Swept Frequency Capacitive Touch Sensing

A BACHELOR'S MINI PROJECT

submitted in partial fulfillment

of the requirements for the completion of the ***VIth*** *semester*

of the

UNDER GRADUATE PROGRAM

in

Electronics and Communication Engineering
(B.Tech. in ECE)

Submitted by

Naman Kalkhuria – IEC20100065

Prathak Rastogi – IEC2010093

Prashant Mishra – IEC2010096

Under the Guidance of:

Dr. Ajay Singh Raghuvanshi

IIIT-Allahabad

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY
ALLAHABAD – 211 012 (INDIA)

May, 2013

CANDIDATE'S DECLARATION

I hereby declare that the work presented in this project entitled "**Swept Frequency Capacitive Touch Sensing**", submitted in the partial fulfillment of the completion of the semester VIth of Bachelor of Technology (B.Tech.) program, in Electronics and Communication Engineering at Indian Institute of Information Technology, Allahabad, is an authentic record of my original work carried out under the guidance of **Dr. Ajay Singh Raghuvanshi** due acknowledgements have been made in the text of the project to all other material used. This semester work was done in full compliance with the requirements and constraints of the prescribed curriculum.

Place: Allahabad
Date:

Naman Kalkhuria - IEC2010065
Prathak Rastogi – IEC2010093
Prashant Mishra – IEC2010096

CERTIFICATE FROM SUPERVISOR

I/We do hereby recommend that the mini project report prepared under my/our supervision by "Naman Kalkhuria (IEC2010065), Prathak Rastogi (IEC2010093), Prashant Mishra (IEC2010096) " titled "**Swept Frequency Capacitive Touch Sensing**" be **accepted** in the partial fulfillment of the requirements of the completion of VIth semester of Bachelor of Technology in Electronics and Communication Engineering **for Examination.**

Date:
Place: Allahabad, IIITA

Dr. Ajay Singh Raghuvanshi

Committee for Evaluation of the Thesis

_____ _____

_____ _____

ACKNOWLEDGEMENTS

This project could not have been accomplished without **Dr. Ajay Singh Raghuvanshi** who not only served as our supervisor but also encouraged us throughout project. He guided us through the debugging process, never accepting less than our best efforts. We thank him for all his help.

Naman Kalkhuria (IEC2010065)

Prathak Rastogi (IEC2010093)

Place: Allahabad

Prashant Mishra (IEC2010096)

Date:

B.Tech. 3rd Year, IIITA

ABSTRACT

In a typical capacitive touch sensor, a conductive object is excited by an electrical signal at a fixed frequency. The sensing circuit monitors the return signal and determines touch events by identifying changes in this signal caused by the electrical properties of the human hand touching the object.

Swept Frequency Capacitive Sensing technique that can not only detect a touch event, but also recognize complex configurations of the human hands and body. Such contextual information significantly enhances touch interaction in a broad range of applications, from conventional touchscreens to unique contexts and materials. For example, in our explorations we add touch and gesture sensitivity to the human body and liquids.

In ***Swept Frequency Capacitive Touch Sensing***, we monitor the response to capacitive human touch over a range of frequencies. Objects excited by an electrical signal respond differently at different frequencies, therefore, the changes in the return signal will also be frequency dependent. Thus, instead of measuring a single data point for each touch event, we measure a multitude of data points at different frequencies. Not only can we determine that a touch event occurred, we can also determine how it occurred. Importantly, this contextual touch information is captured through a single electrode, which could be simply the object itself.

In this project we are using Arduino board as a micro-controller unit and then we are programming it work as PWM generator and ADC to analyze the input signal in advanced touch sensing technique. For this we require the Arduino IDE as a programmer and USB board. Programming on Arduino platform using processing language which is an extension on C/C++ languages. The signal received from the Arduino board is then analyzed using the graphical capabilities of the Processing programming language using the GUINO library.

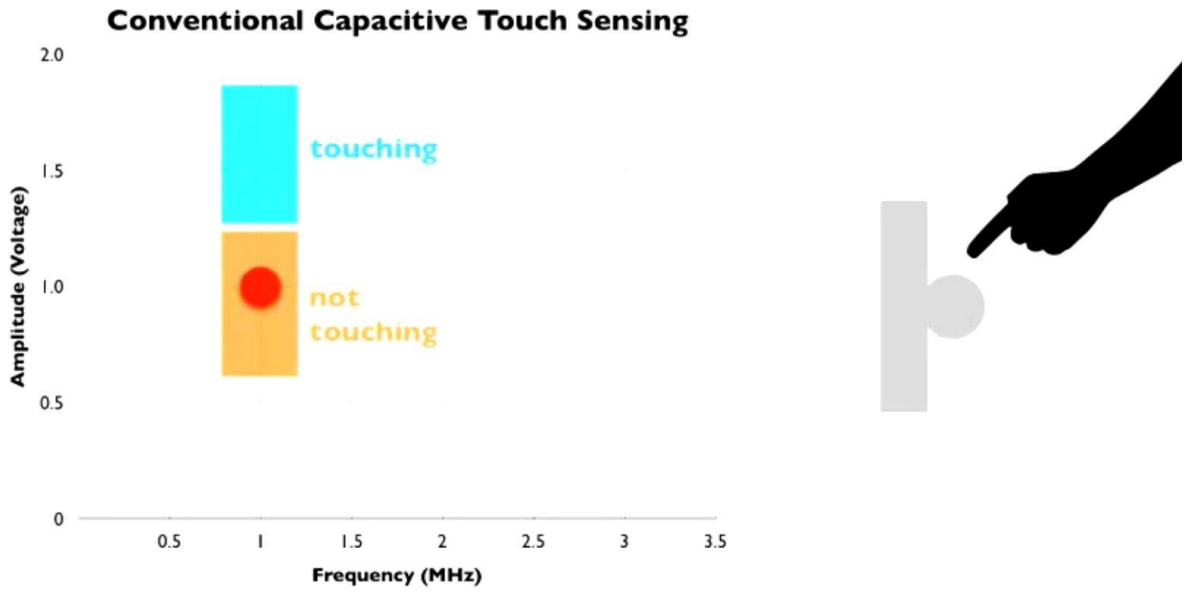
TABLE OF CONTENTS:

1. INTRODUCTION

In ***Swept Frequency Capacitive Touch Sensing***, we monitor the response to capacitive human touch over a range of frequencies. Objects excited by an electrical signal respond differently at different frequencies, therefore, the changes in the return signal will also be frequency dependent. Thus, instead of measuring a single data point for each touch event, we measure a multitude of data points at different frequencies. Not only can we determine that a touch event occurred, we can also determine how it occurred. Importantly, this contextual touch information is captured through a single electrode, which could be simply the object itself.

1.1 Currently Existing Technology:-

In a typical capacitive touch sensor, a conductive object is excited by an electrical signal at a fixed frequency. The sensing circuit monitors the return signal and determines touch events by identifying changes in this signal caused by the electrical properties of the human hand touching the object.



The basic principles of operation in most common capacitive sensing techniques are quite similar: A periodic electrical signal is injected into an electrode forming an oscillating electrical field. As the user's hand approaches the electrode, a weak capacitive link is formed

between the electrode and conductive physiological fluids inside the human hand, altering the signal supplied by the electrode. This happens because the user body introduces an additional path for flow of charges, acting as a charge "sink". By measuring the degree of this signal change, touch events can be detected.

Capacitive sensing is a malleable and inexpensive technology – all it requires is a simple conductive element that is easy to manufacture and integrate into devices or environments. Consequently, today we find capacitive touch in millions of consumer device controls and touch screens. It has, however, a number of limitations. One important limitation is that capacitive sensing is not particularly expressive – it can only detect when a finger is touching the device and sometimes infer finger proximity. To increase the expressiveness, matrices of electrodes are scanned to create a 2D capacitive image. Such space multiplexing allows the device to capture spatial gestures, hand profiles or even rough 3D shapes. However, this comes at the cost of increased engineering complexity, limiting its applications and precluding ad hoc instrumentation of our living and working spaces. Current capacitive sensors are also limited in materials they can be used with. Typically they cannot be used on the human body or liquids.

There are two types of capacitive sensing system: mutual capacitance, where the object (finger, conductive stylus) alters the mutual coupling between row and column electrodes which are scanned sequentially; and self- or absolute capacitance where the object (such as a finger) loads the sensor or increases the parasitic capacitance to ground. In both cases, the difference of a preceding absolute position from the present absolute position yields the relative motion of the object or finger during that time. The technologies are elaborated in the following section.

Capacitance is typically measured indirectly, by using it to control the frequency of an oscillator, or to vary the level of coupling (or attenuation) of an AC signal.

1.2 Analysis of Previous Research in this Area:-

Although electromagnetic signal frequency sweeps have been used for decades in wireless communication and industrial proximity sensors, we are going to use this fundamental frequency sweep method in our touch sensing.

Wireless Communication:

Radio frequency sweep or "Frequency sweep" or "RF sweep" refer to scanning a radio frequency band for detecting signals being transmitted there. This is implemented using a radio receiver having a tunable receiving frequency. As the frequency of the receiver is changed to scan (sweep) a desired frequency band, a display indicates the power of the signals received at each frequency.

Industrial Proximity Sensor:

A **proximity sensor** is a sensor able to detect the presence of nearby objects without any physical contact.

