Project Report

Assignment No - 3

Embedded Systems and Designs

Name: Asit Gautam , Aryan Gupta

Roll Number: 19UEC112 , 19UEC109

Instructor: Dr. Deepak Nair

Department of Electronics and Communication Engineering

The LNMIIT Jaipur, Rajasthan

April 28, 2022

Declaration

This report has been prepared based on my work. Where other published and unpublished source materials have been used, these have been acknowledged.

Student Name: **Asit Gautam , Aryan Gupta**
Roll Number: **19UEC112 , 19UEC109**
Date of Submission: 28/04/2022

Table of Contents

Abstract

TM4C123GH6PM microcontroller is designed around an ARM Cortex-M processor core. TI's **TM4C123GH6PM** is a 32-bit Arm Cortex-M4F based MCU with 80 -MHz, 256 -KB Flash, 32 -KB RAM, 2 CAN, RTC, USB, 64-Pin LQF. We will use IAR embedded ARM workbench IDE for simulation and then we will debug our code in microcomtroller. Here tiva board as an potential measurer is implemented.

Component Name

Texas Instrument's Tiva TM4C123GH6PM microcontroller, batteries or one battery(with potentiometer),jumper wires, breadboard ……..……

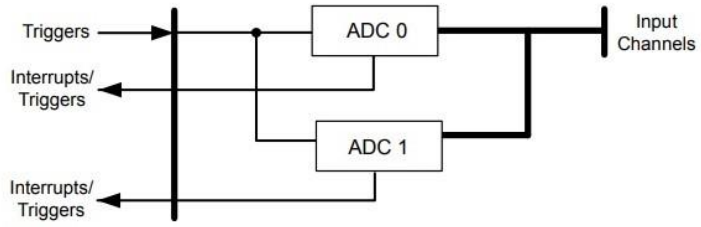
Chapter 1: Introduction

This project is on measuring battery voltage using TM4C123GH6PM.

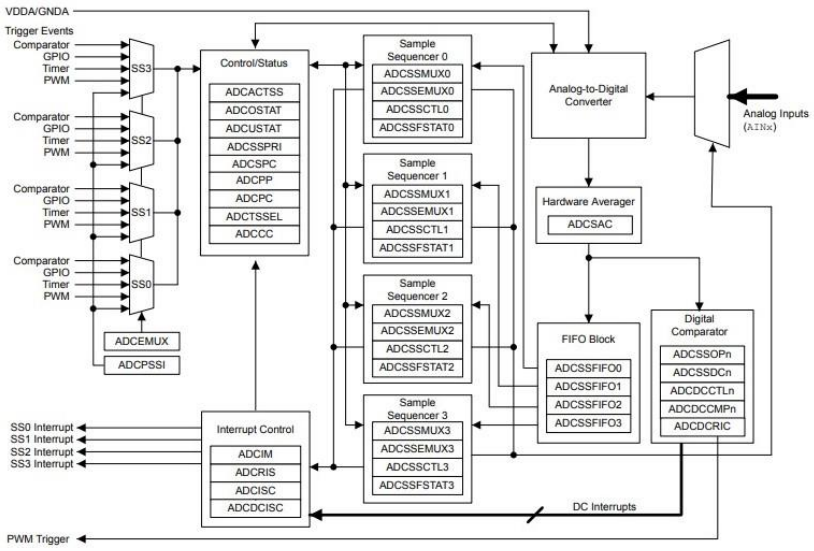
We will use ADC feature as battery discharges with time so basically the output of battery is a large collection of nearly constant voltage(analog value) measured in small intervals of time.

As specified in datasheet an analog-to-digital converter (ADC) is a peripheral that converts a continuous analog voltage to a discrete digital number (binary number). Two identical converter modules are included, which share 12 input channels. The TM4C123GH6PM ADC module has 12-bit conversion resolution and 12 input channels, plus an internal temperature sensor. Each ADC module contains four programmable sequencers that allows the sampling of multiple analog input sources without controller intervention. Each sample sequencer provides flexible programming with fully configurable input source, trigger events, interrupt generation, and sequencer priority. In addition, the conversion value can optionally be diverted to a digital comparator module.

