Coursework 2016/2017

Designing, Developing and Testing an Embedded Open-loop Controller

Report
USB oscilloscope and waveform generator

Name: Ravi M. Damodaran, Qinghui Liu (Brian)
Group: Ravi & Qinghui

The embedded application we have implemented is: **USB oscilloscope and waveform generator**

Contents:

Part A:

Part B:

Annex:

Table of Figures

1. Introduction

All electronics engineers will come across a big question during their hobby projects, *how can I have an oscilloscope to test my signals?* It's an expensive device which costs thousands of dollars in cash to purchase it. Similarly, a precise function generator also a challenge to accommodate inside the small home laboratory. This project is an attempt to use the cheaply available- powerful 8 BIT microcontroller ATMEGA 2560 to solve this issue. This microcontroller contains a 10bit Analog to Digital Convertor, 4-USART, 6 Timer/Counter, Waveform generators as well. This makes it a heavily armed device and a smart way of tackling its limits can bring a low cost solution for hobbyists problems.

This projects fundamental idea is to push the limits of ADC module on the chip (ATMEGA 2560) to sample the highest possible frequency keeping the *Nyquist Sampling Theorem* as reference. Sampling frequency should be always greater than or equal to two times of the signal frequency.

$$f_{sample} \geq 2f_{signal}$$

Then push this digital converted analog data through UART- (Asynchronous operation is preferred) to the PC serial port. Use a script in MATLAB to read this serial data and visualize it graphically. This compiles the idea of a signal visualizer. Use the PWM- Timer modules to generate high frequency signals with variable pulse width according to the user needs. This can serve as a signal generator. The interest is in highly calibrated time constrain signals rather than generating a random pulse. This need a careful push of these modules to reach its limits to exploits its functionality.

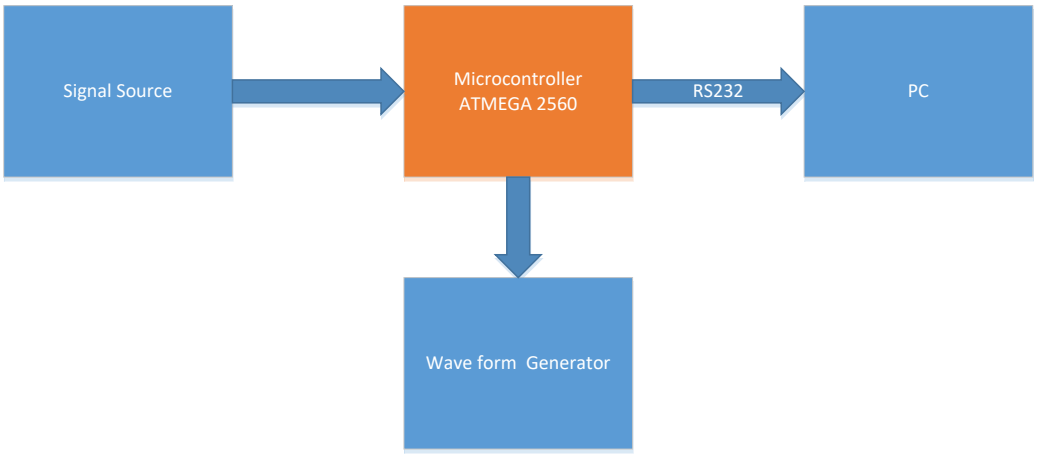


Figure 1 System Overview

The plan is to use an LCD, buttons and few indicator LED's for the user to select the working mode of the device. Human machine interface will be implemented in this manner for user friendly approach. The use case diagram below explains the basic functionality of the system. It is simple and self-explanatory.

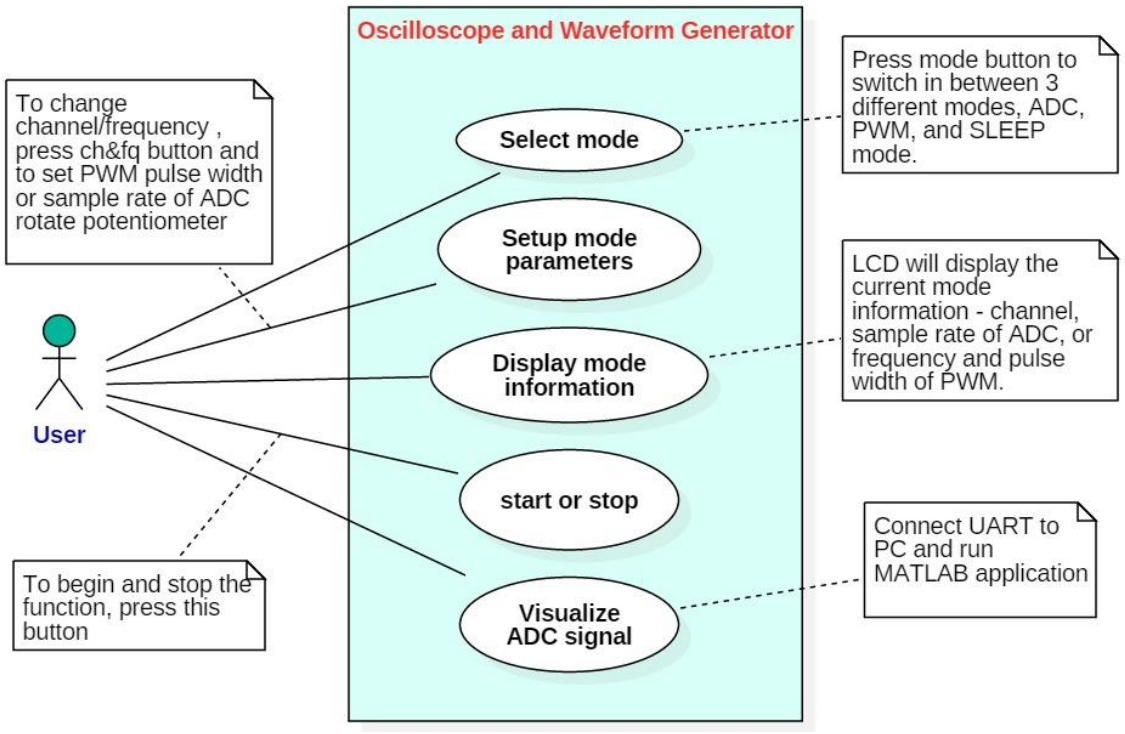


Figure 2 High lever user case diagram

2. Design and Development

2.1. Basic Requirement Analysis

Requirements	**Importance**
Read the analog signal input and send it to the PC for analysis	High
Generate output square wave with variable frequency and duty cycle	High
Have indicator LEDs , input switches	Medium
LCD display for more detailed indication	Low

2.2. Hardware Requirement Analysis

Requirements	**Importance**
Analog signal input- 2 channels	High
Visualizer in PC	High
Sampling rate selection	Medium
Signal generator	High
Signal generation frequency and duty cycle selection	High
User input using buttons	High
LCD display	Medium
LED indicators	Low

2.3. Required Modules and Datasheet Limits

Modules	Specifications	Remarks
Analog to Digital Convertor	Max sampling rate- 1000KHz	It can be pushed to 2000KHz if we use 8 BIT conversion
UART	Max baud rate is 2Mbps	In the double speed configuration
PWM and Timer (timer 0 and 2)	It is 8 bits and its pre-scalar values can be changed from 0 to 1024	It holds two 8 BIT compare registers A and B where we can generate separate waveforms
2 wire serial Interface	7 bit addressing and 400 KHz data rate	It used to interface LCD
Hardware interrupts	Used to interface Buttons	Int 3, 4, 5 are used

2.4. Characteristics of ADC Module and its Limits

To determine the minimum and maximum clock frequency required to use on ADC, the following analysis is required. The following characterization of ADC has been done before the system design. ADC Characteristic curve- (ADC Value- Input Voltage) on different sampling rates.

2.4.1. ADC Setting

1. ADC is on free running mode
2. 8 Bit resolution used (To limit the number of clock cycles required for transferring data)
3. Sampling time = 13x ADC clk time
4. Processor is running on 16 MHz clk
5. The output value of ADCH is measured from debug registers in ATMEL STUDIO

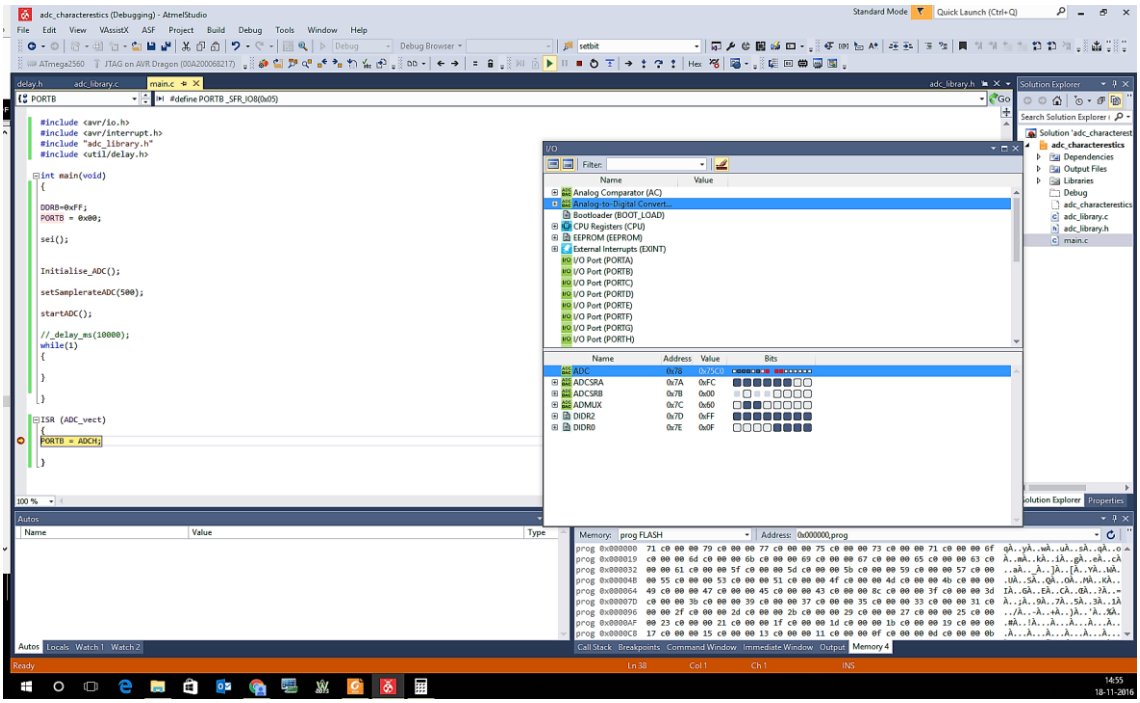


Figure 3 ADC experiment

2.4.2. Experiment Procedure and Observation

1. Input of ADC is fed with a precise voltage generator with high output impedance. The range set from 0-5 V
2. Sampling rate is changed from 125 KHz to 8 MHz (This has a scaling factor of 1/13 since it takes 13 clk cycles for conversion.