A proximity sensor often emits an electromagnetic field or a beam of electromagnetic radiation (infrared, for instance), and looks for changes in the field or return signal. The object being sensed is often referred to as the proximity sensor's target. Different proximity sensor targets demand different sensors. For example, a capacitive photoelectric sensor might be suitable for a plastic target; an inductive proximity sensor always requires a metal target.

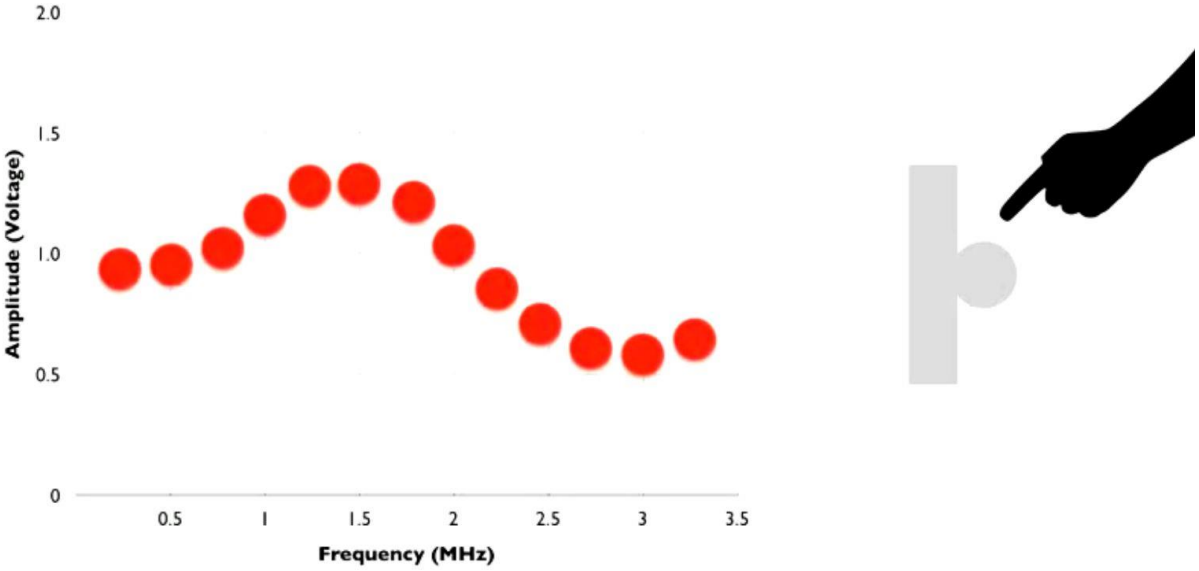
1.3 Problem Definition and Scope:-

In our day-to-day task we come in contact with various types of machines and we always yearn that these machines work according to our will without even telling them to do so. So to make this happen we need a type of interaction with these types of machines.

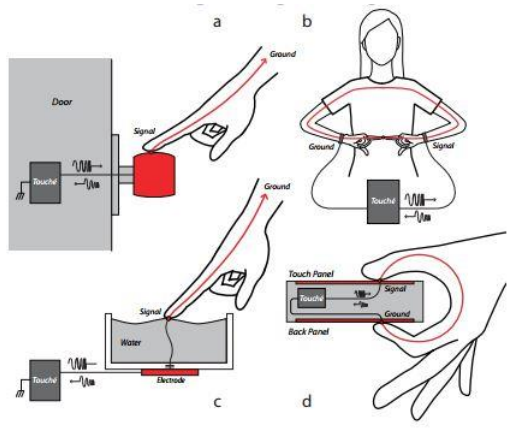
The most common type of interaction we make with objects or machines is touch. So if we can easily make an object sensitive to our touch and make them understand the way we touch; this could help in automating various tasks. By this thought this project came into our mind.

In a typical capacitive touch sensor, a conductive object is excited by an electrical signal at a fixed frequency. The sensing circuit monitors the return signal and determines touch events by identifying changes in this signal caused by the electrical properties of the human hand touching the object.

In ***Swept Frequency Capacitive Touch Sensing***, on the other hand, we monitor the response to capacitive human touch over a range of frequencies. Objects excited by an electrical signal respond differently at different frequencies, therefore, the changes in the



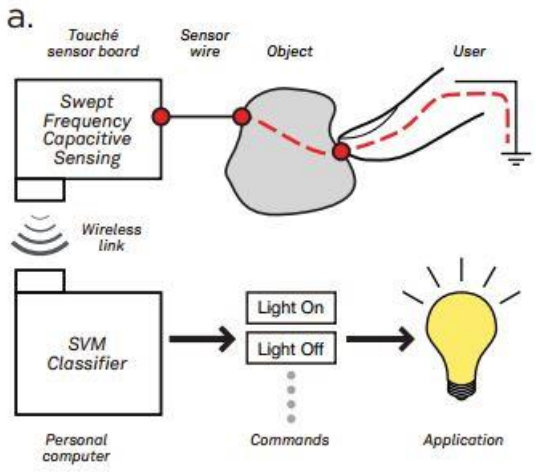
return signal will also be frequency dependent. Thus, instead of measuring a single data point for each touch event, we measure a multitude of data points at different frequencies. Not only can we determine that a touch event occurred, we can also determine how it occurred. Importantly, this contextual touch information is captured through a single electrode, which could be simply the object itself.



Our project can be implemented in following:

- For detecting the touch on a door knob.
- For detecting configuration of fingers touching to each other.
- For detecting touch on liquid surfaces.
- For detecting the touch and configuration of touch on normal touch sensitive screens.

1.4 Formulation of the Present Problem:-



Generally, *swept frequency capacitive sensing* is implemented by Digital Signal Processing and machine learning. But we are using Micro-controller for implementing this technique of sensing the touch.

In this project we implement the technique of Swept Frequency Capacitive Touch Sensing. By the help of this technique we can easily detect the touch but we can also detect the configuration of touch. We are implementing this technique using the micro-controller board instead of Digital Signal Processor.

The human body is conductive, e.g., the average internal resistance of a human trunk is ~100 Ω Skin, on the other hand, is highly resistive, ~1M Ω for dry undamaged skin. This would block any weak constant electrical (DC) signal applied to the body.

Alternating current (AC) signal, however, passes through the skin, which forms a capacitive interface between the electrode and ionic physiologic fluids inside the body. The body forms a charge "sink" with the signal flowing though tissues and bones to ground, which is also connected to the body through a capacitive link. The resistive and capacitive properties of the human body oppose the applied AC signal. This opposition, or electrical impedance changes the phase and amplitude of the AC signal. Thus, by measuring changes in the applied AC signal we can 1) detect the presence of a human body and also 2) learn about the internal composition of the body itself.

The amount of signal change depends on a variety of factors. It is affected by how a person touches the electrode, e.g., the surface area of skin touching the electrode. It is affected by the body's connection to ground, e.g., wearing or not wearing shoes or having one or both feet on the ground. Finally, it strongly depends on signal frequency. This is because at different frequencies, the AC signal will flow through different paths inside of the body. Indeed, just as DC signal flows through the path of least resistance, the AC signal will always flow through the path of least impedance.

The human body is anatomically complex and different tissues, e.g., muscle, fat and bones, have different resistive and capacitive properties. As the frequency of the AC signal changes, some of the tissues become more opposed to the flow of charges, while others less, thus changing the path of the signal flow.

Therefore, by sweeping through a range of frequencies in capacitive sensing applications, we obtain a wealth of information about 1) how the user is touching the object, 2) how the user is connected to the ground and 3) the current configuration of the human body and individual body properties. The challenge here is to reliably capture the data and then find across-user commonalities – static and temporal patterns that allow an interactive system to infer user interaction with the object, the environment, as well as the context of interaction itself.

SFCS presents an exciting opportunity to significantly expand the richness of capacitive sensing.

1.5 Organization of the Thesis:-

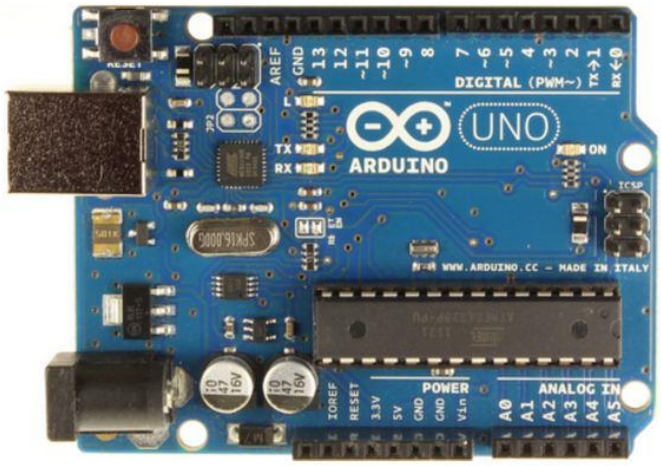
- Chapter 2, tells about the various Hardware and software we used in the preparation of the device.
- Chapter 3, gives the details of various theoretical tools we used in the project.
- Chapter 4, describes the code we used in the project.
- Chapter 5, gives the detailed photos of various Input and Output.
- Chapter 6, gives the conclusion of the project.
- Chapter 7, tells about the future work which can be done in this field.
- Chapter 8, gives the various references we used.