The TM4C123GH6PM microcontroller contains two identical ADC modules. Both modules, ADC0 and ADC1, share 12 analog input channels. Both ADC modules operates independently and can therefore execute different sample sequences, sample any of the analog input channels at any time, and generate different interrupts and triggers.



The two modules connected to analog inputs and the system bus.



ADC Module block diagram

This diagram shows that we can have 5 types of interrupts, it also shows the flow of different registers that we would be needing.

Trigger control –>> Controller (software) – Timers – Analog Comparators – PWM – GPIO

1.1 Sample sequencers

Sample sequencers handles the sampling control and data capture. All of the sequencers are identical in implementation except for the no. of samples that can be captured and the depth of the FIFO. Table below shows the max. number of samples that each sequencer can capture and its corresponding FIFO depth.

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

FIFO basically means data storing unit which varies for the four sequencers.

1.2 Interrupts

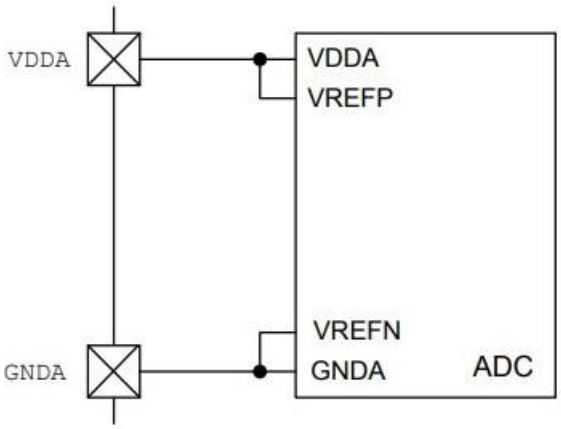
The register configurations of sample sequencers and digital comparators dictate which events generate raw interrupts, but don’t have control over whether the interrupt is actually sent to the interrupt controller. The ADC module's interrupt signals are controlled by the state of the MASK bits in the ADC Interrupt Mask (ADCIM) register. Interrupt status can be viewed at two locations: the ADC Raw Interrupt Status (ADCRIS) register, which shows the raw status of the various interrupt signals; and the ADC Interrupt Status and Clear (ADCISC) register, which

shows active interrupts that are enabled by the ADCIM register. Sequencer interrupts are cleared by writing a 1 to the corresponding IN bit in ADCISC. Digital comparator interrupts are cleared by writing a 1 to the ADC Digital Comparator Interrupt Status and Clear (ADCDCISC) register**.**

***We are using GPIO triggering in this for interrupt.

1.3 Voltage Reference

The ADC uses internal signals VREFP and VREFN as references to produce a conversion value from the selected analog input. VREFP is connected to VDDA and VREFN is connected to GNDA, as shown in figure.



ADC Voltage Reference

mV per ADC code = (VREFP - VREFN) / 4096

It is not necessary that VREFN should be ground, one can use differential sampling also in which VREFN will be no zero.

There are other features in ADC but these were important ones, others can be seen in datasheet.

Chapter 2: CODE

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

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

#include "TM4C123GH6PM.h"

#include "stdint.h"

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

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

volatile static uint32_t adcResult = 0;


//Function for interrupt from ADC

void ADC1SS3_Handler (void)

{

  adcResult = ADC1 ->SSFIFO3;  //read adc coversion result from SS3 FIFO

  ADC1->ISC = (1<<3);

}

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

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




int main()

{

  //Initialization sequence for the ADC is as follows:


  //1. Enable the ADC clock using the RCGCADC register (see page 352).

  SYSCTL->RCGCADC =(1<<1);

  GPIOF->PUR |= 0x10;        // Enable Pull Up resistor PF4

  //2. Enable the clock to the appropriate GPIO modules via the RCGCGPIO register (see page 340).

  //To find out which GPIO ports to enable, refer to “Signal Description” on page 801.

