with ADF7020-1 (uplink receiver) and CMX7164 (downlink transmitter) has been established, COM initializes Memory Card. After receiving expected

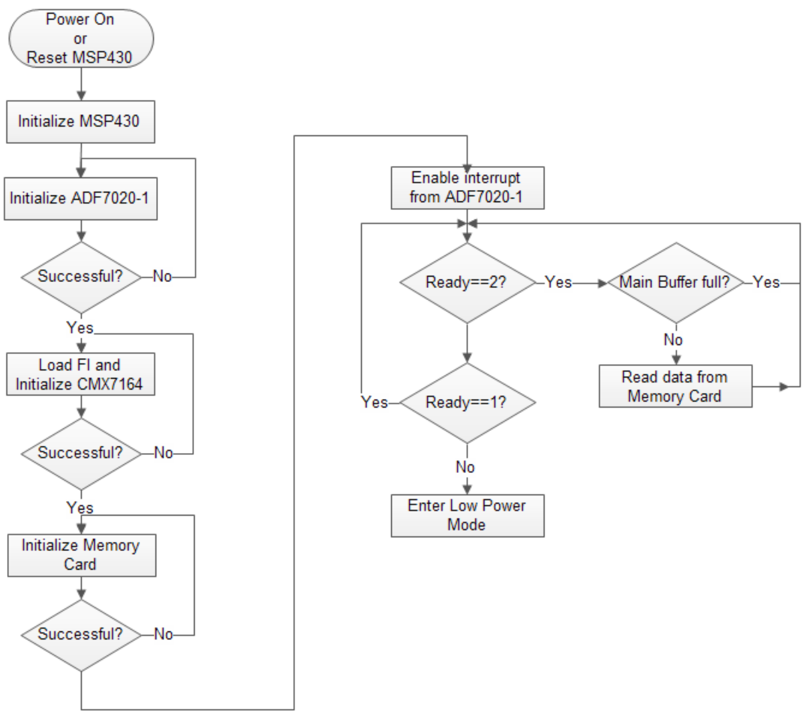


Figure 4.3: Flow chart for General Block

response from memory card, COM starts listening to the tele-commands.

Next, COM enables the interrupt associated with ADF7020-1. Since, ADF7020-1 gives an interrupt when it detects a valid Frame Sequence, COM should be ready to respond to it immediately. Initially COM will be in Idle State (Ready=0) and hence it will enter Low Power mode (LPM). This helps reducing power consumption when COM is not in use. Since CPU and Master Clock are disabled in LPM, all CPU activities are seized. COM stays in LPM and just waits for an interrupt from ADF7020-1.

When interrupted by ADF7020-1, ADF Block is executed which in turn can enable CMX7164 interrupt based on the received telecommand. Either ADF Block or CMX-A Block can alter *Ready* parameter which specifies the state of COM (i.e. Idle, House-keeping or Data Transfer). After serving the respective Interrupt Service Routine (ISR), when program execution returns to General Block, COM will decide whether to read data from Memory Card or not based on the parameters *Ready* and *Empty*.

Software maintains a circular queue whose entries are *ptr[0], *ptr[1] and *ptr[2]. These are the character pointers that stores the address of 512 bytes character array each. Properties of this Queue is as follows:

- When `Front == Rear` and `Empty==1`, Queue is empty
- When `Front == Rear` and `Empty==0`, Queue is full

- When data block is read from memory card, it is added at `Rear` and then `Rear = (Rear+1) mod 3`. Now if `Rear==Front` then `Empty=0` (i.e. Queue if Full).
- Data is transmitted from `Front` and then `Front = (Front+1) mod 3`. If `Front==Rear` then `Empty=1` (i.e. Queue is Empty).

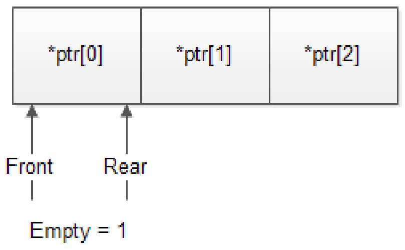


Figure 4.4: Queue for data frames

When COM reads one block of data from Memory card it is stored at the rear end of queue i.e. *ptr[Rear]. When COM transmits data to CMX7164, the data is extracted from *ptr[Front]. Since COM is interrupt driven, it can transfer data to CMX7164 and read blocks from Memory card at the same time. Thus when COM is busy transferring one of the three buffer, it can keep filling the empty buffers. Buffer is ready to receive new data from Memory card once it has been transferred to CMX7164.

4.2.2 ADF Block

ADF Block comprises the Interrupt Service Routine (ISR) associated with ADF7020-1 interrupt. As explained earlier, ADF7020-1 generates an interrupt when it detects a valid Frame Sequence. When COM receives this interrupt it follows following steps:

- Stops listening to ADF7020-1 interrupt.
- It enables SPI for receiving data from ADF7020-1 (USCI-A0) and also enables interrupt associated with UCA0RXBUF as soon it receives a byte.
- Inside ISR of UCA0RXBUF each received byte is stored in an array till we receive an End Flag.
- Once End Flag is encountered, the received character array is compared with stored commands to identify what action needs to be taken. At this point ADF7020-1 interrupt is enabled again.

Advantage of using UCA0RXBUF interrupt is that COM can perform many other works in the interval of receiving two bytes. Since uplink speed is 1200 bps, one byte will be received in 6.66 msec. COM is working at 16 MHz, hence there are approximately 106 k clock cycles in this duration. So rather than waiting for whole frame to come, UCA0RXBUF interrupt allows us to use 106 k clock cycles in more productive work.

Works done by COM after receiving a complete frame is shown in the flow-chart of figure 4.5. Parameter controlled by this block is *Command*. *Command* is just a numerical representation of the telecommand that has been received.

Few unique things have to be done when COM starts transmitting for first time (i.e. when COM is in Idle State (Ready==0) and it receives SABM telecommand). This situation is handled by ADF block itself. As shown in the flow-chart, all conditions associated with Idle State (*Ready==0*) is handled by ADF Block. Controls over other COM states are left over as the job of CMX-A Block.

ADF Block changes COM state from Idle (*Ready=0*) to Housekeeping (*Ready=1*) when it receives a new SABM frame.

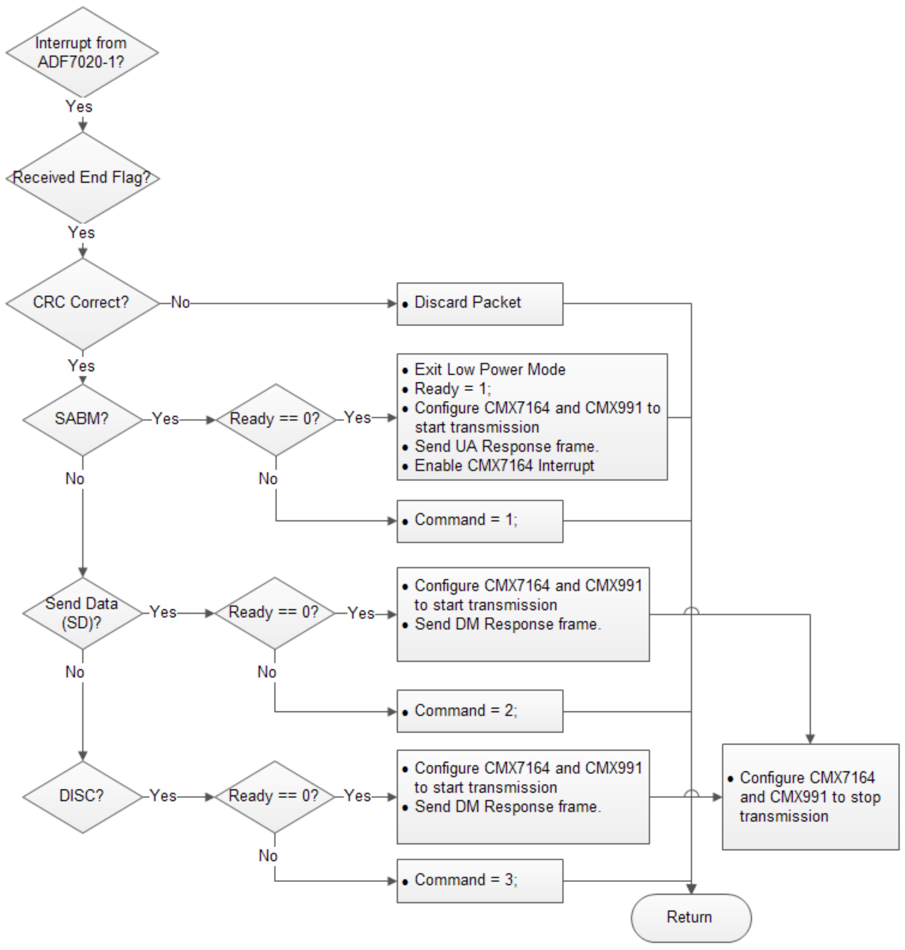


Figure 4.5: Flow chart for ADF Block

4.2.3 CMX-A Block

As explained in previous section, ADF Block enables interrupt from CMX7164 only when SABM frame is received in Idle state. Also, it was mentioned in CMX7164 Section that interrupt will be issued by CMX7164 as soon the CMD Buffer is empty. When CMX7164 is already transmitting data, this interrupt means that CMX7164 is asking for more data before it runs out of it.

