Stepper Motor Control using ADC and PWM

Neel Parag Patel
Computer Engineering Department, College of Engineering
San Jose State University, San Jose, CA 94303
E-mail: neel.p.patel@sjsu.edu

Abstract

This report documents the setup and implementation of various components that make up the working of Analog to Digital conversion as well as Pulse Width Modulation. The report also includes the drivers and user space application specifically designed for the arm board and linux OS. A potentiometer was given as the input to the ADC and the speed of motor was varied by varying the resistance of potentiometer. One of the main challenges of this exercise was to combine the knowledge of PWM and ADC to write a user application program that passes the digital value obtained after ADC stage as the frequency parameter to PWM. It was overcome by reading ADC structure and operation from S3C6410 datasheet.

1. Introduction

ADC is a very important technology in today's world. Every time an analog sensor needs to be interface with a device, ADC is the key component that is utilized.

The objective of this lab was to implement the ADC block and drive a stepper motor using PWM (pulse width modulation) signal from the mini6410 Board. The ADC value was used to manipulate the Frequency parameter of PWM. So that turning the potentiometer (changing the current) would change the speed of the stepper motor. The potentiometer was connected to the development kit. On the output side, the driver board was connected to the development kit. Analog input was given to the ADC and the output of ADC was used to manipulate the PWM wave. Which in turn varied the speed of the stepper motor.

2. Methodology

This section explains the method that was carried out in order to achieve the result, which was to vary the speed of the motor by varying the pot position of potentiometer. This was achieved by modifying the driver code provided by FriendlyARM and also writing a user application program that combined the operations of both, the ADC as well as the PWM.

2.1. Objectives and Technical Challenges

Implementation of this lab required working on two parts which were equally important, the software, which needed to be within the limits of our development kit and linux development environment and the hardware, which was used to showcase the actual functioning of our drivers.

The first task to be carried out was to identify the pins of S3C6410 which will be used to interface various components needed for this lab, i.e. Potentiometer and Driver Board. The ADC pin was connected to the Potentiometer and the PWM pins were connected to the driver board. The potentiometer used is a 10kohm pot and the motor used is a NEMA 17 two phase stepper motor.

The next task was to modify the device drivers provided for PWM to take duty cycle into consideration. For ADC the driver provided by the manufacturers was used. The user space program was written such that it would prompt user to input the direction and duty cycle. It was also modified to take into consideration the ADC value to manipulate the PWM frequency. The executable file of the user space program and the kernel object files of the three drivers. i.e. led(GPIO), PWM and ADC were generated and copied onto a USB drive. This USB drive was connected to the mini6410 and inputs were given via serial communication using "putty". The resistance of potentiometer was varied and the speed change of stepper motor was observed.

2.2. Problem Formulation and Design

Our design incorporates separate equally important areas which will help us achieve the goal of this lab. Figure 1 below shows the simple block diagram of the lab setup. This was the basic setup that was required in order to achieve the output of this lab.

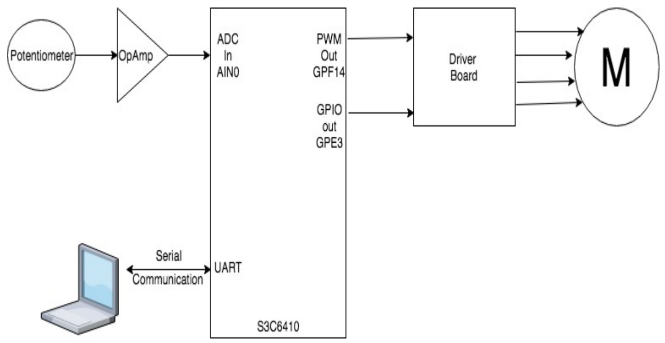


Figure 1. System Block Diagram

Below are the steps that were necessary for the completion of this lab and get a successfully working project.

1. Study about Linux operating system
2. Study about the S3C6410 architecture and peripherals
3. Setup the FriendlyARM development environment on your device
4. Study the manufacturer provided codes to understand the programming aspects
5. Write a simple code to manipulate GPIO state
6. Write a code to manipulate the PWM frequency and Duty Cycle
7. Write a code to take Analog input from the potentiometer and convert it to a digital value.
8. Write a code to implement FFT which is Fast Fourier Transform to visually validate the data.
9. Combine all these codes in order to achieve the objectives of this lab
10. Downloading "Putty" to establish serial communication with the development board.
11. Soldering required components and basic setup of the hardware.
12. Dumping the code and drivers to the board to observe the output and result of the lab.

