ADC Sensor Interface and Motor Drive Integration Testing

Himanshu Gunjal
Computer Engineering Department, College of Engineering
San Jose State University, San Jose, CA 95192
E-mail: himanshu.gunjal@sjsu.edu

Abstract

This paper describes the design and development of hardware and software components required for testing and verification of ADC conversion of a sensory interface and its integration to drive a stepper motor using Friendly ARM mini 6410 development kit with Samsung S3C6410 processor. Fast Fourier Transform is used to visually analyze the power spectrum and for the validation of the ADC data. The speed of the motor is observed to be varying in proportion to the change in the analog input from the potentiometer and the results are verified from the FFT output.

1. Introduction

In real world, most of the data is characterized by analog signals. For example, in an IIOT device-NH3/NH4 sensor electrode, the output of the solution under test is an analog voltage signal which marks the concentration of ammonia/ammonium. Hence, there exits a need to convert this analog input signal to valid digital pulses which can thus be then utilized to perform various applications.

In this project, we feed voltage generated from a potentiometer to the ARM11 processor which in turn powers a stepper motor driver to drive a motor. The speed of the motor is indirectly controlled from the voltage value generated by the potentiometer. The range of analog voltage is first brought to the full range of 0-3.3V DC for the ARM11 processor using op-amp circuit.

2. Methodology

This section deals with the main objective of this project and the technical challenges that were faced during the implementation. It also describes the abstract design of the project with the help of block diagrams and circuit schematics. In-dept implementation of the hardware and software aspects of the design will be taken up in the next section.

2.1. Objectives and Technical Challenges

The project can be divided into many small stages which can be clubbed together to formulate the final system. The various steps can be depicted as follows:

1. Setup an input circuit consisting of a potentiometer that feeds on a power supply and delivers a range of voltage to the ADC pin of the ARM board as a function of time.
2. A pre-processing circuit consisting of op-amps is used to get a full dynamic range from 0V to 3.3V DC.
3. To keep the output voltage stable, a unit gain buffer op-amp was setup as a load. The Rf and R1 registers of the op-amp are shorted of a non-inverting type op-amp to achieve a unit gain buffer op-amp.
4. The ADC pin then converts this analog voltage to a digital pulse.
5. The software consists of an application program to get inputs from user and driver programs to interface the components. These are discussed in detail in the implementation section.
6. The user application program also performs FFT calculation to validate the data generated by the ADC.
7. PWM is performed using the ARM processor. Duty cycle and direction is set from the application program and this is sent to the motor driver board to rotate the motor.
8. The speed of the motor is observed to change in proportion to the analog value set from the potentiometer.

The technical challenges that were faced were to get the full dynamic range of 0V to 3.3V DC to the ADC pin. Also, setting up the motor driver and integrating the motor was a challenge. Calibration changes were required after looking at the power spectrum and using the threshold value to get valid ADC output.

2.2. Problem Formulation and Design

A potentiometer is powered with a 5V DC supply and ground and is connected to an op-amp circuit. The op-amp circuitry is used to stabilize the output from the potentiometer before passing it into the ADC pin of the ARM11. For this, we use a unit gain buffer op-amp. To achieve unit gain, a non-inverting op-amp was used with Rf and R1 shorted.

PWM was implemented and the user application program was used to pass the duty cycle and direction of rotation of the motor to the stepper motor driver board. The stepper motor thus in turn rotates the motor with those characteristics.

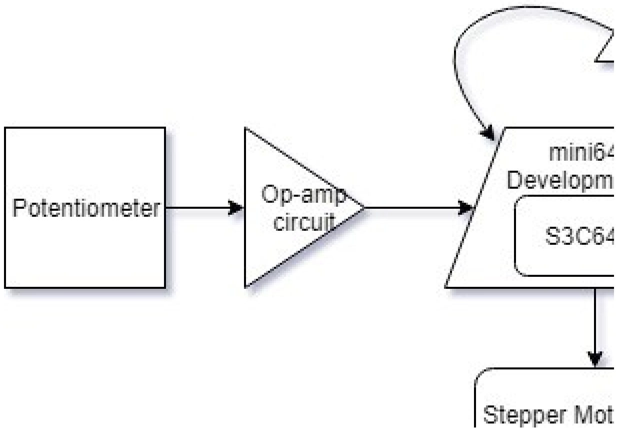

Figure 1: System Block Diagram

3. Implementation

The driver and application program are loaded onto a USB device and connected the development kit. Putty is used to load them onto the kernel of the ARM11. The application program is executed on the S3C6410 processor and the motor is made to rotated as a result of the analog voltage generated by the potentiometer.

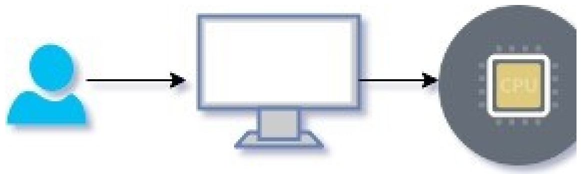

Figure 2. Implementation Flow

The detailed implementation is divided into hardware and software design and discussed in detail further in this section.

3.1. Hardware Design

The hardware design can be divided into 2 main components. The input circuit which contains the sensory interface, the processor board and a motor which is to be controlled by the input analog interface. These components are discussed in detail in this section.

3.1.1 Input Circuit:

A potentiometer is used as an analog voltage generator. The potentiometer is rotated from minimum to maximum to generate a range of voltages as function of time.

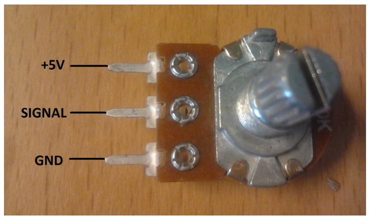

Figure 3: A Potentiometer With Pin Connections

The potentiometer is then connected to an operational amplifier circuit to stabilize the output and bring it to a full range of 0V to 3.3V DC. This is then supplied to the ADC pin of the ARM11 development kit board.