Each time the interrupt is generated by CMX7164, COM executes CMX Block. CMX-A sub-block is completely bypassed if no new telecommand has been received (i.e. *Command* == *0*). If COM receives a new command then CMX-A sub-block responds according to the designed protocol. Job of CMX-B sub-block is to fill the data into CMD Buffer, but CMX-A directs it by specifying what it should fill based on the received telecommand. If CMX-A finds that Ground Station is asking for link termination or some unexpected event has occurred, then CMX-A can proceed for link termination and in this case CMX-B will be bypassed.

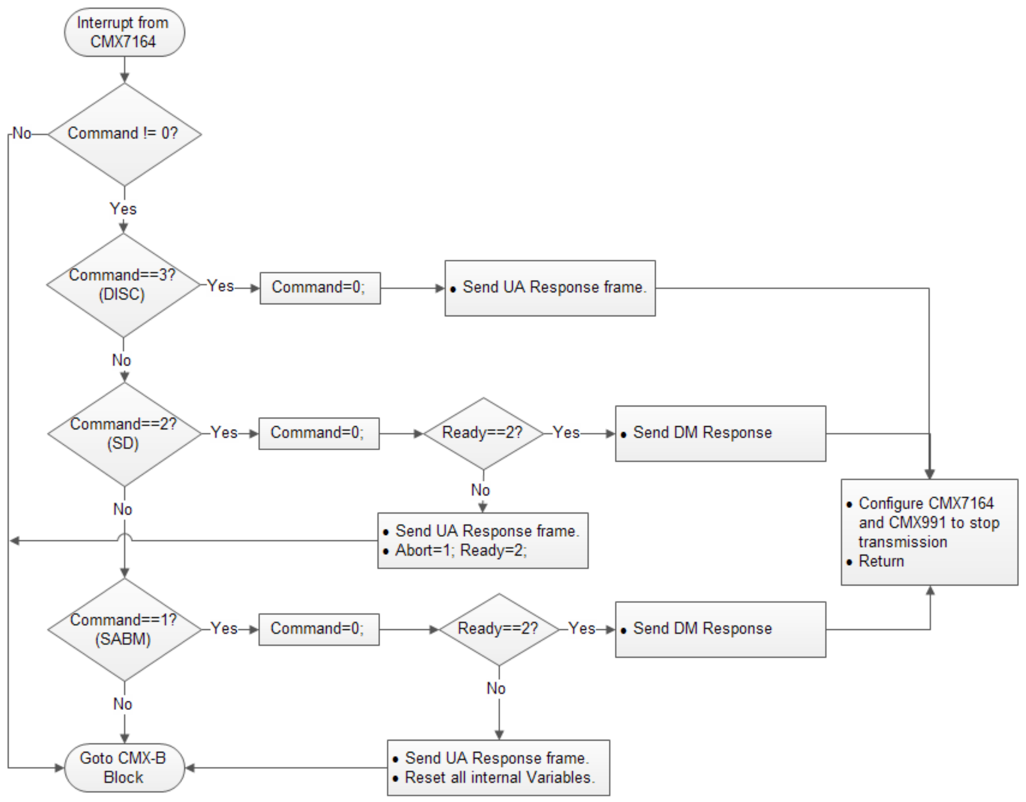


Figure 4.6: Flow chart for CMX-A sub-block

4.2.4 CMX-B Block

This block is concerned with filling the CMD buffer of CMX7164. As soon a new SABM frame has been received COM starts transmitting Preambles. Preamble is nothing but a series of 10101... which are used for clock synchronization at the Ground Station. Using streaming C-BUS transaction, maximum of 15 bytes can be transferred at a time. CMX7164 generates an interrupt when CMD Buffer is empty, i.e. when the transferred bytes have been read from CMD Buffer. Since downlink transmission speed is 19.2 kbps, it takes 6.25 msec. for CMX7164 to transmit 15 bytes and hence generate an interrupt. Following flow-chart describes the job of this block.

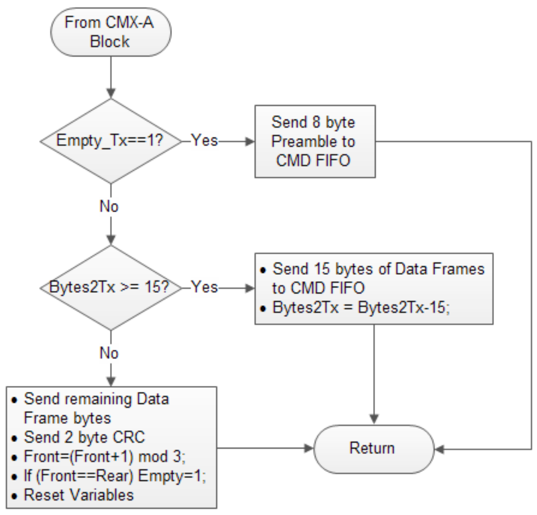

From CMX-A Block
Empty_Tx==1?
Yes
Send 8 byte Preamble to CMD FIFO
No
Bytes2Tx >= 15?
Yes
• Send 15 bytes of Data Frames to CMD FIFO
• Bytes2Tx = Bytes2Tx-15;
No
• Send remaining Data Frame bytes
• Send 2 byte CRC
• Front=(Front+1) mod 3;
• If (Front==Rear) Empty=1;
• Reset Variables
Return

Figure 4.7: Flow chart for CMX-B sub-block

4.2.5 Interrupt Management

All the software blocks that we have implemented in MSP430 are interrupt driven. Complete code uses four ISR (Interrupt Service Routines) which are described below:

- **Port-2 ISR**: Interrupt signal given by ADF7020-1 is connected to P2.0 pin of MSP430. ADF7020-1 issues an interrupt when it detects Frame Synchronization field in the received bits. On receiving an interrupt from ADF7020-1, PORT-2 ISR is executed.
- **PORT-1 ISR**: Interrupt signal from CMX7164 is connected to P1.0 of MSP430
- **USCIAB0 Rx ISR**: This ISR is executed when MSP430 received one byte of information through either USCI_A0 or USCI_B0. Though USCI_A0 stores the received byte in register UCA0RXBUF and USCI_B0 stores the received byte in UCB0RXBUF, they both share a common ISR. Data and Clock provided by

ADF7020-1 are collected by MSP430 via USCI_A0 module. Since we have not activated interrupt associated with USCI_B0, USCIAB0 Rx ISR will be executed only when USCI_A0 module receives a byte in UCA0RXBUF from ADF7020-1.

- **WDT Timer ISR**: As mentioned earlier, we want to reset the COM system after every 10 minutes. Watchdog timer keeps a count of the time elapsed and triggers system reset when the timer expires.

Interrupt priorities are fixed and it defines what interrupt is serviced when more than one interrupt is pending simultaneously. As mentioned in the datasheet of MSP430F2619, interrupt from Watchdog timer has highest priority, followed by USCI_A0 Receive, PORT-2 (ADF7020-1) and PORT-1 (CMX7164). Interrupt from ADF7020-1 was given priority over that from CMX7164 because CMX7164 has the facility of buffering the information that has to be transmitted. This allows us to service ISR associated with PORT-2 (ADF7020-1) while continuing seamless transmission from CMX7164. Figure 4.8 shows the interrupt flow that is implemented in our code. In this figure all the edges emanating from a single point represents the steps that has to be performed simultaneously.

