Digital Back End Electronics For HEPD

A THESIS

to be submitted by

Satish Kumar R

EE11M061

for the award of the degree

of

MASTER OF TECHNOLOGY

under the guidance of

Prof. Nitin Chandrachoodan

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

CHENNAI-600036

CERTIFICATE

This is to certify that the thesis titled **“Digital Back End Electronics For HEPD”,** submitted by Mr. Satish Kumar R, to the Indian Institute of Technology Madras, Chennai for the award of the degree of Master of Technology, is bonafide record of research work done by him under my supervision. The contents of the this thesis, in full or parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Nitin Chandrachoodan
Project Guide
Assistant Professor
Dept. of Electrical Engineering
IIT-Madras, Chennai-600036

Place: Chennai

Date: 29 May 2013

ACKNOWLEDGEMENT

I would like to take this opportunity to express my sincere gratitude to my project guide **Dr. Nitin Chandrachoodan,** Department of Electrical Engineering Department. He put a consider amount of time and effort to explain the concepts related to project and helped me to understand the project clearly. His guidance has broadened my Knowledge and thinking level. He has been a constant source of motivation, moral support and inspiration throughout the study which made me more confident.

Furthermore I would also like to acknowledge with much appreciation the staff of Embedded Systems Lab and IE lab who gave the permission to use all required equipment to complete the task. A special thanks goes to my team mates Sanak, Akshay, Nitin, Ankith and sai Surya Teja for our philosophical debates, exchanges of knowledge and skills. Many thanks go to my lab mates Vikas, Karthikeyan, Chaitanya and Amit spending their time with me and for all the fun we had in the last year.

Satish Kumar R

Abstract

The main aim of this project is to Design, Implement and Interface a Digital Payload Block for IITM Student Satellite.

This Payload block has to sample the analog input voltage at a very high rate which is around 278 KHz. Once sampling is done, this sampled voltage has to be mapped in to corresponding energy value using a voltage to energy look-up table. Create a Histogram of this energy Vs number of incident particles (say electrons and protons).Histogram represents the energy spectrum of the particles. Histogram data has to be send to external memory or any other subsystems present on the Satellite at a regular interval of time. On the other hand it has to interface this block with other subsystems present on the satellite in order to communicate with them and send health data to the master block of the satellite.

A complete system design, implementation of the algorithm and the interface with other subsystems is done. Finally design of this block and some of the test results are given showing how this subsystem can be used in any future nanosatellite system.

Contents

List of Tables

List of Figures

ABBREVIATION

ADC	Analog to Digital Converter
API	Application Peripheral Interface
BEE	Back End Electronics
CAN	Controller Area Network
COM	Communication
DAC	Digital to Analog Converter
DMA	Direct Memory Access
DMIPS	Drhystone Million Instructions Per Second
EPS	Electrical Power Sub system
FEE	Front End Electronics
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FRAM	Ferroelectric Random Access Memory
GPIO	General Purpose Input Output
HEPD	High Energy Particle Detector
I2C	Inter Integrated Circuit
ISR	Interrupt Service Routine
MAC	Media Access Control
MIPS	Million Instructions Per Second
OBC	On Board Computer
PHD	Peak Hold Detector
PMT	Photo Multiplier Tube
SCLK	Serial Clock
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

Chapter 1

Introduction

The IITMSAT project goal is to study the energy spectrum of charged particles in the upper ionosphere and understand the effects of solar storms, lightning storms and seismic activity.The ionosphere starts from a height of about 50kms from the surface of the earth and extends to more than a 1000km. Electrons are stripped off the gas molecules, resulting in ions, by the ultra-violet radiation of the Sun as well as incident X-rays.The ionosphere is thus a shell of trapped charged particles such as electrons, protons and other charged atoms and molecules which plays an important part in radio wave propagation and influences many other earthly phenomenon. Changes in the particle concentration can help in weather prediction and earthquake predictions.

Recent satellite studies indicate an anomalous increase in the charge particle flux in the lower layers of the ionosphere, a few hours before the occurrence of an earthquake. The IIT Madras Student Satellite aims to expand the existing know-how on this phenomenon, to ultimately develop an earthquake prediction model. Continuous monitoring of the ionosphere using HEPD will allow us to obtain the energy spectrum of anomalous charge particle bursts that may occur in correlation with earthquakes.[10]

1.1 Problem Statement

To design a Digital Payload Block which can be used to process the data and to store the processed data in the external memory. Design of this block includes selection of micro-controller, external memory and other hardware components. And finally to interface this Digital Payload Block with other Subsystems on the Satellite

Functions to be implemented[1]

- Sample the input voltage signals from the Front End Electronics (FEE), provided the coincidence logic is satisfied. Convert the average voltage of the input signal to energy (from a given table).
- Check for the particle type (proton, electron, others) from the energies associated with the signals from FEE.
- Generate histograms (of energy vs. particle counts) with fine (0.1 sec) and coarse temporal resolution (3 sec) for protons and electrons.
- Detect particle bursts (sudden increase in particle counts).
- Store histograms with fine and coarse temporal resolution if BEE detects a particle burst, else store only histograms with coarse temporal resolution.
- Send stored data to the on-board computer of the satellite when requested.
- Control and regulate the power-supply to different electronic components in HEPD.
- Monitor temperature characteristics of important components such as PMT, scintillators, Preamplifier etc. through temperature sensors.
- Monitor current and voltage of different components in HEPD through appropriate sensors.

1.2 Detector Configuration

The High Energy Particle Detector is a solid state charged particle detector, capable of measuring the energy of charged particles that are incident on it. Figure 1.1 is a schematic describing the basic configuration and working of the High Energy Particle Detector.

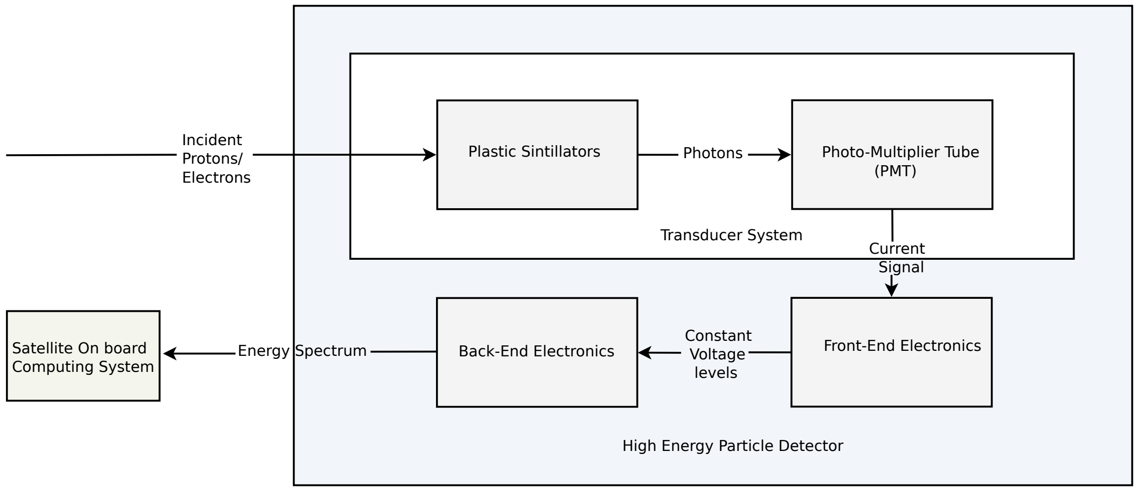


Figure 1.1: HEPD Schematic [1]

The detector consists of a Transducer System made of plastic scintillators which converts incident charged particles to photons. The total number of photons produced is proportional to the energy deposited by the charged particle in the plastic scintillators. These photons are then routed to a Photo- multiplier tube (PMT) that produces photo-electrons proportional to the number of photons incident on the PMT. The PMT then transmits an amplified current pulse with the area under the pulse proportional to the number of photo-electrons produced. The current pulse is then converted to a voltage signal by the analogue Front-End Electronics (FEE), such that the amplitude of this voltage signal is also proportional to the number of photo-electrons. The FEE provides a steady voltage output to the digital Back-End Electronics (BEE). The BEE then processes the input voltages to calculate the energy and type of the incident particle. The BEE finally stores the energy spectrum of protons and electrons incident on the detector, which will be transmitted to the ground station .[10]

Chapter 2

Topology and Hardware

Satellite has 5 subsystems

- HEPD
- OBC
- ADCS
- COM
- EPS.

Each subsystem has its own functionalities

- **HEPD**
  - To measure the count rates of high energy protons and electrons in the ionosphere and to generate the energy spectrum of these particles.
- **OBC**
  - It controls all the functions of the Satellite. The control of data flow between other subsystems is handled by OBC. Health of the satellite is monitored by this subsystem.

- **ADCS**
  - ADCS subsystem is to deduce the satellite's attitude by sensing environmental variables, determine the correction torque to be applied using a suitable control algorithm, and apply the torque using actuators.
- **COM**
  - It consists of tele-command receiver: It is used to receive data from the ground station at a pre-defined modulation scheme, demodulate it and provide the data to on-board computation system.The Telemetry and payload data transmitter system: It transmits data collected and processed by payload and on-board computer respectively.The Ground Segment consists of uplink and downlink antennas.
- **EPS**
  - Electrical power subsystem (EPS) is mainly responsible for supplying continuous and regulated power to on-board subsystems like ADCS, OBCS, Communication, payload, sensors etc.

On a satellite we have 2 kinds of data

- Science Data
- Health Data.

Data rate of Science data is very high when compare to Health data, it is in the order of 1000 times greater. Now it is very important that how these two kinds of data is transferred between these five subsystems and finally to the ground station. So we need a proper topology by which these subsystems can talk to each other and can have an efficient way of data transfer between these subsystems. Through out our project we have been implementing different topologies, by learning new things at each and every step.

2.1 Topologies Considered

2.1.1 Topology 1

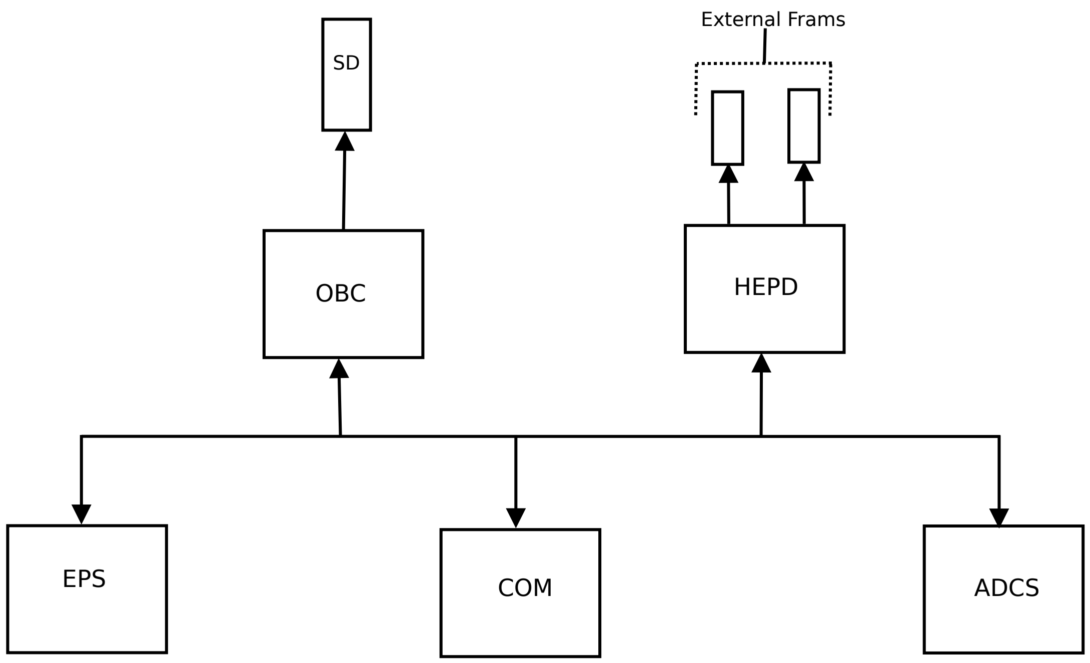


Figure 2.1: Topology 1

This topology was proposed by satellite team. The main bus which connects all five subsystems uses I2C protocol for communication. The OBC block has its own external memory (SD-CARD) which is connected through SPI protocol. SD-CARD is used to store science data and the health data. HEPD block has two FRAMs. FRAMs are used for temporary store of science data before transferring it to external memory through HEPD block and OBC block. Two FRAMs are there in order to switch the data between these two FRAMs because the FRAM size is limited to 1MByte so that we can transfer data while writing to one and reading from other.

Disadvantages

- As the data rates for HEPD is very high it keeps on sending the data to the OBC through main bus. This makes main bus busy.
- During transmission of data to ground station data transfer happens through OBC to

COM block and makes main bus busy. This time is high enough and is likely to be 5 -10 minutes. During this time HEPD cannot transfer the science data to SD-CARD. And this is serious problem due to limitation of FRAMs size, because FRAM cannot hold that much data.

2.1.2 Topology 2

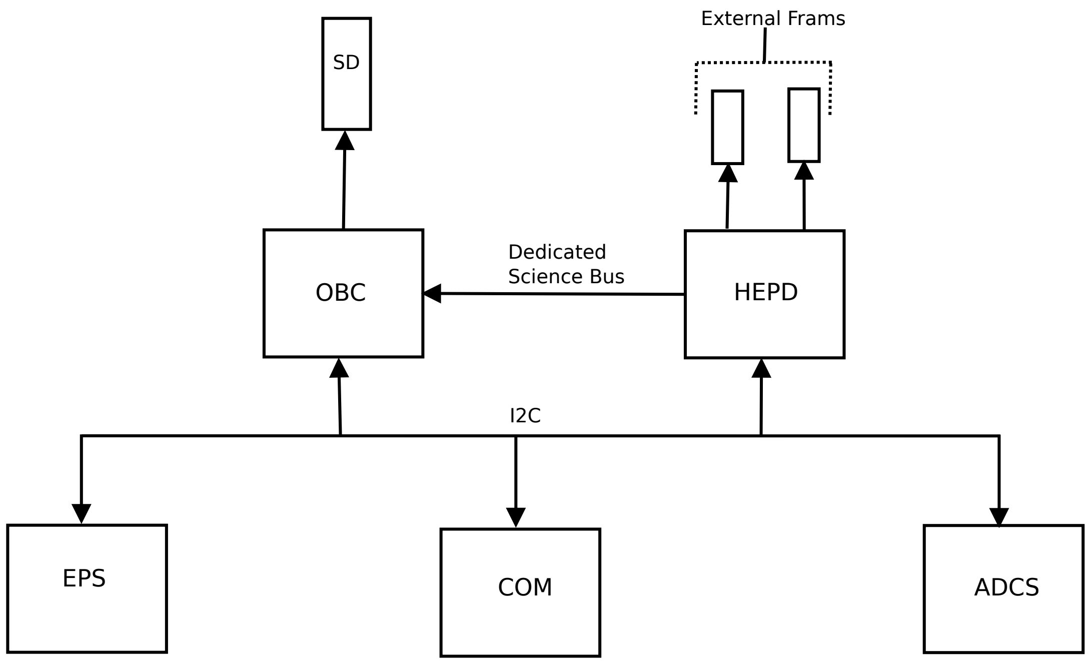


Figure 2.2: Topology 2

In order to overcome the problem faced in topology 1 we have chosen this topology. In this topology we have given a dedicated path to science data. This dedicated path lets you to write the data to SD-CARD even though main bus is busy.