  SYSCTL->RCGCGPIO = (1<<4)|(1<<5); // Port E will be used for Input, Port F will be used for LED PF3 and
switch PF4
```

```
GPIOE->DIR &= ~(1<<1); //changing port E pin 2 to input  PE1



// Configure the LED Pin of Port F
GPIOF->DIR |= 0x08;     //using switch 1 as input and led green as output
GPIOF->DEN = 0xFF;      //port F as digital



//3. Set the GPIO AFSEL bits for the ADC input pins (see page 671). To determine which GPIOs to
//configure, see Table 23-4 on page 1344.
GPIOE->AFSEL = (1<<1);    //PE1

//4. Configure the AINx signals to be analog inputs by clearing the corresponding DEN bit in the
//GPIO Digital Enable (GPIODEN) register (see page 682).
GPIOE->DEN &= ~(1<<1);

//5. Disable the analog isolation circuit for all ADC input pins that are to be used by writing a 1 to
//the appropriate bits of the GPIOAMSEL register (see page 687) in the associated GPIO block.
GPIOE->AMSEL = (1<<1);

//6. If required by the application, reconfigure the sample sequencer priorities in the ADCSSPRI
//register. The default configuration has Sample Sequencer 0 with the highest priority and Sample
//Sequencer 3 as the lowest priority.

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



/*------------------------------------------------------------------------------*/
/*------------------------------------------------------------------------------*/
//Sample Sequencer configuration should be as follows:
```

```
  //1. Ensure that the sample sequencer is disabled by clearing the corresponding ASENn bit in the
  //ADCACTSS register. Programming of the sample sequencers is allowed without having them
  //enabled. Disabling the sequencer during programming prevents erroneous execution if a trigger
  //event were to occur during the configuration process.
  ADC1->ACTSS &= ~(1<<3);

  //2. Configure the trigger event for the sample sequencer in the ADCEMUX register.
  ADC1->EMUX = (0x4 <<12);  //  OR  ADC1->EMUX=0x4000
  //*****************************************************************************interrup
part*********************************************************************//

  GPIOF->ADCCTL = (1<<4);
  GPIOF->IM = (1<<4);

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

  //3. When using a PWM generator as the trigger source, use the ADC Trigger Source Select
  //(ADCTSSEL) register to specify in which PWM module the generator is located. The default
  //register reset selects PWM module 0 for all generators.
  // Not Needed

  //4. For each sample in the sample sequence, configure the corresponding input source in the
  //ADCSSMUXn register.
  ADC1->SSMUX3 = 2; //1st sample input select( a value of 0x2 indicates the input is AIN2 that is PE1 pg.801)

  //5. For each sample in the sample sequence, configure the sample control bits in the corresponding
  //nibble in the ADCSSCTLn register. When programming the last nibble, ensure that the END bit
  //is set. Failure to set the END bit causes unpredictable behavior.
  ADC1->SSCTL3 = 0x6;  //no TS0 D0, yes IE0 END0
```

```
  //6. If interrupts are to be used, set the corresponding MASK bit in the ADCIM register.
  ADC1->IM = (1<<3);

  //7. Enable the sample sequencer logic by setting the corresponding ASENn bit in the ADCACTSS
  //register.
  ADC1->ACTSS |= (1<<3);
  ADC1->ISC = (1<<3);    //status and clear

  NVIC_EnableIRQ(ADC1SS3_IRQn);

  while (1)
{
    if (adcResult<1861 && adcResult>1620)   //voltage between 1.3v and 1.5v
    {
      GPIOF->DATA |= (1<<3);
    }
    else
    {
      GPIOF->DATA &= ~(1<<3);
    }
  }
}