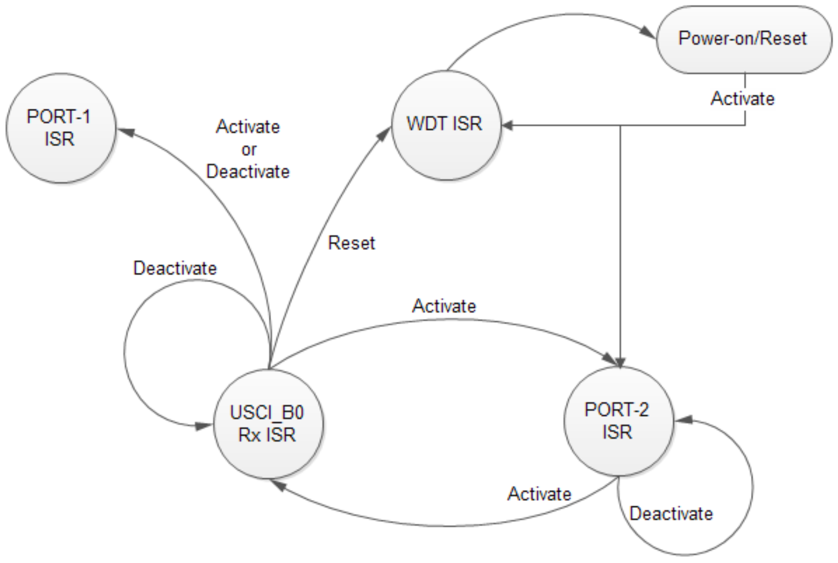


Figure 4.8: Interrupt flow

After the COM is powered-on/Reset, only PORT-2 and WDT Timer interrupt is enabled. Upon receiving a valid Frame Sequence, PORT-2 ISR disables itself and enables USCI_A0 interrupt. Here COM collects the telecommand byte by byte until an End flag is encountered. Once a complete telecommand has been received, USCI_A0 Rx ISR disables itself, enables PORT-1 ISR again and based on the received telecommand it can either activate or deactivate PORT-2 ISR. While all these things are processing, WDT

Timer is running in parallel which performs COM reset after every 10 minutes. When USCI_A0 Rx ISR detects a new SABM request, WDT Timer will reset its countdown and will start counting again to generate System reset 10 minutes after the last "new" SABM frame was received.

CHAPTER 5

Test Setup

This chapter describes the various test set-up we build at each stage of our project. Our first aim was to establish a reliable communication link between two ADF7020-1 FSK transceivers. For this, as shown in figure 5.1, one ADF7020-1 was used as a FSK transmitter which accepts AX.25 encoded frames from host MSP430 while another ADF7020-1 receives the FSK signal, demodulates it and passes it to client MSP430. Client MSP430 extracts data from AX.25 frames and passes it to PC via UART. Data received by PC were observed in Windows Hyperterminal.

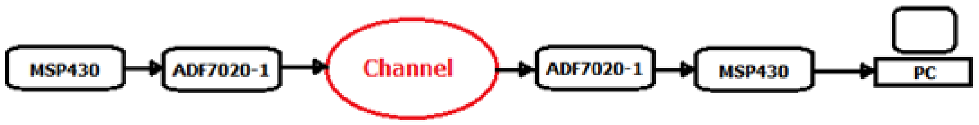


Figure 5.1: ADF7020-1 to ADF7020-1 communication

Next step was to verify the performance of uplink receiver in the presence of Doppler. For this we established the arrangement as shown in figure 5.2. As mentioned earlier, we performed Matlab simulation to obtain the Doppler shift vs. Time relationship. These values were put into the USRP code to generate a real-time Doppler corrupted FSK signal. This Doppler corrupted signal was fed as input to ADF7020-1 and the received data was observed in Hyperterminal. ADF7020-1 was able to track the signal with Doppler shift up to $\pm 50kHz$.

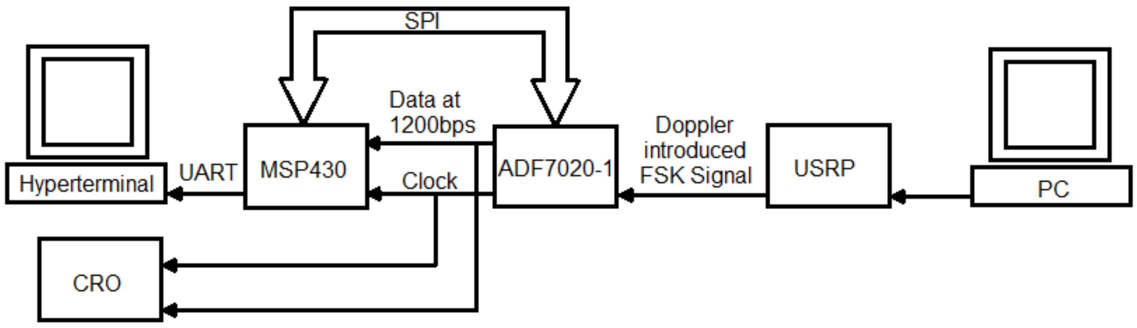


Figure 5.2: Test setup to verify Doppler Correction

Finally CMX7164, CM991 and ADF4351 were integrated with ADF7020-1 and MSP430 to create a full working prototype of on-board satellite communication system. A prototype for ground station was also constructed which can send telecommand and receives telemetry. Telecommand were issued by ground station based on the user input and satellite response was observed in PC.

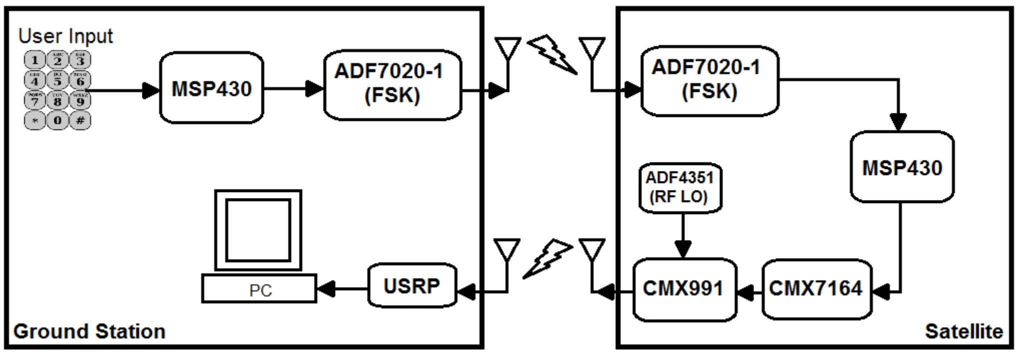


Figure 5.3: Complete prototype for Satellite and Ground Station

CHAPTER 6

Conclusion

By the end of this project we were ready with the final hardware design for on-board communication system of IITM-Sat. Single microcontroller (COM) along with six external modules were combined to form the complete self-contained COM system. Initial testing of this design was performed using the evaluation boards of MSP430F2619, ADF7020-1 (FSK Receiver), CMX7164 (GMSK Baseband Modulator), CMX991 (I/Q up-converter), ADF4351 (Local Oscillator) and RF5110 (Power Amplifier).

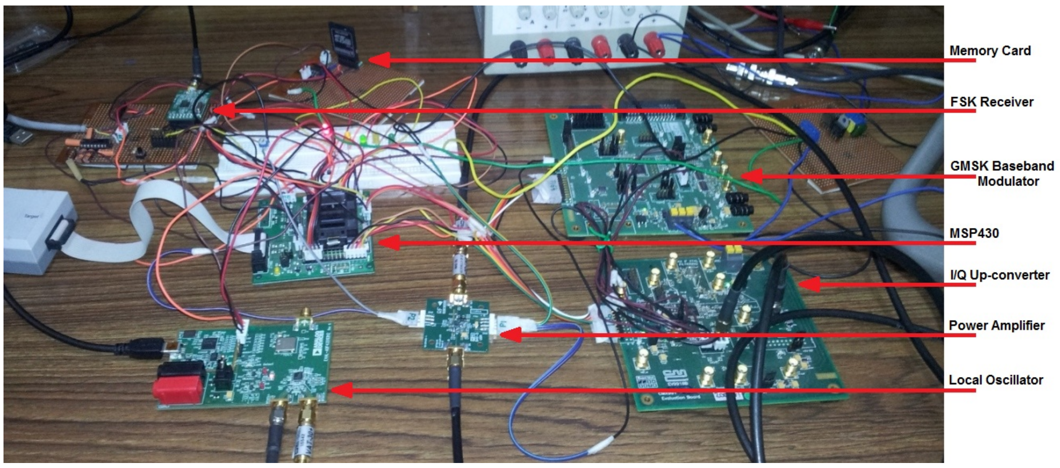


Figure 6.1: Lab setup of COM system using Evaluation boards

Since the performance is verified by experiments on evaluation boards we proceeded to design schematic for integrating all the modules into single PCB. Future testing has to be performed on this single PCB design.

As a communication protocol between ground station and satellite, modified version of AX.25 protocol was implemented. Handshaking mechanism, as defined in the AX.25 specifications, were deployed and verified. Exact protocol can be implemented once all the telecommands are decided.