Disadvantages

- Dedicated path results in more power consumption.
- More the number of paths more the chance of failure.

2.1.3 Topology 3

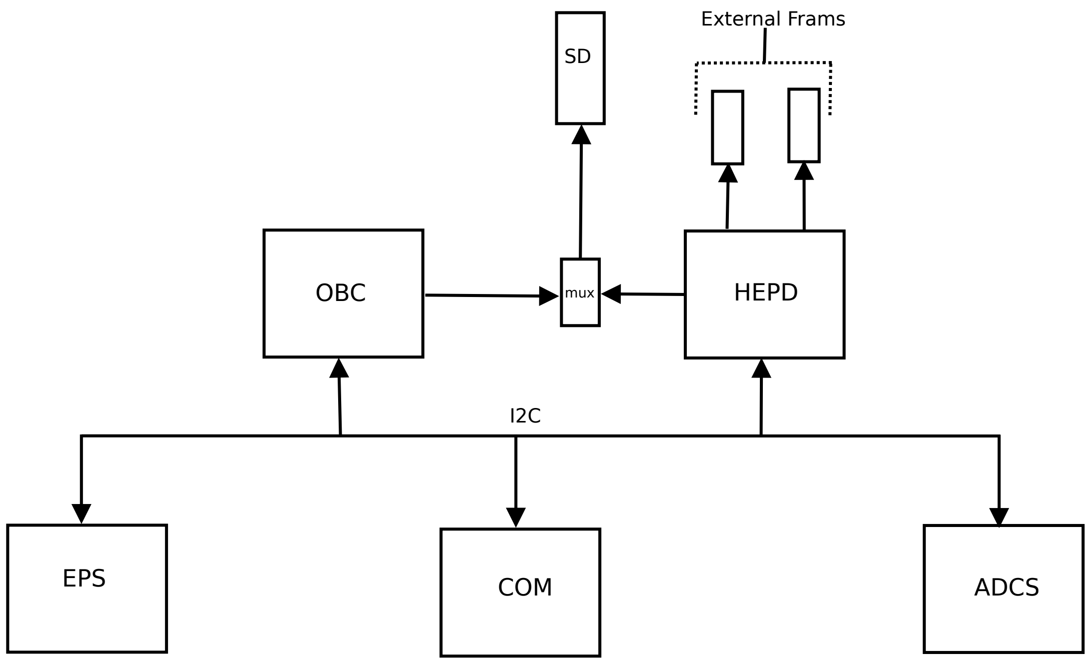


Figure 2.3: Topology 3

Since only two blocks OBC and HEPD communicates with external memory we thought of sharing the external memory between these two sub systems. The control bit of mux lies with OBC.

Disadvantage

During transmission of data to ground station data has to be read by OBC and then transferred to COM which creates unwanted delay.

2.1.4 Topology 4

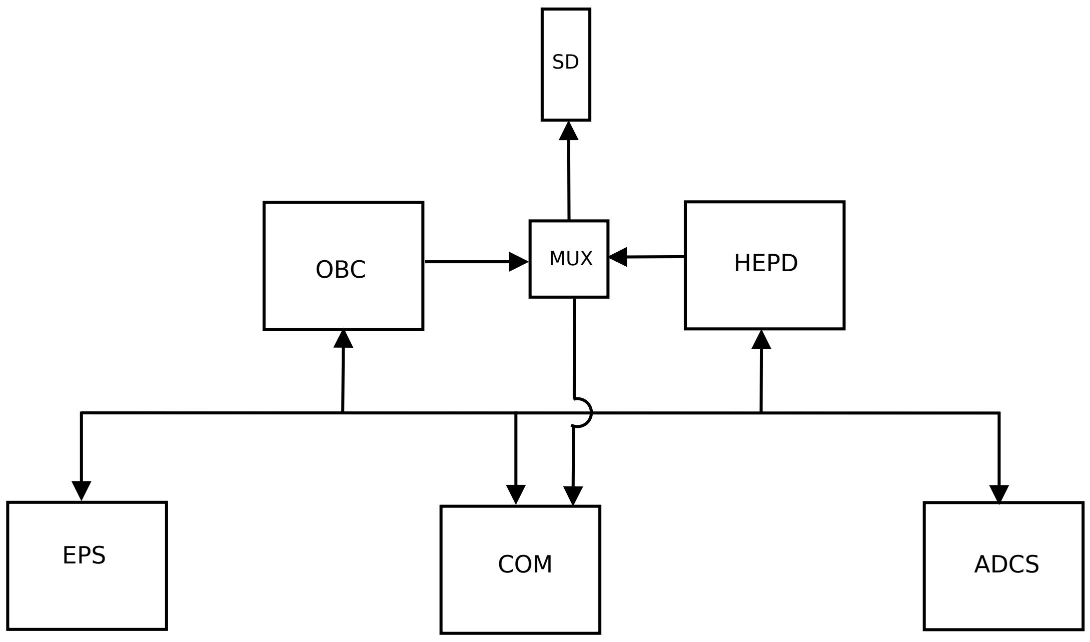


Figure 2.4: Topology 4

This topology was chosen in order to avoid the extra delay faced in topology 3. Here SD-CARD is shared between three sub systems HEPD, OBC and COM. In this topology

- HEPD will write science data to SD-CARD
- OBC will write health data to SD-CARD.
- COM will read the data form SD-CARD.

The control of mux lies with OBC.

Disadvantages

Controlling and synchronizing three subsystems becomes very difficult. We need to add extra buffers for this topology.

2.1.5 Topology 5

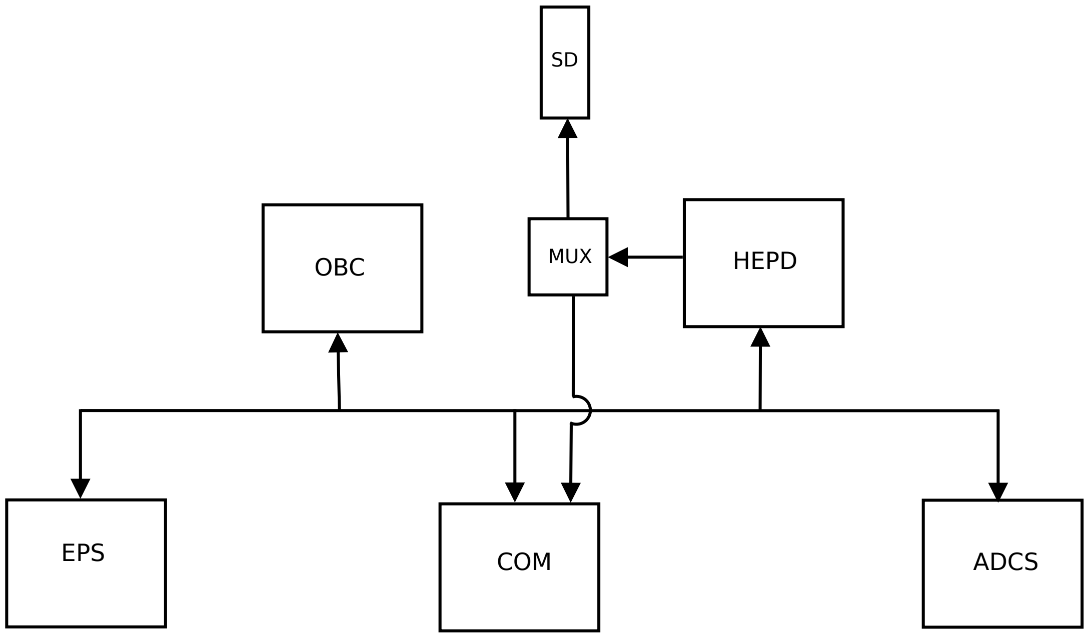


Figure 2.5: Topology 5

This topology was chosen to overcome the complexity of topology 4. In this topology

- HEPD will write the science data to SD-CARD.
- COM will write health data as well as it will read the data back in order to send it to ground station.

2.2 Hardware Selection

Selection of hardware requires to know what type of functions to be performed, what are its requirements and what are its constraints.

Functionalities to be performed is discussed in project statement of Introduction in chapter 1.

2.2.1 Requirements

- Maximum count rate to be detected: 2.8 x 10^5 counts/sec. => 1 particle in every 3.6 μs (considering uniform flux).
- Store the science data in the form of two histograms;
  - coarse temporal resolution (3 s)
  - fine temporal resolution (0.1 s)

2.2.2 Constraints

- Average power consumed < 750 mW.
- Sample an Analog signal within 1 μs.
- Maximum clock frequency of low-power industrial grade micro-controllers : 80 MHz.

After knowing these now we have an idea of what type of microcontroller we need to have. We should also know the application in which the microcontroller is going to be used. In our case it is Satellite, so we need to check some of the primary requirements like:

- Computational Performance
  - Generally microcontroller performance is measured in terms of MIPS. This is the only system which is going to perform all computational tasks so it should have high MIPS and higher operating frequency.
- Power Consumption
  - To get high power on a satellite is very difficult, so all subsystems should consume as less power as possible.
- GPIOs and features

    - On market we get many microcontrollers of similar specifications but different features and resolutions. Select a chip that has all the features with the resolution as per your needs.

- Availability of components
    - Components used in this design needs to be commercially available for a long time to come.
- Memory
    - We should have enough memory in the microcontrollers to program it and to makes any changes in future.
- Online support
    - Selecting an microcontroller which has good online support will help you with your ideas and solve most of your problems as the experience of other users is available for you.
- History
    - We have to check whether anyone has already used these types of hardware in the space and what are their ratings.

After considering all these we have chosen ARM Cortex-M3 based Stellaris LM3S9B92 microcontroller.

2.2.3 Stellaris LM3S9B92 microcontroller features

- ARM Cortex-M3 Processor Core.
- High Performance: 80-MHz operation; 100 DMIPS performance.
- 256 KB single-cycle Flash memory.

- 96 KB single-cycle SRAM.
- Advanced Communication Interfaces: UART, SSI, I2C, I2S, CAN, Ethernet MAC and PHY, USB.
- Analog support: analog and digital comparators, Analog-to-Digital Converters (ADC), on-chip voltage regulator.
- Industrial (-40°C to 85°C) temperature range.[7]

Chapter 3

Payload Algorithm Implementation

The main functionalities of payload are

- Collecting the Input Data
- Process the Input data and make Histograms and finally
- Store the Data in to some external memory.

This chapter will we will discuss how can we capture the Input data, process the captured data and finally convert the processed data in to histogram. The next chapter will discuss about how this data is transferred to external memory. On seeing the functionalities of the payload, we can clearly observe that things are happening in some specific order like, collecting the data , processing the data and storing the data. After knowing the sequence of operations and the priorities between these main functions we can have a Flow chart .

3.1 Flow Chart

Flow chart showing the sequence of operations involved in payload subsystem

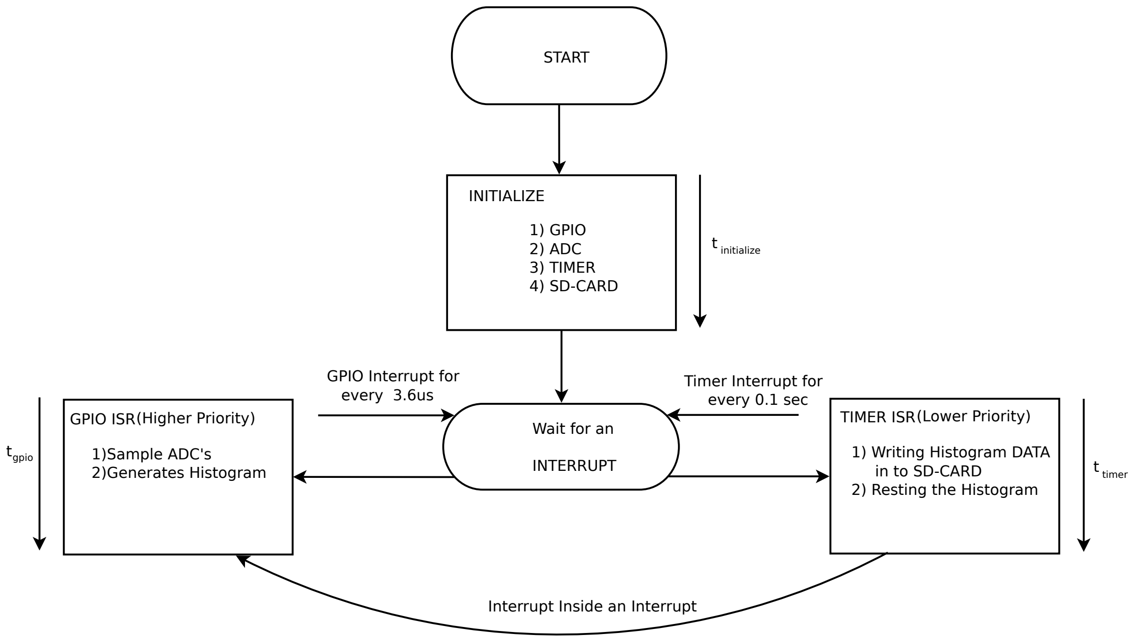

START
INITIALIZE
1) GPIO
2) ADC
3) TIMER
4) SD-CARD
t initialize
GPIO Interrupt for every 3.6us
Timer Interrupt for every 0.1 sec
Wait for an INTERRUPT
GPIO ISR(Higher Priority)
1)Sample ADC's
2)Generates Histogram
tgpio
TIMER ISR(Lower Priority)
1) Writing Histogram DATA in to SD-CARD
2) Resting the Histogram
t timer
Interrupt Inside an Interrupt

Figure 3.1: Flow Chart

and how the interrupts comes, and at what rate they come, and what are all things need to be performed by the ISR. First we will start with sampling the input data and then generating the histogram and then sending the data to external memory.