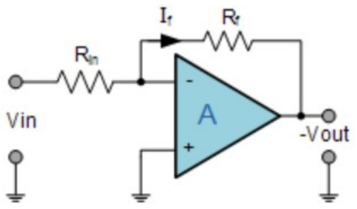

Figure 4: A Potentiometer (Non-inverting)

LM741 was used as the op-amp in this project. It is an 8-pin IC as shown below. The most significant pins are 2,3 and 6, where pin 2 and 3 denote inverting & non-inverting terminals respectively and pin 6 denotes output voltage.

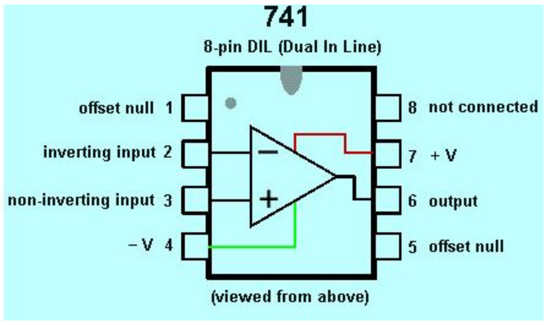

Figure 5: Op-Amp LM741 Pin Diagram

3.1.2 Friendly ARM Mini6410 Development Kit:

Samsung mini6410 is an ARM11 based development kit. It is powered with S3C6410 processor along with a plethora of peripherals such as Audio, Ethernet, Camera etc. It runs a Linux operating system. All the pins from the processor and the kit are discussed here.

3.1.2.1 ADC pin:

The pre-processing circuit supplies a full range of 0V to 3.3V to the ADC input pin which is Pin number 39 on Port 1 (AIN0) as can be verified from the datasheet. The ADC circuit consists of a Sample & Hold block which samples the input analog voltage signal at the rate as pre-defined in the driver program. We are sampling the input current at 1000 samples per second.

3.1.2.2 Pins to control Duty Cycle and Direction:

The process to set values on the PWM output pin and GPP ports using the driver program will be discussed in the software section. Data sheet was referred to find out the exact pins for getting the connections right.

1. PWM output was taken from ARM board to drive control the speed of the motor.
   For this we use GPF14 general purpose port which is located on port 1 at pin number 52.
2. GPP output to control the direction of the motor.
   For this we use GPE3 general purpose port which is located at CON port 1 at pin number 5.

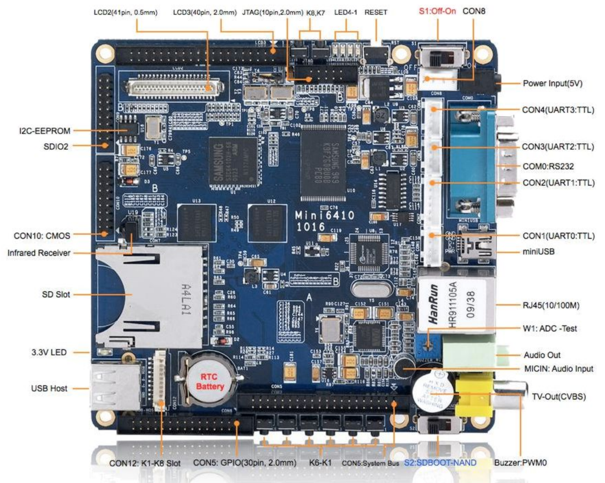

Figure 6: Friendly ARM Mini6410 Development Kit

3.1.3 Stepper Motor Driver Board

Stepper motor drivers regulate the current. If you hook up a motor to 12V, for example, the motor while attempt to draw more current, but the stepper motor driver will not allow that to happen and use high frequency pulses to limit the average current to the desired maximum value.

We used A4988 stepper motor driver to drive the motor. It is a micro-stepping driver for controlling bipolar stepper motors which has built-in translator for easy operation. This means that we can control the stepper motor with just 2 pins from our controller, or one for controlling the rotation direction and the other for controlling the steps.

Now let's close look at the pinout of the driver and hook it up with the stepper motor and the development kit. The 2 pins on the button right side for powering the driver, the VDD and Ground pins that we need to connect them to a power supply of 3 to 5.5 V and in our case that will be our controller, the ARM board which will provide 5 V. The following 4 pins are for connecting the motor. The 1A and 1B pins will be connected to one coil of the motor and the 2A and 2B pins to the other coil of the motor. For powering the motor we use the next 2 pins, Ground and VMOT that we need to connect them to Power Supply from 8 to 35 V. A decoupling capacitor with at least 47 µF can be used for protecting the driver board from voltage spikes. The schematic is as shown below.

The next two 2 pins, Step and Direction are the pins that we actually use for controlling the motor movements. The Direction pin controls the rotation direction of the motor and we need to connect it to GPE3 at CON 1 port as mentioned earlier.

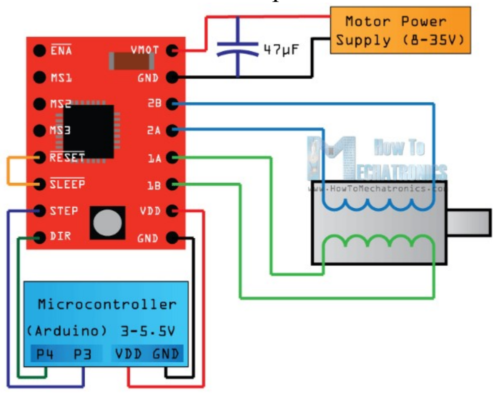

Figure 7: A4988 Driver Board

3.1.3.1 Stepper Motor (NEEMA 17)

This NEMA 17-size hybrid bipolar stepping motor has a 1.8° step angle (200 steps/revolution). Each phase draws 1.7 A at 2.8 V. The motor has four color-coded wires terminated with bare leads: black and green connect to one coil; red and blue connect to the other.