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

main.c

```c
/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/
#include "TM4C123GH6PM.h"
#include "stdint.h"
/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/
volatile static uint32_t adcResult = 0;

//Function for interrupt from ADC
void ADC1SS3_Handler (void)
{
  adcResult = ADC1 ->SSFIFO3;  //read adc coversion result from SS3 FIFO
  ADC1->ISC = (1<<3);
}
/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/



int main()
{
  //Initialization sequence for the ADC is as follows:

  //1. Enable the ADC clock using the RCGCADC register (see page 352).
  SYSCTL->RCGCADC =(1<<1);
  GPIOF->PUR |= 0x10;        // Enable Pull Up resistor PF4
  //2. Enable the clock to the appropriate GPIO modules via the RCGCGPIO register (see page 340).
  //To find out which GPIO ports to enable, refer to "Signal Description" on page 801.
  SYSCTL->RCGCGPIO = (1<<4)|(1<<5); // Port E will be used for Input, Port F will be used for LED PF3 and switch PF4
  GPIOE->DIR &= ~(1<<1); //changing port E pin 2 to input  PE1


  // Configure the LED Pin of Port F
  GPIOF->DIR |= 0x08;     //using switch 1 as input and led green as output
  GPIOF->DEN = 0xFF;      //port F as digital
```

```
//3. Set the GPIO AFSEL bits for the ADC input pins (see page 671). To determine which GPIOs to
//configure, see Table 23-4 on page 1344.
GPIOE->AFSEL = (1<<1);    //PE1

//4. Configure the AINx signals to be analog inputs by clearing the corresponding DEN bit in the
//GPIO Digital Enable (GPIODEN) register (see page 682).
GPIOE->DEN &= ~(1<<1);

//5. Disable the analog isolation circuit for all ADC input pins that are to be used by writing a 1 to
//the appropriate bits of the GPIOAMSEL register (see page 687) in the associated GPIO block.
GPIOE->AMSEL = (1<<1);

//6. If required by the application, reconfigure the sample sequencer priorities in the ADCSSPRI
//register. The default configuration has Sample Sequencer 0 with the highest priority and Sample
//Sequencer 3 as the lowest priority.

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


/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/
//Sample Sequencer configuration should be as follows:


//1. Ensure that the sample sequencer is disabled by clearing the corresponding ASENn bit in the
//ADCACTSS register. Programming of the sample sequencers is allowed without having them
//enabled. Disabling the sequencer during programming prevents erroneous execution if a trigger
//event were to occur during the configuration process.
ADC1->ACTSS &= ~(1<<3);

//2. Configure the trigger event for the sample sequencer in the ADCEMUX register.
ADC1->EMUX = (0x4 <<12);  //  OR  ADC1->EMUX=0x4000
//*************************************************************************interrup part*********************

GPIOF->ADCCTL = (1<<4);
GPIOF->IM = (1<<4);
```

```
//3. When using a PWM generator as the trigger source, use the ADC Trigger Source Select
//(ADCTSSEL) register to specify in which PWM module the generator is located. The default
//register reset selects PWM module 0 for all generators.
// Not Needed

//4. For each sample in the sample sequence, configure the corresponding input source in the
//ADCSSMUXn register.
ADC1->SSMUX3 = 2; //1st sample input select( a value of 0x2 indicates the input is AIN2 that is PE1 pg.801)

//5. For each sample in the sample sequence, configure the sample control bits in the corresponding
//nibble in the ADCSSCTLn register. When programming the last nibble, ensure that the END bit
//is set. Failure to set the END bit causes unpredictable behavior.
ADC1->SSCTL3 = 0x6;  //no TS0 D0, yes IE0 END0

//6. If interrupts are to be used, set the corresponding MASK bit in the ADCIM register.
ADC1->IM = (1<<3);

//7. Enable the sample sequencer logic by setting the corresponding ASENn bit in the ADCACTSS
//register.
ADC1->ACTSS |= (1<<3);
ADC1->ISC = (1<<3);    //status and clear

NVIC_EnableIRQ(ADC1SS3_IRQn);

while (1)
{
  if (adcResult<1861 && adcResult>1620)   //voltage between 1.3v and 1.5v
  {
    GPIOF->DATA |= (1<<3);
  }
  else
  {
    GPIOF->DATA &= ~(1<<3);
  }
}
}
```