3.2 Sampling Input Data

One of the important functionality to be performed by BEE is to sample the Input signal. Input signal to BEE comes from FEE.

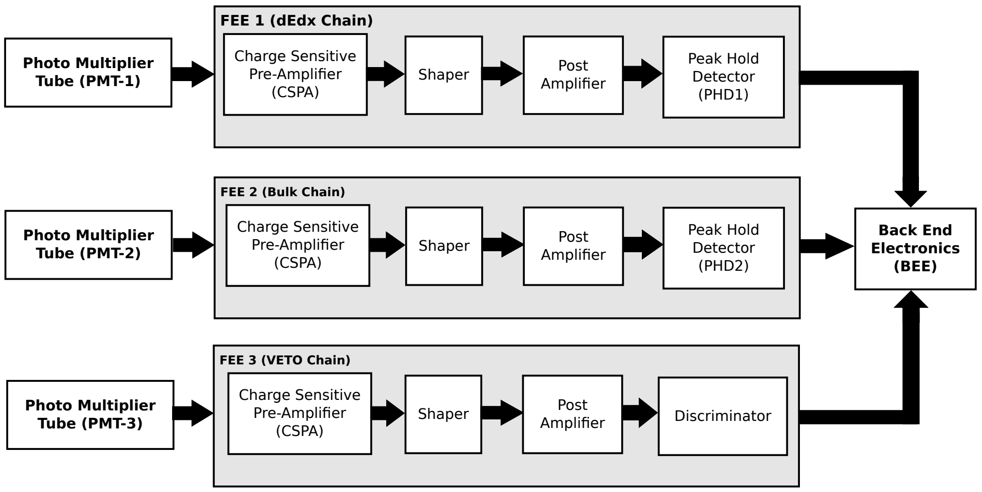

FEE 1 (dEdx Chain)
Photo Multiplier Tube (PMT-1)
Charge Sensitive Pre-Amplifier (CSPA)
Shaper
Post Amplifier
Peak Hold Detector (PHD1)
FEE 2 (Bulk Chain)
Photo Multiplier Tube (PMT-2)
Charge Sensitive Pre-Amplifier (CSPA)
Shaper
Post Amplifier
Peak Hold Detector (PHD2)
Back End Electronics (BEE)
FEE 3 (VETO Chain)
Photo Multiplier Tube (PMT-3)
Charge Sensitive Pre-Amplifier (CSPA)
Shaper
Post Amplifier
Discriminator

Figure 3.2: Functional Block diagram of FEE. [1]

The PHD holds the maximum value of the voltage signal from the post amplifier. This signal is the output of FEE modules 1 and 2. But FEE module 3 has a discriminator, instead of a PHD, that gives a voltage signal if the output from the post amplifier is above a threshold. The output of the FEE modules will then be fed in to the ADC of the BEE for further processing.BEE receives this signal as input to the ADC. ADC is a block available in microcontroller as one of the peripherals which accepts a continuous signal as input and converts to a discrete digital number which can be further processed by the micro controller.

The ADC available in Stellaris offers you with following Features

1. 10-Bit conversion Resolution.
2. 16 shared analog input channels.
3. Single-ended and Differential-input configurations.
4. Maximum sample rate of one million samples/second.
5. Four programmable sample conversion sequencers with corresponding conversion result FIFOs.
6. Efficient transfers using Micro Direct Memory Access Controller (μDMA). [7]

3.3 Block Diagram of ADC

The Stellaris Micro controller have two identical Analog-to-Digital Converter modules called ADC0 and ADC1. These two ADC s module share the same 16 analog input channels.

Each ADC module can

1. Operate Independently.
2. Execute different sample sequences.
3. Any time they can sample any of the analog input channels.
4. Can generate different interrupts and triggers &
5. Can have independent μDMA channels.

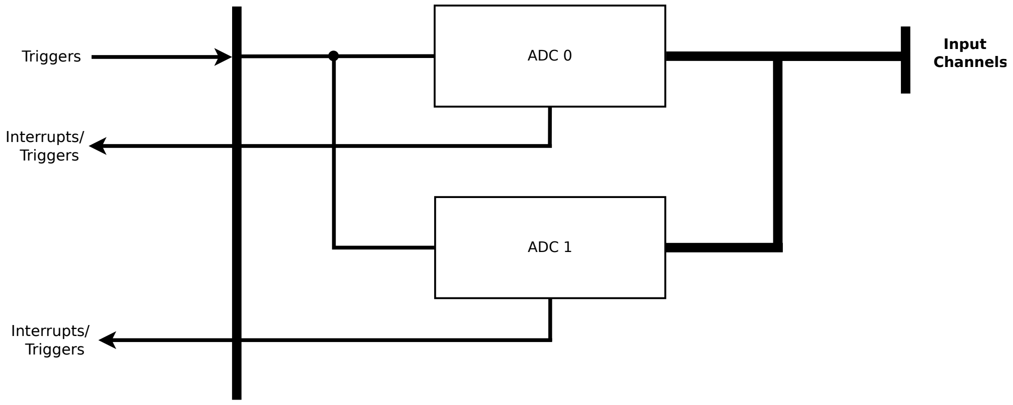


Figure 3.3: Implementation of Two ADC Blocks.[7]

ADC s which are available in Stellaris collects sample data by using a programmable sequence-based approach. ADC module has four sampling sequencers of depth 1,4,8 which allows you to sample different analog sources with a single-trigger event.

3.3.1 Sample Sequencers

We have four sample sequencers which are completely independent of each other. Each sample sequencer has its own set of configuration registers as shown below.

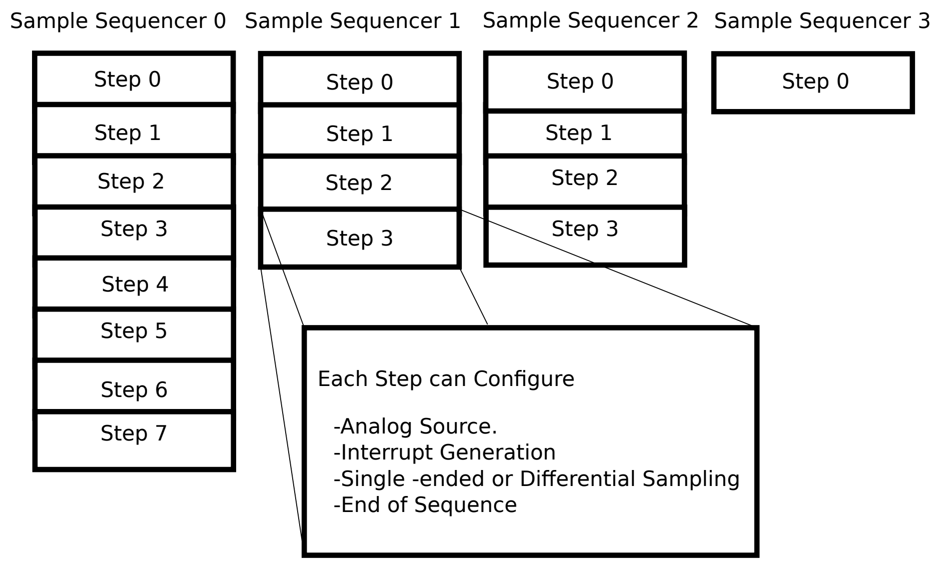
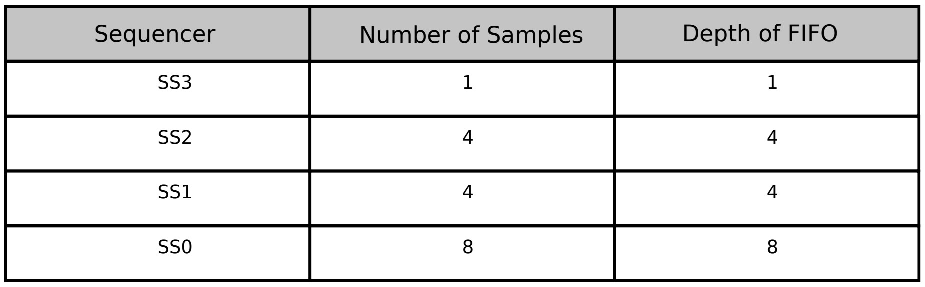


Figure 3.4: Sample Sequencer Structure.[11]

We can see from the above figure all the steps in the sample sequencer are configurable allowing you to select different analog sources like temperature sensor or the signal itself. Each sampler sequence has its own FIFO depth and can collect those many number of samples.



Sequencer	Number of Samples	Depth of FIFO
SS3	1	1
SS2	4	4
SS1	4	4
SS0	8	8

Figure 3.5: Samples and FIFO Depth Sequencers.[7]

The above figure shows the maximum no of samples that each sequencer can capture and its corresponding FIFO depth. The number of samples an ADC can capture from each sample sequencer is also programmable by having an END bit set. It also give you flexibility to handle multiple sample sequencers which are triggered simultaneously by giving them configurable priority.

How it Operates

When a sample sequencer is triggered

- It samples the signal at the programmed sampling rates say 250K, 500K, 1M samples/sec.
- Sampling continues till it encounters the END (END bit can be set for any step in sequence) bit set, indicating end of the sequencer.
- Collect the converted result in the sampler sequencer FIFO.

Each FIFO entry is a 32-bit word, with the lower 10-bits containing the conversion result.

.

3.4 Sampling the Data

In space we don't know the rate at which the particles comes. To capture the particle and to convert it in to signal by the FEE it takes some time and is called **DEAD Time**.

DEAD Time

let us say at t=0 we have a particle incident on the payload scintillators and to convert this particle in to an signal it takes 3 sec, now this 3 sec is the dead time which means to say if we have an another particle coming at t=2 sec then it will not get detected by the FEE because it takes some time to process the signal and in the mean time it is not able to detect the another particle.

3.4.1 Interface between FEE and BEE

The output of peak hold detectors of FEE are connected to the input of ADC of BEE. FEE has some DEAD time and BEE has to sample that data within that DEAD time and send an reset signal to the FEE saying that sampling of data is done. After receiving the reset signal from BEE the FEE peak hold detector output goes low. This is done to to have an indication that for every new particle comes there will be an rising edge at the output of the peak hold detector.

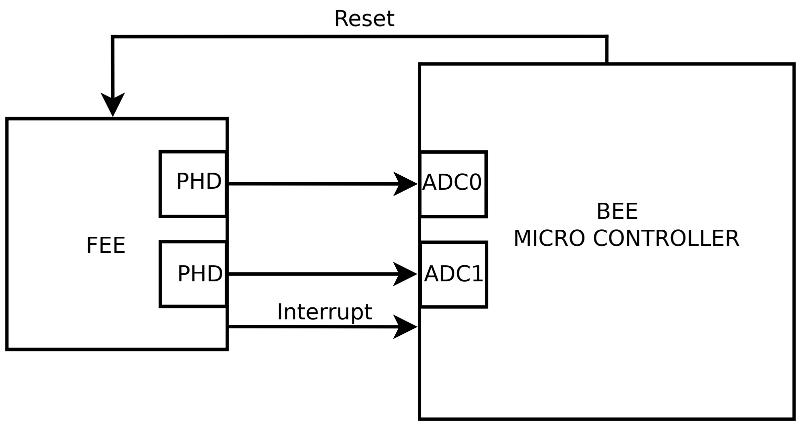


Figure 3.6: Interface of FEE and BEE

.In order to achieve this we are choosing an interrupt based method. Every time FEE detects the particle and generates new signal we get an interrupt to the BEE indicating that a new particle has been detected. So whenever an interrupt comes to the BEE we have chosen an ISR to sample the incoming data and store in a temporary location which can be used for further processing.

Now the Dead time for our system is 3.6μs which means for every 3.6μs we will have a new data. So the output of Peak Hold detector looks like this

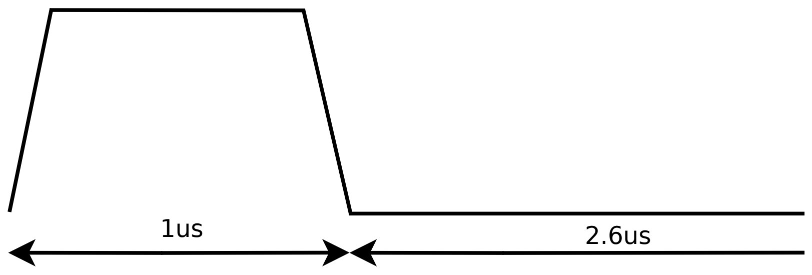


Figure 3.7: Peak Hold Detector Output

.

if we observe the time at which we are getting the input signal is very small i.e frequency of the input signal is

$$\frac{1}{3.6\mu s} = 277.777 \text{ KHz.}$$

which is at a very high rate. The microcontroller which is chosen has a clock speed of 80 MHz.

If the input comes at every 3.6us & If the micro controller operates at 80 MHz.

The no of clock cycles we can get to perform every ISR is

$$time \times frequency of operation = clock cycles$$

$$3.6\mu s \times 80MHz = 288$$

so we have 288 clock cycles which is quite good amount of clock cycles to perform an ISR.

3.5 Method Adopted to Sample Input Data

For our experiment we need to sample the input data, process the data and then we need to send the data to an external memory in a given interval of time. Sampling the input data we have planned to do in an ISR. Now how much time you spend in the ISR depends on the approach you have chosen. Smallest the time to do an ISR is the best approach and we need this so that we can save the time and clock cycles so that we can do other things. So the methods we adopted to do this task are

- DMA
- Direct Hardware Register method.

3.5.1 DMA Approach

3.5.1.1 Why DMA

Transferring of data can be done using DMA without processor intervention. In a general computing system when data has to be transferred between say memory and interfaces, CPU reads the data and then it transfers. Problem with this approach is if we have a large amount of data to be transferred then CPU does all the transfer and it gets stuck over there for long time and meanwhile it can't do any other processing. Transferring of data doesn't require any processing or computing so instead of CPU which is capable of doing some computing if any other device can do this task it will be very efficient way of transferring the data and usage of CPU. DMA does this work it takes over the CPU or Controller and does this transferring when ever CPU is not using the memory and peripherals. In th meanwhile CPU can do other tasks like handling interrupts and carrying out internal operations which doesn't require memory.