Voltage input	Theoretical Value	ADC Value (f=125KHz)	ADC Value (f=250KHz)	ADC Value (f=500KHz)	ADC Value (f=1000KHz)	ADC Value (f=2000KHz)	ADC Value (f=4000KHz)	ADC Value (f=8000KHz)
0	0	0	0	0	0	1	3	12
0.06	3	1	1	1	1	1	3	12
0.07	4	3	3	3	3	2	6	12
0.1	5	4	4	4	4	4	7	12
1	51	57	57	57	57	57	60	63
2	102	117	117	118	118	118	120	124
3	153	177	178	177	177	177	176	192
4	204	238	238	238	238	238	233	227
4.27	218	255	255	255	255	255	249	243
4.3	219	255	255	255	255	255	252	245
4.32	220	255	255	255	255	255	254	248
4.34	221	255	255	255	255	255	255	250
4.4	224	255	255	255	255	255	255	252
4.5	230	255	255	255	255	255	255	255
5	255	255	255	255	255	255	255	255

2.4.3. Plot input voltage vs ADC value

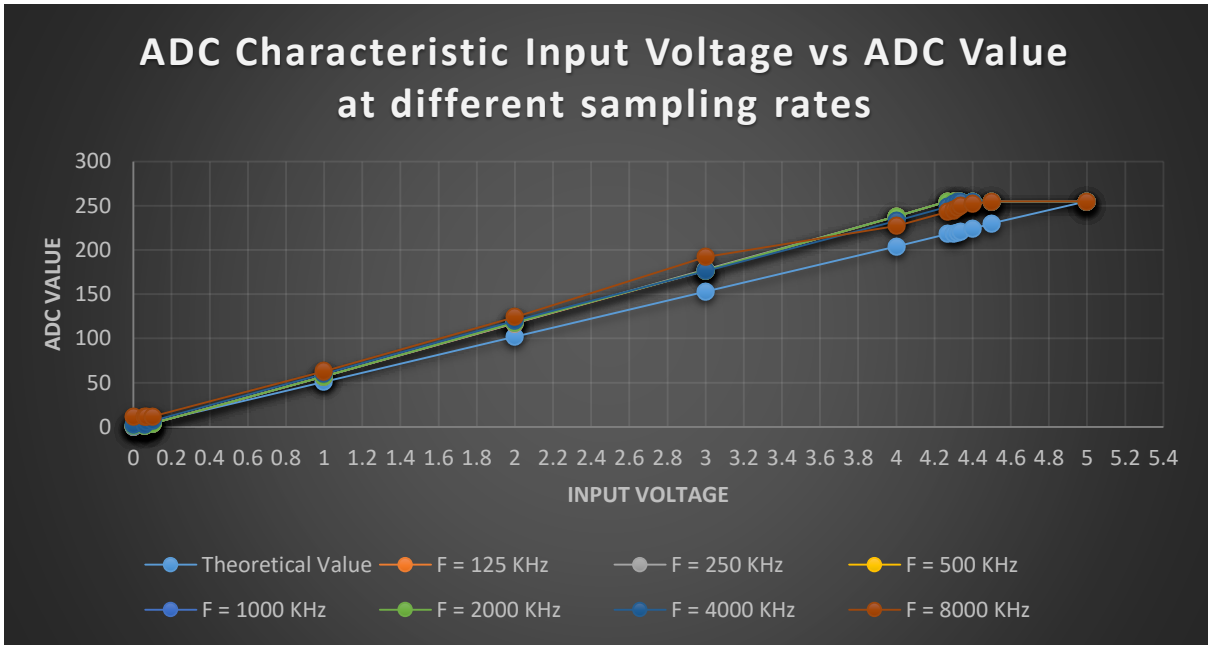


Figure 4 Input Voltage Vs ADC Value

2.4.4. Conclusion

From the above experiment, it is being concluded that the ADC is even designed to work on a maximum clock rate 1000 KHz, it is quite stable at 4000 KHz. Our attempt is to utilize the chip's maximum capability; hence the ADC clock can be selected as 4000 KHz.

2.5. System Design Based on Worst Case Execution Time Analysis

The design should be based on the below structure. T sampling is the sampling time, T conversion is the conversion time. Here we use free running mode, so there is no special time is required for sampling. During the conversion of the captured signal, next data sampling will occur. In effect Tsampling = 0.

$$T_{ADC} = T_{conversion} = 13ADC_{clk}$$

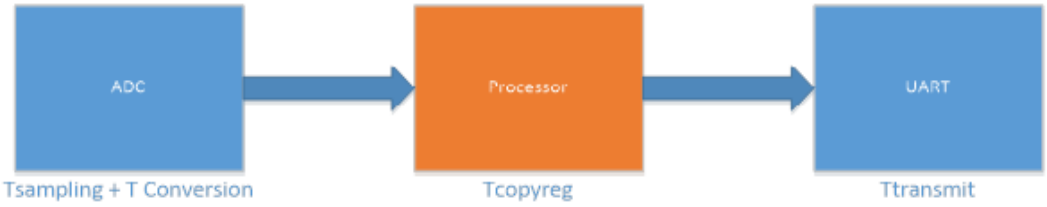


Figure 5 Concept of operation

$$T_{ADC} \geq T_{copy\ register} + T_{transmit}$$

2.5.1. Analysis Based on the Equation

	System Clock	UART	ADC
	16 MHz	2 MHz	ADC clk time required is 4000ns/13 = 307 ns
	62.5 ns time period	Baud rate = 2000000	Max ADC clk frequency = 3.25 MHz
		Time for sending 8 bits = (1/2000000)×8 = 4000 ns	Nearby value is 2 MHz by setting the presale to 8
Required clock speed	16 MHz	2MHz	2 MHz

2.5.2. Block Diagram

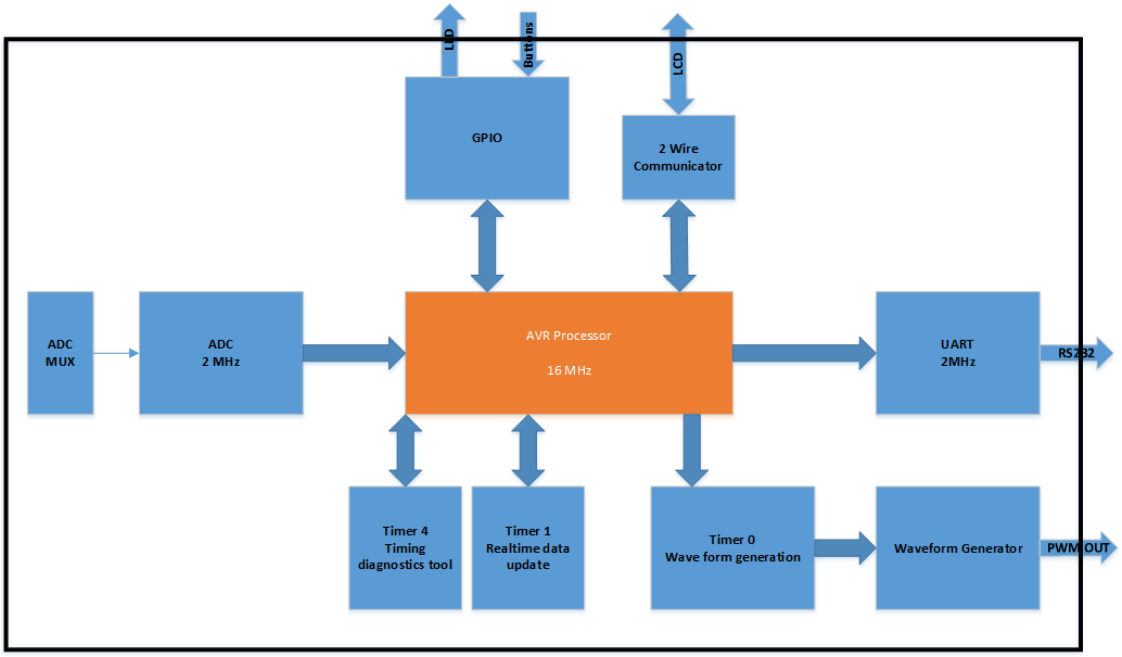


Figure 6 Block Diagram of system

2.5.3. Functionality Explanation

Module	**Functionality**
ADC	Sample input signal and convert to digital
GPIO	For input control switches and output indicator LED's
Timer 0 & Waveform Generator	For generating precise PWM signal / Sine Wave
Timer 1	Update Global variable on real time used for LCD display and LED's
Timer 4	It's a diagnostics tool for debugging and advanced users to test the code execution in detail
UART	For sending analog samples to PC
2 Wire communicator	Used to interface the LCD

2.5.4. Designed Values of ADC Sample Rate and PWM Frequencies

ADC Sample Rates (Theoretical considering 13 ADC CLK for conversion)	**PWM Frequencies**
9 KHz	62.25 kHz
19 KHz	7.8 KHz
38 KHz	1 KHz
76 KHz	245 Hz
152 KHz	

2.5.5. User Instructions

If user need to select ADC Mode

- Power ON the module
- PUSH the Mode button and select ADC Mode
- Then the GREEN LED will be turned ON and LCD will display ADC MODE
- Now push the channel select button repeatedly to select the input channel number
- The channel number will be displayed on the LCD and RGB LED
- Will light GREEN -Channel 1, Blue -Channel 2, RED -Channel 3
- Use the POT meter to select the sampling rate and it will be displayed on the LCD.
- Then push the start button to start the operation. Now the Green LED will be blinking to indicate ADC is running.
- During the runtime, MODE button and Channel button will be disabled.
- Connect UART to PC and run the MATLAB code to see the input signal.
- To stop operation, push the Start button again.