core_cm4.h

```
__STATIC_INLINE void NVIC_EnableIRQ(IRQn_Type IRQn)
{
/*  NVIC->ISER[((uint32_t)(IRQn) >> 5)] = (1 << ((uint32_t)(IRQn) & 0x1F));  enable interrupt */
  NVIC->ISER[(uint32_t)((int32_t)IRQn) >> 5] = (uint32_t)(1 << ((uint32_t)((int32_t)IRQn) & (uint32_t)0x1F)); /* enable interrupt */
}
```

cstartup.h

This handler function will be only used if a function with same name is not in main.c.

```
    ADC1SS3_Handler //51th interrupt

/* ************************************************
************************************************/
};


/* Provided below is the section providing functions to handle
exceptions and interrupts */
/*************************************************
*************************************************/

#pragma call_graph_root = "interrupt"
__weak void NMI_Handler( void ) { while (1) {} }
#pragma call_graph_root = "interrupt"
__weak void HardFault_Handler( void ) { while (1) {} }
#pragma call_graph_root = "interrupt"
__weak void MemManage_Handler( void ) { while (1) {} }
#pragma call_graph_root = "interrupt"
__weak void BusFault_Handler( void ) { while (1) {} }
#pragma call_graph_root = "interrupt"
__weak void UsageFault_Handler( void ) { while (1) {} }
#pragma call_graph_root = "interrupt"
__weak void SVC_Handler( void ) { while (1) {} }
#pragma call_graph_root = "interrupt"
__weak void DebugMon_Handler( void ) { while (1) {} }
#pragma call_graph_root = "interrupt"
__weak void PendSV_Handler( void ) { while (1) {} }
#pragma call_graph_root = "interrupt"
__weak void SysTick_Handler( void ) { while (1) {} }
/* ************************************************
************************************************/

/* 3) Add here the function for the interrupt here*/
/* No need to worry about the correct position w.r.t datasheet
//#pragma call_graph_root = "interrupt"
//__weak void TIMER0A_Handler( void ) { while (1) {} }
#pragma call_graph_root = "interrupt"
__weak void ADC1SS3_Handler( void ) { while (1) {} }
/* ************************************************
```

Chapter 3: Explanation

Now TM4C123GH6PM is a 32 bit microcontroller. We use GPIO port, in which memory mapping is present for connecting peripheral devices. Each GPIO port can be accessed through one of two bus apertures. Firstly we need to understand that each of the six ports present, has there specific base address, with register map which can accessed using offset values. We used GPIODATA, GPIODEN, GPIODIR register to enable and change direction to output of port E and port F. Port E and F are accessed using RCGCGPIO to enable Clock gating. Pointers to the addresses of registers are used to access these registers.