3.5.1.2 Features of DMA in Stellaris

The μDMA provides following features

- 32-Channel configurable μDMA controller.
- Supports
  - Memory-to-Memory,
  - Memory-to-Peripheral, and
  - Peripheral-to-Memory in multiple transfer modes.
- Basic for simple transfer scenarios.
- Ping-pong for continuous data flow
- Highly flexible and configurable channel operation.
  - Independently configured and operated channels
  - Dedicated channels for supported on-chip modules
  - Primary and secondary channel assignments
- Data sizes of 8, 16, and 32 bits.

- Transfer size is programmable in binary steps from 1 to 1024.
- Source and destination address increment size of byte, half-word, word, or no increment.
- Maskable peripheral requests.[7]

3.5.1.3 Ping-Pong Method

This method is used for continuous data flow.

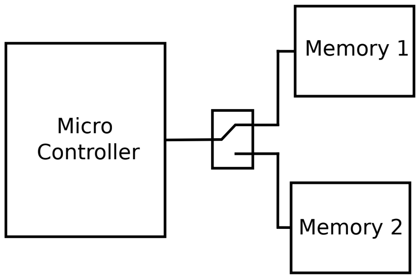


Figure 3.8: Popping method

By this approach if the data becomes full in memory location 1 then micro controller starts transferring the data to second memory location 2, mean while we can collect he data from location 1 and by this way transfer of data can be done continuously.

Problems with this approach

- Because the interrupt rate of input data was too high filling up the memory locations was at a faster rate when compared to retrieving the data back from memory locations and processing it for Energy conversion and other processing.[9]
- Every time the transfer of specified number of bytes is done using DMA the channel gets disable and we have to enable it again for next transfer which takes approximately 80 clock cycles which is very high in an interrupt based approaches.

3.5.2 Direct Hardware Register method

In this method we used directly the hardware registers. In this approach number of clock cycles to execute an instruction reduced very much say more than 50 %.

3.6 Results and conclusion of ADC

Earlier we have used API functions provided by the StellarisWare to trigger ADC but it was taking 20 clock cycles and when we used direct hardware register for triggering the ADC it took only 10 clock cycles.

By using hardware register method we can reduce the number of clock cycles to more than half or at least half for many API functions provided by the StellarisWare.

The output of an ADC for a Sine wave as a input with frequency 100 KHz and the sampling frequency of 1MHz is shown below ,

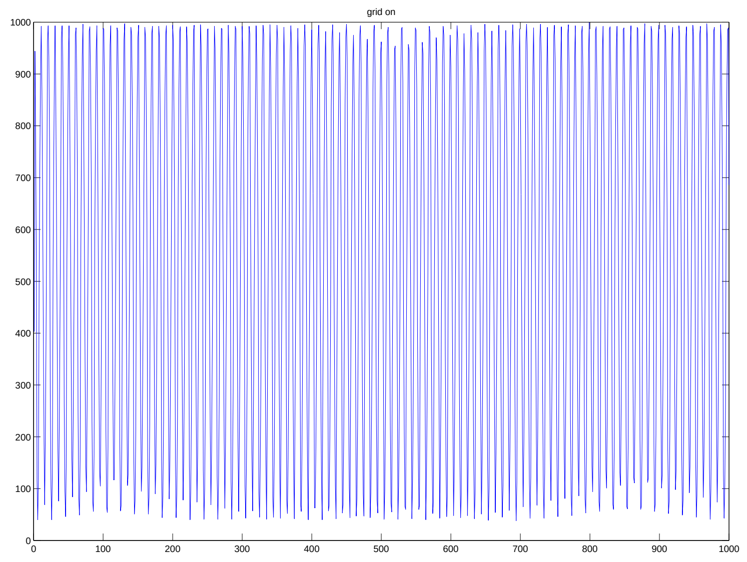


Figure 3.9: Time Domian

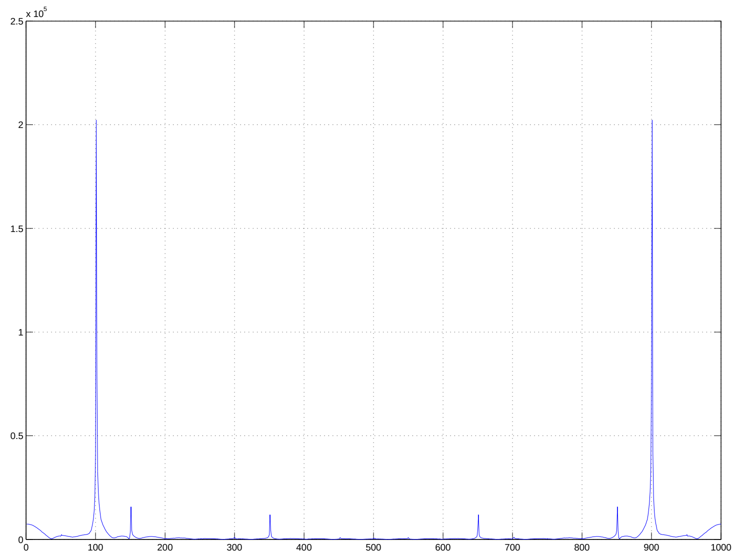


Figure 3.10: Frequency Domain

from the above figure we can observe that the peak occurs at 100 KHz which is nothing but the frequency of the given input signal.
From the above experiment conducted we found no errors in the ADC operation.

3.7 Histogram Generation

First we initialize the required peripherals and different functional blocks like ADCs, Timers and Power-ON SD-Card. The time required to initialize these blocks and the peripheral, let say “$t_{\text{initialize}}$”. Now we wait for an interrupt to occur. There are two interrupts in the system

- GPIO
- TIMER

3.7.1 GPIO Interrupt

This is an external GPIO interrupt. The input of this GPIO comes from FEE for every 3.6μs and for a microcontroller operating at 80 MHz the corresponding

CLOCK CYCLES = 288

so for every 3.6μs our system receives an interrupt from an FEE. This is Input to the system so it is very important and in any case we have to respond to this interrupt by halting others. Because of this requirement we made this interrupt as the Highest priority interrupt.

Functions to be Performed by GPIO Interrupt

- Whenever an GPIO interrupt comes it has to trigger the ADCs and sample the data.
- Convert the voltage in to Energy by using an LOOK-UP table.
- Generate histograms (of energy vs. particle counts) with fine (0.1 sec) and coarse temporal resolution (3 sec).

3.7.1.1 LOOK-UP Table

We are using an 1-D array as a look-up table where each address location of an array is an output of an ADC. Stellaris offers you a 10-bit ADC so the output of ADC range from 0-1023 values. For easy calculation and to have one to one mapping we have chosen the 1024 energy values ranging from 0-25. Finally we have chosen an array of size 1024 to have this 1024 energy values. Now this array address represents the ADC output and the corresponding value at that address represents the energy value for a particular voltage.

Now the energy look-up table (which is nothing but 1-D array of 1024 values) is filled in this manner,

1st 25 locations are filled with Zeros,

2nd 25 locations are filled with Ones,

“

“

24th 25 locations are filled with a value 23, and rest are filled with with a value 24.

Now the if the ADC output voltage

is in the range of 0-24 it gives an energy value of Zero,

and if the range is 25-49 it gives an energy value of One,

“

“

and if the range is 999-1023 it gives an energy value of 24.

Having this energy table the output of ADC is given as an address to the array to get the corresponding energy value.

3.7.1.2 Histogram

Generating histograms (of energy vs. particle counts) is to reduce the data size. The data rates are so high because we are sampling the input data at a sampling frequency of 1MSPS. So in one second 1MBytes of data is getting generated and to handle such large data is very difficult. Such large data are difficult to store and process on the micro-controllers.Microcontrollers doesn’t have enough internal memory to store that much data. Our requirement is only to know how many particles of a particular energy has incident, so it is not needed to store the actual data but the particle count of particular energy is enough, below figure shows how an histogram in created,

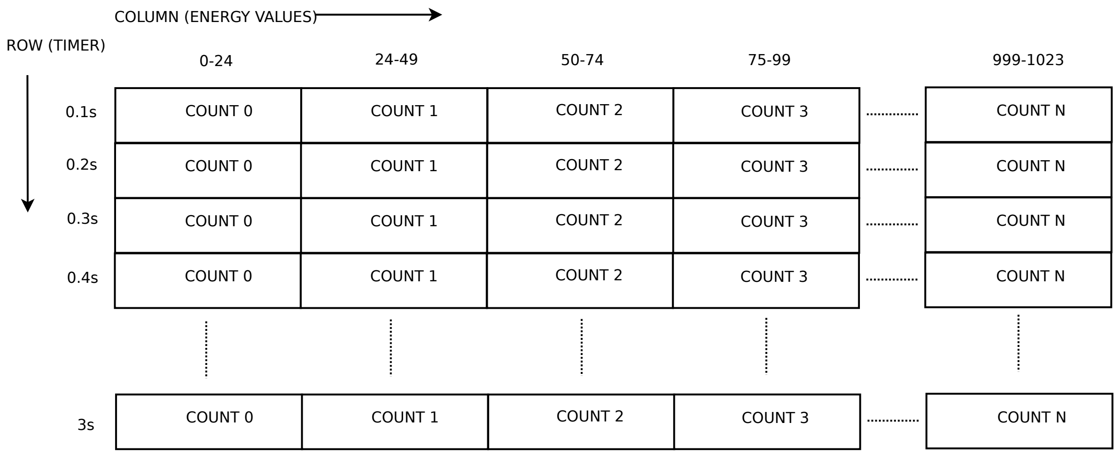


Figure 3.11: Histogram

.Here we create an histogram for every 0.1s and this 0.1s data is called FINE FRAME and for every 3s we have an another histogram running in parallel which is nothing but the COARSE FRAME data. Once FINE FRAME data histogram is created it is stored in a SD-CARD. For every 0.1s we store the FINE FRAME data and for every 3s COARSE FRAME data is stored in a SD-CARD.

3.7.1.3 Time required to serve an GPIO ISR

As mentioned earlier GPIO has to perform three functions

- Whenever an GPIO interrupt comes it has to trigger the ADCs and sample the data.
- Convert the voltage in to Energy by using an LOOK-UP table.
- Generate histograms (of energy vs. particle counts) with fine (0.1 sec) and coarse temporal resolution (3 sec).

we have two ADCs sampling two signals and two FRAMES one is FINE FRAME and other is COARSE FRAME. FINE FRAME and COARSE FRAME has to be update whenever an ADC samples a new data.

So we have

ADC0 updating one FINE FRAME and one COARSE FRAME and

ADC1 also once again updating the same FINE FRAME and the COARSE FRAME.

Functions	**Number of Clock Cycles**
Clr the Interrupt	4
Trigger ADC0	10
Count Fine Frame for ADC0	33
Count Coarse Frame for ADC0	33
Trigger ADC1	10
Count Fine Frame for ADC1	33
Count Coarse Frame for ADC1	33
push+pop	4+13
Total No of Clock Cycles	**177**

Table 3.1: Total Number of Clock Cycles

.Total number of Clock Cycles = 177.

$$\begin{aligned} time \times frequency &= ClockCycles \\ so\, time &= \frac{ClockCycles}{frequency} \\ &= \frac{177}{80\mu s} \\ &= 2.2125\mu s. \end{aligned}$$

so to finish one GPIO interrupt (t_{gpio}) we require 2.2125μs.

$$\begin{aligned} Remaining\, time\, to\, get\, next\, interrupt &= 3.6 - 2.2125 \\ &= 1.3875\mu s \end{aligned}$$

and

$$\begin{aligned} No\, of\, Clock\, cycles\, remaining &= 288 - 177 \\ &= 111\, ClockCylces. \end{aligned}$$

These remaining time is used to serve any other process.

3.7.2 Timer Interrupt

This is an internal interrupt caused by using some of the timers present in the microcontroller. Timer interrupt occurs for every 0.1 s or for every 100 ms. In this interrupt service routine we are handling histogram data transfer which is generated in the GPIO interrupt to external devices like SD-Card or any other microcontroller.

3.7.2.1 Time required to serve a Timer Interrupt

Functions to be performed by timer interrupt

- Histogram data has to be transfer to SD-CARD.
- Reseting the FINE FRAMEs and COARSE FRAMEs.

Time required to send the Histogram data which is 50Bytes to an SD-CARD is 0.8 ms best and the worst case is 3 ms.

Total number of clock cycles required to service this ISR = 161558.

$$\begin{aligned}
time \times frequency &= ClockCylces. \\
so\, time &= \frac{ClockCycles}{frequency} \\
&= \frac{161558}{80\,\mu s} \\
&= 2.02\, ms.
\end{aligned}$$

In order to finish Timer interrupt (t_{timer}) we require 2.02ms time and we have 100 ms of time to get back to this interrupt.

$$\begin{aligned}
Remaining\, time &= 100 - 2.02 \\
&= 97.98
\end{aligned}$$

which is very good amount of time to do any processing on the Histogram data. The number of times the GPIO interrupt occurred in this 2.02 ms is

$$\begin{aligned} &= \frac{Time\,Required\,to\,serve\,this\,interrupt}{rate\,at\,which\,GPIO\,interrupt\,occurs} \\ or\,in\,terms\,of\,clock\,cycle &= \frac{Number\,of\,clock\,cycles\,required\,to\,serve\,this\,interrupt}{rate\,at\,which\,GPIO\,interrupt\,occurs} \\ &= \frac{161558}{288} \\ &= 560 \end{aligned}$$

so these many times an GPIO interrupt is being served while being in the timer interrupt.

3.7.2.2 Justification of time required to serve an Timer Interrupt

Time required to send the Histogram Data to SD-CARD is 0.8 ms.

$$\begin{aligned} Time\,reqiuired\,to\,serve\,an\,GPIO\,interrupt\,560\,times &= \frac{560 \times 177}{80\,\mu s} \\ &= 1.24\,ms. \end{aligned}$$

Time requires to reset the Histogram data will be in some μs so the major part of the time will be sum of SD-CARD time and the GPIO Interrupt Time

$$\begin{aligned} SD - CARD\,time + GPIO\,Interrupt\,time &= 0.8 + 1.24 \\ &= 2.04\,ms \end{aligned}$$

which is approximately equal to 2.02 ms.

Chapter 4

Communication Interface

In order to communicate between two devices microcontrollers offers you with a variety of protocols like

- I2C
- SPI

each of them are suitable for different applications with some advantages and disadvantages.

4.1 I2C

Developed by Phillips Semiconductors in 1980's for providing easy way of communicating between IC's. The name itself says it is an Inter-Integrated Circuit. It is a simple bi-directional 2-wire bus.