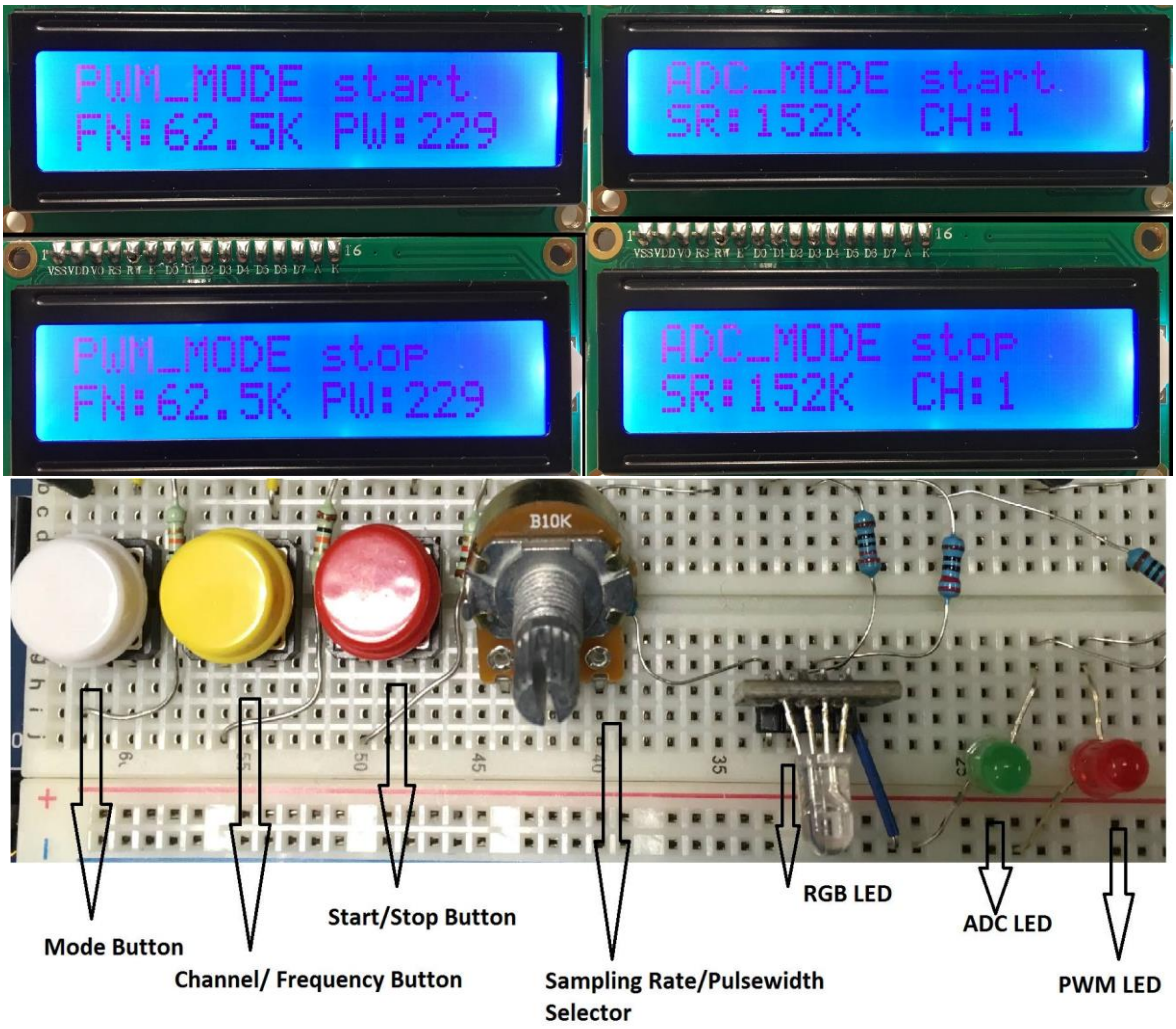


Figure 7 User Interface

If user need to select PWM Mode

- PUSH the Mode button and select PWM Mode
- Then the RED LED will be turned ON and LCD will display PWM MODE
- Now push the frequency select button repeatedly to select the PWM frequency
- Use the POT meter to select the pulse width and it will be displayed on the LCD.
- Then push the start button to start the operation. Now the RED LED will be blinking to indicate PWM is running.
- During the runtime, MODE button and frequency button will be disabled.
- To stop operation, push the Start button again.

3. Hardware Design

3.1. PORT Mapping

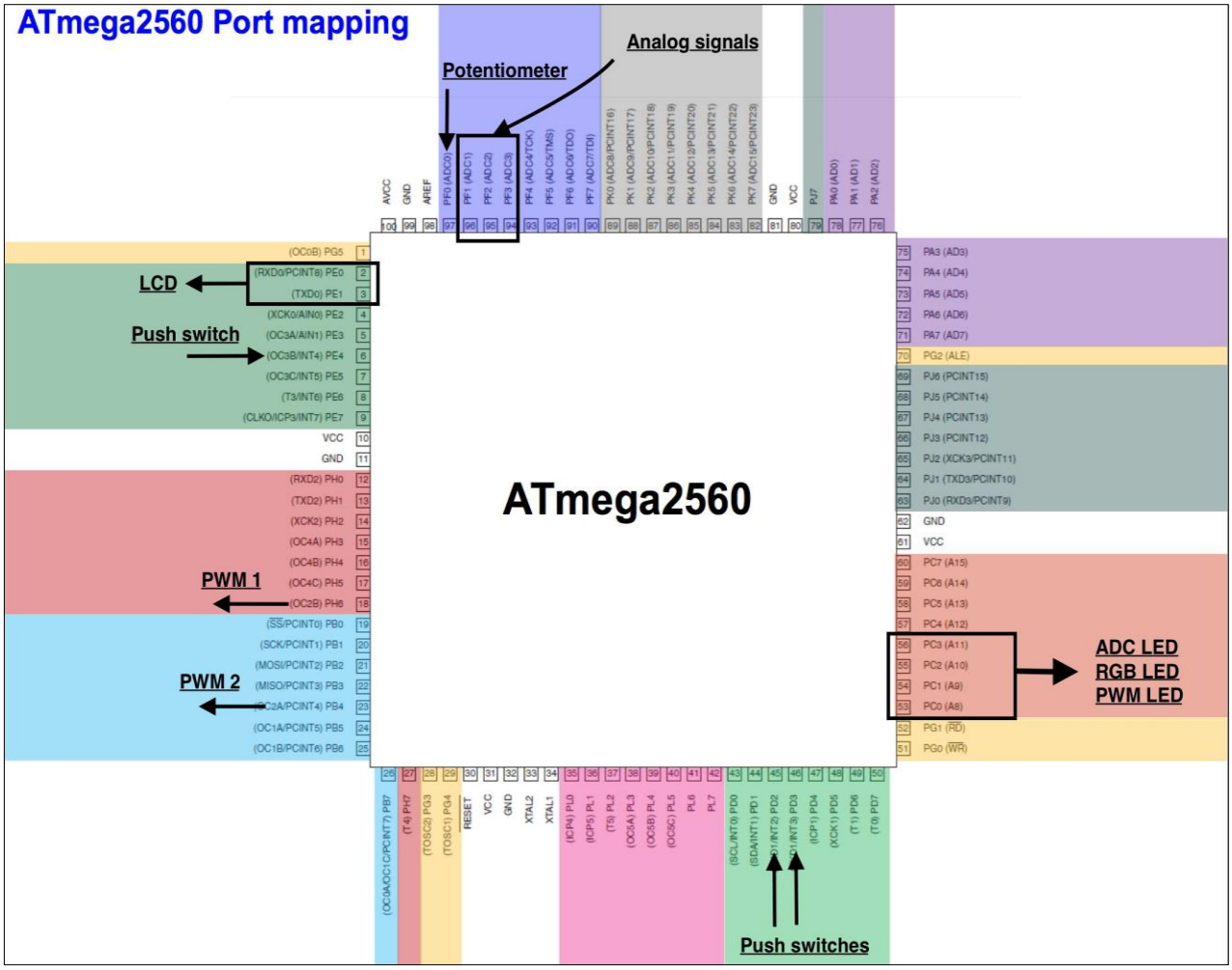


Figure 8 Port Mapping

3.2. Circuit Design

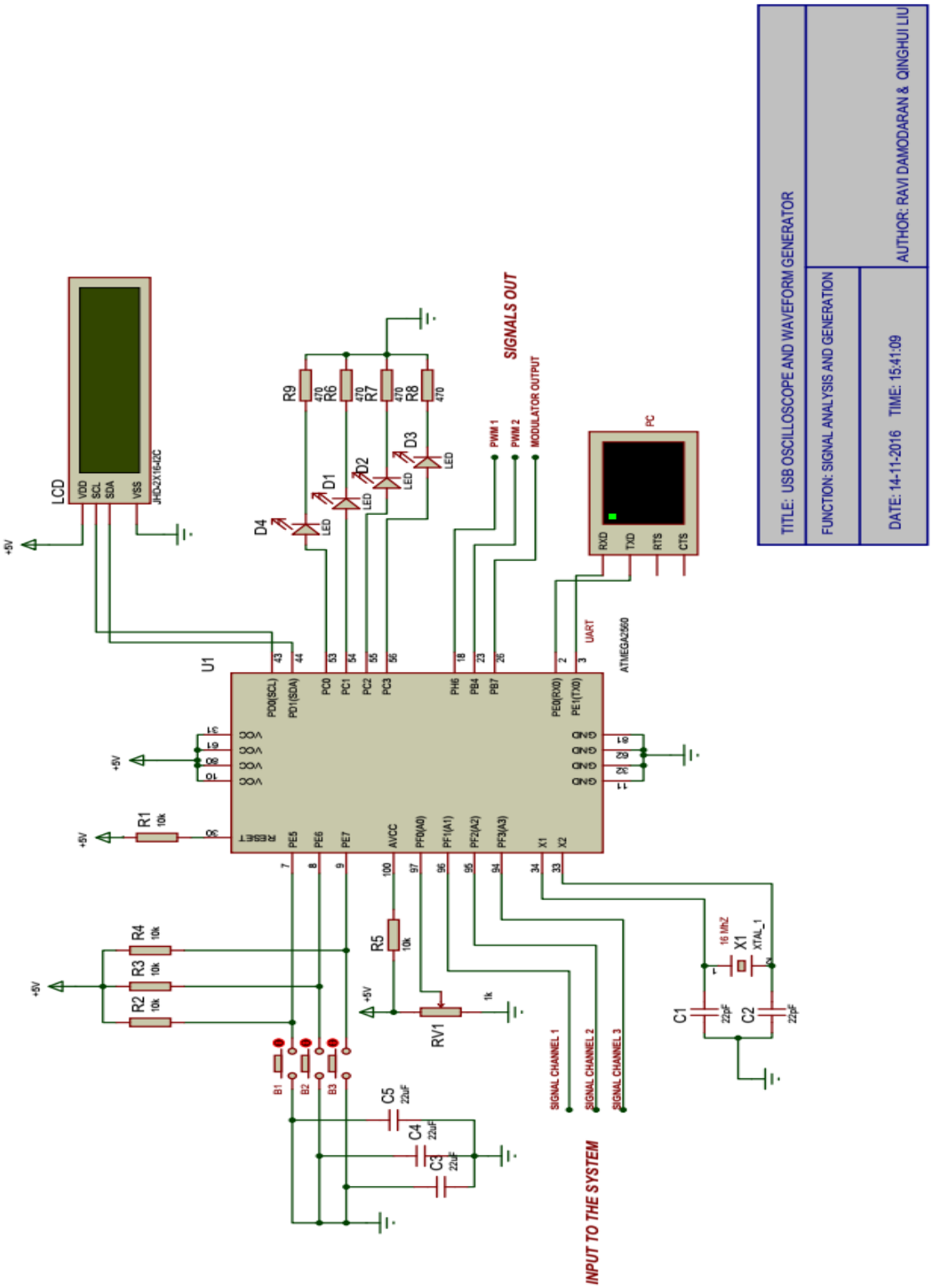


Figure 9 Circuit Diagram

3.3. Button Denounce Design

Here we need a denounce time more than 100 ms to give a stable value for microcontroller to get an interrupt and execute. Low pass filter is basically need to be designed. Equation of Low pass filter is

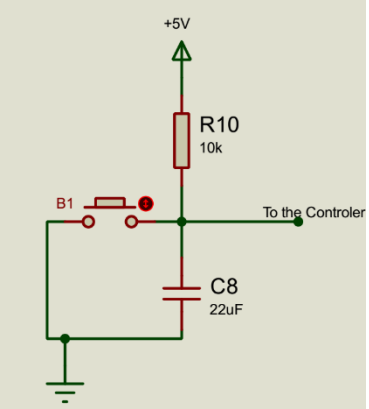


Figure 10 De-bounce Circuit

$Low\ pass\ filter\ cut\ off\ frequencey\ f_c = 1/2\pi RC$

Consider a denounce Frequency nearby 1 Hz, Resistance value = 10K

Then $C = \frac{1}{2\pi 10 * 10^3}$

C = 15μF

Closely available value = 22 μF

R = 10 K
C= 22 μF

4. Firmware Design

The algorithm can be developed based on the detailed analysis of MCU configuration, PORT mapping, Peripheral requirements, Internal requirements and finally the functionality requirements

Detailed flowchart is given to explain each and every function.

4.1. Flow Charts

4.1.1. Main Function and Timer 1 ISR

This flow chart explains the main function of the controller. This function uses to set all input out parameters, Setup timer, UART, ADC etc. essential for the functionality. In this software design all functionality is achieved by interrupt driven routines.