3. Implementation

This section talks about the hardware and software aspects of this lab. It shows the architecture and flow of program in detail. A basic diagram of the setup of this lab is shown below.

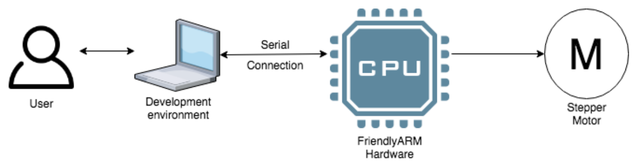


Figure 2. Flow of Implementation

3.1. Hardware Design

As mentioned and shown previously in figure 1, The hardware design entails the assembly and testing of the FriendlyARM board serially connected to our development PC. This board connects to our motor driver board to drive the motor. A Potentiometer is also attached to the ADC block of this board. A detailed description of the hardware used is given below.

3.1.1 FriendlyARM – The FriendlyARM tiny6410 board is a low coast development board with great capabilities. At the heart of this board is a 533 MHz Samsung S3C6410 ARM1176JZF-S CPU with 256 Mb RAM and up to 2Gb of NAND flash. This development board run Linux as its operating system and supports serial communication with our development environment. The images below show the Core module and the SDK board.

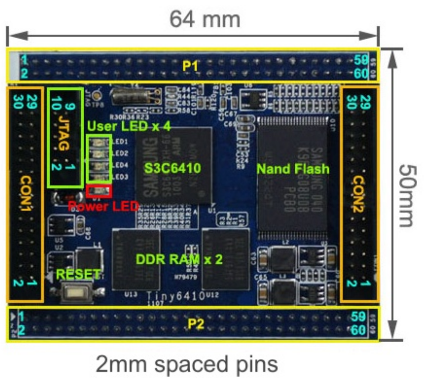


Figure 3. S3C6410 module

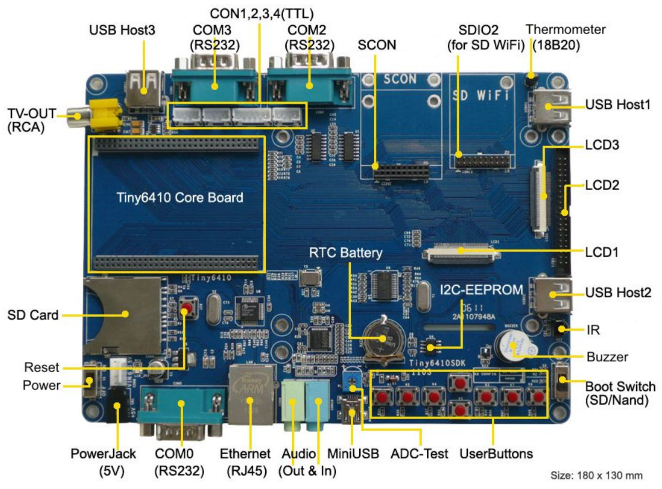


Figure 4. SDK Board

It is worth to note that FriendlyARM provides very good community support for this boards. This board was chosen because of its outstanding functionalities, its ability to interface with analog sensors and motors and its ability to support various operating systems out of which we are using linux.

3.1.2 A4988 Stepper Motor Driver- A4988 is a complete micro-stepping motor driver with built-in translator for easy operation. It can operate bipolar stepper motors in full-, half-, quarter-, eighth- and sixteenth-step modes. This board requires a noise free 6-35V of power supply to drive the motor. Below is a diagram and a brief description of the board.