4.1.1 I2C Features

- Only two bus lines are required namely
  - Serial Data Line (SDA)
  - Serial Clock Line (SCL)

- Each device connected to the bus is software addressable with a unique address.
- Simple Master and Slave relation, master can act as Master-Transmit or as a Master-Receive.
- True Multi-Master bus including collision detection and arbitration to prevent data corruption.
- Serial 8-bit oriented bidirectional data transfers can be done at different rates like
  - 100 Kbits/s in Standard-mode
  - 400 Kbits/s in Fast-mode
  - 1M bits/s in Fast-mode Plus
  - 3.4 Mbits/sec in High-Speed mode
- Serial, 8-bit oriented, unidirectional data transfers up to 5 Mbit/s in Ultra Fast-mode.
- The number of IC's connected to bus is limited by Maximum Bus Capacitance 400 pF.[2]

4.1.2 I2C Communication

- Simple procedure to have communication between two IC's.
- Each IC connected to bus has it's own ID called address of the IC.
- Master IC starts the communication by sending or initiating the clock.

These are the steps in which I2C communication happens,

- When SDA and SCL are both high. The bus is 'free'.
- Master IC puts the message on the bus saying I have STARTED to use the bus. All other ICs then LISTEN to the bus data and checks their own address in order to know that master wants to talk to them or with someone else.

- Provide on the CLOCK (SCL) wire a clock signal. It will be used by all the ICs as the reference time at which each bit of DATA on the data (SDA) wire will be correct (valid) and can be used. The data on the data wire (SDA) must be valid at the time the clock wire (SCL) switches from 'low' to 'high' voltage.

- Master sends out a unique binary 'address' (name) of the IC that it wants to communicate with in serial form.

- Master keeps a message (one bit) on the bus telling whether it wants to SEND or RECEIVE data from the other chip.

- Master waits for the other IC to ACKNOWLEDGE (using one bit) that it recognized its address and is ready to communicate.

- If the other IC acknowledges all is OK, data can be transferred else communication will be stopped or master resends the address again.

- Every time master sends an 8-bit words of data and after every 8-bit data word the master IC expects the receiving IC to acknowledge the transfer is going OK.

- When all the data is finished the master chip must free up the bus and it does that by a special message called 'STOP'.

Every transaction on the I2C bus is nine bit long, having a 8-bit data and one bit acknowledge bit. Each byte has to have its acknowledge bit, and data must be transferred MSB first. Below figure shows how an I2C bus look

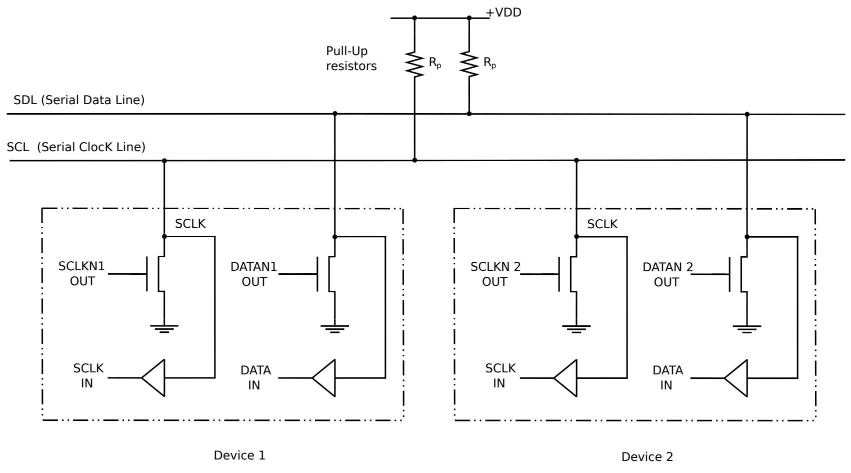


Figure 4.1: I2C BUS.[2]

4.1.2.1 VALID DATA

During transmission of data or the address on the SDA line data should be stable when the SCL have a HIGH value. The change of data line from HIGH to LOW or LOW to HIGH can happen only when the clock signal is LOW.Below figure shows the validity of data

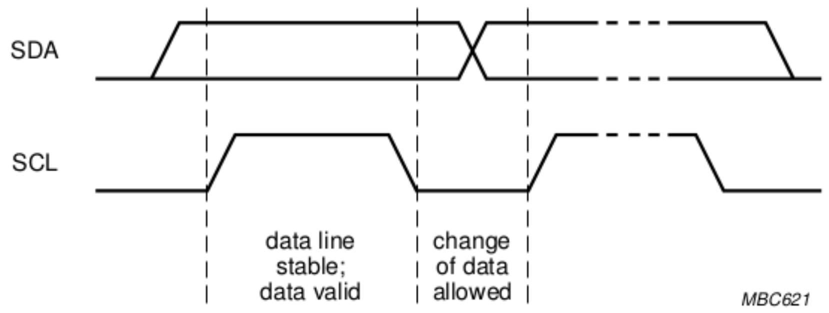


Figure 4.2: Bit Transfer on I2C Bus.[2]

4.1.2.2 START and STOP conditions

To know when the communication to begin and when to end the I2C bus protocol has START and STOP special conditions.

- **Start condition**

  - A change in data value from HIGH to LOW while SCL is HIGH is indicated as START condition.

- **Stop Condition**
  - A change in data value from LOW to HIGH while SCL is HIGH is indicated as STOP condition.

below figure shows how the STOP and START conditions are generated,

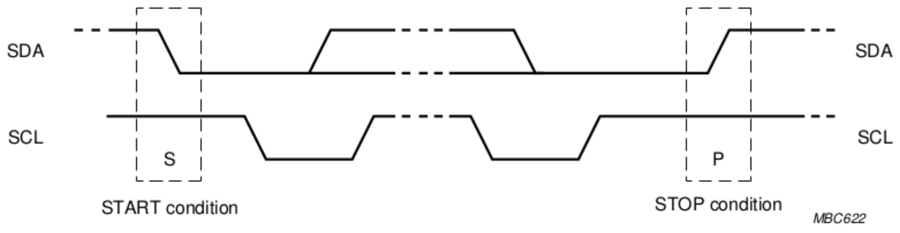


Figure 4.3: Start and Stop conditions.[2]

.

Note:-If data changes when SCL is HIGH it represents START or STOP conditions. Between START and STOP data should not change when SCL is HIGH otherwise it considered as a unnecessary START or STOP condition.

START and STOP conditions are always generated by master. After START condition is generated bus is considered as busy and after STOP condition is generated bus is considered as free. Master can send repeated START conditions for continuous transfer of data.

4.1.2.3 DATA TRANSFER

Every time you put the data on the SDA line it should be 8-bit long. There is no restriction of how many such bytes you put. Every byte should be followed by an acknowledgement bit. While transferring data MSB bit should be transfered first. Slave can hold the clock line LOW and keep the master in to wait state until it receives the data.

4.1.2.4 Acknowledge

After every byte master generates an acknowledge clock cycle. During acknowledge the Transmitter releases the SDA line and receiver must pull down the SDA line during acknowledge clock cycle. The acknowledge bit has to satisfy the data validity condition.

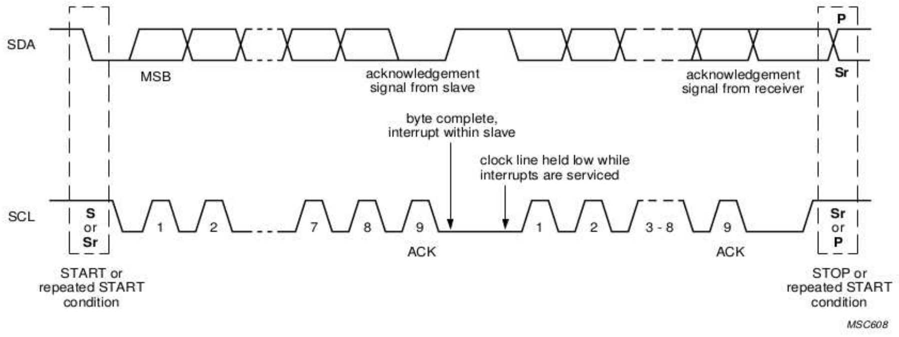

SDA
SCL
MSB
acknowledgement signal from slave
acknowledgement signal from receiver
byte complete, interrupt within slave
clock line held low while interrupts are serviced
S or Sr
1
2
7
8
9
ACK
1
2
3 - 8
9
ACK
Sr or P
P
Sr
START or repeated START condition
STOP or repeated START condition
MSC608

Figure 4.4: Data Transfer on the I2C Bus.[2]

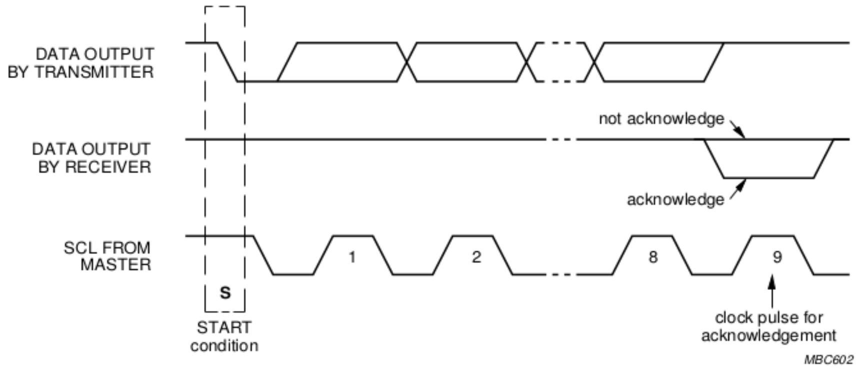


Figure 4.5: Acknowledge on the I2C Bus.[2]

4.1.2.5 Data Transfers using 7-bit Addresses

After generation of START condition Slave addresses of length 7-bit is transferred fallowed by an data direction bit R/S bit.

If R/S =0, it says master will send data i.e master transmit mode and

if R/S =1, it says master will receive data i.e master receive mode.

Data transfer is always terminated by a STOP condition which is generated by master.

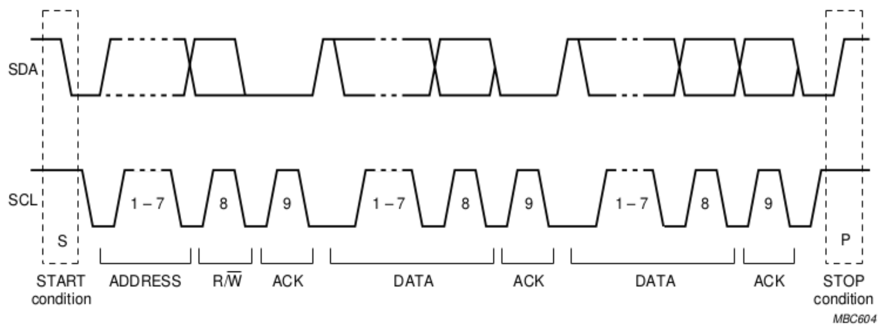


Figure 4.6: Complete Data Transfer.[2]

4.1.3 I2C for HEPD

In order to transfer the data from one microcontroller to other we need to have some kind of protocol to communicate between two devices. So in order to send the health data from the HEPD to the OBC we have chosen the I2C communication. Since the health data is not much and is served for every few seconds I2C is the best approach. And another reason of choosing the I2C is I2C is the Main Bus which communicate with each and every other microcontroller. For the main bus OBC is the master and every other device is slave.

4.1.3.1 Experiments Performed

We have chosen

OBC microcontroller which is MSP430 as the Master &

HEPD microcontroller which is Stellaris LM3S9B92 as the Slave.

HEPD as a SLAVE

HEPD is a Stellaris microcontroller which has two I2C modules I2C0 and I2C1. We have used I2C1 to communicate with master which is MSP430.

In this setup master will send 100 bytes of data to he slave at a speed of 100KHz. The correctness of the communication is checked using the UART connection which is connected to slave. UART setup is done in slave device such that for every time you send data to slave it will display the same on the monitor.

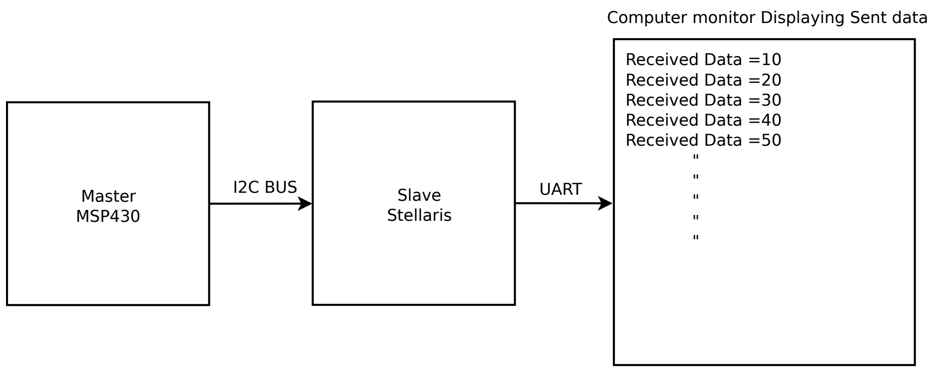


Figure 4.7: I2C Setup

.

Results and Observations

$$Clock Speed = 100 KHz$$

$$No of Bytes to be transferred = 100 Bytes.$$

In I2C communication for every byte you transfer you will have to send an acknowledge bit.

So totally we have 100 bytes + one bit for every byte you transfer.

$$\begin{aligned} So number of bits transferred &= 900 bits \\ so the Transfer rate &= \frac{900}{100} \\ &= 9 ms. \end{aligned}$$

I2C communication between MSP430 and Stellaris is performed and it is successfully done. No errors were found in data transfer.

4.2 SPI

Serial Peripheral interface is developed by Motorola and is also called four-wire serial bus. Used for communicating the microprocessor/microcontroller to the peripheral devices.

Operates in full Duplex mode and communication happens in master/slave mode where the master device initiates the data frame. SPI is also called as Synchronous Serial Interface (SSI).
Stellaris microcontroller has two SSI modules with following features:

- Programmable interface operation .
- Master or Slave Operation.
- Programmable bit rate.
- Two separate FIFO's one for Transmit and one for receive each 16 bits wide and 8 locations deep.
- Data size is programmable and it can vary from 4 to 16 bits wide.
- Standard FIFO-based interrupts and End-of-Transmission interrupt .
- Efficient way of transferring data using μDMA.[3]

4.2.1 Basic Connection

SPI have 4 signal wires

- **Master Out Slave In** (MOSI)
  - Signal gets generated from master and slave acts as a receiver.
- **Master In Slave Out** (MISO)
  - Signal gets generated from slave and master acts as a receiver.
- **Serial Clock** (SCLK or SCK)
  - Signal is generated by master to synchronize data transfer between master and slave.