2. DESCRIPTION OF HARDWARE AND SOFTWARE USED

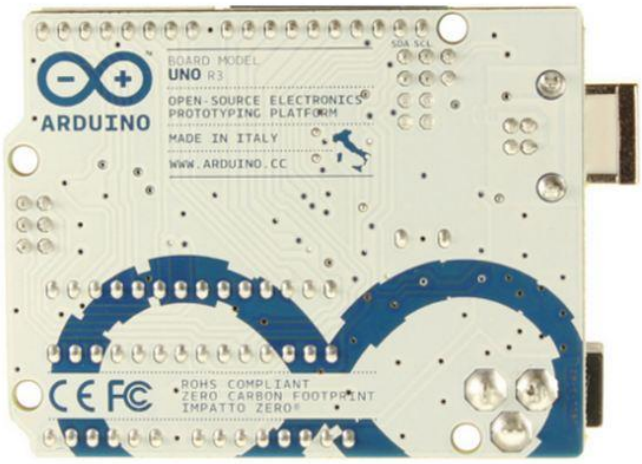
2.1 HARDWARE:

2.1.1 Micro-controller Board: Arduino Uno R3

The Arduino Uno is a microcontroller board based on the ATmega328. It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button. It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with an AC-to-DC adapter or battery to get started.



Arduino Uno R3 Front



Arduino Uno R3 Back

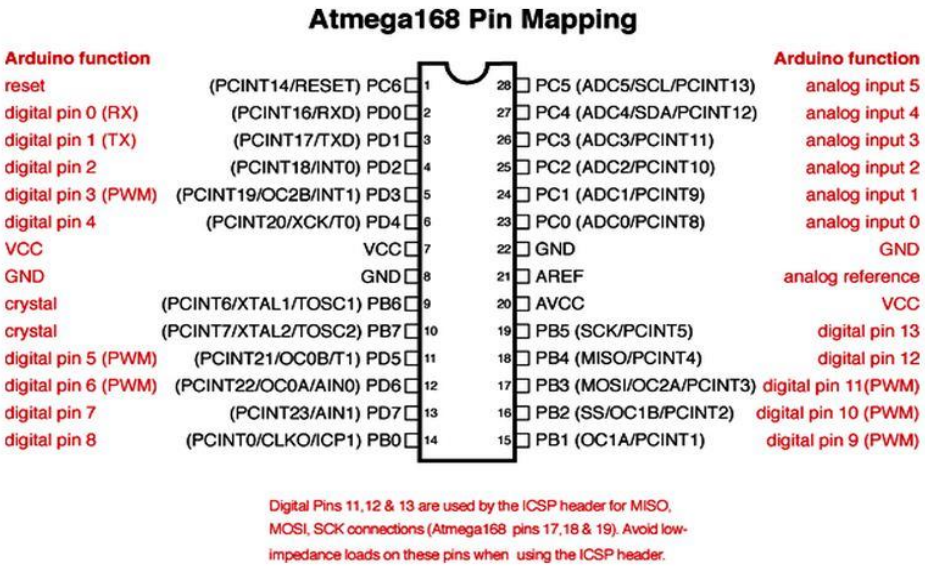
Summary

Microcontroller	ATmega328
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V

Digital I/O Pins	14 (of which 6 provide PWM output)
Analog Input Pins	6
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328) of which 0.5 KB used by boot loader
SRAM	2 KB (ATmega328)
EEPROM	1 KB (ATmega328)
Clock Speed	16 MHz

ATmega168/328-Arduino Pin Mapping

Note that this chart is for the DIP-package chip. The Arduino Mini is based upon a smaller physical IC package that includes two extra ADC pins, which are not available in the DIP-package Arduino implementations.



2.1.2 Other Components in the Sensor PCB:

Capacitor: 15nF, 100pF

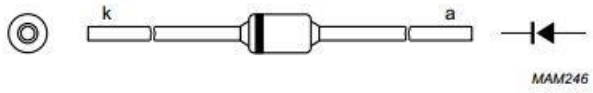
Inductor: 10mH

Resistor: 12KΩ, 1MΩ, 3.3KΩ

Diode: 1N4148

2.1.3 Diode: 1N4148

FEATURES



The diodes are type branded.

Fig.1 Simplified outline (SOD27; DO-35) and symbol.

• Hermetically sealed leaded glass SOD27 (DO-35) package

• High switching speed: max. 4 ns

• General application

• Continuous reverse voltage: max. 100 V

• Repetitive peak reverse voltage: max. 100 V

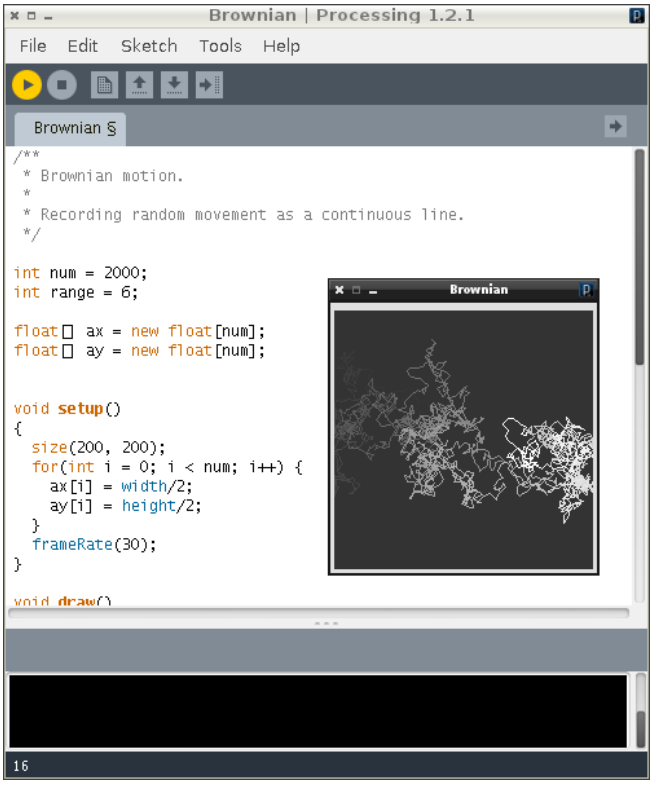
• Repetitive peak forward current: max. 450 mA.

APPLICATIONS

• High-speed switching.

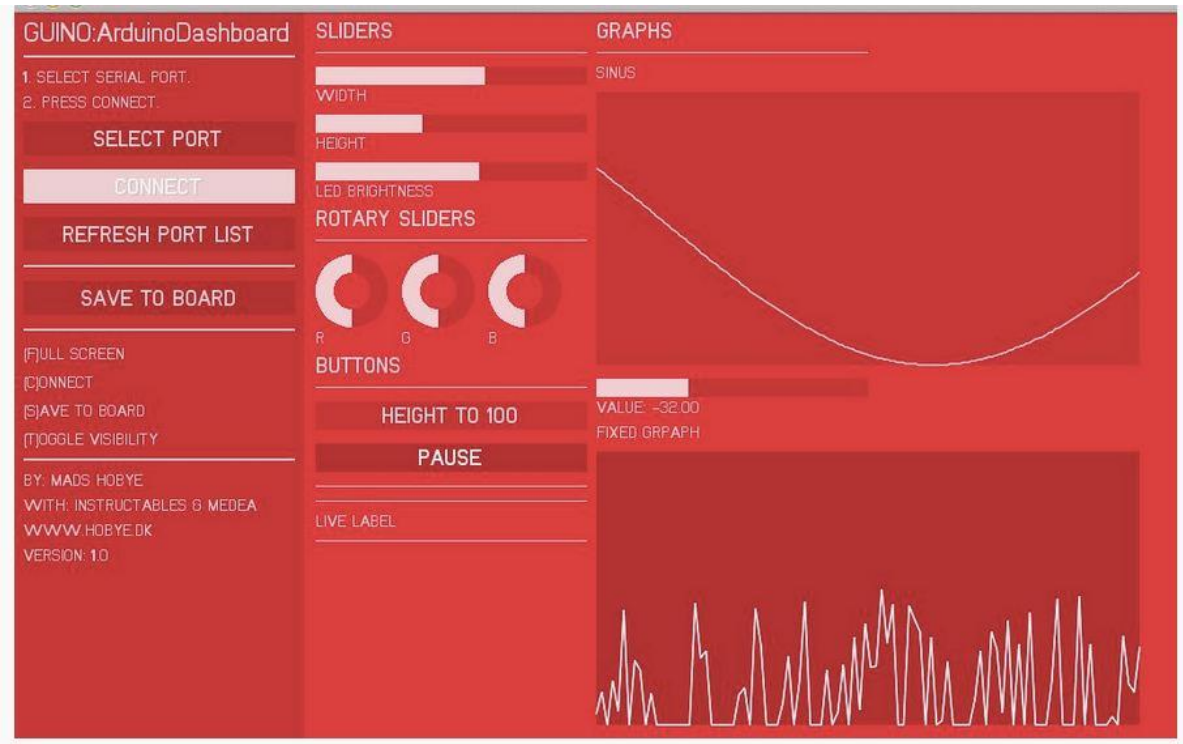
2.2 SOFTWARE:

2.2.1 Processing Programming Language:



Processing includes a *sketchbook*, a minimal alternative to an integrated development environment (IDE) for organizing projects. Processing also allows for users to create their own classes within the PApplet sketch. This allows for complex data types that can include any number of arguments and avoids the limitations of solely using standard data types such as: int (integer), char (character), float (real number), and color (RGB, ARGB, hex).

2.2.2 GUINO:



The GUINO program has the following possibilities and features:

1. Custom design your interface from the Arduino board

You define which sliders, graphs and buttons you need for your interface. You do this in your Arduino sketch which means that the gui program acts as a slave to the sketch. All information is stored in your board.