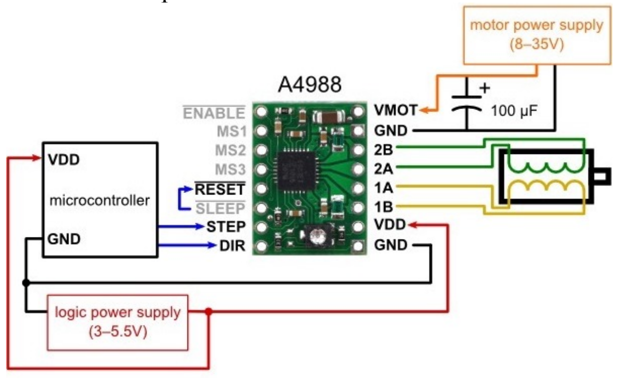


Figure 5. A4988 driver board

1. **STEP-** Logic Input. Any transition on this pin LOW to HIGH will drive the motor one step. Direction and Size of step is controlled by the DIR and MSx pins on the board.
2. **DIR-** Logic Input. This pin determines the direction of spin of the stepper motor. This pin is connected to GPIO to manipulate the direction.
3. **RESET-** Logic Input. When low all step commands are ignored. Must be pulled to high to enable STEP control.
4. **1A-** H-Bridge 1 Output A. Half of the connection point for bi-polar stepper motor coil B.
5. **1B-** H-Bridge 1 Output B. Half of the connection point for bi-polar stepper motor coil B.
6. **2A-** H-Bridge 2 Output A. Half of the connection point for bi-polar stepper motor coil A.
7. **2b-** H-Bridge 2 Output B. Half of the connection point for bi-polar stepper motor coil A.

3.1.3 NEMA 17 Stepper Motor- This is the stepper motor that was used to demonstrate the PWM and its speed was changed by varying the resistance of the potentiometer. This is a plug and play ready motor which easily connects to the driver board via the four wires that it has. An image of NEMA17 motor is given below

Figure 6. NEMA 17 Motor

3.1.4 Bill of Material- Apart from these materials, several other components were used to design and implement this lab. Such as:

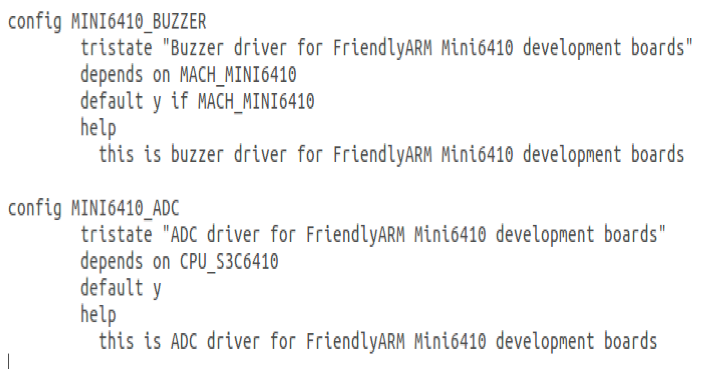
1. Potentiometer
2. OpAmp(IC741)
3. Serial Connection Cable

3.2. Software Design

To achieve successful result of this lab we would have to first install the development environment on our pc. This was needed to write the drivers for the hardware. These drivers were written in “C/C++” code in linux using the “gedit” editor. A serial communication software “putty” needed to be installed in order to communicate with our development board. The user application program was compiled and built before transferring to the board.

It was also required of us to implement the drivers as modules. To achieve that, following steps were taken:

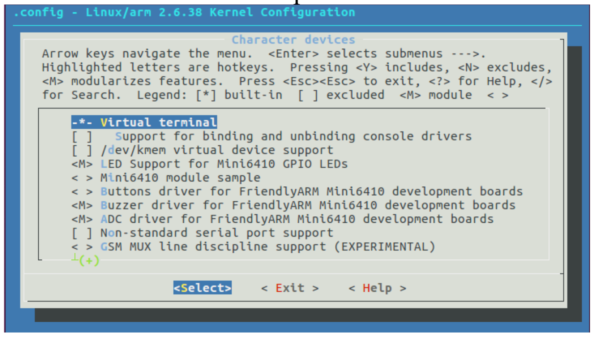
1. First the **Kconfig** needed to be modified in order to let the compiler know that the driver need to be compiled as a module.



```
config MINI6410_BUZZER
        tristate "Buzzer driver for FriendlyARM Mini6410 development boards"
        depends on MACH_MINI6410
        default y if MACH_MINI6410
        help
          this is buzzer driver for FriendlyARM Mini6410 development boards

config MINI6410_ADC
        tristate "ADC driver for FriendlyARM Mini6410 development boards"
        depends on CPU_S3C6410
        default y
        help
          this is ADC driver for FriendlyARM Mini6410 development boards
```