- **Slave Select or Chip Select**
  - Signal is generated by master and is used to select the slave device. It is an active low signal.

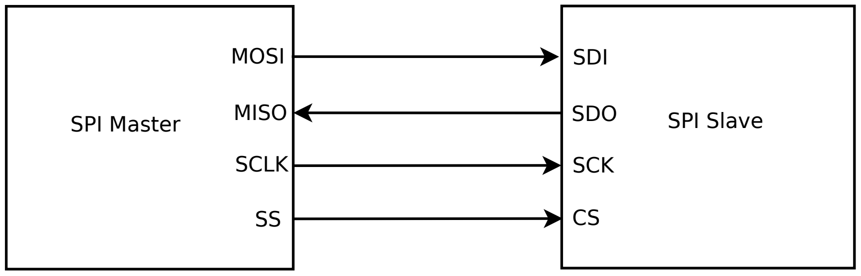


Figure 4.8: SPI basic connection

Among the four wires two wires MOSI & MISO are considered to be data line signal and the other two are called control signals.

4.2.2 Single Master and Single Slave

In this type of configuration one device acts as master and the other acts as a slave. Master device controls the data flow by generating the clock signal and selecting the slave.

4.2.2.1 How do they Communicate

Master device first generates the clock. The master then selects the slave for communicating by pulling the chip select signal low. In a multi master configuration all other slaves will be tri-stated and a single slave whose address is matched is connected to the master.

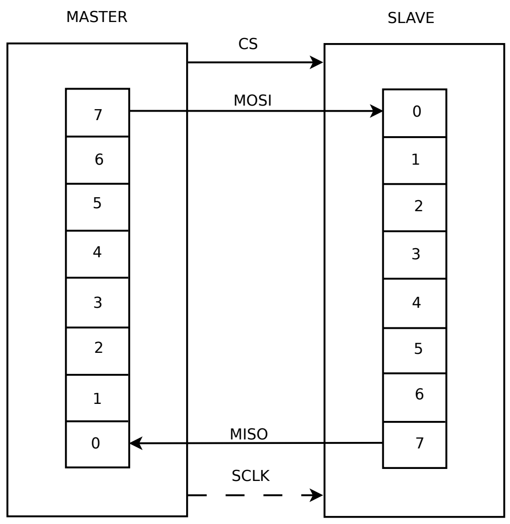


Figure 4.9: Communication using two Shift Registers.[3]

.

This is a simple shift register configuration. Full-duplex data transmission is possible here during each clock cycle. Master sends on bit of data on the MOSI line and slave reads that bit and it transmits one bit of data simultaneously on the MISO line and master reads that data. First MSB bit from the master shifts out and simultaneously in LSB bit it receives the new data. After say 8-bits of data has been transferred then we say a byte has transfered from master to slave and from slave to master. If user wants further communication i.e more data to be transferred the user loads the new data in to the shift register. Data size is not limited to 8-bits.

4.2.3 Other Configurations

If the microcontroller wants to talk to multiple peripherals there are two ways to setup this

- Cascaded slaves or daisy-chained slaves.
- Independent slaves or parallel configuration.

Daisy-chained slave configuration:

In this configuration all the clock lines and chip select are connected together. Data flow out from master microcontroller and through all peripherals connected to the chain and comes to master microcontroller. The data out of first slave device is connected to second slave device and the data out of second slave device is connected to third and this continues and finally the last slave device data out is connected to the master device. Thus forming a wide shift register. Only one chip select is enough in this configuration.

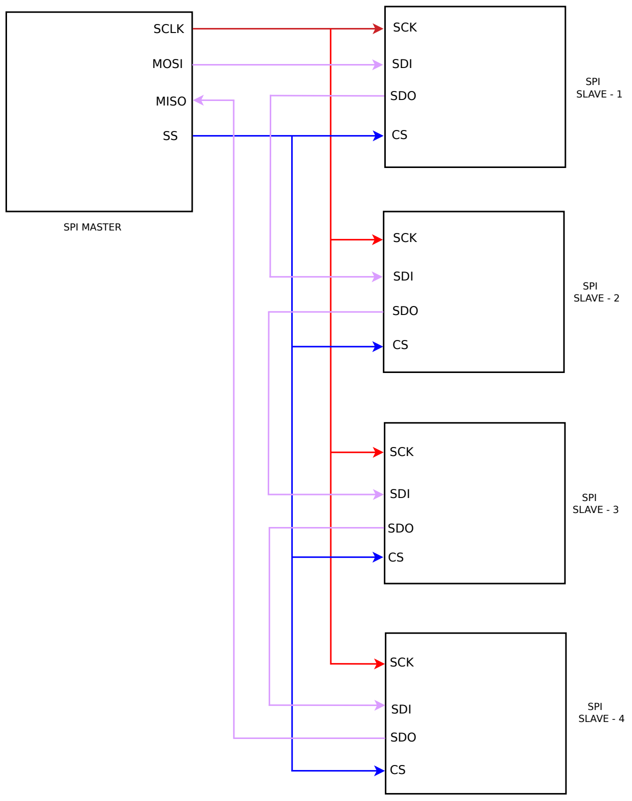


Figure 4.10: Daisy-Chain Configuration.[3]

.

Independent slaves :

All the clock lines (SCLK) are connected together.

- All the MISO data lines are connected together.
- All the MOSI data lines are connected together.
- But the Chip Select (CS) pin from each peripheral must be connected to a separate Slave Select (SS) pin on the master-microcontroller.

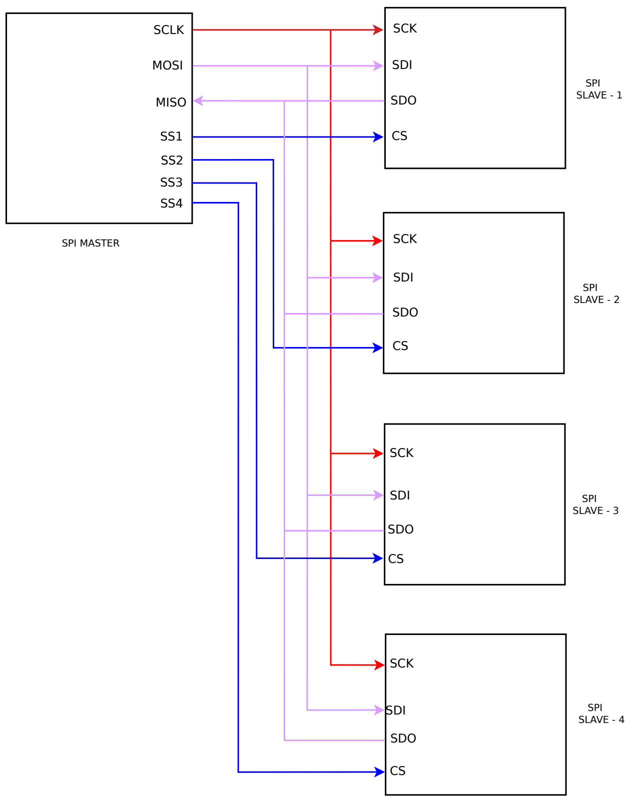


Figure 4.11: Independent Slave Configuration.[3]

.

4.2.4 SPI for HEPD

In order to communicate with external memory (SD-CARD) we have chosen simple one master and one slave SPI interface.

4.2.4.1 SD-Card Pin Configuration

Micro SD-CARD has 8 pins and the following figure gives the details of the micro SD-CARD pin configuration and how are they used.

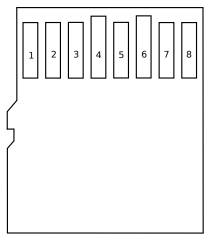


Figure 4.12: Micro SD-Card Pin Diagram

Micro SD-pin	Name	I/O	Description
1	NC	-	Unused
2	nCS	I	SD-Card chip select (active LOW)
3	DI	I	SPI Serial Data in (MOSI)
4	VDD	S	Power
5	CLK	I	SPI Serial Clock (SCLK)
6	VSS	S	Ground
7	DO	O	SPI Serial Data Out (MISO)
8	NC	-	Unused

Table 4.1: Pin Description

4.2.4.2 Interface between Microcontroller and the SD-CARD

Stellaris has port A which can be used for SPI communication below figure shows the interface between stellaris microcontroller and the SD-CARD

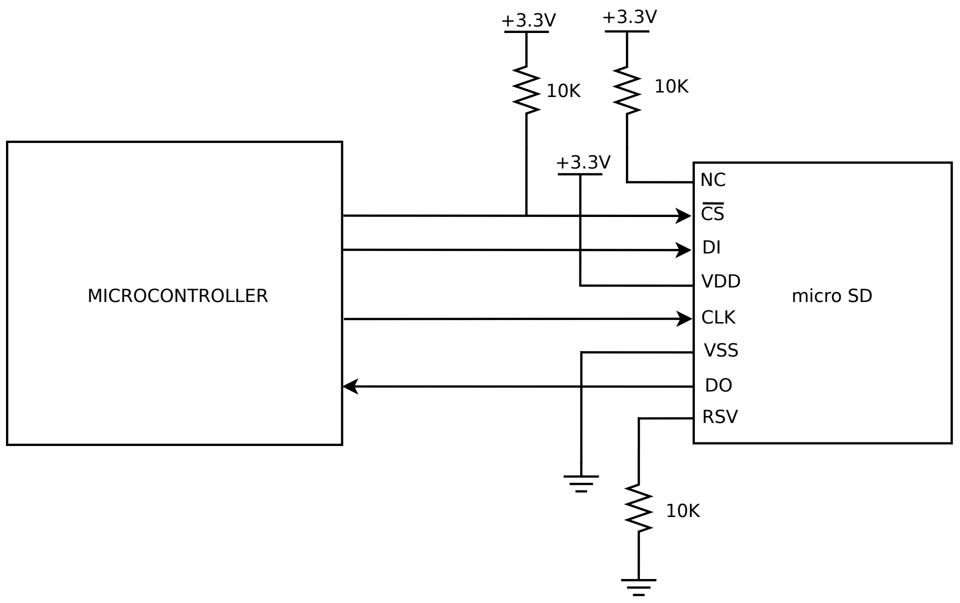


Figure 4.13: Interface between Stellaris and the SD-CARD

.

4.2.4.3 Fat-File System

A file system defines the structure and the rules used to read,write, and maintain information stored on a disk. One of the serious drawback in that when files are deleted and new files written to the media, directory fragments tend to become scattered over the entire media, making reading and writing a slow process. Initially we thought of writing the Histogram data in to SD-CARD using Fat-File System. Since the data rates for writing is too high so we thought of going for RAW data only. Another reason for this is the data was of same type like the HEPD data and the health data so we thought we can maintain it by using some extra bits to get the details like time stamping and the type of data.[5]

Timings in Fat-File System

$$Time\,required\,to\,create\,a\,new\,file = 25ms.$$

$$Time\,required\,to\,write\,50\,Bytes\,of\,data = 20\,ms$$

$$Time\,required\,to\,write\,1\,KByte\,of\,data = 20\,ms$$

$$Time\,required\,to\,write\,54\,MBytes\,of\,data = 180\,ms$$

we ca see the difference when file size increases seek time increases and the speed decreases.

Timing for writing RAW data

Time required to write 50 Bytes of data = 0.8 ms best and the worst case is 3 ms.

4.3 Results and Observations

By interfacing the microcontroller and the SD-CARD and writing some data on to it we found no errors and the SPI communication was successful.

WRITING TIME on SD Card:

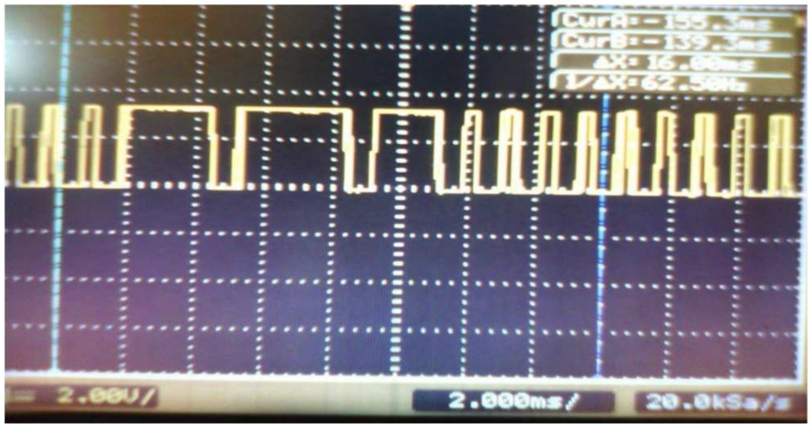


Figure 4.14: Writing to SD-Card

Observations Operating Stellaris at 80 MHz for RAW Data

- Time to write 500 Bytes is 0.8 ms (typical) and 3.5 ms (worst).
- Response time for write command is 3.08 μs (typical).
- Data response from micro SD is 1.43 μs (typical).

READING TIME from SD Card:

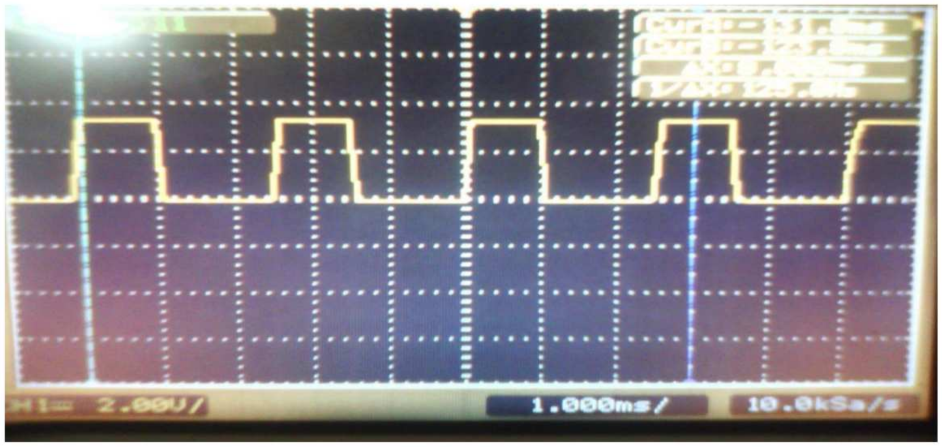


Figure 4.15: Reading From SD-CARD

- Time to Read 500 Bytes = 1 ms (typical).
- Response time for Read command is 44.58 μs (typical).

Chapter 5

FRAM

5.1 Introduction

We thought of going for FRAM because the read and write time for FRAM is very less when compared to SD-CARD.