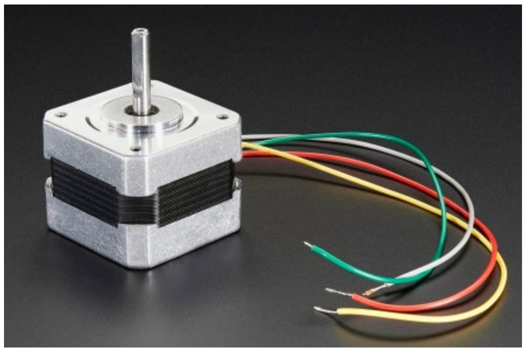

Figure 8: NEEMA17 Stepper Motor

3.1.3.2 Optical Encoder (LPD3806)

To implement PID (Proportional Integral Differential) control, we tried using LPD3806 optical encoder.
An optical rotary encoder is used to measure rotational speed, angle and acceleration of an object. In this project it is used to precisely measure rotation of our motor. Rotary encoders are used to provide direct physical feedback of motor position and speed of rotation. Unlike potentiometers, it can turn infinitely with no end stop. Rotary encoders come in various kinds of resolutions. The number of pulses or steps generated per complete turn for LPD3806 is 600 pulses per resolution for a single phase. Therefore, two phase output leads to 2400 pulses per revolution.

3.1 Working of the encoder to generate a feedback signal:
A rotary encoder has two square wave outputs (A and B) which are 90◦ out of phase with each other. Every time the A signal pulse is on the falling edge, the value of the B pulse is read. From Fig 2, when the encoder is turned clockwise, the B pulse is always positive. The B pulse is negative when the encoder is turned counter-clockwise.
By connecting both outputs with our ARM processor, it is possible to determine the direction of turn. By counting the number of A pulses, we can determine how far it has turned. The two outputs (A and B) represent the motion of the encoder disc as a quadrature modulated pulse train. By adding a third index signal that pulses once for each revolution, the exact position of the rotor can be known.

3.2 Color coding for connection is as follows:
Red- 5V DC, Black/Yellow- Ground, Green- A phase, White- B phase.

3.3 Measuring RPM Using the encoder:
As RPM increases, the period (T) and pulse width (W) becomes smaller. Both period and pulse width are proportional to RPM. The frequency (or period) of either A or B signal gives the RPM of the motor. Counting the number of pulses gives motor position. A-B phase provides motor direction.

3.2. Software Design

The software design for this project can be divided into 4 major parts. First would be to set up the IDE for the ARM processor. Second would be to set up the Application program in C/C++ to take input from the user and pass it to the driver program which brings up to point number three. Setting up all the driver programs to manipulate the input data and result in the controlling the direction and rotation speed of the motor. Lastly, we perform visual validation of the data which is generated by the ADC by performing fast Fourier transform to generate the power spectrum. Also, we tried to integrate an optical encoder in our system to form a closed loop feedback circuit. Each of these steps will be discussed in detail in this section.

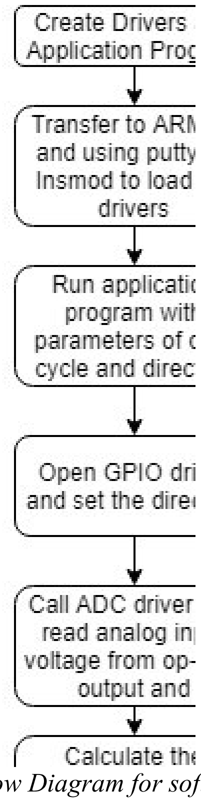

Figure 9: Flow Diagram for software design

3.2.1 Setting Up The Environment

Mini S3C6410 uses Linux operating system as the development platform. We installed and built Linux version 2.6.38 on our local machine and installed gcc cross compiler toolchain. We built all the necessary object files and kernel drivers. Then we write the driver programs and build it as a module. Then we run MenuConfig to modify Kconfig file to implement our code. We use the make command to build the .ko file of our driver module. Transfer these .ko files along with the application program to the ARM kit using a USB and then build the kernel using insmod command.

3.2.2 Device Drivers:

This project can be viewed as a combination of 3 subprojects. First, we setup GPIO driver which is used to decide the direction of the motor rotation. Second, a PWM driver to control the speed of rotation of the motor. And third, ADC driver to convert analog input voltage into digital values. These drivers are provided by the manufacturer and can be found present while installing the packages. We have modified the packages corresponding to our requirements.

1. **GPIO Driver:** We identify a port General purpose E port as the port to send the direction data to the motor. 1 corresponding to counter-clockwise rotation and 0 corresponding to clockwise rotation. To achieve this, the configuration of GPECON is modified and GPE3 pin is set as output. Its unique address is found from the data sheet and we use a pointer that points to the address of the SPR. GPEDAT – a register is used to hold the data while doing read/write operation. GPECON[15:12] when set to 0001 translates it as an output pin. So, we set the 12th pin to 1. This is accomplished by masking operation.

Algorithm:

1. Do insmod to insert the driver module into the kernel image.
2. From the user application file open and link the device driver file.
3. Run init() which is initialization function used to initialize GPE3 port as Direction output port. This is done by setting GPECON[15:12] to 0001 as can be verified from the data sheet.
4. Ioctl() runs to execute the program as per the code.
5. After complete execution of the entire rogrm, perform “rmmod” to remove this module from the driver.

2. **PWM Drivers:** User application program accepts input from the user i.e frequency and duty cycle and passes them to the device driver along with the file descriptor. The device driver is initialized to a frequency of 1000Hz and 50% duty cycle. These values are changed from the user's input. Also, the TCNTB0 and TCMPB0 buffer values are set as per the input. The TCNTB0 buffer decides the total time period of the PWM cycle. TCMPB0 buffer decides the duty cycle of the PWM.
The logic behind setting the duty cycle to control the speed of the motor by pulse width modulation is explained as below.

Step 1. The down-counter is initially loaded from the Timer Count Buffer register (TCNTBn).

Step 2. For each clock cycle, it counts down its value. When the down count value in TCNTBn matches TCMPBn (Timer Compare Buffer) register, the output level changes. Thus, the compare register determines the turn-on time of a PWM output.

Step 3. When down-counter TCNTBn reaches 0, the interrupt is generated, one cycle is finished.

Step 4. TCNTBn is automatically reloaded to start the next cycle and this is repeated.

Algorithm:

1. Do insmod to insert the driver module into the kernel image.
2. From the user application file open and link the device driver file.
3. Run init() which is initialization function used to initialize GPF14 port as PWM output port.
4. Ioctl() runs to execute the program as per the code.
5. After complete execution of the entire rogrm, perform "rmmod" to remove this module from the driver.