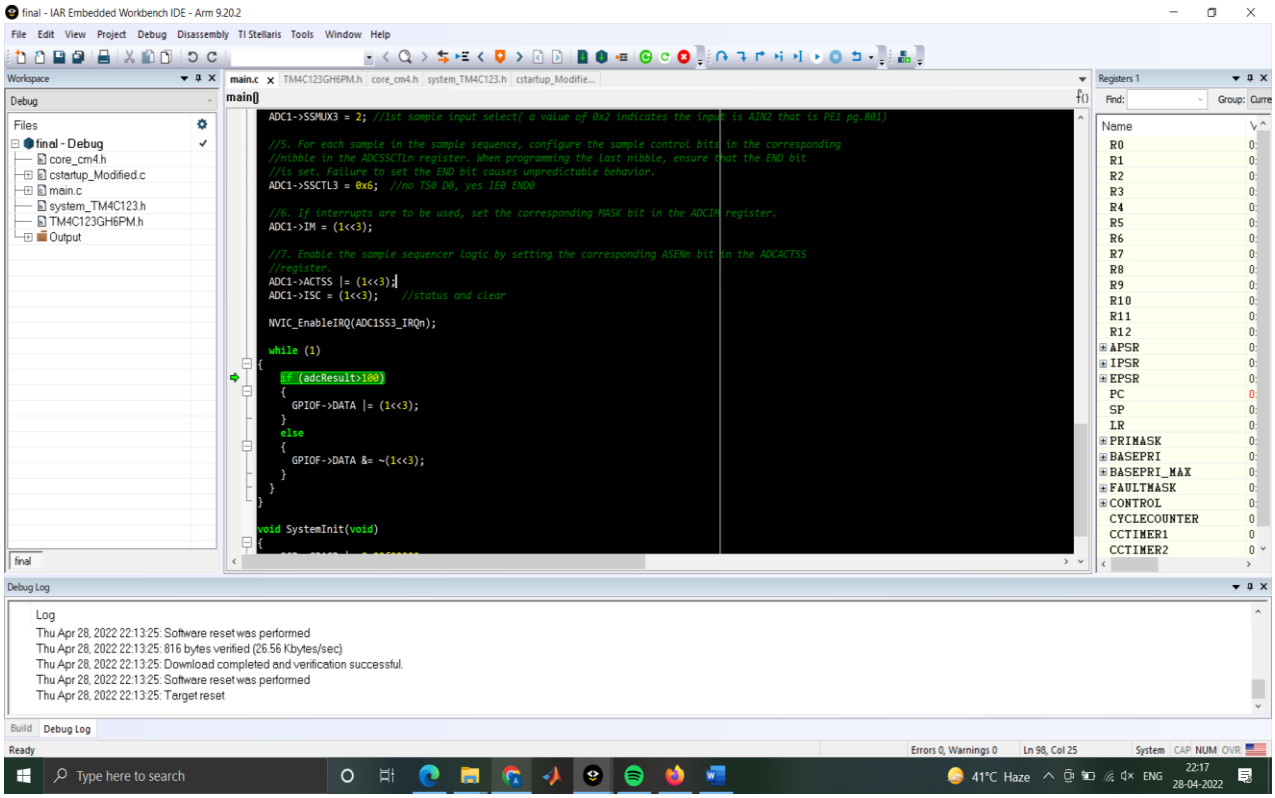
While loop is used to create pattern for non finite time. While debugging we can check the memory, count variable values, we can also check the program counter values in the register dialog box .

Code is quiet self explanatory as the comments are properly mentioned, even the steps are mentioned in code only as comments. So in brief we are using PE1 pin as analog input,PF3 and PF4 for led green and switch, direction, and digital or analog is set .Next when configuring we will use sample sequencer 3,so one sample at a time will be taken, trigger event is set to GPIO, which is set to the switch, so switch will be used to trigger the sampling, one thing to notice is that switch is pulled up. Then there are registers which acts as flag which also need to be taken care of. Triggering is used so that microcontroller doesn't waste its resources by always checking the values coming through input pin. Whenever switch will be pushed a trigger event will be generated which will lead to interrupt handler function where the analog value will be read. To measure the voltage we just need to read this analog input and can see it in our system(pc) which can be done using UART or to keep the program simple we can apply boundary condition as done here, whenever the value ranging from 1.3V(just in case if battery is somewhat discharged) to 1.5V(converted to samples using (x/3.3)*4096) green LED will light up, by this we can check if the battery provided is of 1.5V or not. Limit to measure the voltage is 3.3V,so don't exceed the input value. To get different values or samples to measure one can use potentiometer with the battery source.

Now one important application of this is to draw the characteristic graph of various cells or batteries, that is the relation between the time and discharging voltage. Using UART we can store the voltages value with the time and then this data can be used to generate the graph.

Now one future improvement can be made by making the microcontroller to do some other work like generating pwm or anything and then using this triggering to measure the voltage, so that project will show how TM4C123GH6PM can be used to measure battery voltages as a side job without using much resources.

Chapter 4: Output



Screenshot while debugging the program in tiva board, while debugging adcResult>100 condition was used just to test different conditions.

Bibliography

- https://www.youtube.com/playlist?list=PL3ZmPhf-_xiWxQ8nnbAcRo2CNSHxXxxVW
- https://www.ti.com/lit/gpn/tm4c123gh6pm
- https://www.youtube.com/watch?v=UReClONV0O8&list=PL3ZmPhf-_xiU7BSoYx4vHZ6n115Wc7pCZ