Ferroelectric memory, FRAM (Ferroelectric RAM) is a non-volatile memory offers

- high-speed writing.
- low power consumption and
- long rewriting endurance.

It is a nonvolatile but operates in other respects as a RAM. The read and write timings are almost same. The FRAM we have chosen is a product of RAMTRON and the FRAM IC is FM18W08.

This FRAM has

- 15-Adress lines
- 8 data lines and
- 3 control lines.

The three control lines are active low and are

- OE (Output Enable)
- WE (Write Enable
- CE (Chip Enable).

OE is used for reading the data and WE is used for writing data only after CE is active. Below shows the difference between conventional SRAM and the FRAM signaling[4]

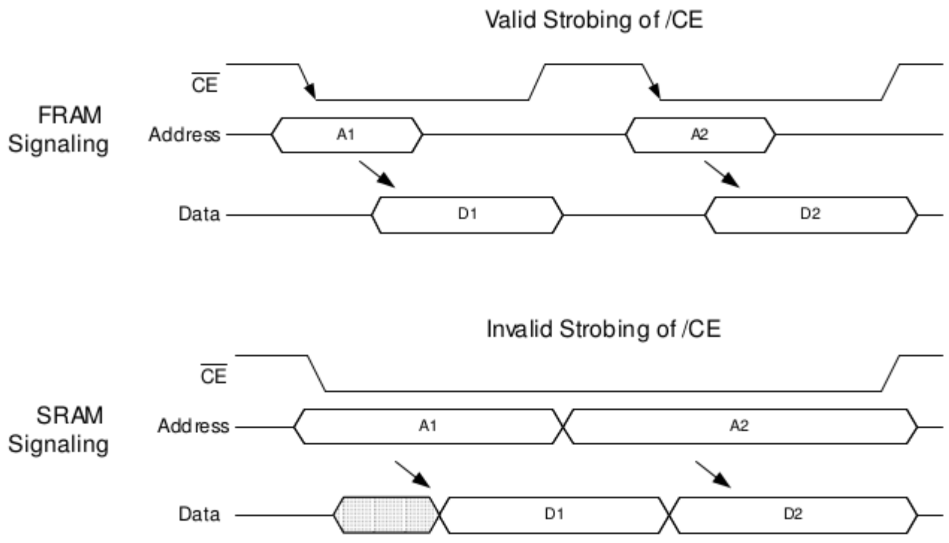


Figure 5.1: Chip enable and memory address relationship.[4]

5.2 Memory Operations

- Read
- Write

5.2.1 Read Operation

- First thing we have to do is to load the address on to the address lines.
- Second we have to make the CE bit active and this makes the address is latched.After the address has been latched, the address value may be changed upon satisfying the

hold time parameter. Unlike an SRAM, changing address values will have no effect on the memory operation after the address is latched.

- Now third step make OE bit active, and the data for the corresponding address you can read now.

5.2.2 Write Operation

- First thing we have to do is to load the data on to the data lines.

- FM18W08 supports both CE active and WE active controlled write cycles. In both cases, the address is latched on the falling edge of CE.

  CE Controlled

  In a CE controlled write, the WE signal is made active prior to beginning the memory cycle. That is, WE is low when CE falls. In this case, the part begins the memory cycle as a write.

  WE Controlled

  In a WE controlled write, the memory cycle begins on the falling edge of CE. The WE signal falls after the falling edge of CE. Therefore, the memory cycle begins as a read.

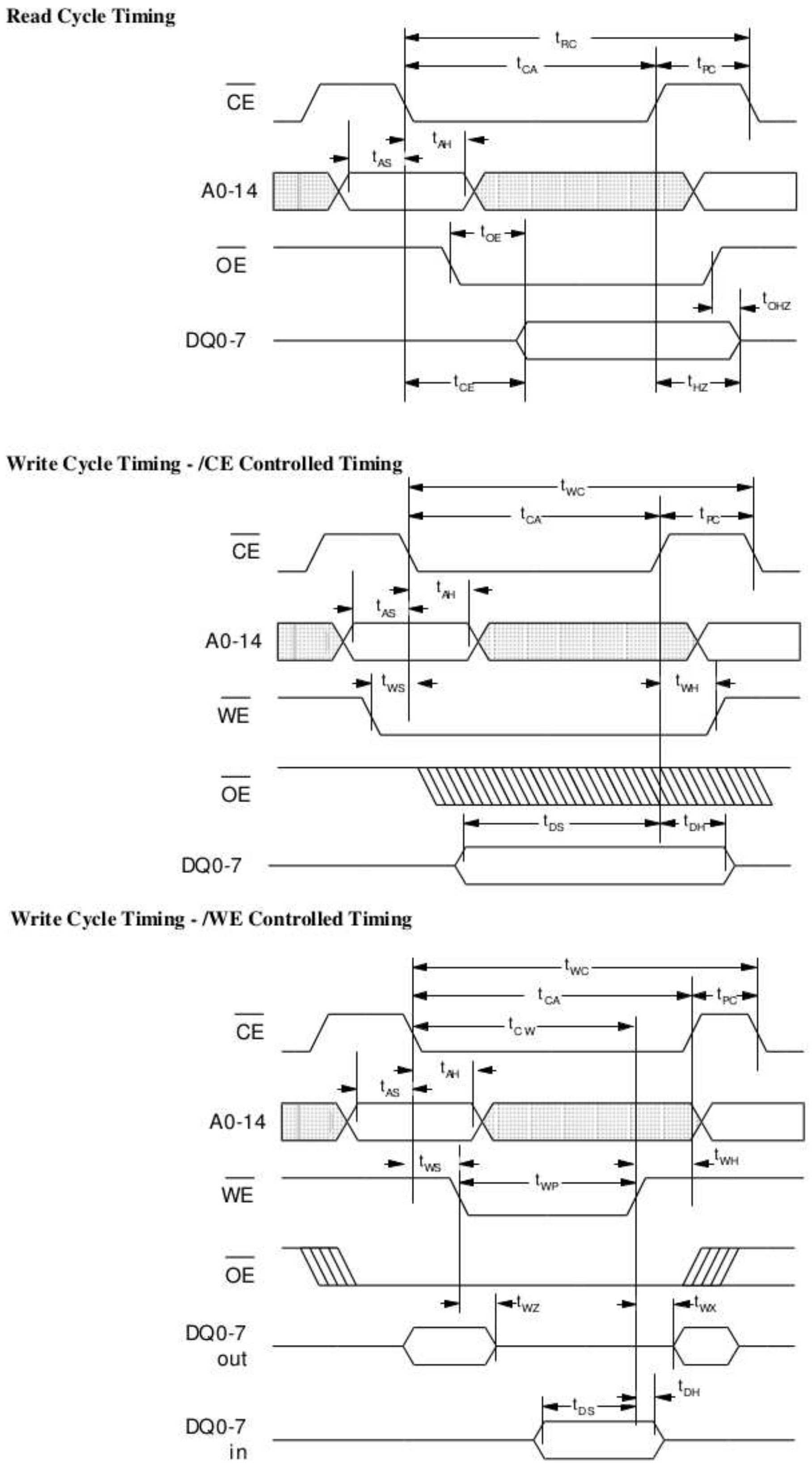


Figure 5.2: Write Cycles.[4]

.

5.3 Interface Between Stellaris and FRAM

FM18W08 is a parallel FRAM. While interfacing we need 15 address lines and 8 data lines and 3 control line to connect the FRAM, below figure shows you how the interface looks like

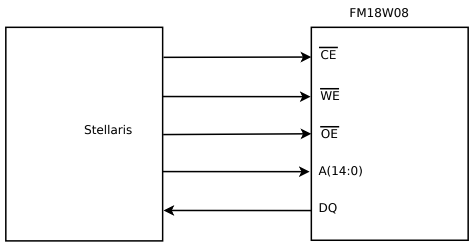


Figure 5.3: Interface between stellaris and FRAM

5.4 Topology Considered

For this setup we thought of new topology and is shown below

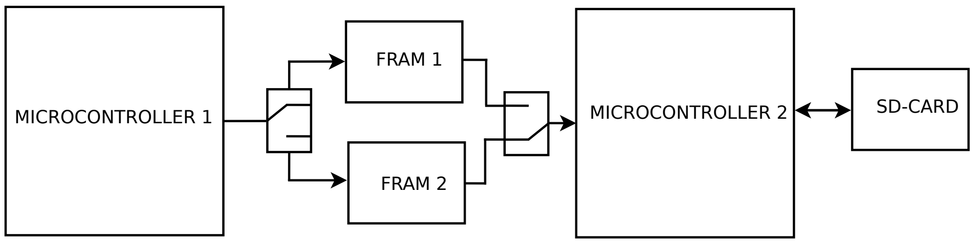


Figure 5.4: FRAM based Topology

.In this setup while microcontroller 1 sends data to FRAM 1, microcontroller 2 will read the data from FRAM 2 and store it in SD-CARD and vice versa.

Problems with this topology

In this topology we need to have buffers for address,data and control lines and that makes it complex.

5.5 Results and Conclusion

The writing speed and th reading speed are as quick as SRAM. Only difference is before writing and reading we need to make CE active. FRAM is not useful for our design because of complexity caused by the buffers and the limited memory size in the markets (max 1 MByte).

Chapter 6

Test Case to Check this Algorithm

In order to check this algorithm we have used two approaches

- FPGA
- MATLAB

6.1 FPGA

In this method we have used SPARTAN-3e FPGA board having 12-bit DAC [6]. In order to test this algorithm we have generated a test case as shown below using FPGA DAC

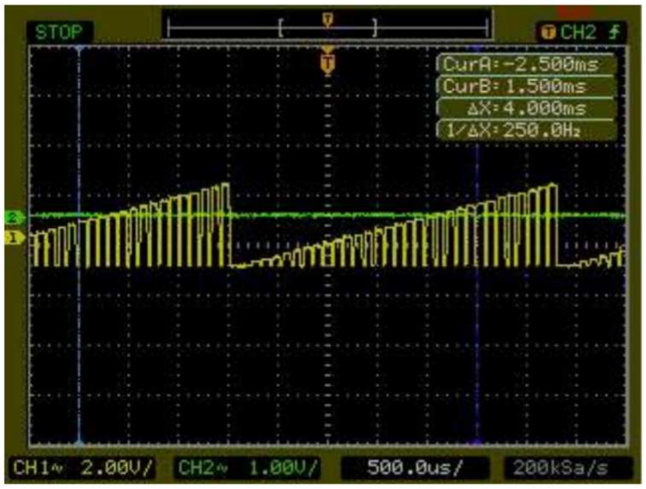


Figure 6.1: FPGA DAC Output

to generate this output we have considered only last MSB 5-bit so the output ranges from 0-31 vlots.Why only last 5-bits because the output voltage resolution of DAC was too close enough to be recognized by the stellaris ADC which has its own resolution.

$$
\begin{aligned}
Resolution of the stellaris ADC &= \frac{Vmax - Vmin}{1024} \\
&= \frac{3}{1024} \\
&= 2.9\,mv
\end{aligned}
$$

So Resolution of this ADC is 2.9 mv that means any signal if it varies below this voltage I cannot predict correct output Digital data.

Now assume that we have an energy conversion table for these DAC output values, that is from 0-31. So we have an energy table i.e 1-D array of 32 energy values, and for simplicity we have loaded these 32 locations form 0-31 as 0-31 only.

e.g-

the zero address value have the energy value 0

one address value have the energy value 1

“

“

31st address value have the energy value 31.

Now with this setup it is very easy to test the correctness of the algorithm. Now if we apply this signal as an input to the ADC of stellaris then it should get the output from 0-31 and it should continue.

Results and Conclusion for this Approach

ADC Output	ADC Output	ADC Output	ADC Output	ADC Output
0	0	0	0	0
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	4	4	4	4
5	5	5	5	5
6	6	6	6	6
7	7	7	7	7
8	8	8	8	8
9	9	9	9	9
10	10	10	10	10
11	11	11	11	11
12	12	12	12	12
13	13	13	13	13
14	14	14	14	14
15	15	15	15	15
17	17	17	17	17
18	18	18	18	18
19	19	19	19	19
20	20	20	20	20
21	21	21	21	21
22	22	22	22	22
23	23	23	23	23
24	24	24	24	24
25	25	25	25	25
26	26	26	26	26
28	28	28	28	28
29	29	29	29	29
30	30	30	30	30
31	31	31	31	31
31	31	31	31	31

Table 6.1: ADC output for this input Signal

from above table we can see the outputs are not continuous instead we should get 0-31 continuously. But this is acceptable in any real system which is having some Non-Linearities in the system.

Problem with this approach was the FPGA DAC output rise time was too high and is in

μs. So we could not able to check it for Higher frequencies.

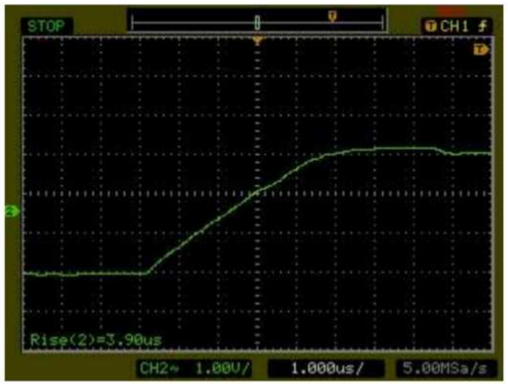


Figure 6.2:

and this experimental setup was made for only 12.08 KHz which is at a very less from the operating frequency of an real PAYLOAD system.

6.2 Matlab

We have mimic this algorithm in matlab. We have run this algorithm in matlab and in the Stellaris microcontroller, and we got the results, and both of them were almost same when driven by a particular frequency of a sinusoidal signal.