3. **ADC Driver:** This driver reads the analog value of voltage generated by the potentiometer and converts it to digital pulses. The input is varied by rotating the potentiometer. The op-amp circuitry coupled with the potentiometer generates a full range of 0 to 3.3 V DC analog signal which is stored in the buffer register. Then this data is sampled at a sampling frequency of 1000 samples per seconds as set in this driver program. As the potentiometer reaches its maximum value, the voltage reaches 3.3VDC and as a result the motor runs at its maximum speed. Similarly when the potentiometer is brought down to 0V, the motor is brought to a halt.

Algorithm:

1. Acquire ADC I/O control
2. Read input voltage value and start ADC conversion by setting sampling rate to 1000 samples per second.
3. Wait for interrupt after completing the conversion.
4. Read the ADC values
5. Release ADC I/O.
6. Send ADC values to user program.
7. Return.

3.2.3 User Application Program: A C/C++ program is required to take input from user and then integrate the three drivers to achieve the successful implementation of the report. The program prompts the user and takes input of duty cycle and direction of rotation of the motor. This uses several file descriptors to open various device drivers. IOCTL command is used to pass values to the corresponding drivers. ADC output is varied by changing the potentiometer. The ADC output is passed on to rotate the motor thus controlling its speed. Also, the ADC values are subjected to Fast Fourier Transform analysis to verify the validity of the convertion.

Algorithm:

1. In Putty, use insmod to insert the 3 drivers and run the application program with parameters 50 for duty cycle and 1 for direction of motor.
2. Open the Device file which was automatically created in the system when we inserted the new kernel module.
3. Read the 256 ADC values generated by rotating the potentiometer.
4. Manipulate PWM to control the speed of the motor.
5. Calculate FFT for these values.
6. Calculate the power spectrum.
7. Find the highest frequency component and check if Nyquest's Theorem is satisfied. This can be done visually.
8. Identify the neighborhood of the highest frequency component. We considered 10% to the left and 10% to the right of the highest frequency component.
9. Calculate the summation of this memory.
10. If this result is less than 15% of the total energy, we conclude Nyquest's theorem is satisfied.

11. If yes, print that data is valid. If not, print data is invalid.
12. Return.

4. Testing and Verification

The system was put under test by establisheing serial communication between the development kit and Linux environment. The 3 drivers executable files and the application program was then copied into an USB and transferred into the ARM kit. Using putty, the drivers were installed with the help of insmod command. Then the user application was executed by providing duty cycle as 50 and direction of rotation of motor set as 1.

The PWM output was first observed on a an oscilloscope as depicted below.

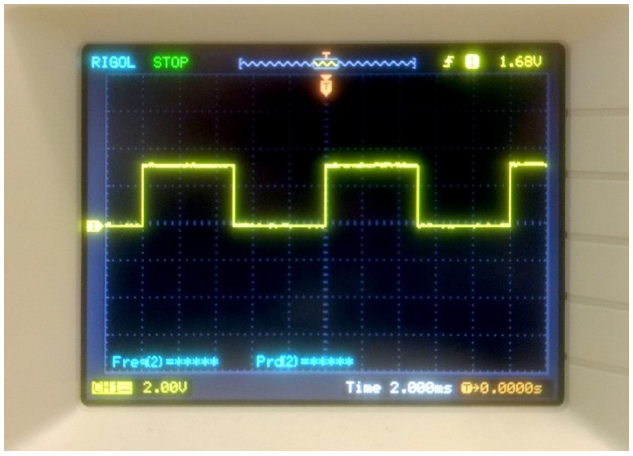

Figure 10: PWM output on an oscilloscope

Upon its prover verification this was then passed on to the driver board which in turn powered the motor.

The potentiometer was then varied to get multiple analog voltage readings which then controlled the speed of the motor. The entire setup was captured as shown in the following figure.

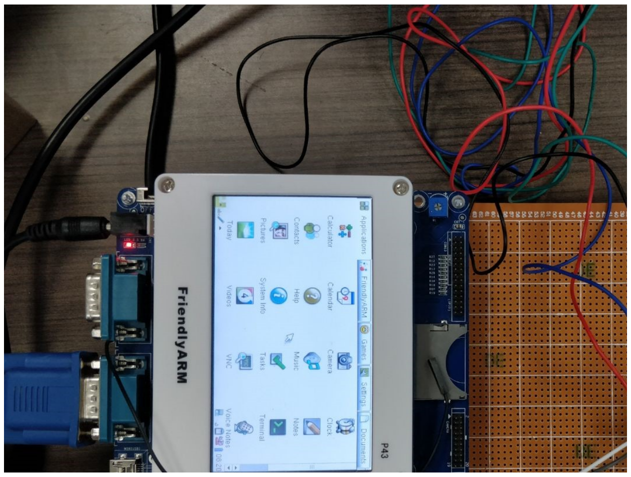

Figure 11: Entire setup with input pre-processing circuit, ARM board connections and motor

To verify if the ADC conversion is valid we performed Fast Fourier Transform on the signal to generate power spectrum. To verify this, the application program read the ADC values and performed the necessary calculations. The following steps were made.

1. Take 256 different values from the ADC output.
2. Perform FFT on the signal
3. Identify the highest frequency component. This will be at around (N/2-1) which is the 127th value.
4. Identify the neighborhood of this point. For this we considered 10 points to the left and right of the spectrum.
5. Aggregate the energy possessed by these values and check if this is less than 15% of the total enery. 15% was considered as the threshold.
6. If this was found true, the program returned that the data is valid. Else it displayed that the values are invaid.
7. Tweaks were made at the op-amp circuitry to make the conversion valid.

The digital value generated from the analog value can be plotted in the ADC interpolation graph and can be verified as below.