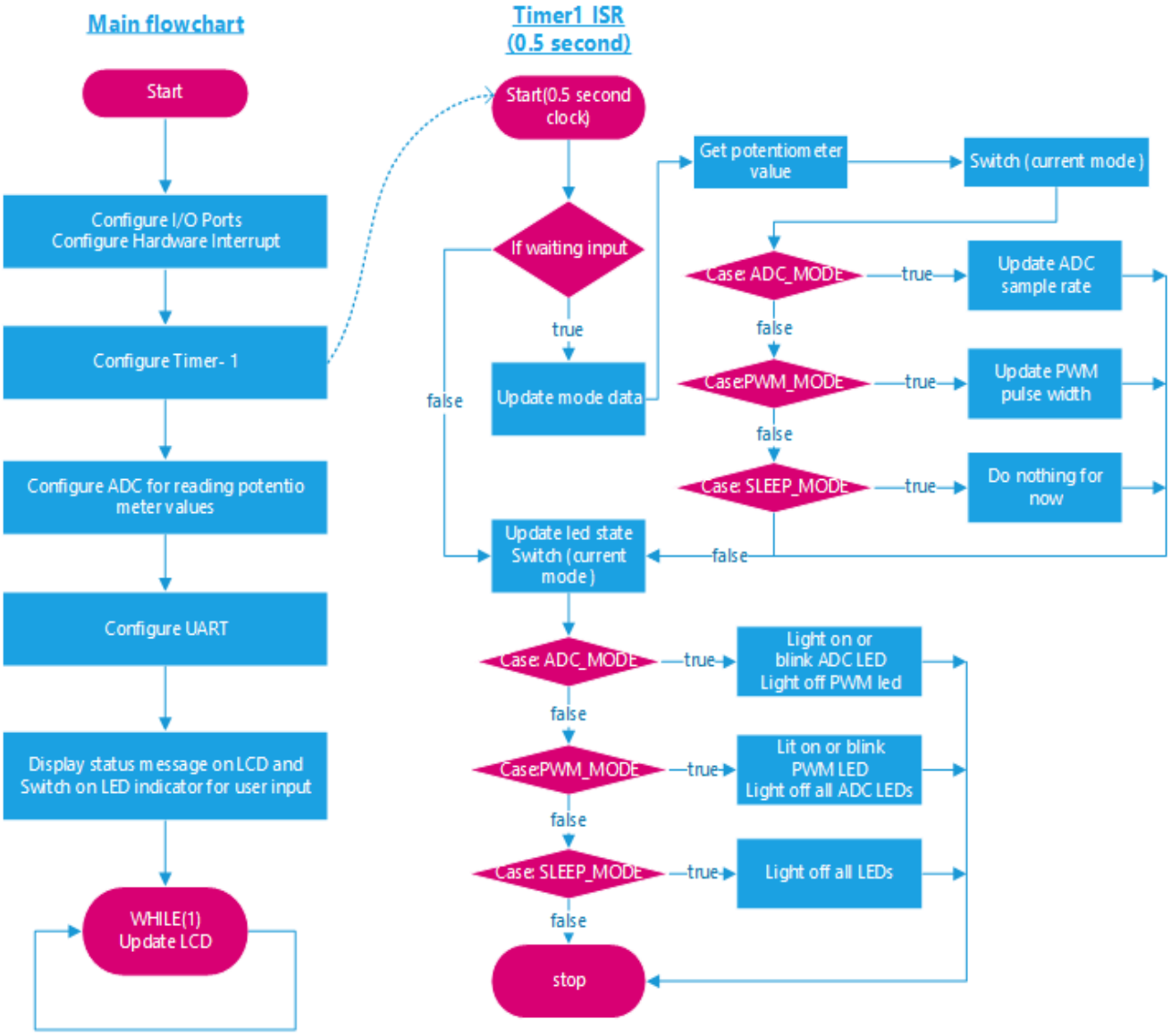


Figure 11 Main flow chart and Timer 1 ISR

4.1.2. Hardware Interrupts

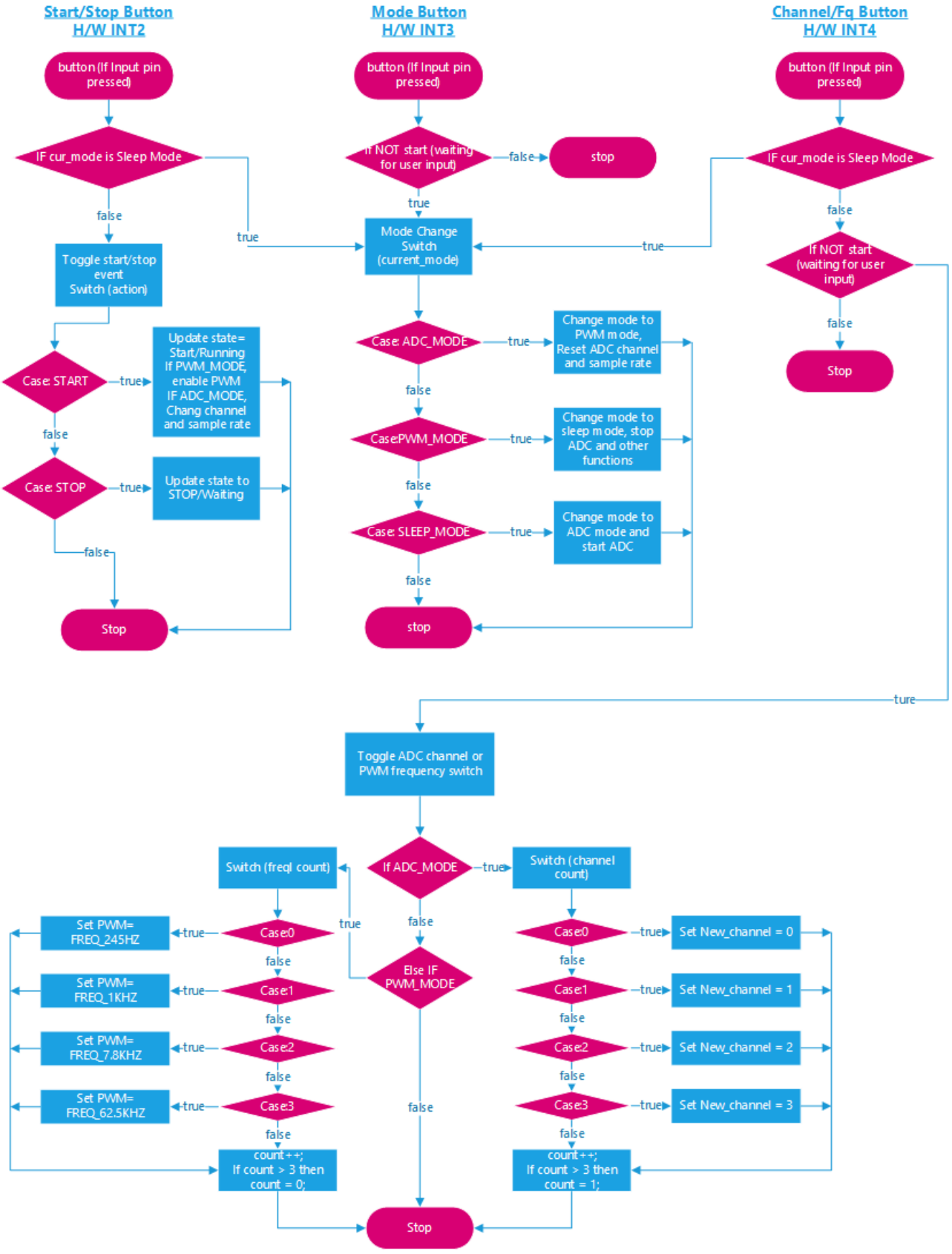


Figure 12 Hardware Interrupts

4.1.3. ADC Interrupt and PWM Function

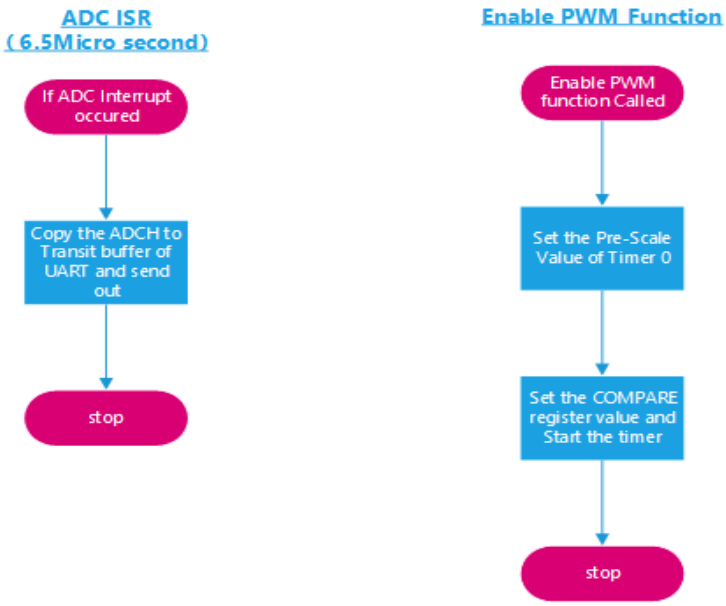
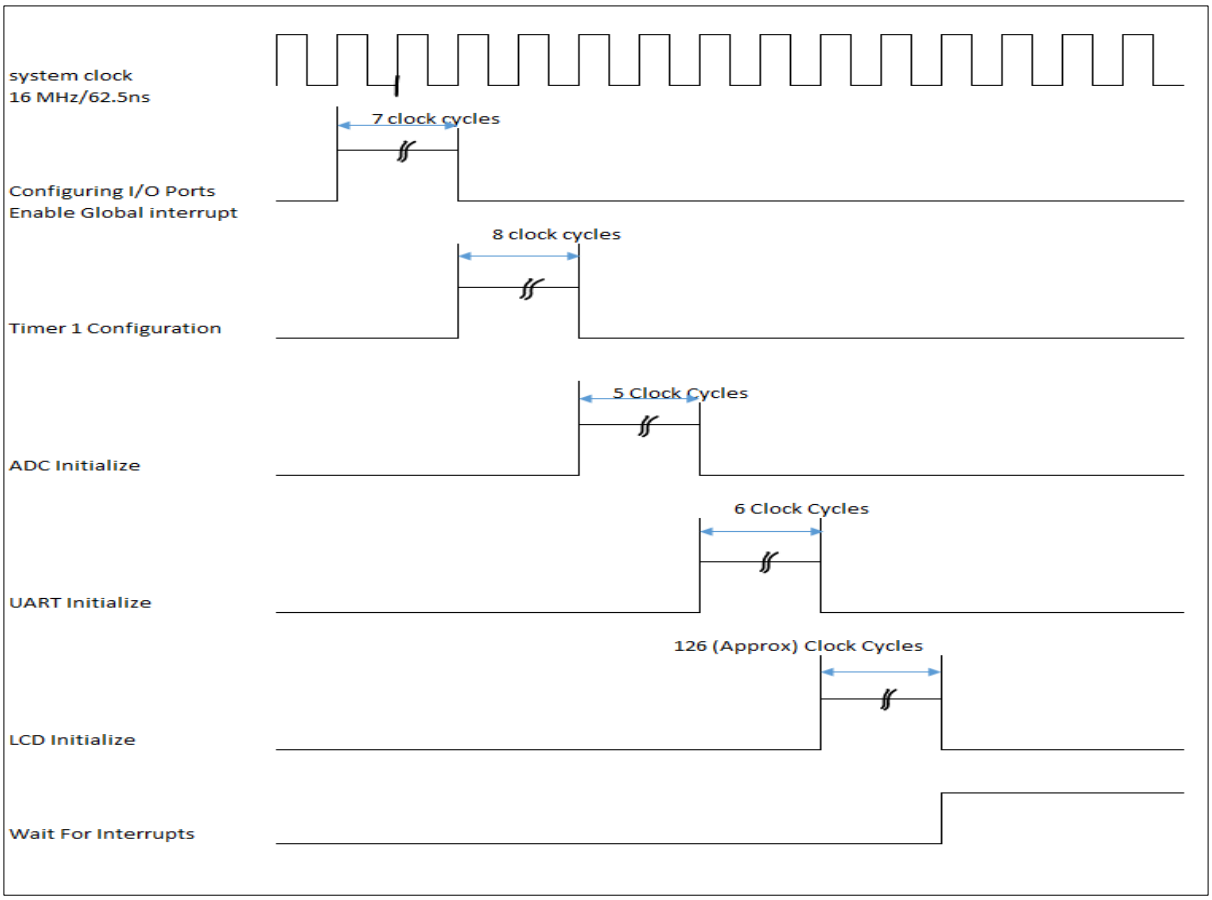