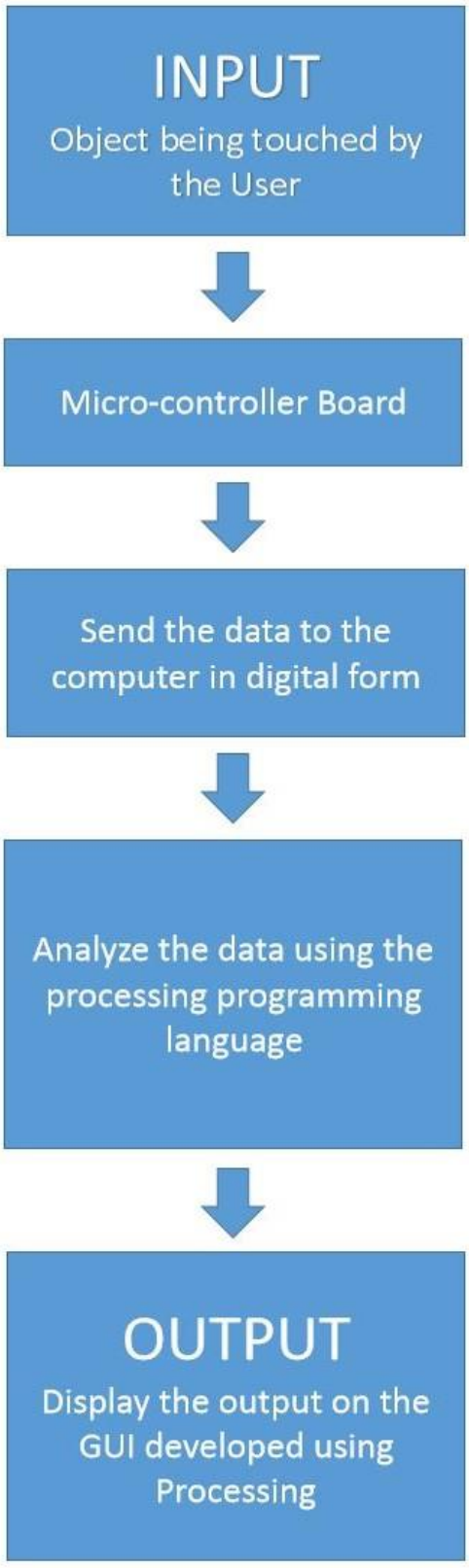
2. Visualize and manipulate realtime data

Whether you are making an RGB light controller or a robot arm, getting a graphical feedback is crucial to understand what is going on inside the board. This enables you to understand whether it is your hardware or the code that is causing problem. Further the sliders and buttons enables you to tweak the individual parameters in realtime. This way you can see what effect different thresholds have on the interaction.

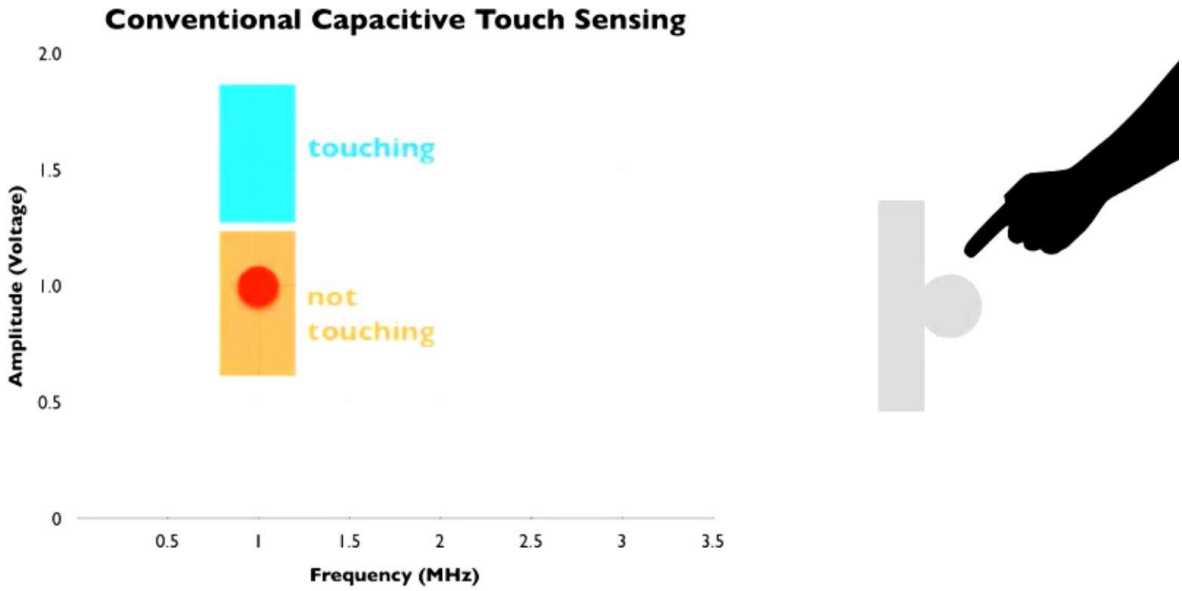
3. Save the parameters in the boards memory

When you have tweaked the parameters you can save them to the EEProm of the board. The parameters will be auto loaded next time you power on the board, even if the computer is not connected.

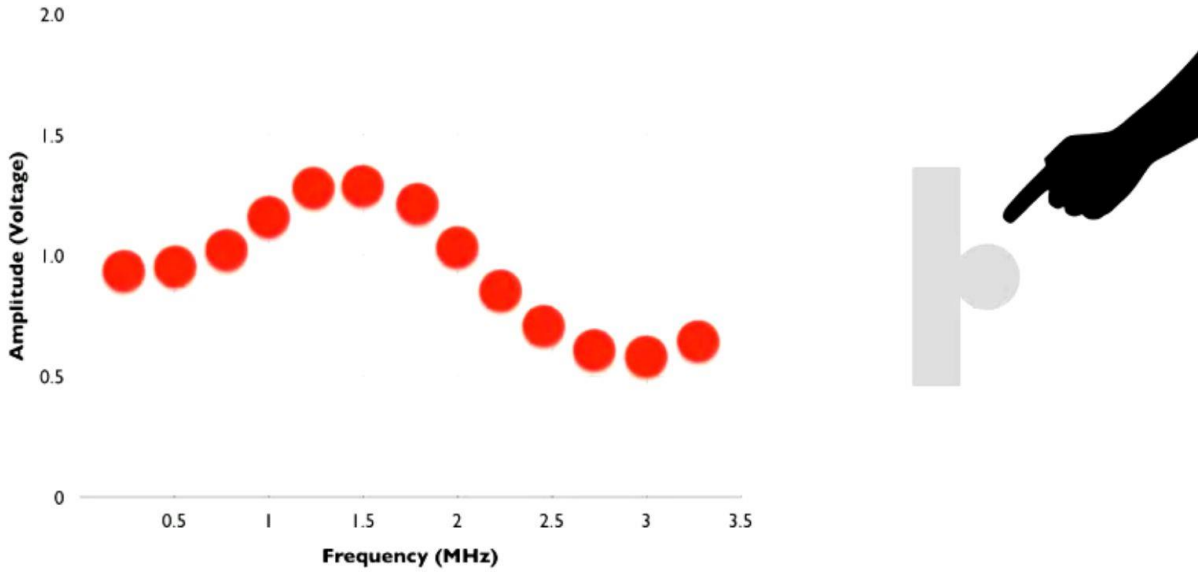
3. Theoretical Tools – Analysis and Development



In a typical capacitive touch sensor, a conductive object is excited by an electrical signal at a fixed frequency. The sensing circuit monitors the return signal and determines touch events by identifying changes in this signal caused by the electrical properties of the human hand touching the object.

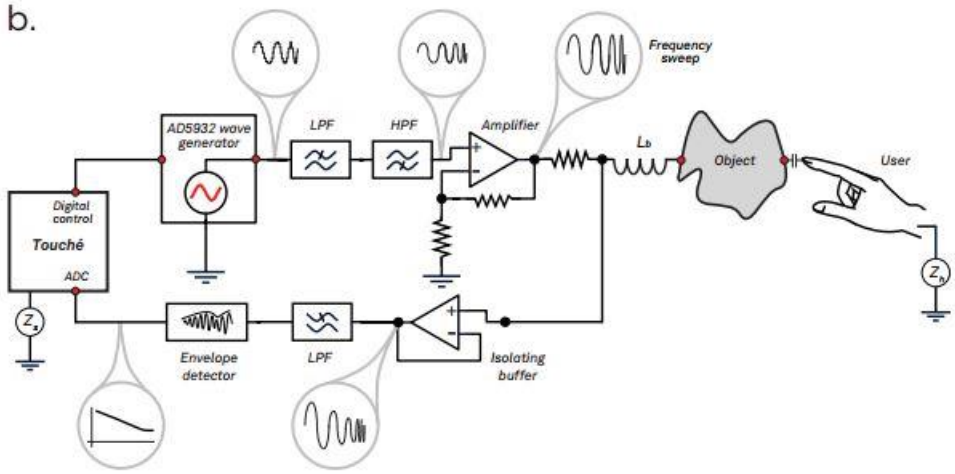


In ***Swept Frequency Capacitive Touch Sensing***, on the other hand, we monitor the response to capacitive human touch over a range of frequencies. Objects excited by an electrical signal respond differently at different frequencies, therefore, the changes in the return signal will also be frequency dependent. Thus, instead of measuring a single data point for each touch event, we measure a multitude of data points at different frequencies. Not only can we determine that a touch event occurred, we can also determine how it occurred.