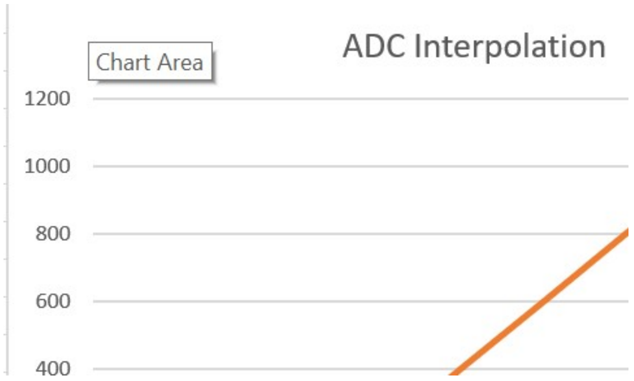

Figure 12: ADC Interpolation

The power spectrum graph based on the calculation was obtained as shown below.

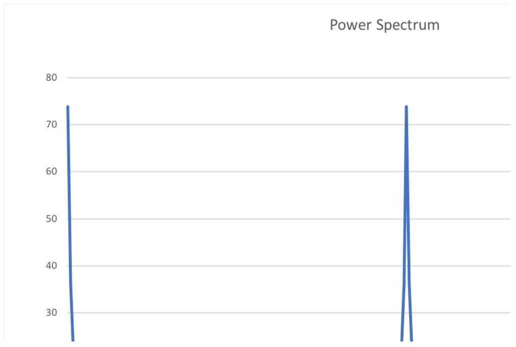

Figure 13: Power Spectrum

This was an alternate method to visually verify that Nyquest's criteria was satisfied. Nyquest's criteria originally checks the condition that frequency of sampling should be greater than twice the maximum frequency to be verified.

5. Conclusion

The development of a system to take input from a sensory interface and run a motor using digital signal was successfully implemented. We were able to convert the analog voltage generated by varying the potentiometer into a series of digital pulse. This was verified n the oscilloscope and then this powered a motor to run at a speed controlled by the value of input voltage. The ARM11 kit was successfully integrated with this system to perform the function as desired. Furthermore, data validation was performed to validate the ADC conversion using the Fast Fourier Transform method by visually observing the power spectrum.

Scope for improvement for this project would be to add an optical sensor to check the speed of the motor and generate a controlled feedback system with a closed loop PID calculation to further add control on the speed of rotation of the motor. This was tried in our project, but satisfactory values were not achieved. Overall, the implementation was a success as the results were verified.

6. Acknowledgement

I would like to thank our professor Dr. Harry Li for his guidance and explanation of the concept. I would also like to thank my project team mates and other classmates for helping me understand the design and setting up the circuit.

7. References

[1] H. Li, "Author Guidelines for CMPE 146/242 Project Report", Lecture Notes of CMPE 146/242, Computer Engineering Department, College of Engineering, San Jose State University, March 6, 2006, pp. 1.
[2] FriendlyARM Mini 6410 SBC (Single-Board Computer) with 533 MHz Samsung S3C6410 ARM11 processor can be found at http://www.friendlyarm.net/products/mini6410.
[3] A4988 Stepper Motor Driver Carrier https://www.pololu.com/product/1182.
[4] NEMA 17 Stepper motor: http://reprap.org/wiki/NEMA_17_Stepper_motor.
[5] Working with incremental rotary sensory encoder: https://toriilab.blogspot.com/2016/09/a-rotary-encoder-is-used-to-measure.html

8. Appendix

This section includes all the source codes for this project.

8.1 GPIO Driver Program:

```
#include <linux/miscdevice.h>
#include <linux/delay.h>
#include <asm/irq.h>
//#include <mach/regs-gpio.h>
#include <mach/hardware.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/mm.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/delay.h>
#include <linux/moduleparam.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/ioctl.h>
#include <linux/cdev.h>
#include <linux/string.h>
#include <linux/list.h>
```

```
#include <linux/pci.h>
#include <asm/uaccess.h>
#include <asm/atomic.h>
#include <asm/unistd.h>

#include <mach/map.h>
#include <mach/regs-clock.h>
#include <mach/regs-gpio.h>

#include <plat/gpio-cfg.h>
#include <mach/gpio-bank-e.h>
#include <mach/gpio-bank-k.h>

#define DEVICE_NAME "himanshu_gpio"

static long sbc2440_leds_ioctl(struct file *filp,
unsigned int cmd)
{
        switch(cmd)
        {
                unsigned tmp;
        case 1:
                tmp                    =
readl(S3C64XX_GPEDAT);
                tmp |= (1 << 3);
                writel(tmp,
S3C64XX_GPEDAT);
                return 0;

        case 0:

                tmp                    =
readl(S3C64XX_GPEDAT);
                tmp &= ~(1 << 3);
                writel(tmp,
S3C64XX_GPEDAT);
                return 0;

        default:
                return -EINVAL;
        }
}

static struct file_operations dev_fops = {
        .owner                 =
THIS_MODULE,
        .unlocked_ioctl    = sbc2440_leds_ioctl,
};

static struct miscdevice misc = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = DEVICE_NAME,
        .fops = &dev_fops,
};
```

```
static int __init dev_init(void)
{
        int ret;

        {
                unsigned tmp;
                tmp                    =
readl(S3C64XX_GPECON);
            tmp &= ~(0xF<<12);
                tmp = (tmp |(1<<12));
                writel(tmp,
S3C64XX_GPECON);

                tmp                    =
readl(S3C64XX_GPEDAT);
                tmp &= ~(1 << 3);
                writel(tmp,
S3C64XX_GPEDAT);
        }

        ret = misc_register(&misc);

        printk
(DEVICE_NAME"\tinitialized\n");

        return ret;
}

static void __exit dev_exit(void)
{
        misc_deregister(&misc);
}

module_init(dev_init);
module_exit(dev_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("FriendlyARM Inc.");
```

8.2 PWM Driver Program:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/poll.h>
#include <asm/irq.h>
#include <asm/io.h>
#include <linux/interrupt.h>
#include <asm/uaccess.h>
#include <mach/hardware.h>
#include <plat/regs-timer.h>
#include <mach/regs-irq.h>
#include <asm/mach/time.h>
```

```
#include <linux/clk.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/miscdevice.h>

#include <mach/map.h>
#include <mach/regs-clock.h>
#include <mach/regs-gpio.h>

#include <plat/gpio-cfg.h>
#include <mach/gpio-bank-e.h>
#include <mach/gpio-bank-f.h>
#include <mach/gpio-bank-k.h>