APPENDIX A

APPENDIX

A.1 MSP430F2619 Connections

$MSP\ Pin\ no.$	$Description$	$Direction$
59	PDRF for ADF4351	Out
60	CE for ADF4351	Out
61	CE for ADF7020-1	Out
2	MISO for ADF7020-1	In
3	MOSI (Common for ADF7020-1 and ADF4351)	Out
4	SLE for ADF7020-1	Out
5	SCLK (Common for ADF7020-1 and ADF4351)	Out
6	SLE for ADF4351	Out
12	Interrupt from CMX7164	In
13	Reset CMX7164	Out
14	CSN for CMX991 (CSN_B)	Out
15	Reset CMX991	Out
16	CSN for CMX7164 (CSN_A)	Out
20	Interrupt from ADF7020-1	In
21	To V_{APC} of Power Amplifier	Out
28	Clock from ADF7020-1	In
29	C-BUS MOSI (Common from CMX7164 and CMX991)	Out
30	C-BUS MISO (Common from CMX7164 and CMX991)	In
31	C-BUS Clock (Common from CMX7164 and CMX991)	Out
32	MOSI from ADF7020-1	In
44	Clock for SD Card	In
45	Slave Select $\overline{SS}$ for SD Card	Out
46	MOSI for SD Card	Out
47	MISO from SD Card	In

Table A.1: Pin connection of MSP430F2619

A.2 ADF.h

```c
#define ADF7020 1
#define ADF4351 2
#define CE_ADF7020   P6OUT |= BIT2
#define CE_ADF4351   P6OUT |= BIT1
#define RFon_ADF4351   P6OUT |= BIT0
#define RFoff_ADF4351   P6OUT &= ~BIT0

unsigned char SIL[4] = {0x00, 0x00, 0x01, 0xC7};
unsigned char RSSI[4] = {0x00, 0x00, 0x01, 0x47};
unsigned char RET[4];

unsigned char Reg7020_0[4] = {0x79, 0x04, 0x43, 0xB0};
unsigned char Reg7020_1[4] = {0x00, 0x03, 0x90, 0x11};
unsigned char Reg7020_3[4] = {0x00, 0x62, 0x80, 0x93};
unsigned char Reg7020_4[4] = {0x01, 0x00, 0x00, 0x54};
unsigned char Reg7020_5[4] = {0x94, 0xFE, 0x7E, 0x35};
unsigned char Reg7020_6[4] = {0x0C, 0x48, 0x1E, 0xC6};        //lna mode =1, Optimized for Delta_f = 2.5kHz
unsigned char Reg7020_9[4] = {0x00, 0xBA, 0x31, 0xE9};
unsigned char Reg7020_11[4] = {0x00, 0x10, 0x35, 0x5B};

unsigned char Reg4351_5[4] = {0x00, 0x58, 0x00, 0x05};
unsigned char Reg4351_4[4] = {0x00, 0xAC, 0x80, 0x3C};
unsigned char Reg4351_3[4] = {0x00, 0x00, 0x04, 0xB3};
unsigned char Reg4351_2[4] = {0x00, 0x00, 0x4E, 0x42};
unsigned char Reg4351_1[4] = {0x08, 0x00, 0x80, 0x09};
unsigned char Reg4351_0[4] = {0x00, 0x54, 0x00, 0x00};



const char SABM_pkt[] = {'I','I','T','M','-','S','a','t','␣','S','A','B','M',0xCF,0x67,0x00}; // CRC is 0xCF67
const char SD_pkt[]   = {'I','I','T','M','-','S','a','t','␣','S','D','␣','␣',0xF4,0xD4,0x00}; // CRC is 0xF4D4
const char TERM_pkt[] = {'I','I','T','M','-','S','a','t','␣','T','E','R','M',0x41,0xF9,0x00}; // CRC is 0x41F9

int Offset_Rx=0;
char Rx[80];

void write(int, unsigned char *);
void read(unsigned char *);
int ADF7020_initialize(void);
void ADF4351_initialize(void);

void ADF4351_initialize(void)
{
  RFon_ADF4351;
  CE_ADF4351;
  delay(5); //Try reducing

  write(ADF4351, Reg4351_5);
  write(ADF4351, Reg4351_4);
  write(ADF4351, Reg4351_3);
  write(ADF4351, Reg4351_2);
  write(ADF4351, Reg4351_1);
  write(ADF4351, Reg4351_0);
}

int ADF7020_initialize()
{
  CE_ADF7020;
  delay(5);
  read(SIL);
  if (RET[0]==0x20 && RET[1]==0x18)
  {
    write(ADF7020, Reg7020_0);
    write(ADF7020, Reg7020_1);
    write(ADF7020, Reg7020_3);
```

```
      write(ADF7020, Reg7020_4);
      write(ADF7020, Reg7020_5);
      write(ADF7020, Reg7020_6);
      delay(50);
      write(ADF7020, Reg7020_9);
      delay(50);
      write(ADF7020, Reg7020_11);
      return 1;
   }
   return 0;
}

void write(int ADF, unsigned char Reg[4])
{
   unsigned char D;
      for(int i=0;i<=3;i++)
      {
        D = Reg[i];
        for(int j=1;j<=8;j++)
        {
           delay(5);
           P6OUT &= ~BIT6;            //CLK low
           if(D&0x80)
              P6OUT |= BIT4;
           else
              P6OUT &= ~BIT4;
           D<<=1;
           delay(5);
           P6OUT |= BIT6;             //CLK high
        }
      }
      delay(5);
      P6OUT |= BIT6;                          //CLK low


      if (ADF == 1)
      {
        P6OUT |= BIT5;                //SLE high
        delay(5);
        P6OUT &= ~BIT5;               //SLE low
        delay(5);
      }
      else
      {
        P6OUT |= BIT7;                //SLE high
        delay(5);
        P6OUT &= ~BIT7;               //SLE low
        delay(5);
      }

}

void read(unsigned char c[4])
{
      unsigned char a[4]={0,0,0,0},b,D;

      for(int i=0;i<=3;i++)
      {
        D = c[i];
        for(int j=1;j<=8;j++)
        {
           delay(5);
           P6OUT &= ~BIT6;            //CLK low
           if(D&0x80)
              P6OUT |= BIT4;
           else
              P6OUT &= ~BIT4;
           D<<=1;
```

```
            delay(5);
            P6OUT |= BIT6;              //CLK high
        }
    }
    delay(5);
    P6OUT &= ~BIT6;                 //CLK low

    P6OUT |= BIT5;                  //SLE high
    delay(5);
    P6OUT |= BIT6;                              //CLK high
    delay(5);
    P6OUT &= ~BIT6;                 //CLK low
    for(int i=0;i<=3;i++)
      for(int j=1;j<=8;j++)
    {
        delay(5);
        P6OUT |= BIT6;              //CLK high
        if(P6IN & BIT3)
          b = 0x01;
        else
          b=0x00;
        a[i] = a[i]<<1;
        a[i] = a[i] + b;

        delay(5);
        P6OUT &= ~BIT6;             //CLK low
    }
    P6OUT &= ~BIT5;                 //SLE low
    RET[0] = a[0];
    RET[1] = a[1];
    RET[2] = a[2];
    RET[3] = a[3];
}
```