Figure 13 ADC Interrupt and PWM Function

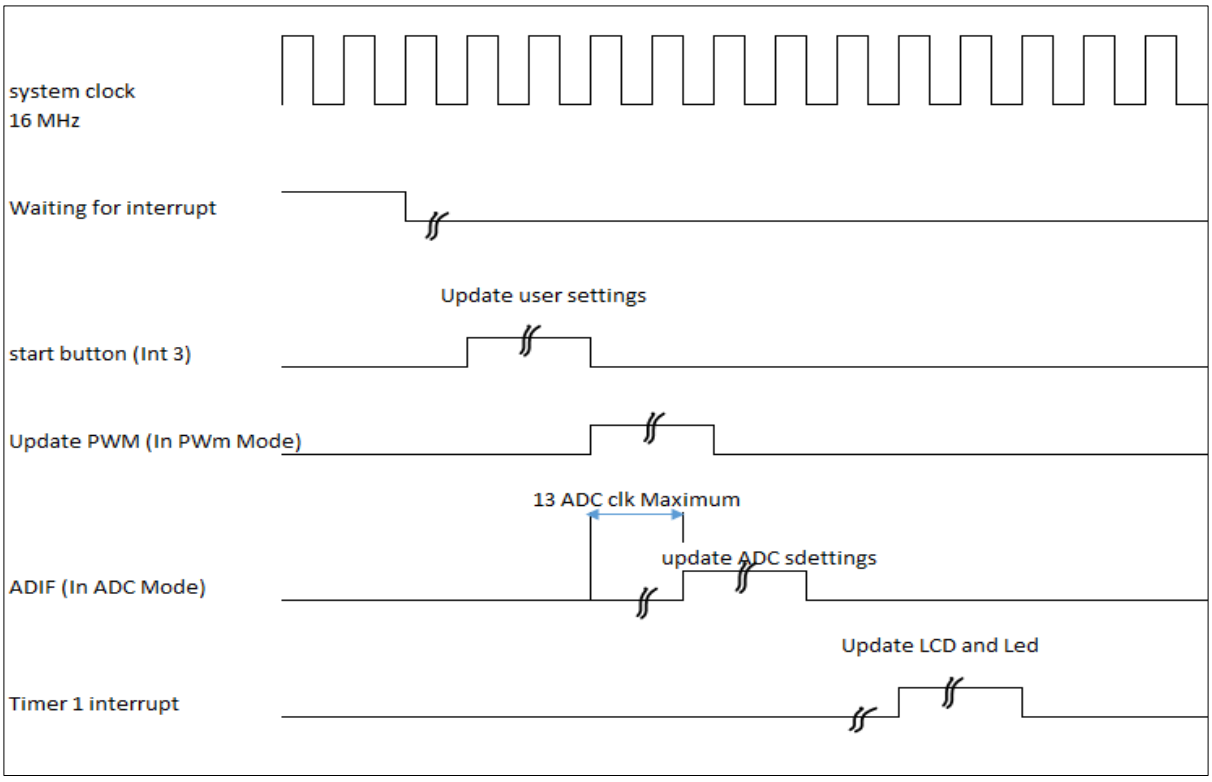
4.2. Timing Diagrams

Timing diagrams are used for the time constrain execution analysis of the software code inside MCU. This will help to plan the best possible function arrangement inside the micro controller.

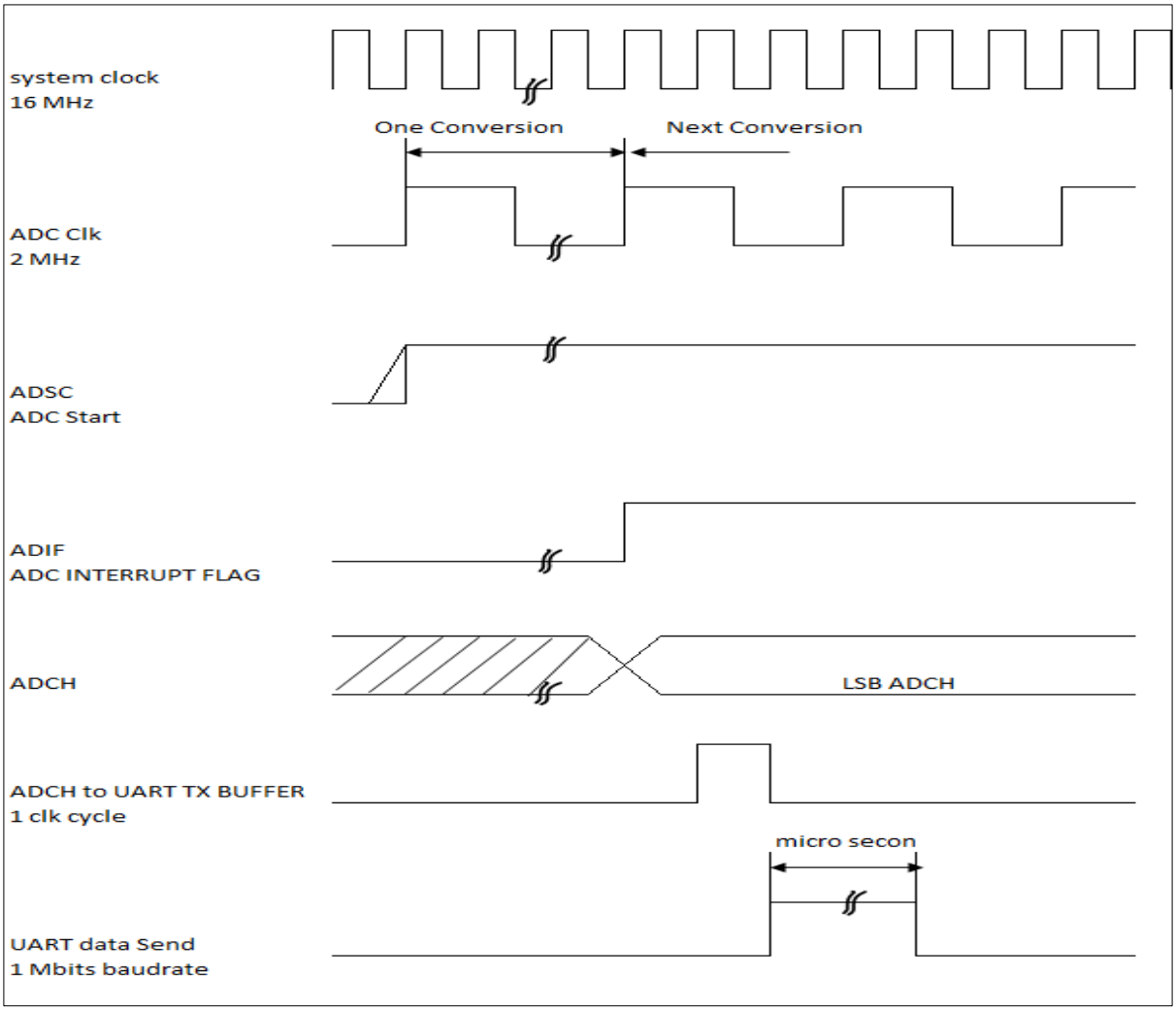
4.2.1. Main Function Flow



4.2.2. Interrupt Handler



4.2.3. ADC and UART



4.2.4. Worst Case Execution Time of ADC combined with UART

ADC sample + Conversion = 13 ADC Clk cycles = 500 ns * 13 = 6.5 μS(ADC CLK = 2 MHz)

$$\boldsymbol{WCET = T_{ADC} + T_{copy\ register} + T_{transmit}}$$

$$\boldsymbol{T_{ADC} = T_{conversion} = 13ADC_{clk} = 6.5\ \mu S\ \text{at}\ (\text{ADC Clk} = 2\ \text{MHz})}$$

$$\boldsymbol{T_{copy\ register} = 0.0625\ \mu S}\ \ (Sys\ Clk = 16\ MHz)$$

$$\boldsymbol{T_{transmit}} = \frac{1}{2000000} * 8 = \boldsymbol{4\mu S}\ \ (Sys\ Clk = 16\ MHz\ and\ baudrate = 2Mbps)$$

Hence

$$\boldsymbol{WCET} = 6.5\mu S + 0.0625\mu S + 4\mu S = \boldsymbol{10.5625\ \mu S}\ (Sys\ Clk = 16\ MHz)$$

5. Graphical User interface

A MATLAB algorithm is used to capture the serial port data (data from UART) and plot it as it is. There is no digital to analogue conversion is happening for the time being. The data is plotted

in Y axis against time to visualize it. This algorithm can be modified in future to correctly display the frequency / period of signal.

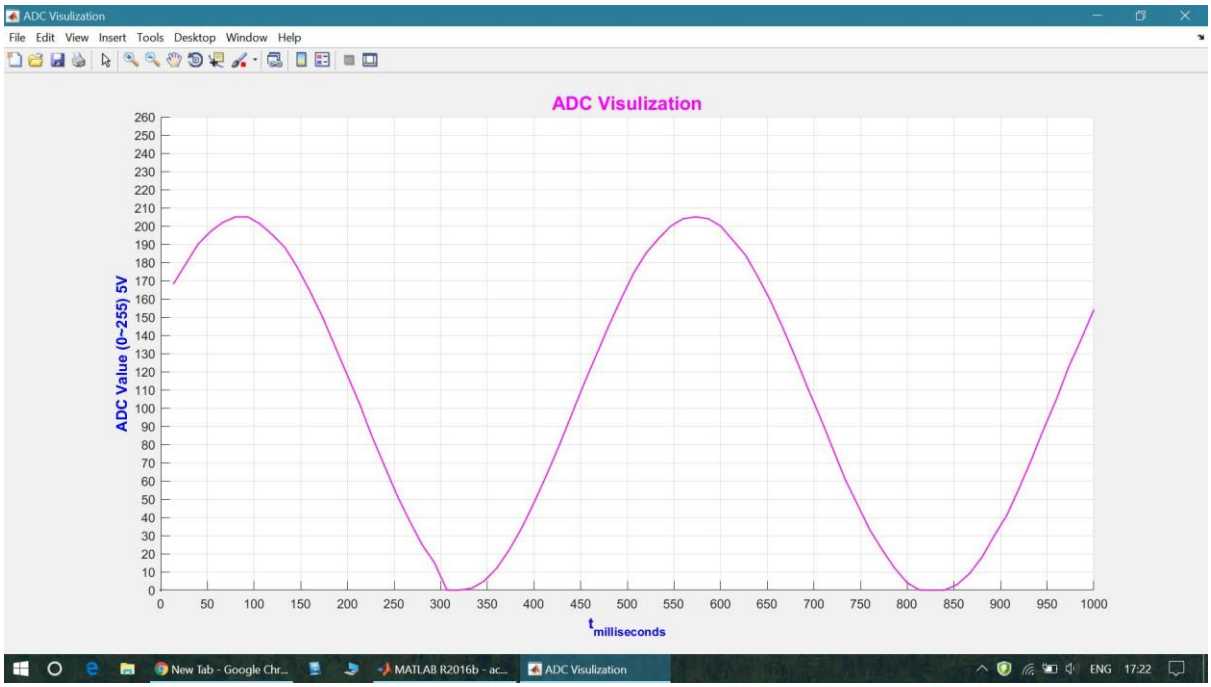


Figure 14 Graphical User Interface in MATLAB

6. Testing

The development has been done based on iterative software coding and testing procedure. The initial software components and libraries are developed based on the system requirements. Each library has been tested individually before integrating. Then the main function body has been created based on the state machine approach. We are not using any kind of real time operating/ scheduling algorithms. All the codes have been implemented in the state machine. The code is developed on interrupt driven routines rather than main function while loop to make the software efficient.