#define DEVICE_NAME     "himanshu_pwm"

#define PWM_IOCTL_SET_FREQ
        1
#define PWM_IOCTL_STOP
        0

static struct semaphore lock;

/* freq:  pclk/50/16/65536 ~ pclk/50/16
  * if pclk = 50MHz, freq is 1Hz to 62500Hz
  * human ear : 20Hz~ 20000Hz
  */
static void PWM_Set_Freq( unsigned int
freq,unsigned long duty )
{
        unsigned long tcon;
        unsigned long tcnt;
        unsigned long tcfg1;
        unsigned long tcfg0;

        struct clk *clk_p;
        unsigned long pclk;

        unsigned tmp;
        //unsigned int duty;


        tmp = readl(S3C64XX_GPFCON);
        tmp &= ~(0x3U << 28);
        tmp |=  (0x2U << 28);
        writel(tmp, S3C64XX_GPFCON);

        tcon = __raw_readl(S3C_TCON);
        tcfg1 = __raw_readl(S3C_TCFG1);
        tcfg0 = __raw_readl(S3C_TCFG0);

        //prescaler = 50
        tcfg0                          &=
~S3C_TCFG_PRESCALER0_MASK;
        tcfg0 |= (50 - 1);

        //mux = 1/16
        tcfg1                          &=
~S3C_TCFG1_MUX0_MASK;
        tcfg1 |= S3C_TCFG1_MUX0_DIV16;

        __raw_writel(tcfg1, S3C_TCFG1);
        __raw_writel(tcfg0, S3C_TCFG0);

        clk_p = clk_get(NULL, "pclk");
        pclk  = clk_get_rate(clk_p);
        //tcnt  = (pclk/50/16)/freq;
        tcnt  = (pclk/50/16)/freq;
        duty =(tcnt*duty)/100;
        __raw_writel(tcnt, S3C_TCNTB(0));
        __raw_writel(duty, S3C_TCMPB(0));



        tcon &= ~0x1f;
        tcon |= 0xb;                //disable
deadzone,    auto-reload,    inv-off,    update
TCNTB0&TCMPB0, start timer 0
        __raw_writel(tcon, S3C_TCON);



        tcon &= ~2;
        //clear manual update bit
        __raw_writel(tcon, S3C_TCON);
}

void PWM_Stop( void )
{
        unsigned tmp;
        tmp = readl(S3C64XX_GPFCON);
        tmp &= ~(0x3U << 28);
        writel(tmp, S3C64XX_GPFCON);
}

static   int   s3c64xx_pwm_open(struct   inode
*inode, struct file *file)
{
        if (!down_trylock(&lock))
                return 0;
        else
                return -EBUSY;
}


static   int   s3c64xx_pwm_close(struct   inode
*inode, struct file *file)
{
        up(&lock);
        return 0;
}

//static long s3c64xx_pwm_ioctl(struct file
*filep, unsigned int cmd, unsigned long
arg1,unsigned long arg2)
static long s3c64xx_pwm_ioctl(struct file *filep,
unsigned int arg1, unsigned long arg2)
{

if (arg1 == 0)
{
PWM_Stop();
return -EINVAL;
}

PWM_Set_Freq(arg1,arg2);
        return 0;
}


static struct file_operations dev_fops = {
   .owner                    =
THIS_MODULE,
   .open             =
s3c64xx_pwm_open,
   .release          =
s3c64xx_pwm_close,
   .unlocked_ioctl   =
s3c64xx_pwm_ioctl,
};

static struct miscdevice misc = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = DEVICE_NAME,
        .fops = &dev_fops,
};

static int __init dev_init(void)
{
        int ret;

        sema_init(&lock, 1);
        ret = misc_register(&misc);