A.3 CML.h

```
#define CMX7164 1
#define CMX991 2

#define CON_7164    P1OUT &= ~BIT4
#define DIS_7164    P1OUT |= BIT4
#define CON_991     P1OUT &= ~BIT2
#define DIS_991     P1OUT |= BIT2
#define Rx_FIFO_Level 0x4F
#define Rx_FIFO_Word 0x4D
#define Delay1 1

#define Enable_CMX7164       P1OUT |= BIT1
#define Disable_CMX7164      P1OUT &= ~BIT1
#define Enable_CMX991        P1OUT |= BIT3
#define Disable_CMX991       P1OUT &= ~BIT3


#define Enable_Tx_CMX7164        write16(CMX7164, 0x6B,0x0042)   // Preamble, FS1, Raw Data
#define Go_Idle_CMX7164          write16(CMX7164, 0x6B,0x0000)

void CMX7164_initialize(void);
void CMX991_initialize(void);
int FI_Load(void);
char read8(int, char);
unsigned int read16(int, char);
void write8(int, char, char);
void write16(int, char, unsigned int);
void delay(int);




void CMX7164_initialize()
{
// Program Block 5
  read16(CMX7164, 0x7E);
  write16(CMX7164, 0x6B, 0x0350);        // PB5.3, IDLE, PLL Off
  write16(CMX7164, 0x6A, 0xA000);
        // GPIOA set as an automatic output, it will go high as soon as a Tx command is received
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 0x8028);
        // GPIOB set as an automatic output, it will go low $28 x 1/20 symbol periods after a Tx command is received
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 0x0000);
        // GPIOC is not automatically controlled Ű it is manually controlled
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 0x0000);
        // GPIOD is not automatically controlled Ű it is manually controlled
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 0x8050);
        // AUXDAC1 Ű the RAMDAC will automatically start ramp up $50 x 1/20 symbol periods after a Tx command is received
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 0x8040);
        // Modulation will start $40 x 1/20 symbol periods after a Tx command is received
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 0x8000);
        // RAMDAC will automatically start ramp down as soon as modulation ends
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 0x8000);
        // GPIOA will go low as soon as modulation ends
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 0xA080);
        // GPIOB will go high $80 x 1/20 symbol periods after modulation ends
  while(!(0x4000 & read16(CMX7164, 0x7E)));
```

```
// Program Block 1
  write16(CMX7164, 0x6B, 0x0110);
  write16(CMX7164, 0x6A, 2);
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 0x2019);
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 128);
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 1);
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 16);
  while(!(0x4000 & read16(CMX7164, 0x7E)));
  write16(CMX7164, 0x6A, 0x2799);
  while(!(0x4000 & read16(CMX7164, 0x7E)));

// Program Block 4
  write16(CMX7164, 0x6B, 0x0140);
  write16(CMX7164, 0x6A, 0x0100);                // BT = 0.3
  while(!(0x4000 & read16(CMX7164, 0x7E)));


}

int FI_Load()
{
  char ch1;
  unsigned int val1, val2;
  int count;

  Enable_CMX7164;
  delay(30000);

  CON_7164;
  delay(Delay1);
  UCB0TXBUF = 0x01;
  while (UCB0STAT & UCBUSY);
  DIS_7164;
  delay(10000);


  //Read RxFIFO Level until three device check appears
  for(int i=1;i<=3;i++)
  {
    read16(CMX7164, Rx_FIFO_Word);
  }

  // Block
  write16(CMX7164, 0x49,DB1_LEN);
  write16(CMX7164, 0x49,DB1_PTR);
  count = 0;
  while(count < DB1_LEN)
  {
    ch1 = read8(CMX7164, 0x4B);
    if (ch1>=0x80) return 0;
    CON_7164;
    delay(Delay1);
    UCB0TXBUF = 0x49;
    while (UCB0STAT & UCBUSY);
    for (int i=0;i<128-(int)ch1;i++)
    {
      UCB0TXBUF = db1[count]>>8;
      while (UCB0STAT & UCBUSY);
      UCB0TXBUF = db1[count++];
      while (UCB0STAT & UCBUSY);
      if (count==DB1_LEN) break;
    }
    DIS_7164;
    delay(Delay1);
  }

  val1 = read16(CMX7164, 0x4D);  // Should be 0x0024
  val2 = read16(CMX7164, 0x4D);  // Should be 0xC156

  // Block 2
  write16(CMX7164, 0x49,DB2_LEN);
  write16(CMX7164, 0x49,DB2_PTR);
  count = 0;
  while(count < DB2_LEN)
  {
    ch1 = read8(CMX7164, 0x4B);
    if (ch1>=0x80) return 0;
    CON_7164;
    delay(Delay1);
    UCB0TXBUF = 0x49;
    while (UCB0STAT & UCBUSY);
    for (int i=0;i<128-(int)ch1;i++)
    {
      UCB0TXBUF = db2[count]>>8;
      while (UCB0STAT & UCBUSY);
      UCB0TXBUF = db2[count++];
      while (UCB0STAT & UCBUSY);
      if (count==DB2_LEN) break;
    }
    DIS_7164;
    delay(Delay1);
  }
  val1 = read16(CMX7164, 0x4D);  // Should be 0xFFF2
  val2 = read16(CMX7164, 0x4D);  // Should be 0xF1F0

  // Activation Block
  write16(CMX7164, 0x49,ACTIVATE_LEN);
  write16(CMX7164, 0x49,ACTIVATE_PTR);

  count=0;
  while(!(read16(CMX7164, 0x7E) & 0x4000))
  {
    if(count++ >= 1000) return 0;
  }

  val1 = read16(CMX7164, 0x4D);  // Should be 0x7164
  val2 = read16(CMX7164, 0x4D);  // Should be 0x1002

  // To indicate that FI has been loaded correctly
  if (val1==0x7164 && val2==0x1002) return 1;
  else return 0;
}

void write8(int reg, char add, char val)
{
  if (reg == CMX7164) CON_7164;
  else if (reg == CMX991) CON_991;
  delay(Delay1);
  UCB0TXBUF = add;
  while (UCB0STAT & UCBUSY);
  UCB0TXBUF = val;
  while (UCB0STAT & UCBUSY);
  if (reg == CMX7164) DIS_7164;
  else if (reg == CMX991) DIS_991;
  delay(Delay1);
}

void write16(int reg, char add, unsigned int val)
{
  if (reg == CMX7164) CON_7164;
  else if (reg == CMX991) CON_991;
```

```
    delay(Delay1);
    UCB0TXBUF = add;
    while (UCB0STAT & UCBUSY);
    UCB0TXBUF = val>>8;
    while (UCB0STAT & UCBUSY);
    UCB0TXBUF = val;
    while (UCB0STAT & UCBUSY);
    if (reg == CMX7164) DIS_7164;
    else if (reg == CMX991) DIS_991;
    delay(Delay1);
}

char read8(int reg, char add)
{
    char ch;
    if (reg == CMX7164) CON_7164;
    else if (reg == CMX991) CON_991;
    delay(Delay1);
    UCB0TXBUF = add;
    while (UCB0STAT & UCBUSY);
    UCB0TXBUF = 0xAA;
    while (UCB0STAT & UCBUSY);
    while (!(IFG2 & UCB0RXIFG));
    ch = UCB0RXBUF;
    if (reg == CMX7164) DIS_7164;
    else if (reg == CMX991) DIS_991;
    delay(Delay1);
    return ch;
}

unsigned int read16(int reg, char add)
{
    unsigned int val=0;
    if (reg == CMX7164) CON_7164;
    else if (reg == CMX991) CON_991;
    delay(Delay1);
    UCB0TXBUF = add;
    while (UCB0STAT & UCBUSY);
    UCB0TXBUF = 0xAA;
    while (UCB0STAT & UCBUSY);
    while (!(IFG2 & UCB0RXIFG));
    val = UCB0RXBUF;
    val = val<<8;
    UCB0TXBUF = 0xAA;
    while (UCB0STAT & UCBUSY);
    while (!(IFG2 & UCB0RXIFG));
    val = val + UCB0RXBUF;
    if (reg == CMX7164) DIS_7164;
    else if (reg == CMX991) DIS_991;
    delay(Delay1);
    return val;
}

void CMX991_initialize()
{
    Disable_CMX991;
    delay(1000);
    Enable_CMX991;
    delay(1000);

// Needs external RF LO for output at 435MHz
    write8(CMX991, 0x11, 0xC3);
    write8(CMX991, 0x14, 0x52);
    write8(CMX991, 0x15, 0x21);
    write8(CMX991, 0x16, 0x00);
    write8(CMX991, 0x23, 0x07);
    write8(CMX991, 0x22, 0x08);
    write8(CMX991, 0x20, 0xC0);
```

```
  write8(CMX991, 0x21, 0xA0);
}

void delay(int a)
{
  for(int i=1;i<=a;i++);
}
```