Figure 7. Kconfig

2. Need to run **MenuConfig** and modify the setting of char devices to compile driver as module.



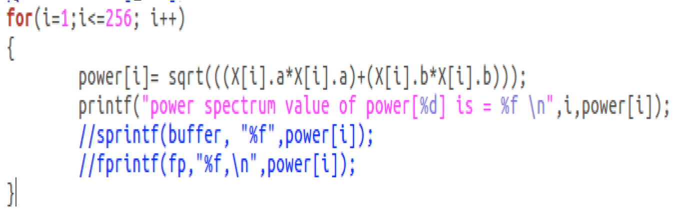
```
.config - Linux/arm 2.6.38 Kernel Configuration
                         Character devices
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

      -*- Virtual terminal
      [ ]   Support for binding and unbinding console drivers
      [ ] /dev/kmem virtual device support
      <M> LED Support for Mini6410 GPIO LEDs
      < > Mini6410 module sample
      < > Buttons driver for FriendlyARM Mini6410 development boards
      <M> Buzzer driver for FriendlyARM Mini6410 development boards
      <M> ADC driver for FriendlyARM Mini6410 development boards
      [ ] Non-standard serial port support
      < > GSM MUX line discipline support (EXPERIMENTAL)
      ↓(+)

                    <Select>    < Exit >    < Help >
```

Figure 8. MenuConfig

3. Once the above modifications are done we can use the **Make** command to build the kernel object(**.ko**) file of the driver.
4. We move these .ko files to a USB drive to transfer them to the board.
5. Once on the board, we can install these modules by using the **insmod** command.
6. **Chmod** command can be used to change permissions of the module.

3.2.1 User Application Program- A user application program needed to be made in order to call the device drivers that would perform necessary operations in order to achieve the end result. This program was written in C/C++ coding language. Basically, when run, this program prompts the user to enter the direction of spin of them motor as well as the duty cycle. This uses several file descriptors to open various Device drivers. IOCTL command is used to pass values to the corresponding drivers. In the main loop it constantly takes in the buffer length, reads the ADC value and uses this value to vary the PWM frequency parameter that is to be passed to the PWM driver. Value of ADC is varied by varying the pot position of the Potentiometer. Data Validation is also performed by using Fast Fourier Transform. This also calculates the power spectrum of the signal.

Fast Fourier Transform: FFT was implemented in order to calculate the power spectrum and validate the data. 256 different values of ADC were first read and the power spectrum of this signal was recorded. Then the center point i.e. 127th point was found and we selected points that lay 10% to the left and right of this point. Then we summed the power spectrum of these values and confirmed that this was 6%(which was less than 15% that is the threshold point) of the total energy of the power spectrum. This proved that the data was valid. A snippet of the code used to calculate power spectrum is given below.



```
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]);
}
```

Figure 9. Power Spectrum calculation

The flow chart for the lab is as given below:

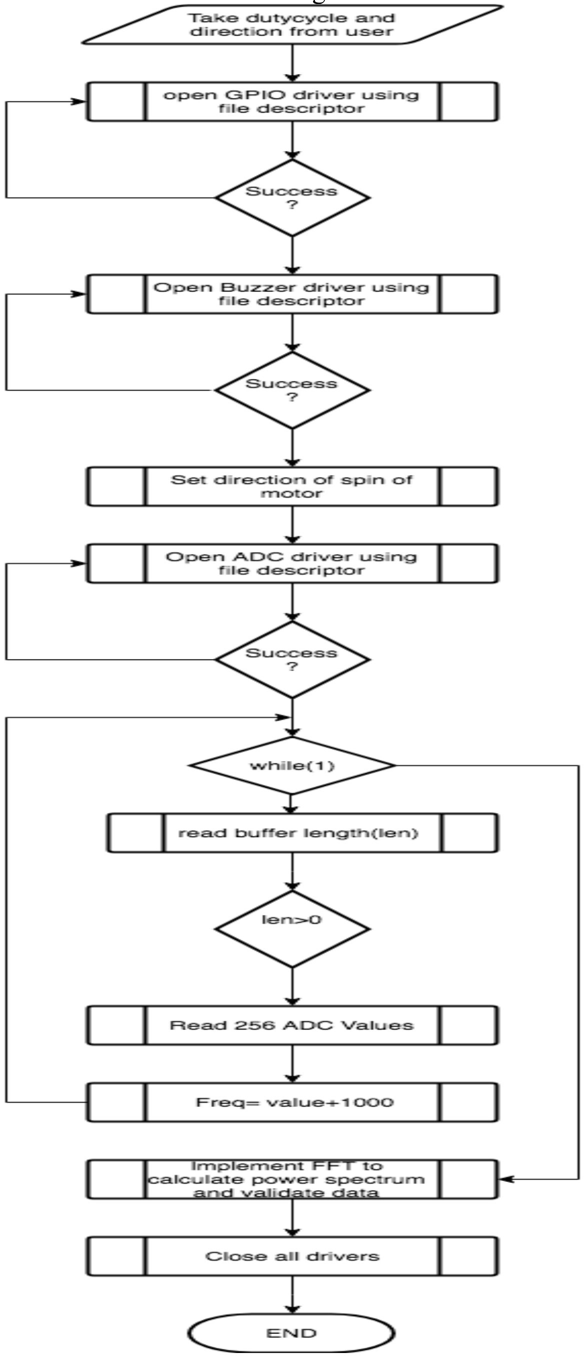

Take dutycycle and direction from user
open GPIO driver using file descriptor
Success ?
Open Buzzer driver using file descriptor
Success ?
Set direction of spin of motor
Open ADC driver using file descriptor
Success ?
while(1)
read buffer length(len)
len>0
Read 256 ADC Values
Freq= value+1000
Implement FFT to calculate power spectrum and validate data
Close all drivers
END

Figure 10. Flow chart

3.2.2 Device Drivers- For implementation of this lab we needed 3 distinct device drivers, GPIO, PWM and ADC. These drivers were provided by the manufacturer and we needed to modify these according to our need. These drivers are also coded in "C/C++" language. The following modifications were done to these drivers:

1. GPIO driver: It was made to accept a parameter that was taken from the user via terminal and set GPE3 (CON1.5) pin which is a general purpose pin as output by modifying the GPECON register. This pin was used to change the direction of spin.
2. PWM driver: This driver was modified to take duty cycle into consideration, duty cycle was input by the user via the terminal. This driver was used to modify the GPFCON register to set GPF14(P1.52) which is the PWM pin as output. This was done by setting the 28th bit of GPFCON as '1'. The duty cycle was written into the S3C_TCMPB register, which is the timer counter buffer register. The frequency was used to set the TCNTB register, which is the timer counter buffer. As the ADC value changed the frequency was manipulated accordingly resulting in change in the speed of motor.
3. ADC driver: This driver was provided by the manufacturer. Basically, what it does is it reads the analog value into the buffer register in an interrupt service routine, this analog value is varied by varying the pot position of the potentiometer. This driver returns the digital value to user space program. These digital values range between 0-1023 since the ADC is operated in 10-bit mode. MUX_SEL register was modified to select AIN0 channel.

The source code of drivers and the user application program are provided in the appendix.

4. Testing and Verification

For testing this lab, Serial communication was established between the mini6410 development kit and the Linux development environment This was accomplished using "putty". The executable for the user application and .ko files for the driver were generated and copied onto a USB drive. This drive was plugged into the board and driver were installed using **insmod** command and then application was executed using **./pwm_test** command. Before changing the frequency using the potentiometer, the motor was run with fix frequency provided directly to the driver. The PWM output was observed on an oscilloscope. Given below is a diagram of the PWM square wave as seen on the oscilloscope.