        printk
(DEVICE_NAME"\tinitialized\n");
        return ret;
}

static void __exit dev_exit(void)
{
        misc_deregister(&misc);
}

module_init(dev_init);
module_exit(dev_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("FriendlyARM Inc.");
MODULE_DESCRIPTION("S3C6410 PWM
Driver");
```

8.3 ADC Driver Program:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/input.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/serio.h>
#include <linux/delay.h>
#include <linux/clk.h>
#include <linux/wait.h>
#include <linux/sched.h>
#include <linux/cdev.h>
#include <linux/miscdevice.h>

#include <asm/io.h>
#include <asm/irq.h>
#include <asm/uaccess.h>

#include <mach/map.h>
#include <mach/regs-clock.h>
#include <mach/regs-gpio.h>
#include <plat/regs-timer.h>
#include <plat/regs-adc.h>

#undef DEBUG
//#define DEBUG
#ifdef DEBUG
#define                    DPRINTK(x...)
{printk(__FUNCTION__"(%d):
",__LINE__);printk(##x);}
#else
#define DPRINTK(x...) (void)(0)
#endif

#define DEVICE_NAME      "adc_lab"

static void __iomem *base_addr;

typedef struct {
    wait_queue_head_t wait;
    int channel;
    int prescale;
} ADC_DEV;

//#ifdef
CONFIG_TOUCHSCREEN_MINI6410
//extern int mini6410_adc_acquire_io(void);
```

```
//extern                              void
mini6410_adc_release_io(void);
//#else
static              inline                int
mini6410_adc_acquire_io(void) {
      return 0;
}
static              inline                void
mini6410_adc_release_io(void) {
      /* Nothing */
}
//#endif

static int __ADC_locked = 0;

static ADC_DEV adcdev;
static volatile int ev_adc = 0;
static int adc_data;

static struct clk          *adc_clock;

#define __ADCREG(name)      (*(volatile
unsigned long *)(base_addr + name))
#define ADCCON
      __ADCREG(S3C_ADCCON)        //
ADC control
#define ADCTSC
      __ADCREG(S3C_ADCTSC)        //
ADC touch screen control
#define ADCDLY
      __ADCREG(S3C_ADCDLY)        //
ADC start or Interval Delay
#define ADCDAT0
      __ADCREG(S3C_ADCDAT0)       //
ADC conversion data 0
#define ADCDAT1
      __ADCREG(S3C_ADCDAT1)       //
ADC conversion data 1
#define ADCUPDN
      __ADCREG(S3C_ADCUPDN)       //
Stylus Up/Down interrupt status

#define PRESCALE_DIS                  (0
<< 14)
#define PRESCALE_EN
      (1 << 14)
#define PRSCVL(x)                     ((x)
<< 6)
#define ADC_INPUT(x)                  ((x)
<< 3)
#define ADC_START
      (1 << 0)
#define ADC_ENDCVT
      (1 << 15)
```

```
#define START_ADC_AIN(ch, prescale) \
      do { \
            ADCCON = PRESCALE_EN |
PRSCVL(prescale) | ADC_INPUT((ch)) ; \
            ADCCON |= ADC_START; \
      } while (0)


static   irqreturn_t   adcdone_int_handler(int
irq, void *dev_id)
{
      if (__ADC_locked) {
            adc_data    =    ADCDAT0    &
0x3ff;

            ev_adc = 1;

      wake_up_interruptible(&adcdev.wait);

            /* clear interrupt */
            __raw_writel(0x0, base_addr +
S3C_ADCCLRINT);
      }

      return IRQ_HANDLED;
}

static  ssize_t  s3c2410_adc_read(struct  file
*filp,  char  *buffer,  size_t  count,  loff_t
*ppos)
{
      char str[20];
      int value;
      size_t len;

      if (mini6410_adc_acquire_io() == 0) {
            __ADC_locked = 1;


      START_ADC_AIN(adcdev.channel,
adcdev.prescale);


      wait_event_interruptible(adcdev.wait,
ev_adc);
            ev_adc = 0;

            DPRINTK("AIN[%d]            =
0x%04x,  %d\n",  adcdev.channel,  adc_data,
ADCCON & 0x80 ? 1:0);

            value = adc_data;

            __ADC_locked = 0;
            mini6410_adc_release_io();
```

```
	} else {
		value = -1;
	}

	len = sprintf(str, "%d\n", value);
	if (count >= len) {
		int r = copy_to_user(buffer, str,
len);
		return r ? r : len;
	} else {
		return -EINVAL;
	}
}

static int s3c2410_adc_open(struct inode
*inode, struct file *filp)
{
	init_waitqueue_head(&(adcdev.wait));

	adcdev.channel=0;
	adcdev.prescale=0xff;

	DPRINTK("adc opened\n");
	return 0;
}

static int s3c2410_adc_release(struct inode
*inode, struct file *filp)
{
	DPRINTK("adc closed\n");
	return 0;
}


static struct file_operations dev_fops = {
	owner:	THIS_MODULE,
	open:	s3c2410_adc_open,
	read:	s3c2410_adc_read,
	release:	s3c2410_adc_release,
};

static struct miscdevice misc = {
	.minor	= MISC_DYNAMIC_MINOR,
	.name	= DEVICE_NAME,
	.fops	= &dev_fops,
};

static int __init dev_init(void)
{
	int ret;

	base_addr                              =
ioremap(SAMSUNG_PA_ADC, 0x20);
	if (base_addr == NULL) {
		printk(KERN_ERR "Failed to
remap register block\n");
		return -ENOMEM;
	}

	adc_clock = clk_get(NULL, "adc");
	if (!adc_clock) {
		printk(KERN_ERR "failed to
get adc clock source\n");
		return -ENOENT;
	}
	clk_enable(adc_clock);

	/* normal ADC */
	ADCTSC = 0;

	ret        =        request_irq(IRQ_ADC,
adcdone_int_handler,      IRQF_SHARED,
DEVICE_NAME, &adcdev);
	if (ret) {
		iounmap(base_addr);
		return ret;
	}

	ret = misc_register(&misc);

	printk
(DEVICE_NAME"\tinitialized\n");
	return ret;
}

static void __exit dev_exit(void)
{
	free_irq(IRQ_ADC, &adcdev);
	iounmap(base_addr);

	if (adc_clock) {
		clk_disable(adc_clock);
		clk_put(adc_clock);
		adc_clock = NULL;
	}

	misc_deregister(&misc);
}

module_init(dev_init);
module_exit(dev_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("FriendlyARM
Inc.");
```