A.4 SD_Card.h

```
#define SD_CS_ASSERT       P5OUT &= ~BIT0
#define SD_CS_DEASSERT     P5OUT |= BIT0

#define GO_IDLE_STATE              0
#define SEND_OP_COND               1
#define SEND_CSD                   9
#define SET_BLOCK_LEN              16
#define READ_SINGLE_BLOCK          17
#define WRITE_SINGLE_BLOCK         24
#define CRC_ON_OFF                 59

char buffer[513];

unsigned int SD_init(void);
unsigned char SD_sendCommand(unsigned char cmd, unsigned long int arg, char crc);
unsigned char SD_readSingleBlock(unsigned long int startBlock);
unsigned char SD_writeSingleBlock(unsigned long startBlock);

unsigned char SD_writeSingleBlock(unsigned long int startBlock)
{
  unsigned char response;
  unsigned int i, retry=0;

  response = SD_sendCommand(WRITE_SINGLE_BLOCK, startBlock<<9, 0); //write a Block command
  if(response != 0x00) //check for SD status: 0x00 - OK (No flags set)
  return response;

  SD_CS_ASSERT;

    UCB1TXBUF=(0xFE);    //Send start block token 0xfe (0x11111110)
    while (UCB1STAT & UCBUSY);

  for(i=0; i<512; i++)     //send 128 bytes data
  {
    UCB1TXBUF=('a'+(i%26));
    while (UCB1STAT & UCBUSY);
  }

    UCB1TXBUF=(0xFF);              //transmit dummy CRC (16-bit), CRC is ignored here
    while (UCB1STAT & UCBUSY);
    UCB1TXBUF=(0xFF);
    while (UCB1STAT & UCBUSY);

    UCB1TXBUF=(0xFF);
    while (UCB1STAT & UCBUSY);
    response = UCB1RXBUF;

  if( (response & 0x1f) != 0x05) //response= 0xXXX0AAA1 ; AAA='010' - data accepted
  {                              //AAA='101'-data rejected due to CRC error
    SD_CS_DEASSERT;              //AAA='110'-data rejected due to write error
    return response;
  }

    UCB1TXBUF=(0xFF);
    while (UCB1STAT & UCBUSY);
    response = UCB1RXBUF;
  while(!response) //wait for SD card to complete writing and get idle
  {
    UCB1TXBUF=(0xFF);
    while (UCB1STAT & UCBUSY);
    response = UCB1RXBUF;
    if(retry++ > 0xfffe){SD_CS_DEASSERT; return 1;}
  }

  SD_CS_DEASSERT;
```

```
    UCB1TXBUF=(0xFF);              //just spend 8 clock cycle delay before reasserting the CS line
    while (UCB1STAT & UCBUSY);

  SD_CS_ASSERT;           //re-asserting the CS line to verify if card is still busy

    UCB1TXBUF=(0xFF);
    while (UCB1STAT & UCBUSY);
    response = UCB1RXBUF;
  while(!response) //wait for SD card to complete writing and get idle
  {
    UCB1TXBUF=(0xFF);
    while (UCB1STAT & UCBUSY);
    response = UCB1RXBUF;
    if(retry++ > 0xfffe){SD_CS_DEASSERT; return 1;}
  }

  return 0;
}

unsigned char SD_readSingleBlock(unsigned long int startBlock)
{
  unsigned char response;
  unsigned int i, retry=0;

  response = SD_sendCommand(READ_SINGLE_BLOCK, startBlock<<9,0); //read a Block command
  //block address converted to starting address of 512 byte Block
  if(response != 0x00) //check for SD status: 0x00 - OK (No flags set)
    return response;

SD_CS_ASSERT;

    UCB1TXBUF=(0xFF);
    while (UCB1STAT & UCBUSY);
    response = UCB1RXBUF;
  while(response != 0xfe) //wait for start block token 0xfe (0x11111110)
  {
    UCB1TXBUF=(0xFF);
    while (UCB1STAT & UCBUSY);
    response = UCB1RXBUF;
    if(retry++ > 0xfffe){SD_CS_DEASSERT; return 1;} //return if time-out
  }

  for(i=0; i<512; i++) //read 128 bytes
  {
      UCB1TXBUF=(0xFF);
      while (UCB1STAT & UCBUSY);
      buffer[i] = UCB1RXBUF;
  }
      buffer[512] = 0x00;
      UCB1TXBUF=(0xFF);              //receive incoming CRC (16-bit), CRC is ignored here
      while (UCB1STAT & UCBUSY);
      UCB1TXBUF=(0xFF);
      while (UCB1STAT & UCBUSY);


      UCB1TXBUF=(0xFF);              //extra 8 clock pulses
      while (UCB1STAT & UCBUSY);
SD_CS_DEASSERT;

return 0;
}

unsigned int SD_init(void)
{
unsigned char i, response, retry=0 ;

SD_CS_ASSERT;
do
```

```
{
    for(i=0;i<10;i++)
    {
     UCB1TXBUF=(0xFF);
     while (UCB1STAT & UCBUSY);
    }
    response = SD_sendCommand(GO_IDLE_STATE, 0, 0x95);//send 'reset & go idle' command
    retry++;
    if(retry>0xfe) return 0;       //time out
} while(response != 0x01);         // Wait till SD Card enters Idle State

SD_CS_DEASSERT;

     UCB1TXBUF=(0xFF);
     while (UCB1STAT & UCBUSY);
     UCB1TXBUF=(0xFF);
     while (UCB1STAT & UCBUSY);

retry = 0;

do
{
     response = SD_sendCommand(SEND_OP_COND, 0, 0); //activate card's initialization process
     response = SD_sendCommand(SEND_OP_COND, 0, 0); //resend command (for compatibility with some cards)
     retry++;
     if(retry>0xfe) return 0; //time out
}while(response);

SD_sendCommand(CRC_ON_OFF, 0, 0); //disable CRC; deafault - CRC disabled in SPI mode
SD_sendCommand(SET_BLOCK_LEN, 512, 0); //set block size to 128

return 1; //normal return
}

unsigned char SD_sendCommand(unsigned char cmd, unsigned long int arg, char crc)
{
  unsigned char response, retry=0;

  SD_CS_ASSERT;
     UCB1TXBUF=(cmd | 0x40);
     while (UCB1STAT & UCBUSY);
     UCB1TXBUF=(arg>>24);
     while (UCB1STAT & UCBUSY);
     UCB1TXBUF=(arg>>16);
     while (UCB1STAT & UCBUSY);
     UCB1TXBUF=(arg>>8);
     while (UCB1STAT & UCBUSY);
     UCB1TXBUF=(arg);
     while (UCB1STAT & UCBUSY);
     UCB1TXBUF=(crc);
     while (UCB1STAT & UCBUSY);

     response = UCB1RXBUF;
     while(response == 0xff)
     {
       UCB1TXBUF=(0xff);
       while (UCB1STAT & UCBUSY);
       response = UCB1RXBUF;
       if (retry++ > 0xfe) break;
     }

     UCB1TXBUF=(0xff);
     while (UCB1STAT & UCBUSY);
  SD_CS_DEASSERT;

  return response;
}
```