6.1. Individual Software Component Test

Module	Test Status	Reliability	Timing Aspects	Correctness	Remark
ADC Module	Success	Satisfactory	Satisfactory	Satisfactory	Pushed the clock frequency to different rates to test its capability.
UART	Success	Satisfactory	Satisfactory	Satisfactory	Pushed the baud rate to maximum to test its capability.
2 Wire Serial interface	Success	Satisfactory	Above average	Satisfactory	Used a tested 3rd party library (lcdpcf8574.h)
Hardware Interrupts	Success	Above average	Satisfactory	Above average	Used the hardware de-bounce for smoother working of the push buttons
Timer 8/16 Bit	Success	Satisfactory	Satisfactory	Satisfactory	Tested different pre-scalar and compare-overflow interrupts

Timer-PWM-Wave Generator	Success	Satisfactory	Satisfactory	Satisfactory	Altered the pre-scalar value of the FAST PWM mode in Timer 0 and achieved a precision of 62.5 ns second pulse and frequency of 62.5 KHz

6.2. System Integration and Testing

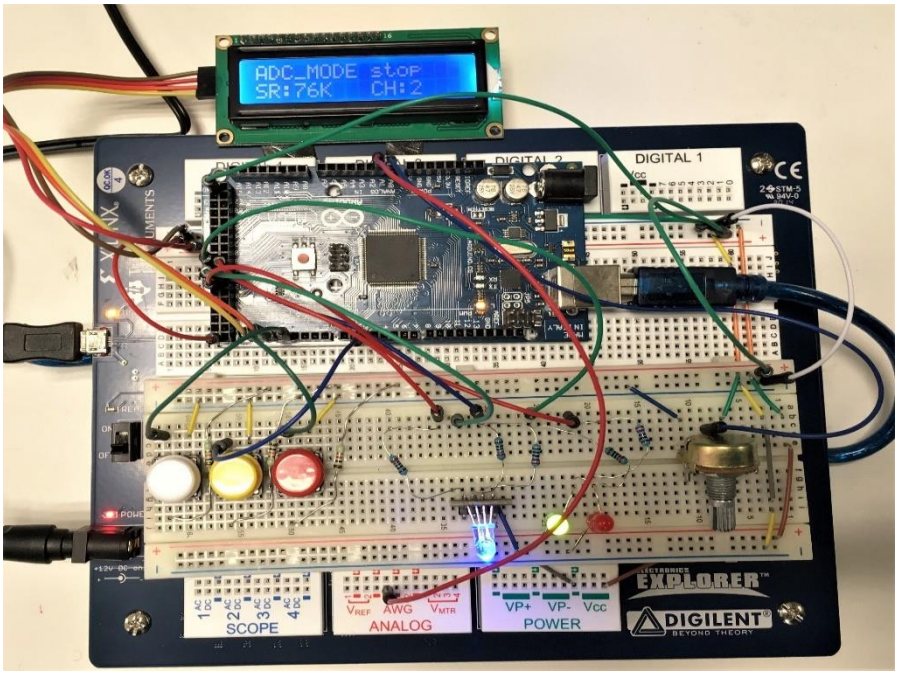


Figure 15 System Prototype

Module	Test Status	Reliability	Timing Aspects	Correctness	Remark
ADC Mode	Success	Satisfactory	Satisfactory	Satisfactory	It can sample on 4 different rates and can have 3 input channels
PWM Mode	Success	Satisfactory	Satisfactory	Satisfactory	We can have 4 frequencies and can vary the duty cycle from 0 to 100
Sleep Mode	Under development and test				This mode is aimed to save power when the device is not in use for a long time
Overall functionality	Success	Above Average	Satisfactory	Satisfactory	This should undergo more tests to improve reliability and in future this is capable of adding more features

7. Program Code Listings

- Please refer to Annex 1
- Download all source code and project files here: https://www.dropbox.com/s/wqnw7ukhud9nbv4/IO_WA.rar?dl=0

8. Critical evaluation and conclusion

Our original design is aimed to create a low-cost signal visualizer using ADC module embedded on the chip ATMEGA 2560 that controls the sampling time of the input signal. This data can be send to a PC through UART associated with AVR and be visualized in real-time waveform by a PC-based software.

We eventually implemented very well all our original design requirements. We pushed the limits of ADC module to sample the highest possible frequency up to around 152KHz (at 2MHz ADC clock) and UART to send data by up to 2Mbps (at Sys clock 16MHz). We also implemented A MATLAB algorithm to capture the serial data from UART and plot it in a real-time waveform. After carefully testing, the result meets very well with the intended design.

Moreover, we added one add-valued PWM mode – waveform generator which can generate 4-frequency (up to 62.5Khz) waveforms with adjustable duty cycles (0 -100). After testing, its performance is quite good. And finally, we also tried to implement a sleep mode aimed to save power when the device is not in use for a long time, unfortunately, due to limited time, we had no chance to fully implemented it, but our software architecture is easy to extend with more features in future.

- *Development and testing method*
  The development has been done based on iterations and test-driven method. We add and test one feature at a time, and then wrap code into a library for integration and test before moving on. What' more, our application is developed on interrupt driven routines that make the system efficient and code clearly structured. Additionally, we use GitHub to make version control in case we need to roll-back old version.

- *Improvements or enhancements*

If time permit, we would like to develop and implement more features, including two/four ADC channels automatic alternation, sleep mode, and more types of waveforms generated, and optimize the whole software architecture and source code.

All in all, we've successfully implemented all our design specification and extended it to more add-valued features. From this project, I've gained a lot of practical experience in the embedded system design, prototyping, testing, etc., and learned more programming and debugging skills with embedded C and assembly on embedded systems platforms and interfacing. All this strengthened my understanding of embedded systems concepts in both theoretical and practical way.

9. Annex 1

9.1. Libraries

```
//standard lib
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "string.h"
//private libs
#include "mylib/usart_library.h"   // for USART transmission, modified from
Richard's sample code
#include "mylib/diagnostic.h"      // for Timing diagnostic, modified from
Richard's sample code
#include "mylib/adc_library.h"     // for ADC function,
#include "mylib/pwm_library.h"     // for PWM function
#include "mylib/lcdpcf8574.h"      // 3-part lib of lcd driver
#include "mylib/pcf8574.h"         // 3-part lib of lcd driver
```

9.2. Function declarations

```
//declare functions
void InitialiseGeneral();
void Initialise_Btn_INTs();
void Enable_Btn_INTs();
void Disable_Btn_INTs();
void InitialiseTimer1();
void Initializse_LCD();
void Update_LCD();
void update_mode_data();
void update_led_state();
void update_channel_led();

void toggle_mode_change();
void toggle_channel_pwmFreq_switch();
void toggle_start_stop_evt();
void change_adc_channel(unsigned char new_channel);

//macro for bit operation
#define setbit(port, bit) (port) |= (1 << (bit))       // Set Bit ON
#define clearbit(port, bit) (port) &= ~(1 << (bit))    // Set Bit Off
#define blinkbit(port, bit) (port) ^= (1 <<(bit))      // XOR bit fro blink
```

9.3. Definitions of variable constants

```
//define ADC channels
#define CHANNEL0 0b01100000 // for potentiometer
#define CHANNEL1 0b01100001 // for ADC mode
#define CHANNEL2 0b01100010 // for ADC mode
#define CHANNEL3 0b01100011 // for ADC mode

#define SAMPLE_RATE_76K 76  //76KHz for ADC default
#define FREQ_245HZ      0   //to generate 245hz waveform
#define FREQ_1KHZ       5   //to generate 1Khz waveform
#define FREQ_7_8KHZ     40  //to generate 7.8Khz waveform
#define FREQ_62_5KHZ    65  //to generate 62.5Khz waveform

#define BAUD_2M   2000000   //baud rate default for usart setup
#define BAUD_96K  9600      //baud rate 9600;

// define LED port mapping
#define ADC_LED      DDC0   //PORTC0
#define RGB_LED_R    DDC1   //PORTC1
#define RGB_LED_G    DDC2   //PORTC2
#define RGB_LED_B    DDC3   //PORTC3
#define PWM_LED      DDC4   //PORTC4

// define H/W interrupts for buttons
#define START_STOP_BUTTON  INT2_vect
#define MODE_BUTTON        INT3_vect
#define CHANNEL_BUTTON     INT4_vect
```

9.4. Global variables and type definitions

```
// comment this line when don't using time diagnostic
#define TIMING_DIAGNOS  1

// default string length for mode information
#define STR_LEN  10

// define a struct for store mode data
typedef struct MODE_Info
{
	char mode_name[STR_LEN];    // static mode title
	char label_SRorFN[STR_LEN]; // static SR or FN string
	char label_CHorPW[STR_LEN]; // static CH or PW string
	char mode_state[STR_LEN];   // start or stop state
	char data_chpw[STR_LEN];    // real time channel or pulse width data
	char data_srfn[STR_LEN];    // real time sample rate or freq data
	uint8_t freq;               // default 5kHz for PWM
	uint8_t pulse_width;        //default 1 for PWM
	uint8_t sample_rate;        //default 76kHz      for ADC
	uint8_t new_channel;        //default 0 for ADC
} MODE_DATA;

//declare & initialize 3 modes info struct variables
MODE_DATA adc_info =
{
	.mode_name = "ADC_MODE ",
	.label_SRorFN = "SR:",
	.label_CHorPW = "CH:",
	.mode_state = "stop",
	.data_srfn = "76K  ",
	.data_chpw = "1  ",
	.new_channel = CHANNEL1,
	.sample_rate = SAMPLE_RATE_76K,
	.freq = 0,
	.pulse_width = 0
};

MODE_DATA pwm_info =
{
	.mode_name = "PWM_MODE ",
	.label_SRorFN = "FN:",
	.label_CHorPW = "PW:",
	.mode_state = "stop",
	.data_srfn = "62.5K",
	.data_chpw = "20 ",
	.freq = FREQ_62_5KHZ,
	.pulse_width = 0,
	.new_channel = 0,
	.sample_rate = 0
};

MODE_DATA sleep_info =
{
	.mode_name = "SLEEP_MODE",
	.label_SRorFN = "",
	.label_CHorPW = "",
	.mode_state = "",
	.data_srfn = "",
	.data_chpw = "",
	.freq = 0,
	.pulse_width = 0,
	.new_channel = 0,
	.sample_rate = 0
};
```

```
//declare a struct pointer link to on-going mode
static MODE_DATA* mode_info = &adc_info;

//define two basic operation modes
typedef enum  {ADC_MODE, PWM_MODE, SLEEP_MODE}  OPT_MODE;
OPT_MODE cur_mode = ADC_MODE;

//define 2 actions for button 3
typedef enum {STOP, START} BTN_ACTION;
BTN_ACTION goto_action = START;

uint8_t channel_count = 2;
uint8_t freq_count = 0;
```