Importantly, this contextual touch information is captured through a single electrode, which could be simply the object itself.

The basic principles of operation in most common capacitive sensing techniques are quite similar: A periodic electrical signal is injected into an electrode forming an oscillating electrical field. As the user's hand approaches the electrode, a weak capacitive link is formed between the electrode and conductive physiological fluids inside the human hand, altering the signal



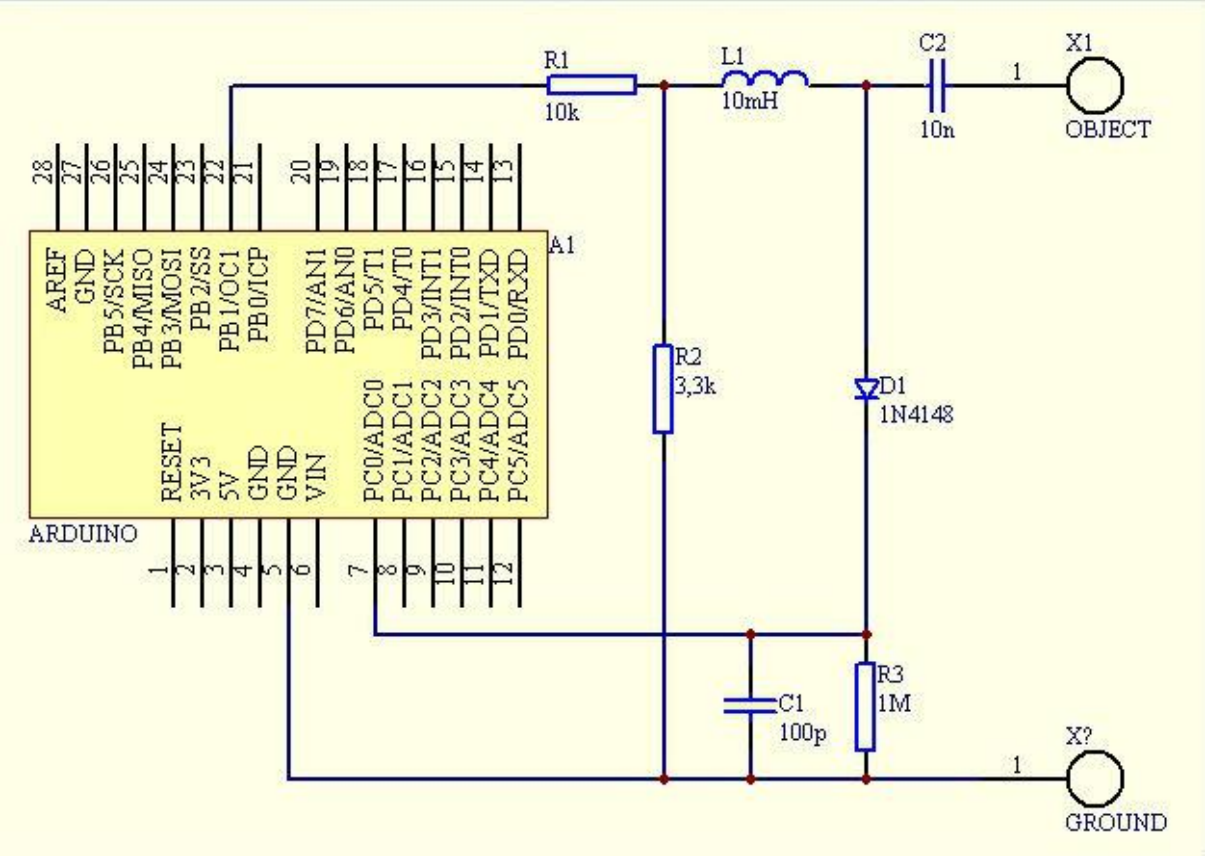
supplied by the electrode. This happens because the user body introduces an additional path for flow of charges, acting as a charge "sink". By measuring the degree of this signal change, touch events can be detected.

There is a wide variety of capacitive touch sensing techniques. One important design variable is the choice of signal property that is used to detect touch events, e.g., changes in signal phase or signal amplitude can be used for touch detection. The signal excitation technique is another important design variable. The choice of topology of electrode layouts, the materials used for electrodes and substrates and the specifics of signal measurement resulted in a multitude of capacitive techniques, including charge transfer, surface and projective capacitive, among others.

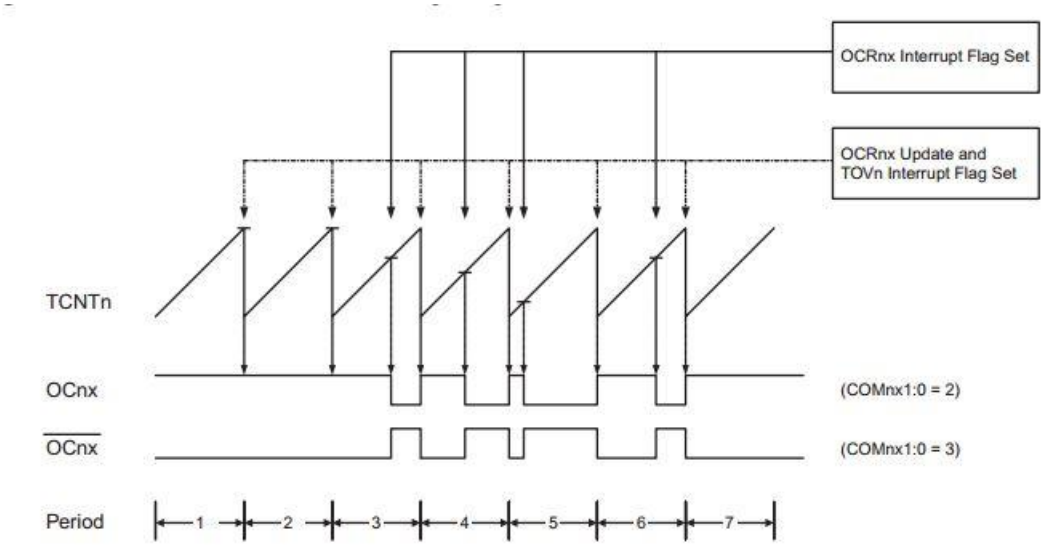
Capacitive sensing is a malleable and inexpensive technology – all it requires is a simple conductive element that is easy to manufacture and integrate into devices or environments. Consequently, today we find capacitive touch in millions of consumer device controls and touch screens. It has, however, a number of limitations. One important limitation is that capacitive sensing is not particularly expressive – it can only detect when a finger is touching the device and sometimes infer finger proximity. To increase the expressiveness, matrices of electrodes are scanned to create a 2D capacitive image. Such space multiplexing allows the device to capture spatial gestures, hand profiles or even rough 3D shapes. However, this comes at the cost of increased engineering

complexity, limiting its applications and precluding ad hoc instrumentation of our living and working spaces. Current capacitive sensors are also limited in materials they can be used with. Typically they cannot be used on the human body or liquids.

We advocate a different approach to enhance the expressivity of capacitive sensing – by using frequency multiplexing. Instead of using a single, pre-determined frequency, we sense touch by sweeping through a range of frequencies. We refer to the resulting curve as a capacitive profile and demonstrate its ability to expand the vocabulary of interactive touch without increasing the number of electrodes or the complexity of the sensor itself.



Pulse Width Modulation:



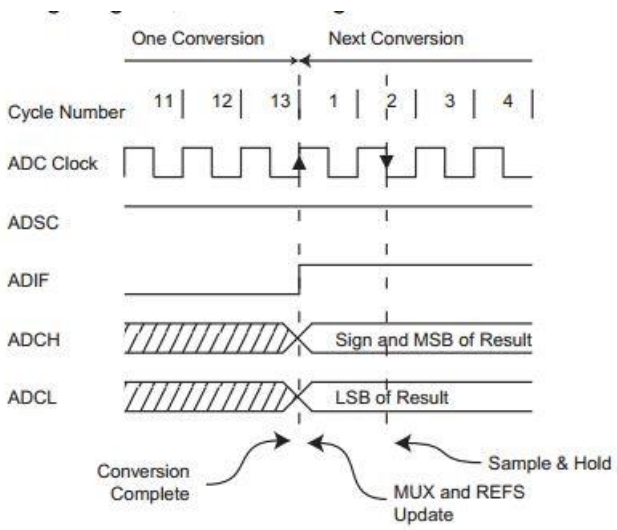
The fast Pulse Width Modulation or fast PWM mode (WGM02:0 = 3 or 7) provides a high frequency PWM waveform generation option. The fast PWM differs from the other PWM option by its single-slope operation. The counter counts from BOTTOM to TOP then restarts from BOTTOM. TOP is defined as 0xFF when WGM2:0 = 3, and OCR0A when WGM2:0 = 7. In non-inverting Compare Output mode, the Output Compare (OC0x) is cleared on the compare match between TCNT0 and OCR0x, and set at BOTTOM. In inverting Compare Output mode, the output is set on compare match and cleared at BOTTOM. Due to the single-slope operation, the operating frequency of the fast PWM mode can be twice as high as the phase correct PWM mode that use dual-slope operation. This high frequency makes the fast PWM mode well suited for power regulation, rectification, and DAC applications. High frequency allows physically small sized external components (coils, capacitors), and therefore reduces total system cost.