A.5 Main Code

```
#include <msp430f2619.h>
#include <stdlib.h>
#include <string.h>
//#include "7164-1.0.0.4.h"
#include "7164-1.0.0.2.h"
#include "CML.h"
#include "ADF.h"
#include "SD_Card.h"


#define SABM     1
#define SD       2
#define DISC     3
#define Frame0   4
#define Frame1   5
#define Frame2   6
#define Frame3   7
#define Frame4   8
#define Frame5   9
#define Frame6   10
#define Frame7   11

#define Length 512       // 128
#define Frame_Len 523    //139           // Frame_Len is the constant length of each data frame

const unsigned int crcTable[0x100] =
{
 0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50A5, 0x60C6, 0x70E7,
 0x8108, 0x9129, 0xA14A, 0xB16B, 0xC18C, 0xD1AD, 0xE1CE, 0xF1EF,
 0x1231, 0x0210, 0x3273, 0x2252, 0x52B5, 0x4294, 0x72F7, 0x62D6,
 0x9339, 0x8318, 0xB37B, 0xA35A, 0xD3BD, 0xC39C, 0xF3FF, 0xE3DE,
 0x2462, 0x3443, 0x0420, 0x1401, 0x64E6, 0x74C7, 0x44A4, 0x5485,
 0xA56A, 0xB54B, 0x8528, 0x9509, 0xE5EE, 0xF5CF, 0xC5AC, 0xD58D,
 0x3653, 0x2672, 0x1611, 0x0630, 0x76D7, 0x66F6, 0x5695, 0x46B4,
 0xB75B, 0xA77A, 0x9719, 0x8738, 0xF7DF, 0xE7FE, 0xD79D, 0xC7BC,
 0x48C4, 0x58E5, 0x6886, 0x78A7, 0x0840, 0x1861, 0x2802, 0x3823,
 0xC9CC, 0xD9ED, 0xE98E, 0xF9AF, 0x8948, 0x9969, 0xA90A, 0xB92B,
 0x5AF5, 0x4AD4, 0x7AB7, 0x6A96, 0x1A71, 0x0A50, 0x3A33, 0x2A12,
 0xDBFD, 0xCBDC, 0xFBBF, 0xEB9E, 0x9B79, 0x8B58, 0xBB3B, 0xAB1A,
 0x6CA6, 0x7C87, 0x4CE4, 0x5CC5, 0x2C22, 0x3C03, 0x0C60, 0x1C41,
 0xEDAE, 0xFD8F, 0xCDEC, 0xDDCD, 0xAD2A, 0xBD0B, 0x8D68, 0x9D49,
 0x7E97, 0x6EB6, 0x5ED5, 0x4EF4, 0x3E13, 0x2E32, 0x1E51, 0x0E70,
 0xFF9F, 0xEFBE, 0xDFDD, 0xCFFC, 0xBF1B, 0xAF3A, 0x9F59, 0x8F78,
 0x9188, 0x81A9, 0xB1CA, 0xA1EB, 0xD10C, 0xC12D, 0xF14E, 0xE16F,
 0x1080, 0x00A1, 0x30C2, 0x20E3, 0x5004, 0x4025, 0x7046, 0x6067,
 0x83B9, 0x9398, 0xA3FB, 0xB3DA, 0xC33D, 0xD31C, 0xE37F, 0xF35E,
 0x02B1, 0x1290, 0x22F3, 0x32D2, 0x4235, 0x5214, 0x6277, 0x7256,
 0xB5EA, 0xA5CB, 0x95A8, 0x8589, 0xF56E, 0xE54F, 0xD52C, 0xC50D,
 0x34E2, 0x24C3, 0x14A0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
 0xA7DB, 0xB7FA, 0x8799, 0x97B8, 0xE75F, 0xF77E, 0xC71D, 0xD73C,
 0x26D3, 0x36F2, 0x0691, 0x16B0, 0x6657, 0x7676, 0x4615, 0x5634,
 0xD94C, 0xC96D, 0xF90E, 0xE92F, 0x99C8, 0x89E9, 0xB98A, 0xA9AB,
 0x5844, 0x4865, 0x7806, 0x6827, 0x18C0, 0x08E1, 0x3882, 0x28A3,
 0xCB7D, 0xDB5C, 0xEB3F, 0xFB1E, 0x8BF9, 0x9BD8, 0xABBB, 0xBB9A,
 0x4A75, 0x5A54, 0x6A37, 0x7A16, 0x0AF1, 0x1AD0, 0x2AB3, 0x3A92,
 0xFD2E, 0xED0F, 0xDD6C, 0xCD4D, 0xBDAA, 0xAD8B, 0x9DE8, 0x8DC9,
 0x7C26, 0x6C07, 0x5C64, 0x4C45, 0x3CA2, 0x2C83, 0x1CE0, 0x0CC1,
 0xEF1F, 0xFF3E, 0xCF5D, 0xDF7C, 0xAF9B, 0xBFBA, 0x8FD9, 0x9FF8,
 0x6E17, 0x7E36, 0x4E55, 0x5E74, 0x2E93, 0x3EB2, 0x0ED1, 0x1EF0
};

//const unsigned char flag[4] = {0xFF, 0xFF, 0xFF, 0xFF};
const unsigned char flag[11] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
const unsigned char spare[6] = {0x55, 0x55, 0x55, 0x55, 0x55, 0x55};
const unsigned char UA_pkt[] = {0x7E, 'I','I','T','M','-','S','a','t','␣','U','A','␣','␣',0x0D,0x00};
const unsigned char DM_pkt[] = {0x7E, 'I','I','T','M','-','S','a','t','␣','D','i','s','c',0x0D,0x00};
const unsigned char Null[] = {0x00};
const int Res_Len = 16;

const int gen = 0x1021;
char *ptr[3];
char Pkt0[Length+15],Pkt1[Length+15],Pkt2[Length+15];
int Front=0, Rear=0, Empty=1, Abort=0, Ready = 0,NS = 0;
long int WDTcount=0;


int Command=0;
void MSP_initialize(void);
void First_Tx(void);
void Send_Frame(const unsigned char*);
void End_Tx(void);
void receive(void);

 void main()
{
  BCSCTL1 = CALBC1_16MHZ;
  DCOCTL = CALDCO_16MHZ;
 __enable_interrupt();
  IE1 |= WDTIE;
  WDTCTL = WDTPW + WDTTMSEL + WDTCNTCL; // Stop watchdog timer

  ptr[0] = Pkt0;
  ptr[1] = Pkt1;
  ptr[2] = Pkt2;

  MSP_initialize();
  Disable_CMX991;
  while(!ADF7020_initialize());
    P4OUT |= BIT3;
  while(!FI_Load())
  {
    Disable_CMX7164;
    delay(1000);
  }
  CMX7164_initialize();
    P4OUT |= BIT4;
  while(!SD_init());
    P4OUT |= BIT2;

    P2IE = 0x01; // Interrupt is called when SYNC is detected by ADF7020
  while(1)
  {
    if (Ready==2)
    {
      if ((Rear==Front) & (!Empty))       // If Full
        delay(5);
      else
        receive();
    }
    else if (Ready==1)
    {
      delay(5);
    }
    else if (Ready==0)
    {
     LPM1;
    }
  }
}

#pragma vector = WDT_VECTOR
__interrupt void WDT_ISR (void)
{
  // Housekeeping can be done here to check if everything is fine. If some module is not working, Reset
  WDTcount++;
  if (WDTcount>=292969)
    WDTCTL = 0;                     //Generate PUC Reset
  else
    WDTCTL = WDTPW + WDTTMSEL + WDTCNTCL;
}

void receive()
{
  SD_readSingleBlock(0);            // <<== Changes
  sprintf(ptr[Rear],"%s%c%s",flag,'0'+(NS++)%3,buffer); //Cannot accept 0x00 inside string
  Rear = (Rear+1)%3;
  if (Front==Rear)  Empty=0;
}

#pragma vector = PORT2_VECTOR
__interrupt void PORT2_ISR (void)
{
  P2IE = 0x00;
  P2IFG = 0x00;                                   // Flag must be cleared manually
  P4OUT &= 0xF0;
  Offset_Rx = 0;
  UCA0CTL1 &= ~UCSWRST;
  IE2 |= UCA0RXIE;
}


#pragma vector = USCIAB0RX_VECTOR
__interrupt void USCIAB0_RX_ISR (void)
{
  static unsigned int accumulator = 0xFFFF;
  unsigned char c;
  c = UCA0RXBUF;
  if (c==0x0D || Offset_Rx >40)             // Maximum Packet Size = 40
  {
    if (accumulator != 0)
    {
      P4OUT = 0xFF;
      while(1)
      {
      delay(30000);
      delay(30000);
      delay(30000);
      delay(30000);
      delay(30000);
      P4OUT ^= 0xFF;
      }
    }
    accumulator = 0xFFFF;
    Rx[Offset_Rx] = NULL;
    if (!strcmp(Rx,SABM_pkt))                     //SABM
    {
      P4OUT |= BIT3+BIT0;
      if (Ready==0)
      {
        LPM1_EXIT;
        WDTcount = 0;
        Ready = 1;
        First_Tx();
        Send_Frame(UA_pkt);
        P1IE = 0x01;                        // Enable Port1 Interrupt
        write16(CMX7164, 0x6C, 0x8100);     // Write in IRQ_Mask to activate Cmd_done
      }
      else
      {
        Command = SABM;
      }
    }
    else if (!strcmp(Rx,SD_pkt))          //SD (Send Data)
    {
      P4OUT |= BIT3+BIT1;
      if (Ready==0)
      {
        First_Tx();
        Send_Frame(DM_pkt);
        End_Tx();
      }
      else
      {
        Command = SD;
      }
    }
    else if (!strcmp(Rx,TERM_pkt))        //DISC
    {
      P4OUT |= BIT3+BIT1+BIT0;
      if (Ready==0)
      {
        First_Tx();
        Send_Frame(DM_pkt);
        End_Tx();
      }
      else
      {
        Command = DISC;
      }
    }
    UCA0CTL1 = UCSWRST;
    Offset_Rx = 0;
    P2IE = 0x01;
  }
  else
  {
    Rx[Offset_Rx++] = c;
    while (!(UC1IFG & UCA1TXIFG));
    UCA1TXBUF = c;
    accumulator = ((accumulator & 0x00FF) << 8) ^ crcTable[((accumulator >> 8) ^ c ) & 0x00FF];
  }
}

void End_Tx()
{
    P1IE = 0x00;
    write16(CMX7164, 0x49, 0xF000);      // Indicate Burst end is intended
      Ready = 0;
    while(!(read16(CMX7164, 0x7E)&0x0200));      // Wait until the burst ends. Tail length can be programmed in PB3
    Disable_CMX991;
    RFoff_ADF4351;
    Go_Idle_CMX7164;
}

void Send_Frame(const unsigned char *ch)
{
  int quo, count=0;
  char rem;
  quo = Res_Len/15;
  rem = Res_Len%15;
  for (;quo>0;quo--)
  {
    write8(CMX7164, 0x4A,0x1F);    // 15 bytes in each data block, after which CMX7164 will request more data from host
    CON_7164;
    delay(Delay1);
      UCB0TXBUF = 0x48;
      while (UCB0STAT & UCBUSY);
      for (int i=1;i<=15;i++)
      {
```

```
        UCB0TXBUF = ch[count++];
        while (UCB0STAT & UCBUSY);
      }
    DIS_7164;
    delay(Delay1);
  }
  if (rem != 0)
  {
    write8(CMX7164, 0x4A,0x10 + rem); // 15 bytes in each data block, after which CMX7164 will request more data from host
    CON_7164;
    delay(Delay1);
      UCB0TXBUF = 0x48;
      while (UCB0STAT & UCBUSY);
      for (;rem>0;rem--)
      {
        UCB0TXBUF = ch[count++];
        while (UCB0STAT & UCBUSY);
      }
    DIS_7164;
    delay(Delay1);
  }
}

void First_Tx(void)
{
  CMX991_initialize();
  ADF4351_initialize();

  read16(CMX7164, 0x7E);                        // Clear Inerrupt Status Register
  write16(CMX7164, 0x50,0x0080);                // Flush Command FIFO

  write8(CMX7164, 0x4A,0x1F);                   // 15 bytes in each data block, after which CMX7164 will request more data from host
  CON_7164;
  delay(Delay1);
    UCB0TXBUF = 0x48;
    while (UCB0STAT & UCBUSY);
    for (int i=1;i<=15;i++)
    {
      UCB0TXBUF = 0xAA;
      while (UCB0STAT & UCBUSY);
    }
  DIS_7164;
  delay(Delay1);

  Enable_Tx_CMX7164;                            // Preamble, FS1, Raw Data
}

#pragma vector = PORT1_VECTOR
__interrupt void PORT1_ISR (void)
{
  P1IFG = 0x00;
  if (read16(CMX7164, 0x7E)&0x0800)        P4OUT &= ~BIT5;

  if (Command)
  {
    if (Command == DISC)//(B3 && !B2 && B1 && B0)                //DISC
    {
      Command = 0;
      Send_Frame(UA_pkt);
      End_Tx();
      return;
    }
    else if (Command == SD)//(B3 && !B2 && B1 && !B0)            //SD
    {
      Command = 0;
      if (Ready==2)
      {
        Send_Frame(DM_pkt);
        End_Tx();
        return;
      }
      else
      {
        Send_Frame(UA_pkt);
        Abort = 1;
        Ready = 2;
      }
    }
    else if (Command == SABM)//(B3 && !B2 && !B1 && B0)          //SABM
    {
      Command = 0;
      if (Ready==2)
      {
        write16(CMX7164, 0x49, 0x810D);            // Writes Termination character 0x0D
        Send_Frame(DM_pkt);
        End_Tx();
        return;
      }
      else
      {
        Send_Frame(UA_pkt);                  // Assuming only Preamble will be transmitted now
      }
      // Set all initialization parameters here
      Front=0; Rear=0; Empty=1; Abort=0; Ready = 1; NS = 0;
    }
  }
// No else
  {
    static int Bytes2Tx=Frame_Len, Offset=0;
    static unsigned int  accumulator = 0xFFFF;

    if (Empty)
    {
      write8(CMX7164, 0x4A,0x18);   // 4 bytes in each data block, after which CMX7164 will request more data from host
      CON_7164;
      delay(Delay1);
        UCB0TXBUF = 0x48;
        while (UCB0STAT & UCBUSY);
        for (int i=1;i<=8;i++)
        {
          UCB0TXBUF = 0xAA;
          while (UCB0STAT & UCBUSY);
        }
      DIS_7164;
      delay(Delay1);
      return;
    }
    else
    {
      if (Bytes2Tx >= 15)
      {
        write8(CMX7164, 0x4A,0x1F);   // 4 bytes in each data block, after which CMX7164 will request more data from host
        CON_7164;
        delay(Delay1);
          UCB0TXBUF = 0x48;
          while (UCB0STAT & UCBUSY);
          for (int i=1;i<=15;i++)
          {
            UCB0TXBUF = *(ptr[Front]+Offset);
            accumulator = ((accumulator & 0x00FF) << 8) ^ crcTable[((accumulator >> 8) ^ *(ptr[Front] + Offset++)) & 0x00FF];
            while (UCB0STAT & UCBUSY);
          }
        DIS_7164;
        delay(Delay1);
        Bytes2Tx -= 15;
        return;
      }
      else if (Bytes2Tx>0 && Bytes2Tx<15)
      {
        write8(CMX7164, 0x4A,0x10 + (0x0F&(char)Bytes2Tx));     // 4 bytes in each data block, after which CMX7164 will request more data
        CON_7164;
        delay(Delay1);
          UCB0TXBUF = 0x48;
          while (UCB0STAT & UCBUSY);
          for (int i=1;i<=Bytes2Tx;i++)
          {
            UCB0TXBUF = *(ptr[Front]+Offset++);
            accumulator = ((accumulator & 0x00FF) << 8) ^ crcTable[((accumulator >> 8) ^ *(ptr[Front] + Offset++)) & 0x00FF];
            while (UCB0STAT & UCBUSY);
          }
        DIS_7164;
        delay(Delay1);
        Bytes2Tx = 0;
        return;
      }
      else
      {
        write8(CMX7164, 0x4A,0x13);                        // 3 bytes in data block
        CON_7164;
        delay(Delay1);
          UCB0TXBUF = 0x48;
          while (UCB0STAT & UCBUSY);
            UCB0TXBUF = accumulator>>8;
            while (UCB0STAT & UCBUSY);
            UCB0TXBUF = accumulator;
            while (UCB0STAT & UCBUSY);
            UCB0TXBUF = 0x0D;
            while (UCB0STAT & UCBUSY);
        DIS_7164;
        delay(Delay1);
        Front = (Front+1)%3;
        if (Front == Rear)        Empty=1;
        Offset=0;
        Bytes2Tx=Frame_Len;
        accumulator = 0;
        return;
      }
    }
  }
}


void MSP_initialize()
{

// Port1 Setting
  P1REN = BIT0 + BIT1 + BIT3;                                    // Pull-up/Pull-down Enabled
  P1OUT = BIT0 + BIT1 + BIT2 + BIT3 + BIT4;         // Pull Up
  P1DIR = BIT1 + BIT2 + BIT3 + BIT4;
  P1IES = 0x01;                                                              // Negative Edge triggered
  P1IFG = 0x00;
  P1IE = 0x00;

// CMX7164 and CMX991
  UCB0CTL1 = UCSSEL_2 + UCSWRST;
  UCB0CTL0 = UCMSB + UCMST + UCSYNC + UCCKPH;
  UCB0BR0 = 0x08;
  UCB0BR1 = 0x00;
  P3SEL = BIT1 + BIT2 + BIT3;
  P3DIR = BIT1 + BIT3;
  UCB0CTL1 &= ~UCSWRST;           // SPI working at 2MHz

//  UART1 initialization for communcation with PC
  UCA1CTL0 = 0;
  UCA1CTL1 = UCSSEL_2 + UCSWRST;
  UCA1BR0 = 0x34;
  UCA1MCTL = UCBRF_1 + UCBRS_0 + UCOS16;
  P3SEL |= BIT6 + BIT7;
  P3DIR |= BIT6;
  UCA1CTL1 &= ~UCSWRST;             // Transmitting at 19.2 kbps

// For Memory Card Interfacing
  UCB1CTL1 = UCSSEL_2 + UCSWRST;
  UCB1CTL0 = UCMSB + UCMST + UCSYNC + UCCKPL;
  UCB1BR0 = 0x02;
  UCB1BR1 = 0x00;
  P5SEL = BIT1 + BIT2 + BIT3;
  P5DIR = BIT0 + BIT1 + BIT3;
  UCB1CTL1 &= ~UCSWRST;

// For Communication with ADF7020
  UCA0CTL1 = UCSWRST;
  UCA0CTL0 = UCMSB + UCSYNC + UCCKPL;       P3SEL |= BIT0 + BIT4;

// Port2 Interrupt
  P2REN = BIT0;
  P2OUT = 0x00;
  P2IES = 0x00;//BIT0;
  P2IFG = 0x00;
  P2IE = BIT0;

  P4DIR = 0xFF;
  P4OUT = 0x00;

// For ADF bit-banging
  P6REN = BIT0 + BIT1 + BIT2;
  P6DIR |= BIT1 + BIT2 + BIT4 + BIT5 + BIT6 + BIT7;
  P6OUT = 0x00;
}
```