9.5. Main function

```
int main(void)
{
	InitialiseGeneral();

#ifdef TIMING_DIAGNOS
	Initialise_Usart(BAUD_96K);

	Initialise_TimingDiagnostic();     // for timing diagnostic
	g_iCheckPointArrayIndex = 0;       // for timing diagnostic
	TimingDiagnostic_ResetTimer4();
	InitialiseTimer1();
	TimingDiagnostic_CheckPoint(g_iCheckPointArrayIndex++); // timing diagnostic
	TimingDiagnostic_CheckPoint(g_iCheckPointArrayIndex++); // timing diagnostic
	TimingDiagnostic_Display_CheckPoint_DataViaUSART0(g_iCheckPointArrayIndex);
#else
	Initialise_Usart(BAUD_2M);
	InitialiseTimer1();
#endif

	Initializse_LCD();
	Initialise_Btn_INTs();
	Enable_Btn_INTs();
	Initialise_ADC();
	setSamplerateADC(SAMPLE_RATE_76K);
	startADC();

	while (1)
	{
		Update_LCD();
	}
}
```

9.6. Initialise general

```
void InitialiseGeneral()
{
	//Port for connecting switches
	DDRB = 0x00;   // Configure PortB direction for Output
	PORTB = 0x00;
	//Port for connecting output LED
	DDRC = 0xFF;
	PORTC = 0x00;

	sei(); // set Global Interrupt Enable (I) bit
}
```

9.7. Initialise Timer1

```
// Configure to generate an interrupt after a 0.5-Second interval
// For 16Mhz CPU, using 256 prescaler
void InitialiseTimer1()
{
	TCCR1A = 0b00000000; // Clear Timer on 'Compare Match' (CTC) waveform mode)
	TCCR1B = 0b00001100; // CTC waveform mode, use prescaler 256
	TCCR1C = 0b00000000;

	// For 16 MHz clock (with 256 prescaler) to achieve a 1 second interval: 16M/256 =
0xF424 (62,500)
	// to achive 0.5 second interval: 16M/256 /2 = 0x7A12 (decimal 31,250)

	OCR1AH = 0x7A;
	OCR1AL = 0x12;// 0.5 second

	TCNT1H = 0b00000000; // Timer/Counter count/value registers (16 bit) TCNT1H and
TCNT1L
	TCNT1L = 0b00000000;
	TIMSK1 = 0b00000010; // bit 1 OCIE1A      Use 'Output Compare A Match' Interrupt,
}
```

9.8. Initialise LCD and update LCD

```
// initial lcd driver, and light on lcd
void Initializse_LCD()
{
	lcd_init(LCD_DISP_ON);
	lcd_home();
	lcd_led(0);
}

// This function to update the display for new mode information
void Update_LCD()
{
	if (cur_mode != SLEEP_MODE)
	{
		//static information
		lcd_gotoxy(0, 0);
		lcd_puts (mode_info->mode_name);
		lcd_gotoxy(0, 1);
		lcd_puts (mode_info->label_SRorFN);
		lcd_gotoxy(9, 1);
		lcd_puts (mode_info->label_CHorPW);

		//changeable information
		lcd_gotoxy(9,0);
		lcd_puts(mode_info->mode_state);
		lcd_gotoxy(3, 1);
		lcd_puts(mode_info->data_srfn);
		lcd_gotoxy(12,1);
		lcd_puts(mode_info->data_chpw);
		lcd_puts("  ");
	}
	else
	{
		lcd_gotoxy(0, 0);
		lcd_puts (mode_info->mode_name);
		lcd_puts ("    ");
		lcd_gotoxy(0, 1);
		lcd_puts ("                "); // clean the lcd
	}
}
```

9.9. Initialise INTs and enable disable

```
// Initialize 3 INT (2,3,4) for 3 push Button
// INT2 - Start/Stop Button, highest priority
// INT3 - Mode Button, second priority
// INT4 - Channel/frequency Button, lowest priority
void Initialise_Btn_INTs()
{
	EICRA = 0b10100000;  // falling-edge triggered for INT2 & 3
	EICRB = 0b00000010; // falling-edge triggered for INT4
	EIMSK = 0b00000000;  // Initially disabled,
	EIFR = 0b11111111;   // Clear all HW interrupt flags
}

// Enable push buttons interrupts
void Enable_Btn_INTs()
{
	EIMSK = 0b00011100;		// Enable H/W Int 2,3,4 for 3 Buttons
}

// Disable push buttons interrupts sometimes can use it for de-bounce
void Disable_Btn_INTs()
{
	EIMSK = 0b00000000;		// Disable H/W Ints
}
```

9.10. ISR of Timer1, ADC, INT2, INT3 and INT4

```
//This ISR for update mode's parameter from potentiometer input
//and led state by 0.5 second interval
ISR(TIMER1_COMPA_vect)
{
	if (goto_action != STOP) update_mode_data();
	update_led_state();
}

//This ISR to send ADC data to PC
ISR(ADC_vect)
{
   USART_TX_SingleByte(ADCH);
}


// Start/Stop Button Int for start mode settings or stop to waiting for user input
ISR(START_STOP_BUTTON) // Interrupt Handler for H/W INT 2
{
	Disable_Btn_INTs();		// Disable INT to prevent key bounce
	if (cur_mode == SLEEP_MODE)
		toggle_mode_change();
	else
		toggle_start_stop_evt();
	Enable_Btn_INTs();		// Re-enable the interrupt
}

// Mode Button Int for switch among 3 modes (ADC, PWM, SLEEP)
// by call the function of toggling_mode_change function
ISR(MODE_BUTTON)  // Interrupt Handler for H/W INT 3
{
	Disable_Btn_INTs();
	if (goto_action != STOP)
		toggle_mode_change();
	Enable_Btn_INTs();
}

//Channel Button for user changing channels of ADC or frequencies of PWM
ISR(CHANNEL_BUTTON) // Interrupt Handler for H/W INT 4
{
	Disable_Btn_INTs();
	if (cur_mode==SLEEP_MODE)
		toggle_mode_change();
	else if(goto_action != STOP)
		toggle_channel_pwmFreq_switch();
	Enable_Btn_INTs();
}
```

9.11. Implementation of other functions

```
// This function used to update Mode indicator LED
// Green LED - light on indicating ADC mode on
// Red LED - light on indicating PWM mode on
// The mode led will continue blinking with 0.5 second
// interval during starting/running state
void update_led_state()
{
	OPT_MODE temp = cur_mode;
	switch(temp)
	{
		case ADC_MODE:
			if (goto_action == START) setbit(PORTC,ADC_LED);
			else blinkbit(PORTC,ADC_LED);
			clearbit(PORTC,PWM_LED); // lit off PWM LED
			update_channel_led();
		break;
		case PWM_MODE:
			if (goto_action == START) setbit(PORTC,PWM_LED);
			else blinkbit(PORTC,PWM_LED);
			//lit off all ADC and Channel LEDs
			clearbit(PORTC,ADC_LED);
			clearbit(PORTC,RGB_LED_R);
			clearbit(PORTC,RGB_LED_G);
			clearbit(PORTC,RGB_LED_B);
		break;
		case SLEEP_MODE: //lit off all led
			clearbit(PORTC,PWM_LED);
			clearbit(PORTC,ADC_LED);
			clearbit(PORTC,RGB_LED_R);
			clearbit(PORTC,RGB_LED_G);
			clearbit(PORTC,RGB_LED_B);
		break;
	}
}

// This function used to update channel indicator LED
// Green - Channel 1
// Blue - Channel 2
// Red - Channel 3
// White - Channel 0, this channel used for potentiometer
void update_channel_led()
{
	unsigned char cur_channel = adc_info.new_channel;
	switch(cur_channel)
	{
		case CHANNEL0:
			setbit(PORTC,RGB_LED_R);    // Lit ON RGB - Red
			setbit(PORTC,RGB_LED_G);    // Lit ON RGB - Green
			setbit(PORTC,RGB_LED_B);    // Lit ON RGB - Blue
		break;
		case CHANNEL1:
			setbit(PORTC,RGB_LED_G);    // Lit ON RGB - Green
			clearbit(PORTC,RGB_LED_R);  // Lit Off Red
			clearbit(PORTC,RGB_LED_B);   // Lit off RGB
		break;
		case CHANNEL2:
			setbit(PORTC,RGB_LED_B);    // Lit ON RGB - Blue
			clearbit(PORTC,RGB_LED_G);  // Lit Off Green
			clearbit(PORTC,RGB_LED_R);  // Lit Off Red
		break;
		case CHANNEL3:
			setbit(PORTC,RGB_LED_R);    // Lit ON RGB - Red
			clearbit(PORTC,RGB_LED_G);  // Lit Off Green
			clearbit(PORTC,RGB_LED_B); // Lit Off Blue
		break;
	}
}
```

```
// This function for real-time updating mode data from potentiometer input
// Timer1 will call it with 0.5 second during user rotate potentiometer to change
// the parameters like ADC sample rate or PWM pulse-width
void update_mode_data()
{
	uint8_t temp = ADCH;
	switch(cur_mode)
	{
		case ADC_MODE:
			if (temp < 50){
				adc_info.sample_rate = 9;
				strcpy(adc_info.data_srfn," 9K  ");
			}
			else if (temp < 100){
				adc_info.sample_rate = 19;
				strcpy(adc_info.data_srfn,"19K  ");
			}
			else if (temp < 150){
				adc_info.sample_rate = 38;
				strcpy(adc_info.data_srfn,"38K  ");
			}
			else if (temp < 200){
				adc_info.sample_rate = 76;
				strcpy(adc_info.data_srfn,"76K  ");
			}
			else{
				adc_info.sample_rate = 152; //152K may cause some
problem
				strcpy(adc_info.data_srfn,"152K ");
			}
		break;

		case PWM_MODE:
			pwm_info.pulse_width = temp;
			itoa(temp, pwm_info.data_chpw, STR_LEN);
		break;
		case SLEEP_MODE:
			// do some things here for sleep mode in future
		break;
	}
}
// This function will be called by Mode Button Interrupt
// for switching mode among ACD_MODE, PWM_MODE, and SLEEP_MODE
void toggle_mode_change()
{
	OPT_MODE temp = cur_mode;
	switch(temp)
	{
		case ADC_MODE:
			cur_mode = PWM_MODE;
			change_adc_channel(CHANNEL0);	// ch0 for potentiometer
			setSamplerateADC(SAMPLE_RATE_76K); //go back default sample
			mode_info = &pwm_info;
		break;
		case PWM_MODE:
			cur_mode = SLEEP_MODE;
			mode_info = &sleep_info;
			stopADC();
		break;
		case SLEEP_MODE:
			cur_mode = ADC_MODE;
			startADC();
			mode_info = &adc_info;
		break;
	}
}
```

```
/* This function to change channel or pwm frequency by pressing channel button to switch
channels among 1,2,3 in ADC mode or switch pwm frequency settings among 245hz, 1Khz,
7.8khz, and 62.5khz in PWM mode. */
void toggle_channel_pwmFreq_switch()
{
	OPT_MODE temp = cur_mode;
	if (temp == PWM_MODE)
	{
		switch(freq_count)
		{
			case 0:
				pwm_info.freq = FREQ_245HZ;
				strncpy(pwm_info.data_srfn,"245Hz", STR_LEN);
			break;
			case 1:
				pwm_info.freq = FREQ_1KHZ;
				strncpy(pwm_info.data_srfn,"1KHz ", STR_LEN);
			break;
			case 2:
				pwm_info.freq = FREQ_7_8KHZ;
				strncpy(pwm_info.data_srfn,"7.8Kh", STR_LEN);
			break;
			case 3:
				pwm_info.freq = FREQ_62_5KHZ;
				strncpy(pwm_info.data_srfn,"62.5K", STR_LEN);
			break;
		}

		freq_count++;
		if (freq_count > 3)
			freq_count = 0;
	}
	else if(temp == ADC_MODE)
	{
		switch(channel_count)
		{
			case 0:
				adc_info.new_channel = CHANNEL0;
				strcpy(adc_info.data_chpw,"0  ");
			break;
			case 1:
				adc_info.new_channel = CHANNEL1;
				strcpy(adc_info.data_chpw,"1  ");
			break;
			case 2:
				adc_info.new_channel = CHANNEL2;
				strcpy(adc_info.data_chpw,"2  ");
			break;
			case 3:
				adc_info.new_channel = CHANNEL3;
				strcpy(adc_info.data_chpw,"3  ");
			break;
		}
		channel_count++;
		if (channel_count > 3)
			channel_count = 1;
	}
}

// Change ADC channels
void change_adc_channel(unsigned char new_channel)
{
	ADMUX = new_channel;
}
```