In fast PWM mode, the counter is incremented until the counter value matches the TOP value. The counter is then cleared at the following timer clock cycle. The timing diagram for the fast PWM mode is shown in Figure 14-6. The TCNT0 value is in the timing diagram shown as a histogram for illustrating the single-slope operation. The diagram includes non-inverted and inverted PWM outputs. The small horizontal line marks on the TCNT0 slopes represent compare matches between OCR0x and TCNT0.

Analog and Digital Converter:

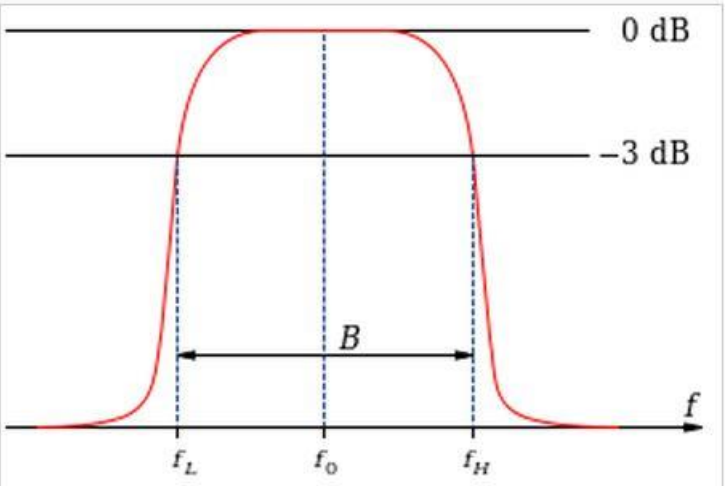
By default, the successive approximation circuitry requires an input clock frequency between 50 kHz and 200 kHz to get maximum resolution. If a lower resolution than 10 bits is needed, the input clock frequency to the ADC can be higher than 200 kHz to get a higher sample rate.

The ADC module contains a prescaler, which generates an acceptable ADC clock frequency from any CPU frequency above 100 kHz. The prescaling is set by the ADPS bits in ADCSRA. The prescaler starts counting from the moment the ADC is switched on by setting the ADEN bit in ADCSRA. The prescaler keeps running for as long as the ADEN bit is set, and is continuously reset when ADEN is low. When initiating a single ended conversion by setting the ADSC bit in ADCSRA, the conversion starts at the following rising edge of the ADC clock cycle.



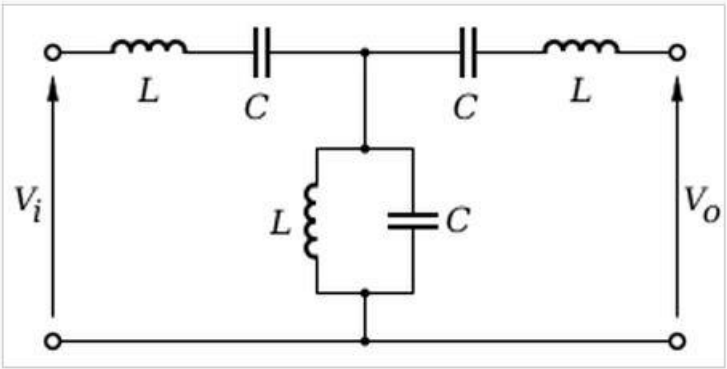
A normal conversion takes 13 ADC clock cycles. The first conversion after the ADC is switched on (ADEN in ADCSRA is set) takes 25 ADC clock cycles in order to initialize the analog circuitry. When the bandgap reference voltage is used as input to the ADC, it will take a certain time for the voltage to stabilize. If not stabilized, the first value read after the first conversion may be wrong.

Band Pass Filter:



A **band-pass filter** is a device that passes frequencies within a certain range and rejects (attenuates) frequencies outside that range. An example of an analogue electronic band-pass filter is an RLC circuit (a resistor–inductor–capacitor circuit).

These filters can also be created by combining a low-pass filter with a high-pass filter. *Bandpass* is an adjective that describes a type of filter or filtering process; it is to be distinguished from passband, which refers to the actual portion of affected spectrum. Hence, one might say "A dual bandpass filter has two passbands." A *bandpass signal* is a signal containing a band of frequencies away from zero frequency, such as a signal that comes out of a bandpass filter. An ideal bandpass filter would have a completely flat passband (e.g. with no gain/attenuation throughout) and would completely attenuate all frequencies outside the passband. Additionally, the transition out of the passband would be instantaneous in frequency. In practice, no bandpass filter is ideal. The filter does not attenuate all frequencies outside the desired frequency range completely; in particular, there is a region just outside the intended passband where frequencies are attenuated, but not rejected. This is known as the filter roll-off, and it is usually expressed in dB of attenuation per octave or decade of frequency. Generally, the design of a filter seeks to make the roll-off as narrow as possible, thus allowing the filter to perform as close as possible to its intended design. Often, this is achieved at the expense of pass-band or stop-band *ripple*.

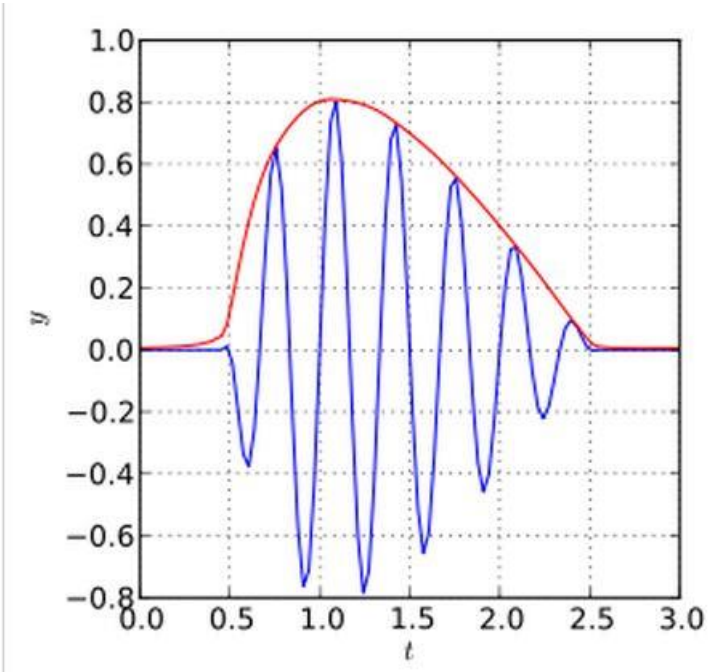


The bandwidth of the filter is simply the difference between the upper and lower cutoff frequencies. The shape factor is the ratio of bandwidths measured using two

different attenuation values to determine the cutoff frequency, e.g., a shape factor of 2:1 at 30/3 dB means the bandwidth measured between frequencies at 30 dB attenuation is twice that measured between frequencies at 3 dB attenuation.

A band-pass filter can be characterised by its Q-factor. The Q-factor is the inverse of the fractional bandwidth. A high-Q filter will have a narrow passband and a low-Q filter will have a wide passband. These are respectively referred to as narrow-band and wide-band filters.

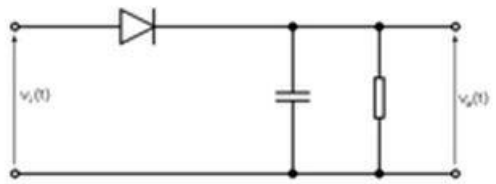
Envelope Detector:



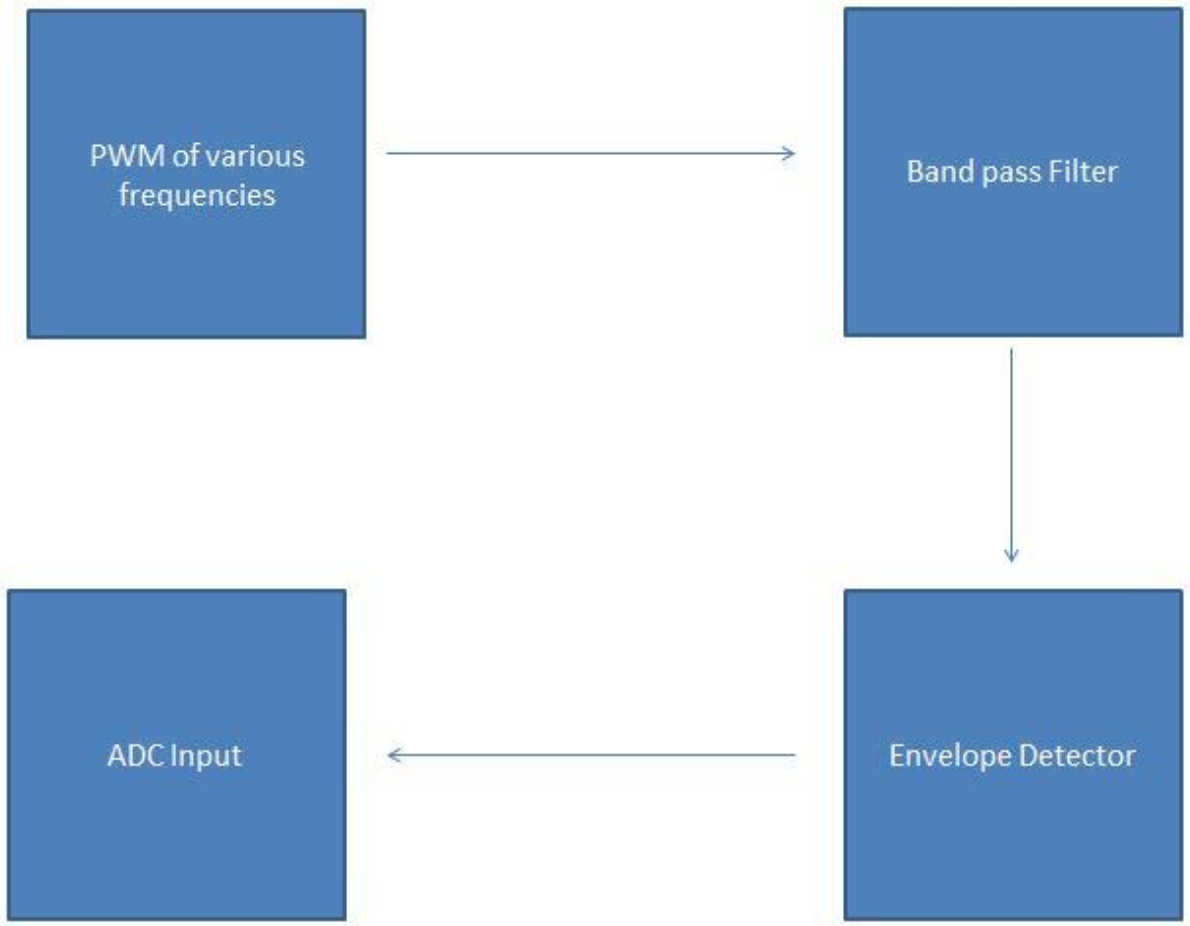
An **envelope detector** is an electronic circuit that takes a high-frequency signal as input and provides an output which is the envelope of the original signal. The capacitor in the circuit stores up charge on the rising edge, and releases it slowly through the resistor when the signal falls. The diode in series rectifies the incoming signal, allowing current flow only when the positive input terminal is at a higher potential than the negative input terminal.

Most practical envelope detectors use either half-wave or full-wave rectification of the signal to convert the AC audio input into a pulsed DC signal. Filtering is then used to smooth the final result. This filtering is rarely perfect and some "ripple" is likely to remain on the envelope follower output, particularly for low frequency inputs such as notes from a bass

guitar. More filtering gives a smoother result, but decreases the responsiveness; thus, real-world designs must be optimized for the application.



The simplest form of envelope detector is the **diode detector** which is shown above. A diode detector is simply a diode between the input and output of a circuit, connected to a resistor and capacitor in parallel from the output of the circuit to the ground. If the resistor and capacitor are correctly chosen, the output of this circuit should approximate a voltage-shifted version of the original (baseband) signal. A simple filter can then be applied to filter out the DC component.



4. DEVELOPMENT OF SOFTWARE

CODE of the Project:

1. Touch.ino

```
//************************************************************************
**********************
// Swept Frequency Capacitive Touch Sensing using Arduino
// Naman - IEC2010065
// Prathak - IEC2010093
// Prashant - IEC2010096
//************************************************************************
**********************


#define SET(x,y) (x |=(1<<y))                    //-Bit set/clear macros
#define CLR(x,y) (x &= (~(1<<y)))                // |
#define CHK(x,y) (x & (1<<y))                    // |
#define TOG(x,y) (x^=(1<<y))                     //-+


#define N 120  //How many frequencies

float results[N];          //-Filtered result buffer
float freq[N];          //-Filtered result buffer
int sizeOfArray = N;
int fixedGraph = 0;
int topPoint = 0;
int topPointValue = 0;
int topPointInterPolated = 0;
int baseline = 0;
```

```
int value = 0;
int width=0, height=0;
int r=0, g=0, b=0;
int flexLabelId=0;
int t=0;


void setup(){
  pinMode(13, OUTPUT);
  digitalWrite(13,LOW);
  // Start the guino dashboard interface.
  // The number is your personal key for saving data. This should be unique for each
sketch
  // This key should also be changed if you change the gui structure. Hence the saved
data vill not match.
  gBegin(34526);

  TCCR1A=0b10000010;        //-Set up frequency generator
  TCCR1B=0b00011001;        //-+

  ICR1=110;
  OCR1A=55;


  pinMode(9,OUTPUT);        //-Signal generator pin
  pinMode(8,OUTPUT);        //-Sync (test) pin


  for(int i=0;i<N;i++)      //-Preset results
    results[i]=0;         //-+
}


void loop(){
```

```
// **** Main update call for the guino
unsigned int d;

int counter = 0;
 topPoint = 0;
 topPointValue = 0;

for(unsigned int d=0;d<N;d++)
{

  int v=analogRead(0);    //-Read response signal
  CLR(TCCR1B,0);          //-Stop generator
  TCNT1=0;                //-Reload new frequency
  ICR1=d;                 // |
  OCR1A=d/2;              //-+
  SET(TCCR1B,0);          //-Restart generator

  delayMicroseconds(1);
  results[d]=results[d]*0.5+(float)(v)*0.5; //Filter results
  if (topPointValue < results[d])
  {
    topPointValue = results[d];
    topPoint =d;
  }

  freq[d] = d;
fixedGraph = round(results[d]);
gUpdateValue(&fixedGraph);
delayMicroseconds(1000);
}
//topPointInterPolated =topPointInterPolated * 0.5f + ((topPoint+
results[topPoint]/results[topPoint+1]*results[topPoint-
1]/results[topPoint])*10.0f)*0.5f;
```

```
  value = topPoint - baseline;
  guino_update();
  gUpdateValue(&topPoint);
  gUpdateValue(&value);
  //gUpdateValue(&topPointInterPolated);
}

// This is where you setup your interface
void gInit()
{
  gAddLabel("Swept Frequency Capacitive Touch Sensing",1);

   gAddSpacer(1);


   gAddSpacer(1);
   //gAddFixedGraph("FIXED GRPAPH",-500,1000,N,&fixedGraph,40);
   gAddSlider(0,N,"TOP",&topPoint);
   //gAddSlider(0,N*10,"Interpolated",&topPointInterPolated);
   gAddSlider(0,800,"Baseline",&baseline);
   gAddSlider(0,300,"Value",&value);


  // The rotary sliders
  gAddLabel("ROTARY SLIDERS",1);
  gAddSpacer(1);

  gAddRotarySlider(0,255,"R",&r);
  gAddRotarySlider(0,255,"G",&g);
  gAddRotarySlider(0,255,"B",&b);
  gSetColor(r,g,b); // Set the color of the gui interface.

  gAddColumn();
```

```
 gAddLabel("GRAPH",1);
 gAddSpacer(1);

 // Last parameter in moving graph defines the size 10 = normal
 //gAddMovingGraph("SINUS",-100,100, &graphValue, 20);
 //gAddSlider(-100,100,"VALUE",&graphValue);
   gAddFixedGraph("OUTPUT GRAPH",-175,175,N,&fixedGraph,30);
 //gAddFixedGraph("Interpolated GRAPH",-175,175,N,&topPointInterPolated,30);
 // The graphs take up two columns we are going to add two
 gAddColumn();
 gAddColumn();
 // Add more stuff here.
 t = topPoint;

 gAddSpacer(1);
 if(t<=47)
   flexLabelId = gAddLabel("NO TOUCH",2);
 else if(t>=49 && t<=52)
   flexLabelId = gAddLabel("ONE FINGER TOUCH",2);
 else if(t>=54 && t<=61)
   flexLabelId = gAddLabel("FINGER PINCH",2);
 else if(t>=63 && t<=77)
   flexLabelId = gAddLabel("GRAB",2);
 else if(t>=85)
   flexLabelId = gAddLabel("FINGER IN WATER",2);
 gAddSpacer(1);

}
```