8.4 Application Program:

```
#include <stdio.h>
#include <termios.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/fs.h>
#include <errno.h>
#include <string.h>
#include <math.h>

#define PWM_IOCTL_SET_FREQ
    1
#define PWM_IOCTL_STOP
    0
#define Ramp_up_constant            10
#define Ramp_down_constant          10
#define deltaT
    1000

#define     ESC_KEY
    0x1b

//FFT calculation for obtaining power
spectrum section
struct Complex
{   double a;     //for storing Real Part
    double b;        //for storing Imaginary
Part
}X[257], U, W, T, Tmp;

void FFT(void)
{
    int
M=7,i=1,j=1,k=1,LE=0,LE1=0,IP=0;
    int N=pow(2.0,M);
    for(k=1;k<=M;k++)
    {
        LE=pow(2.0,M+1-k);
        LE1 = LE / 2;
        U.a = 1.0;
        U.b = 0.0;
        W.a     =     cos(M_PI     /
(double)LE1);
        W.b     =     -sin(M_PI     /
(double)LE1);
        for (j = 1; j <= LE1; j++)
        {
            for (i = j; i <= N; i = i
+ LE)
            {
                IP = i + LE1;
                T.a  =  X[i].a
+ X[IP].a;
                T.b  =  X[i].b
+ X[IP].b;
                Tmp.a       =
X[i].a - X[IP].a;

                Tmp.b       =
X[i].b - X[IP].b;
                X[IP].a     =
(Tmp.a * U.a) - (Tmp.b * U.b);
                X[IP].b     =
(Tmp.a * U.b) + (Tmp.b * U.a);
                X[i].a = T.a;
                X[i].b = T.b;
            }
            Tmp.a = (U.a * W.a) -
(U.b * W.b);
            Tmp.b = (U.a * W.b)
+ (U.b * W.a);
            U.a = Tmp.a;
            U.b = Tmp.b;
        }
    }
    int NV2 = N/2;
    int NM1 = N-1;
    int K = 0;
    j = 1;
    for (i = 1; i <= NM1; i++)
    {
        if (i >= j) goto TAG25;
        T.a = X[j].a;
        T.b = X[j].b;
        X[j].a = X[i].a;
        X[j].b = X[i].b;
        X[i].a = T.a;
        X[i].b = T.b;
TAG25:  K = NV2;
TAG26:  if (K >= j) goto TAG30;
        j = j - K;
        K = K / 2;
        goto TAG26;
TAG30:  j = j + K;
    }
}
//FFT Over

static int getch(void)
{
    struct termios oldt,newt;
    int ch;

    if (!isatty(STDIN_FILENO)) {
```

```
		fprintf(stderr, "this problem
should be run at a terminal\n");
		exit(1);
	}
	// save terminal setting
	if(tcgetattr(STDIN_FILENO, &oldt) <
0) {
		perror("save the terminal
setting");
		exit(1);
	}

	// set terminal as need
	newt = oldt;
	newt.c_lflag &= ~( ICANON | ECHO );
	if(tcsetattr(STDIN_FILENO,TCSANO
W, &newt) < 0) {
		perror("set terminal");
		exit(1);
	}

	ch = getchar();

	// restore termial setting
	if(tcsetattr(STDIN_FILENO,TCSANO
W,&oldt) < 0) {
		perror("restore the termial
setting");
		exit(1);
	}
	return ch;
}

static int fd = -1;
static void close_buzzer(void);
static void open_buzzer(void)
{
	fd = open("/dev/himanshu_pwm", 0);
	if (fd < 0) {
		perror("open pwm_buzzer
device");
		exit(1);
	}

	// any function exit call will stop the
buzzer
	atexit(close_buzzer);
}

static void close_buzzer(void)
{
	if (fd >= 0) {
		ioctl(fd,
PWM_IOCTL_STOP);
		if (ioctl(fd, 2) < 0) {
			perror("ioctl 2:");
		}
		close(fd);
		fd = -1;
	}
}

static void set_buzzer_freq(int freq,int duty)
{
	// this IOCTL command is the key to set
frequency
	int ret = ioctl(fd, freq,duty);  //int ret =
ioctl(fd, PWM_IOCTL_SET_FREQ,
freq,duty);
	if(ret < 0) {
		perror("set the frequency of the
buzzer");
		exit(1);
	}
}
static void stop_buzzer(void)
{
	int ret = ioctl(fd, 0);
	if(ret < 0) {
		perror("stop the buzzer");
		exit(1);
	}
	if (ioctl(fd, 2) < 0) {
		perror("ioctl 2:");
	}
}

static void soft_start(void)
{
	int freq_lo=0;
	int duty=50;
	int i;
	for (i=1000; i>0;i-10)
{
	freq_lo=i;
	set_buzzer_freq(freq_lo,duty);
	usleep(1000);
}
}

int main(int argc, char **argv)
{
	int freq = 800;
	int fd_l;
	int duty;
	int on;
	open_buzzer();
	sscanf(argv[1],"%d", &duty);  // enter
the duty cycle from the terminal
```

```
	sscanf(argv[2],"%d", &on);	// give
motor direction from the termial

	fd_l = open("/dev/himanshu_gpio", 0);
	if (fd_l < 0) {
		fd =
open("/dev/himanshu_gpio", 0);
	}
	if (fd_l < 0) {
		perror("open device leds");
		exit(1);
	}

	ioctl(fd_l, on);
	close(fd_l);

	printf( "\nBUZZER TEST ( PWM
Control )\n" );
	printf( "Press +/- to increase/reduce the
frequency of the BUZZER\n" ) ;
	printf( "Press 'ESC' key to Exit this
program\n\n" );
	////////////ADC
		//for DFT
	int i=0;
	int arr[300];
	float power[256];
	float mean;
		//dft over
	int fd_a ;
	fd_a= open("/dev/adc_lab", 0);
	if (fd_a < 0) {
		perror("open ADC device:");
		return 1;
	}

	// for setting up frequency for rotation

	for(i=1;i<=256;i++) {
		char buffer[30];
		int len = read(fd_a, buffer,
sizeof buffer -1);
		if (len > 0) {
			buffer[len] = '\0';
			int value = -1;
			sscanf(buffer, "%d",
&value);
			//Mapping adc value
to frequency
				if(value<5){
				soft_start();

	stop_buzzer();
				}
						freq= value +
1000;


	set_buzzer_freq(freq,duty);

					printf("ADC Value:
%d\n", value);
					arr[i]=value;
					//getchar();
			} else {
					perror("read ADC
device:");
					return 1;
			}
			usleep(500* 1000);
			printf( "\tFreq = %d\n", freq );
			printf( "\tduty = %d\n", duty );
	}

	for (i = 1; i <= 256; i++)
	{
			X[i].a = arr[i];
			X[i].b = 0.0;
	}
	printf
("*********Before*********\n");
	for (i = 1; i <= 256; i++)
			printf ("X[%d]:real == %f
imaginary == %f\n", i, X[i].a, X[i].b);

	FFT();

	for (i = 1; i <= 256; i++)
	{
			X[i].a = X[i].a/256;
			X[i].b = X[i].b/256;
	}
	printf
("\n\n**********After*********\n");
	for (i = 1; i <= 256; i++)
			printf ("X[%d]:real == %f
imaginary == %f\n", i, X[i].a, X[i].b);
	//char buffer[1000];
	for(i=1;i<=256; i++)
	{
			power[i]=
sqrt(((X[i].a*X[i].a)+(X[i].b*X[i].b)));
			printf("power spectrum value
of power[%d] is = %f \n",i,power[i]);
			//sprintf(buffer,
"%f",power[i]);
			fprintf(fp,"%f,\n",power[i]);
	}
	mean=0;
	for(i=102;i<=154;i++)
	{
		mean= mean+ power[i];
	}
	mean/=52;
	printf("mean is %f \n",mean);
	if(mean<(0.15*(power[127])))
		printf("values are valid \n");
	else
printf("values are not valid \n");

	close(fd_a);
return 0;
}
```