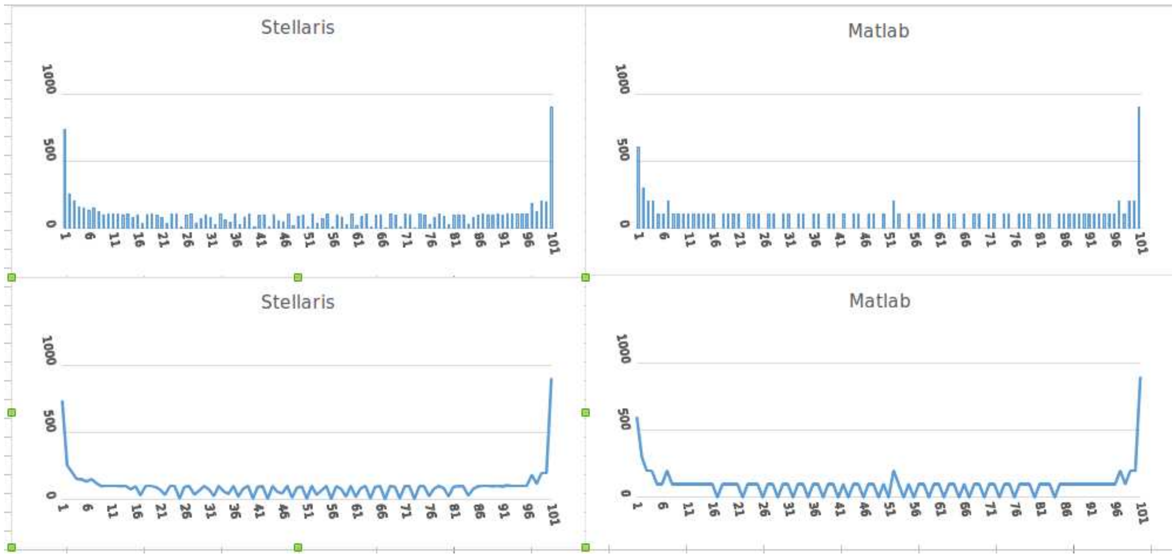


Figure 6.3: Comparison of outputs

Chapter 7

Conclusion and Future Work

7.0.1 Conclusion

Different topologies has been considered for transferring of data between subsystems and an efficient topology has been chosen in terms of data transfers and the hardware. Functionalities such as sampling the input data, generating the Histogram and storing the histogram data in the external memory has been implemented successfully. I2C interface between Stellaris and MSP430 has been done.

7.0.2 Future Work

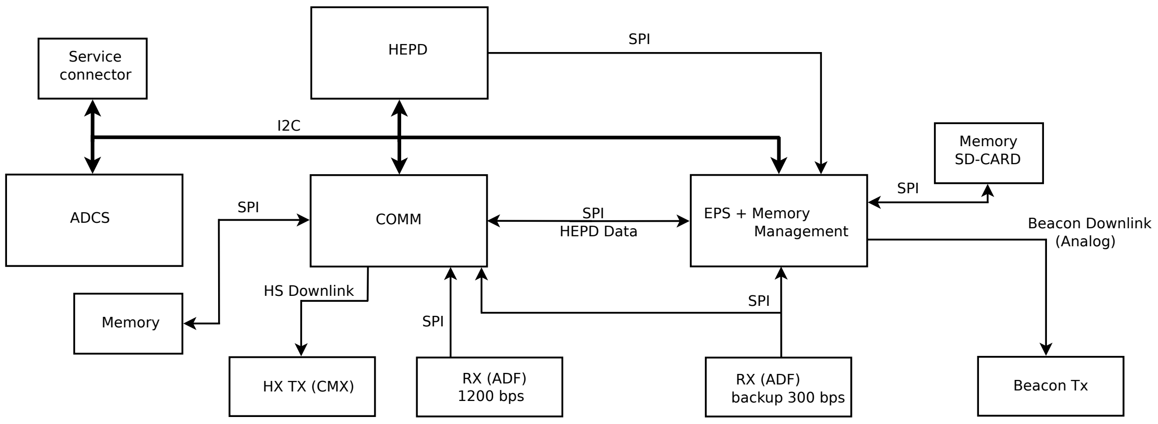


Figure 7.1: New Topology

This topology is has to be tested on a PCB. SPI communication between HEPD and MSP430 has to be done. Integration and interfacing of different sensors like current, voltage and temperature has to be done.

Appendix A

ADC

These are some of the API functions used for ADC in order to configure it.

// Enable the clock to the ADC module

SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC);

//Configure the ADC to sample at 500KSps

SysCtlADCSpeedSet(SYSCTL_SET0_ADCSPEED_500KSPS);

// Disable sample sequences

ADCSequenceDisable(unsigned long **ulBase**, unsigned long **ulSequenceNum**)

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

// Configure sample sequence

ADCSequenceConfigure(unsigned long **ulBase**,
unsigned long **ulSequenceNum**,
unsigned long **ulTrigger**,
unsigned long **ulPriority**);

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

ulTrigger is the trigger source that initiates the sample

sequence; must be one of the ADC_TRIGGER_∗ values.

ulPriority is the relative priority of the sample sequence

with respect to the other sample sequences.

```
// Configure sample sequence
ADCSequenceStepConfigure (unsigned long ulBase,
                          unsigned long ulSequenceNum,
                          unsigned long ulStep ,
                          unsigned long ulConfig) ;
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

ulStep is the step to be configured.

ulConfig is the configuration of this step; must be a logical OR of ADC_CTL_TS, ADC_CTL_IE, ADC_CTL_END, ADC_CTL_D, one of the input channel selects (ADC_CTL_CH0 through ADC_CTL_CH23), and one of the digital comparator selects (ADC_CTL_CMP0 through ADC_CTL_CMP7).

```
// Enable the interrupt
ADCIntEnable(unsigned long ulBase,
             unsigned long ulSequenceNum);
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

```
// Retrieve data from sample sequence
ADCSequenceDataGet(unsigned long ulBase,
                   unsigned long ulSequenceNum,
                   unsigned long *pulBuffer);
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

pulBuffer is the address where the data is stored.

Appendix B

μDMA

These are some of the API functions used for μ**DMA** in order to configure it.

// Enable the uDMA controller peripheral.

SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);

//Enable the uDMA controller.

uDMAEnable();

//Set the base address for the channel control table.

uDMAControlBaseSet(void ***pControlTable**) ;

Parameters:

pControlTable is a pointer to the 1024-byte-aligned base address of the uDMA channel con- trol table.

//Enable attributes of a uDMA channel.

uDMAChannelAttributeEnable(unsigned long **ulChannelNum**, unsigned long **ulAttr**);

Parameters:

ulChannelNum is the channel to configure.

ulAttr is a combination of attributes for the channel.

The **ulAttr** parameter is the logical OR of any of the following:

UDMA_ATTR_USEBURST is used to restrict transfers to use only burst mode.

UDMA_ATTR_ALTSELECT is used to select the alternate control structure for this channel (it is very unlikely that this flag should be used).

UDMA_ATTR_HIGH_PRIORITY is used to set this channel to high priority.

UDMA_ATTR_REQMASK is used to mask the hardware request signal from the peripheral for this channel.

// Configure control parameters for DMA channel

uDMAChannelControlSet(unsigned long **ulChannelStructIndex**,
unsigned long **ulControl**);

Parameters:

ulChannelStructIndex is the logical OR of the uDMA channel number with UDMA_PRI_SELECT or UDMA_ALT_SELECT.

ulControl is logical OR of several control values to set the control parameters for the channel.

// Set the transfer parameters for a uDMA channel control structure

uDMAChannelTransferSet(unsigned long **ulChannelStructIndex**,
unsigned long **ulMode**,
void ***pvSrcAddr**,
void ***pvDstAddr**,
unsigned long **ulTransferSize**);

Parameters:

ulChannelStructIndex is the logical OR of the uDMA channel number with UDMA_PRI_SELECT or UDMA_ALT_SELECT.

ulMode is the type of uDMA transfer.

pvSrcAddr is the source address for the transfer.

pvDstAddr is the destination address for the transfer.

ulTransferSize is the number of data items to transfer.

// Enable a uDMA channel for operation.

uDMAChannelEnable(unsigned long **ulChannelNum**);

Parameters:

ulChannelNum is the channel number to enable.

Appendix C

GPIO

These are some of the API functions used for **GPIO** in order to configure it.

// Enable the peripherals .

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

//Configures pin(s) for use as GPIO inputs.

GPIOPinTypeGPIOInput(unsigned long **ulPort**,
unsigned char **ucPins**);

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

// Set the interrupt type for the specified pin(s).

GPIOIntTypeSet(unsigned long **ulPort**,
unsigned char **ucPins**,
unsigned long **ulIntType**);

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

ulIntType specifies the type of interrupt trigger mechanism.

ulIntType can be one of the following values:

GPIO_FALLING_EDGE

GPIO_RISING_EDGE

GPIO_BOTH_EDGES

GPIO_LOW_LEVEL

GPIO_HIGH_LEVEL.

// Enable interrupts for the specified pin(s).

GPIOPinIntEnable(unsigned long **ulPort**, unsigned char **ucPins**);

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

// Enable the Port interrupts

IntEnable(INT_GPIOB);

Appendix D

Timer

These are some of the API functions used for **Timer** in order to configure it.

// Enable the peripherals used by this example.

SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);

// Configure the timer(s).

TimerConfigure(unsigned long **ulBase**, unsigned long **ulConfig**);

Parameters:

ulBase is the base address of the timer module.

ulConfig is the configuration for the timer.

// Set the timer load value.

TimerLoadSet(unsigned long **ulBase**, unsigned long **ulTimer**, unsigned long **ulValue**)

;

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s) to adjust; must be one of

TIMER_A, TIMER_B, or TIMER_BOTH. Only TIMER_A should

be used when the timer is configured for full-width operation.

ulValue is the load value.

// Enable individual timer interrupt sources.

TimerIntEnable(unsigned long **ulBase**, unsigned long **ulIntFlags**)

ulBase is the base address of the timer module.

ulIntFlags is the bit mask of the interrupt sources to be enabled.

ulIntFlags parameter must be the logical OR of any combination of the following:

TIMER_CAPB_EVENT - Capture B event interrupt

TIMER_CAPB_MATCH - Capture B match interrupt

TIMER_TIMB_TIMEOUT - Timer B timeout interrupt

TIMER_RTC_MATCH - RTC interrupt mask

TIMER_CAPA_EVENT - Capture A event interrupt

TIMER_CAPA_MATCH - Capture A match interrupt

TIMER_TIMA_TIMEOUT - Timer A timeout interrupt.

Appendix E

I2C communication

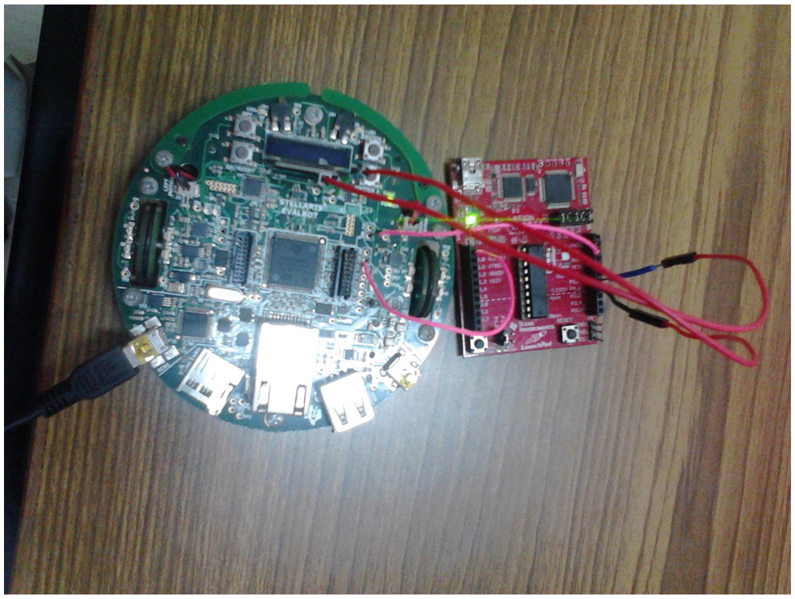


Figure E.1: I2C Communication

The above figure configuration is made to check the I2C communication between Stellaris and the MSP430.

Appendix F

Experimental Setup for Algorithm

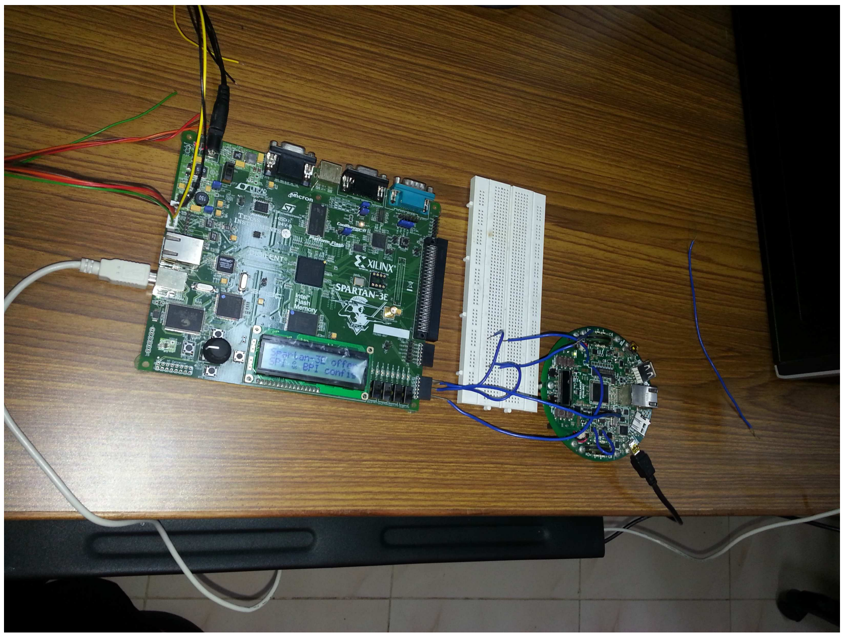


Figure F.1: FPGA DAC to Stellaris ADC

The above setup is made in order to check the correctness of the Algorithm. In this setup the output FPGA DAC is connected to the input of stellaris ADC.

Bibliography

[1] C.-P. I. f. I. Principal Investigator, Co-Principal Investigator from IITM, “High Energy Particle Detector, Project Research Proposal to ISRO-IITM Space Technology Cell,” June 2012.

[2] NXP Semiconductors,The I2C Bus Specification version 2.1 January 2000.

[3] EE Herald, http://www.eeherald.com/section/design-guide/esmod12.html, n.d.

[4] Ramtron, FM18W08ds (Rev. 2.0) January 2012.

[5] Elm-Chan, http://elm-chan.org/fsw/ff/00index_e.html, n.d.

[6] Spartan-3E FPGA Starter Kit ,www.xilinx.com. UG230 (v1.2) January 20, 2011.

[7] Texas Instrument, Stellaris LM3S9B92 Microcontroller Data Sheet (Rev. N) 23 Oct 2012.

[8] Texas Instrument, Stellaris® LM3S9B92 EVALBOT Robotic Evaluation Board user manual 2 March 19, 2011.

[9] Texas Instrument, http://e2e.ti.com, n.d.

[10] IITM Student Satellite,http://iitmsat.iitm.ac.in, n.d.

[11] Luminary Micro, http://chess.eecs.berkeley.edu/eecs149/sp09/docs/an01247.pdf, October 28, 2008.