9.12. Libraries code

```
/*
* adc_library.c
*
* Created: 16-10-2016 20:41:18
* Author: ravim & brian
*/

// Initialise_ADC code modified from Richard's example code
#include <avr/io.h>
#include <avr/interrupt.h>
#include "adc_library.h"

void startADC()
{
	//ADCSRA |= 0b01000000;	// start ADC conversion

	ADCSRA |= (1 << ADEN); // enable adc bit 7
	ADCSRA |= (1 << ADSC); // start conversion bit 6
}

void stopADC()
{
	//ADCSRA &= 0b10111111;	// start ADC conversion

	ADCSRA &= ~(1 << ADEN); //disable adc, bit 7
	ADCSRA &= ~(1 << ADSC); //stop conversion, bit 6
}

void setSamplerateADC(uint8_t rate)
{
	switch (rate)
	{
		case 152:
		ADCSRA = 0b10101011;
		break;

		case 76:
		ADCSRA = 0b10101100;
		break;

		case 38:
		ADCSRA = 0b10101101;
		break;
		case 19:
		ADCSRA = 0b10101110;
		break;
		case 9:
		ADCSRA = 0b10101111;
		break;
	}
}

void Initialise_ADC()
{
// ADMUX ?ADC Multiplexer Selection Register
// bit7,6 Reference voltage selection (00 AREF,01 AVCC, 10 = Internal 1.1V, 11 =
Internal 2.56V)
// bit 5 ADC Left adjust the 10-bit result
```

```c
/*
 * pwm_library.c
 *
 * Created: 17-11-2016 12:23:37
 *  Author: ravim
 */

#include <avr/io.h>

#include "pwm_library.h"

void changePWMDutyCycle(int pwm,int pulseWidth)
 {
        switch (pwm)
        {
                case 0:
                OCR0A=pulseWidth;
                break;
                case 1:
                OCR0B=pulseWidth;
                break;
                case 3:
                OCR2A=pulseWidth;
                break;
                case 4:
                OCR2B=pulseWidth;
                break;
        }
 }

 void enableInvertingPWM(int pwm)
 {
        switch (pwm)
        {
                case 0:
                TCCR0A |=(1<<COM0A1);        //INVERTING MODE
                TCCR0A |=(1<<COM0A0);
                break;
                case 1:
                TCCR0A |=(1<<COM0B1);        //INVERTING MODE
                TCCR0A |=(1<<COM0B0);
                break;
                case 3:
                TCCR2A |=(1<<COM2A1);        //INVERTING MODE
                TCCR2A |=(1<<COM2A0);
                break;
                case 4:
                TCCR2A |=(1<<COM2B1);        //INVERTING MODE
                TCCR2A |=(1<<COM2B0);
                break;
        }
 }
 void enablePWM(int number , int pulseWidth, int frequency)
 {
        // DEFAULT VALUE FOR TIMER 0 AND 2
        TCNT0=0X00;
        TCNT2=0X00;

        switch (number)
        {
```

```
/*
* diagnostic.c
*
* Created: 16-11-2016 21:40:10
* Author: Brian
*/
// This lib modified from Richard's sample code
// Changed the initialise_timing.. removed USART0...
// Changed configure Time4 for 16Mhz CPU and no prescaler
// Changed TimingDiagnostic_Display_CheckPoint_DataViaUSART0()
// for display microsecond precision

#include "diagnostic.h"
#include "usart_library.h"
#include <stdio.h>
#include <stdlib.h>
// *** The Timing Diagnostic support methods are defined from here onwards ***
void Initialise_TimingDiagnostic()
{
	ConfigureTimer4_for_1msPrecisionCheckPointing();
	//USART0_SETUP_9600_BAUD_ASSUME_1MHz_CLOCK();
}

void TimingDiagnostic_CheckPoint(UC iCheckPointArrayIndex)// Record a CheckPoint value
{
	//int iCheckPoint_Value = (TCNT4H * 256) + TCNT4L;
	// 16Mhz, no prescale
	// minus the function call time around 18 clock cycles.
	int iCheckPoint_Value = ((TCNT4H * 256) + TCNT4L) - 18;
	g_iCheckpointArray[iCheckPointArrayIndex] = iCheckPoint_Value;
}

// Display CheckPoint values
// Call this method at the end of execution to avoid disturbing timing whilst recording
CheckPoints
void TimingDiagnostic_Display_CheckPoint_DataViaUSART0(int iLimit_ArrayIndex)
{
	char cArrayIndex[10];
	char cArrayValue[10];
	char cDisplayString[100];
	int lMicroseconds = 0;
	USART0_DisplayBanner_Head();

	for(int iCount = 0; iCount < iLimit_ArrayIndex; iCount++)
	{
		itoa(iCount, cArrayIndex, 10);
		itoa(g_iCheckpointArray[iCount], cArrayValue, 10);
		//lMilliseconds = ((long)g_iCheckpointArray[iCount] * 1024) / 1000;
		lMicroseconds = (int)g_iCheckpointArray[iCount]/16; // 16MHZ no
prescale, microsecond
		sprintf(cDisplayString, "          %4s        %7s    %8d", cArrayIndex,
cArrayValue, lMicroseconds);
		USART0_TX_String(cDisplayString);
	}
	USART0_DisplayBanner_Tail();
}

// Configure to generate an interrupt every 1 MilliSecond (actually every 1.024 mS)
// Thus can time events up to 2^16 milliseconds = approximately 1000 seconds, with
1.024ms precision
void ConfigureTimer4_for_1msPrecisionCheckPointing()
```

```
/*
* usart_library.c
*
* Created: 16-10-2016 20:49:13
* Author: ravim & Brian
*/
// This lib modified from part of Richard's code
// of USART and Timing diagnostic samples

#include <avr/io.h>
#include <avr/interrupt.h>
#include "usart_library.h"
#include "string.h"

#define CR 0x0D
#define LF 0x0A
#define SPACE 0x20
#define UC unsigned char

//**************USART ****************************//

// for 16Mhz
void Initialise_Usart(int32_t baudrate)
{
	// UCSR0A ?USART Control and Status Register A
	// bit 1 UX2 Double the USART TX speed (also depends Baud Rate Registers)
	UCSR0A = 0b00000010;   // Set U2X (Double USART Tx speed, to reduce clocking
error)
	UCSR0B = 0b10011000;   // RX Complete Int Enable, RX Enable, TX Enable, 8-bit
data
	UCSR0C = 0b00000111;   // Asynchronous, No Parity, 1 stop, 8-bit data, Falling
XCK edge
	UBRR0H = 0;
	switch (baudrate)
	{
		case 9600:
			UBRR0L = 207;
		break;
		case 14400:
			UBRR0L = 138;
		break;
		case 19200:
			UBRR0L=103;
		break;
		case 28800:
			UBRR0L=68;
		break;
		case 38400:
			UBRR0L=51;
		break;
		case 57600:
			UBRR0L=34;
		break;
		case 1000000:
			UBRR0L=1;
		break;
		case 2000000:
			UBRR0L=0;
		break;
	}
}
```

9.13. Matlab code

```
%% Real time ADC data ploting
% Author: Qinghui Liu, Date: 2016-11
% This script can be modified to be used on any platform by changing the
% serialPort variable.
% Example:-
% On Linux:     serialPort = '/dev/ttyS0';
% On MacOS:     serialPort = '/dev/tty.KeySerial1';
% On Windows:   serialPort = 'COM1';
%% Create the serial object
clear;
clc;
delete(instrfindall);
serialPort = 'COM5';
serialObject = serial(serialPort);
serialObject.BaudRate =2000000;
serialObject.InputBufferSize = 102400;
fopen(serialObject);
%serialObject.ReadAsyncMode = 'continuous';
%readasync(serialObject);

%% Set up the figure window
time = 0;
adc = 0;
sample = 0;
figureHandle = figure('NumberTitle','off',...
    'Name','ADC Visulization',...
    'Visible','off');
%% define X, Y axes
NX=1000;
NY=260;
XLIMS=[0 NX];
YLIMS=[0 NY];
axesHandle = axes('Parent',figureHandle,'XLim',XLIMS,'YLim',YLIMS,'YLimMode','manual',...
                  'XLimMode','manual','YGrid','on','XGrid','on');
hold on;
%plotHandle = plot(axesHandle,time,w1,time,target1,'LineWidth',2);
plotHandle = plot(axesHandle,time, adc, 'LineWidth', 1);
set(axesHandle,'YTick',[0:10:NY]);
set(axesHandle,'XTick',[0:50:NX]);
% Create xlabel
xlabel('t_{milliseconds}','FontWeight','bold','FontSize',12,'Color',[0 0 1]);

% Create ylabel
ylabel('ADC Value (0~255) 5V','FontWeight','bold','FontSize',12,'Color',[0 0 1]);

% Create title
title('ADC Visulization','FontSize',15,'Color',[1 0 1]);
hl = line('XData',time,'YData',adc,...
          'Color', [1 0 1],...
          'LineWidth',1,...
          'Parent',axesHandle);
%% ADC sample rate setting
ADC_SR = 75000;
plot_step = 1000;
plot_interval = plot_step/ADC_SR;
% interval pause for PC data collection

%% Collect data
count = 1;
set(figureHandle,'Visible','on');
set(plotHandle,'Marker','.');

while true
    sample = fread(serialObject,plot_step);
    time(count) = count * plot_interval * 1000;  %milliseconds
    adc(count) = sample(1);
    set(hl,'XData', time, 'YData',adc);
    drawnow limitrate;
    %pause(0.01);
    count = count + 1;
    if count> (1/plot_interval)
        count = 1;
        set(hl,'XData',get(hl,'XData')+1/plot_interval);
    end
    if ~ishandle(axesHandle), break, end
end
drawnow;
%% Clean up the serial object
fclose(serialObject);
delete(serialObject);
clear serialObject;
```