REFERENCES

[1] **Aboutanios, E.** and **S. Reisenfeld** (2007). Frequency estimation and tracking for low earth orbit satellites.

[2] **Datasheet** (). Transcend 2gb microsd card manual. URL `http://www.comx-computers.co.za/download/transcend/TS2GUSD2-P3.pdf`.

[3] **Datasheet** (2005). Adf7020-1 datasheet. URL `http://www.analog.com/en/rfif-components/rfif-transceivers/adf7020-1/products/product.html#product-documentation`.

[4] **Datasheet** (2012). Adf4351 datasheet. URL `http://www.analog.com/en/rfif-components/pll-synthesizersvcos/adf4351/products/product.html#product-documentation`.

[5] **Datasheet** (Dec 2004). Msp430x2xx family user's guide. URL `http://www.ti.com/lit/ug/slau144i/slau144i.pdf`.

[6] **Datasheet** (May 2012). Cmx7164 datasheet. URL `http://www.cmlmicro.com/products/CMX7164_Multi_Mode_Wireless_Data_Modem/?q=cmx7164&curr=32`.

[7] **Datasheet** (Oct 2012). Cmx991 datasheet. URL `http://www.cmlmicro.com/products/CMX991_RF_Quadrature_Transceiver/?q=CMX991&curr=16`.

[8] **Datasheet** (Rev A1 DS060921). Rf5110 manual. URL `http://www.rfmd.com/CS/Documents/RF5110DS.pdf`.

[9] **Foust, F.** (). Secure digital card interface for the msp430. URL `http://alumni.cs.ucr.edu/~amitra/sdcard/Additional/sdcard_appnote_foust.pdf`.

[10] **Peterson, W. W.** and **D. T. Brown** (). Cyclic codes for error detection. *Proceedings of the IRE*.

[11] **Rife, D. C.** and **R. R. Boorstyn** (1974). Single-tone parameter estimation fiom discrete-time observations. *IEEE Transactions on Information Theory*.

[12] **Shinsuke Hara, Y. T., A. Wannasarnmaytha** and **N. Morinaga** (1997). A novel fsk demodulation method using short-time dft analysis for leo satellite communication systems.

[13] **Tomasi, W.**, *Advanced Electronic Communications System*. Sixth edition, PHI, .