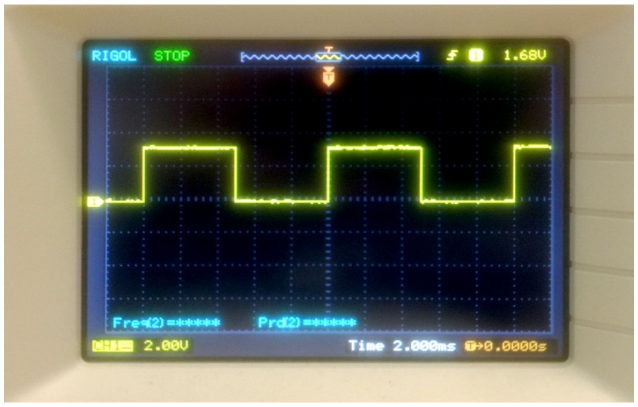


Figure 11. PWM output on oscilloscope

ADC values were read the analog voltage of ADC ranges from 0v-3.3v and digital range is 0-1023. A graph of adc interpolation is given below.

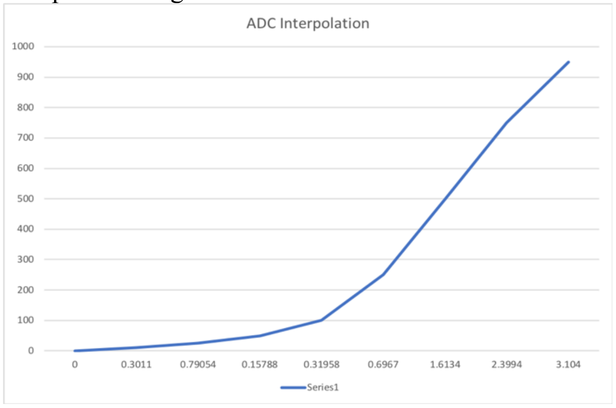


Figure 12. ADC interpolation

The power spectrum of the data was calculated using FFT and data was validated by comparing the power spectrum around the highest point which is 128th point, it was less that 15% of the total power spectrum, hence proving that the data was valid. Given below is a graph of power spectrum.

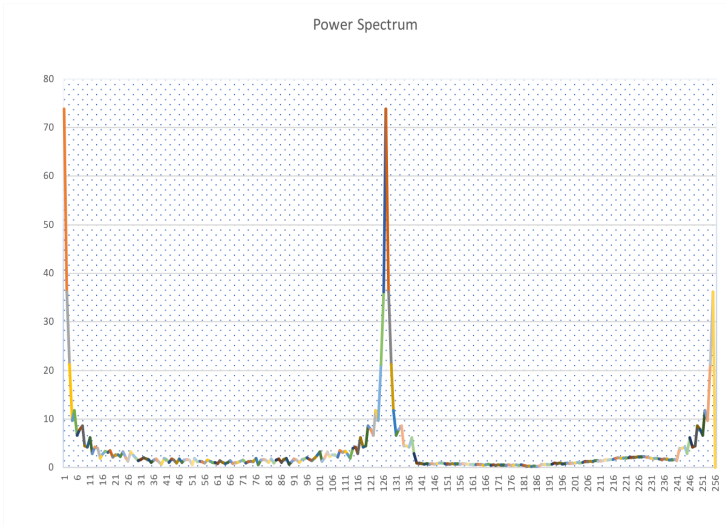


Figure 13. Power Spectrum Graph.

The physical connection on the wire wrapping board were soldered neatly so that there was no case of short circuiting. A picture of the board is given below.

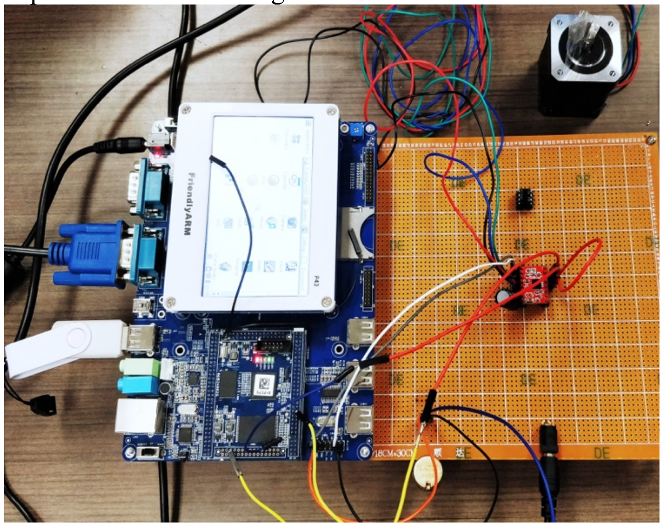


Figure 14. Final setup

5. Conclusion

We were successfully able to manipulate the speed of stepper motor by varying the pot position of the potentiometer. Data validation was also successfully achieved by using Fast Fourier Transform. The ADC value and power spectrum was observed via successfully establishing serial communication with the board.

6. Acknowledgement

I would like to thank Dr. Hua Harry Li for his Guidance throughout this project. I would also like to thank my colleagues who helped me with making the hardware 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] S3C6410 Datasheet

[3]NEMA17 stepper motor https://motion.schneiderelectric.com/downloads/quickreference/NEMA17.pdf

[4]A4988 driver board datasheet https://www.pololu.com/file/0J450/A4988.pdf

[5]FFT code by Prof Harry Li https://github.com/hualili/CMPE242-Embedded-Systems-/blob/master/fft.csssss

8. Appendix

This section contains the source code for the project.

8.1 User 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

#defineESC_KEY              0x1b


/*************FFT*****************
*******/
struct Complex
{       double a;       //Real Part
double b;       //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;
}
}

/*************************************
******/
```

```
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,TCSANOW,
&newt) < 0) {
perror("set terminal");
exit(1);
}

ch = getchar();

// restore termial setting
if(tcsetattr(STDIN_FILENO,TCSANOW,&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/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);
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:");
}
}

int main(int argc, char **argv)
{
//FILE *fp=fopen("PSD.csv","w")
int freq = 1000 ;
int duty;
int on;
int fd_l; //file descriptor for led(GPIO)
int fd_a; //file descriptor for adc

open_buzzer();
sscanf(argv[1],"%d", &duty); // take the duty
cycle from terminal
sscanf(argv[2],"%d", &on);   //take the direction
from the terminal

fd_l = open("/dev/neelassignment", 0);    //open
driver for GPIO to set direction
if (fd_l < 0)
{
fd = open("/dev/neelassignment", 0);
}
if (fd_l < 0) {
perror("open device leds");
exit(1);
}

ioctl(fd_l, on);
close(fd_l);
/***********DFT*****************/
int i =0;
int arr[300];
float power[256];
float mean;
/*******************************/

/***********ADC*****************/
fd_a= open("/dev/adc_lab",0);
if(fd_a < 0)
{
perror("open ADC device:");
return 1;
}

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

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
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(); //doing fast fourier transform

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;
}
```

8.2 PWM driver

```
#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     "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 long freq
)
static void PWM_Set_Freq( unsigned long freq,
unsigned long duty)    // added extra variable for
the duty cycle
{
unsigned long tcon;
unsigned long tcnt;
unsigned long tcfg1;
unsigned long tcfg0;

//unsigned long tduty;

struct clk *clk_p;
unsigned long pclk;

unsigned tmp;

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;

duty = (tcnt*duty)/100;              //modified for
duty cycle

__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 int arg1,
unsigned long arg2 )
{
//switch (cmd) {
//case PWM_IOCTL_SET_FREQ:
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

```
#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"

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 GPIO Driver

```
#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 "leds_lab1"

static long sbc2440_leds_ioctl(struct
file *filp, unsigned int cmd)
{
switch(cmd) {
unsigned tmp;

case 0:tmp = readl(S3C64XX_GPEDAT);
//tmp &= ~(1 << (4 + arg));
tmp  &=  ~(1<<4);     //making  the  pin
LOW.
writel(tmp, S3C64XX_GPEDAT);
return 0;
case 1:
//if (arg > 4) {
//      return -EINVAL;
//}
tmp = readl(S3C64XX_GPEDAT);
//tmp &= ~(1 << (4 + arg));
tmp |= 1<<4;          //making    the
GPE4 pin High to glow the LED.
writel(tmp, S3C64XX_GPEDAT);
//printk (DEVICE_NAME": %d %d\n", arg,
cmd);
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 |= (1<<16);
       //configure GPE as output
//tmp        =         (tmp        &
~(0xffffU<<16))|(0x1111U<<16);
writel(tmp, S3C64XX_GPECON);

tmp = readl(S3C64XX_GPEDAT);
tmp |= (0xF << 4);    //clearing  the
data at GPE
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.");
```