2. GuinoLibrary.ino

```
/*
  GUINO DASHBOARD TEMPLATE FOR THE ARDUINO.
  Done by Mads Hobye as a part of Instructables (AIR Program) & Medea (PhD Student).
  Licens: Creative Commons — Attribution-ShareAlike

  It should be used with the GUINO Dashboard app.

  More info can be found here: www.hobye.dk

  # This is the Guino Protocol Library should only be edited if you know what you are doing.
*/

#include <EasyTransfer.h>
#include <EEPROM.h>

#define guino_executed -1
#define guino_init 0
#define guino_addSlider 1
#define guino_addButton 2
#define guino_iamhere 3
#define guino_addToggle 4
#define guino_addRotarySlider 5
#define guino_saveToBoard 6
#define guino_setFixedGraphBuffer 8
#define guino_clearLabel 7
#define guino_addWaveform 9
#define guino_addColumn 10
#define guino_addSpacer 11
#define guino_addMovingGraph 13
#define guino_buttonPressed 14
```

```
#define guino_addChar 15
#define guino_setMin 16
#define guino_setMax 17
#define guino_setValue 20
#define guino_addLabel 12
#define guino_large 0
#define guino_medium 1
#define guino_small 2
#define guino_setColor  21

boolean guidino_initialized = false;

//This function will write a 2 byte integer to the eeprom at the specified address and address + 1
void EEPROMWriteInt(int p_address, int p_value)
{
  byte lowByte = ((p_value >> 0) & 0xFF);
  byte highByte = ((p_value >> 8) & 0xFF);

  EEPROM.write(p_address, lowByte);
  EEPROM.write(p_address + 1, highByte);
}

//This function will read a 2 byte integer from the eeprom at the specified address and address + 1
unsigned int EEPROMReadInt(int p_address)
{
  byte lowByte = EEPROM.read(p_address);
  byte highByte = EEPROM.read(p_address + 1);

  return ((lowByte << 0) & 0xFF) + ((highByte << 8) & 0xFF00);
}

//create object
```

```
EasyTransfer ET;

struct SEND_DATA_STRUCTURE
{
  //put your variable definitions here for the data you want to send
  //THIS MUST BE EXACTLY THE SAME ON THE OTHER ARDUINO
  char cmd;
  char item;
  int value;
};

// Find a way to dynamically allocate memory
int guino_maxGUIItems = 100;
int guino_item_counter = 0;
int *guino_item_values[100];
int gTmpInt = 0; // temporary int for items without a variable
boolean internalInit = true; // boolean to initialize before connecting to serial

// COMMAND STRUCTURE

//give a name to the group of data
SEND_DATA_STRUCTURE guino_data;
int eepromKey = 1234;
void guino_update()
{

  while(Serial.available())
  {

    if(ET.receiveData())
    {
      switch (guino_data.cmd)
      {
      case guino_init:
```

```
      guino_item_counter = 0;
      guidino_initialized = true;
      gInit();
      break;
    case guino_setValue:
      *guino_item_values[guino_data.item] = guino_data.value;
      guino_data.cmd = guino_executed;
      break;
    case guino_buttonPressed:
      gButtonPressed(guino_data.item);
      break;
    case guino_saveToBoard:
      {

        gInitEEprom();
        for (int i =0; i < guino_item_counter;i++)
        {
          EEPROMWriteInt(i*2+2, *guino_item_values[i]);
        }
      }
      break;
    }
   }
  }
}

void gInitEEprom()
{
  if(EEPROMReadInt(0) != eepromKey)
  {
    EEPROMWriteInt(0, eepromKey);
    for (int i =1; i <guino_maxGUIItems;i++)
    {
```

```
    EEPROMWriteInt(i*2+2, -3276);
   }
  }

}

void gGetSavedValue(int item_number, int *_variable)
{

  if(EEPROMReadInt(0) == eepromKey && internalInit)
  {

   int tmpVar =  EEPROMReadInt((item_number)*2+2);
   if(tmpVar != -3276)
    *_variable = tmpVar;
  }

}

void gBegin(int _eepromKey)
{

  // Sets all pointers to a temporary value just to make sure no random memory
pointers.
  for(int i = 0; i < guino_maxGUIItems; i++)
  {
   guino_item_values[i] = &gTmpInt;
  }
  eepromKey = _eepromKey;

  gInit(); // this one needs to run twice only way to work without serial connection.
  internalInit = false;
  Serial.begin(115200);
  ET.begin(details(guino_data), &Serial);
```

```
  gSendCommand(guino_executed, 0, 0);
  gSendCommand(guino_executed, 0, 0);
  gSendCommand(guino_executed, 0, 0);
  gSendCommand(guino_iamhere, 0, 0);

}

void gSetColor(int _red, int _green, int _blue)
{
  gSendCommand(guino_setColor, 0, _red);
  gSendCommand(guino_setColor, 1, _green);
  gSendCommand(guino_setColor, 2, _blue);
}

int gAddButton(char * _name)
{
  if(guino_maxGUIItems > guino_item_counter)
  {
    gSendCommand(guino_addButton,(byte)guino_item_counter,0);
    for (int i = 0; i < strlen(_name); i++){
      gSendCommand(guino_addChar,(byte)guino_item_counter,(int)_name[i]);
    }
    guino_item_counter++;
    return guino_item_counter-1;
  }
  return -1;
}


void gAddColumn()
{

  gSendCommand(guino_addColumn,0,0);
```

```
}



int gAddLabel(char * _name, int _size)
{
 if(guino_maxGUIItems > guino_item_counter)
 {
  gSendCommand(guino_addLabel,(byte)guino_item_counter,_size);

  for (int i = 0; i < strlen(_name); i++){
   gSendCommand(guino_addChar,(byte)guino_item_counter,(int)_name[i]);
  }

  guino_item_counter++;

  return guino_item_counter-1;
 }
 return -1;



}
int gAddSpacer(int _size)
{
 if(guino_maxGUIItems > guino_item_counter)
 {
  gSendCommand(guino_addSpacer,(byte)guino_item_counter,_size);

  guino_item_counter++;
  return guino_item_counter-1;
 }
 return -1;
```

```
}



int gAddToggle(char * _name, int * _variable)
{
 if(guino_maxGUIItems > guino_item_counter)
 {
  guino_item_values[guino_item_counter] =_variable ;
  gGetSavedValue(guino_item_counter, _variable);
  gSendCommand(guino_addToggle,(byte)guino_item_counter,*_variable);

  for (int i = 0; i < strlen(_name); i++){
    gSendCommand(guino_addChar,(byte)guino_item_counter,(int)_name[i]);
  }

  guino_item_counter++;

  return guino_item_counter-1;


 }
 return -1;
}

int gAddFixedGraph(char * _name,int _min,int _max,int _bufferSize, int * _variable,
int _size)
{
 if(guino_maxGUIItems > guino_item_counter)
 {
  gAddLabel(_name,guino_small);
  guino_item_values[guino_item_counter] =_variable ;
  gGetSavedValue(guino_item_counter, _variable);
  gSendCommand(guino_addWaveform,(byte)guino_item_counter,_size);
```

```
    gSendCommand(guino_setMax,(byte)guino_item_counter,_max);
    gSendCommand(guino_setMin,(byte)guino_item_counter,_min);

gSendCommand(guino_setFixedGraphBuffer,(byte)guino_item_counter,_bufferSize);



    guino_item_counter++;

    return guino_item_counter-1;
  }
  return -1;
}

int gAddMovingGraph(char * _name,int _min,int _max, int * _variable, int _size)
{
  if(guino_maxGUIItems > guino_item_counter)
  {
    gAddLabel(_name,guino_small);
    guino_item_values[guino_item_counter] =_variable ;
    gGetSavedValue(guino_item_counter, _variable);
    gSendCommand(guino_addMovingGraph,(byte)guino_item_counter,_size);
    gSendCommand(guino_setMax,(byte)guino_item_counter,_max);
    gSendCommand(guino_setMin,(byte)guino_item_counter,_min);



    guino_item_counter++;

    return guino_item_counter-1;
  }
  return -1;



}
```

```
int gUpdateLabel(int _item, char * _text)
{

  gSendCommand(guino_clearLabel,(byte)_item,0);
  for (int i = 0; i < strlen(_text); i++){
    gSendCommand(guino_addChar,(byte)_item,(int)_text[i]);
  }




}



int gAddRotarySlider(int _min,int _max, char * _name, int * _variable)
{
  if(guino_maxGUIItems > guino_item_counter)
  {
    guino_item_values[guino_item_counter] =_variable ;
    gGetSavedValue(guino_item_counter, _variable);
    gSendCommand(guino_addRotarySlider,(byte)guino_item_counter,*_variable);
    gSendCommand(guino_setMax,(byte)guino_item_counter,_max);
    gSendCommand(guino_setMin,(byte)guino_item_counter,_min);
    for (int i = 0; i < strlen(_name); i++){
      gSendCommand(guino_addChar,(byte)guino_item_counter,(int)_name[i]);
    }

    guino_item_counter++;

    return guino_item_counter-1;
  }
  return -1;
```

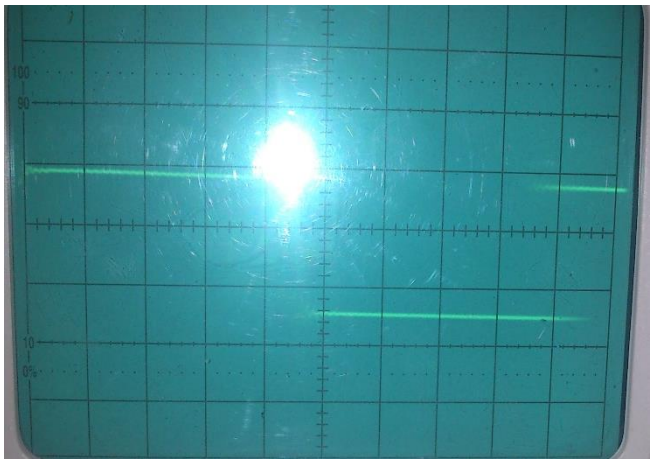
```
}

int gAddSlider(int _min,int _max, char * _name, int * _variable)
{
 if(guino_maxGUIItems > guino_item_counter)
 {
  guino_item_values[guino_item_counter] =_variable ;
  gGetSavedValue(guino_item_counter, _variable);
  gSendCommand(guino_addSlider,(byte)guino_item_counter,*_variable);
  gSendCommand(guino_setMax,(byte)guino_item_counter,_max);
  gSendCommand(guino_setMin,(byte)guino_item_counter,_min);
  for (int i = 0; i < strlen(_name); i++){
    gSendCommand(guino_addChar,(byte)guino_item_counter,(int)_name[i]);
  }

  guino_item_counter++;

  return guino_item_counter-1;
 }
 return -1;

}

void gUpdateValue(int _item)
{

 gSendCommand(guino_setValue,_item, *guino_item_values[_item]);
}

void gUpdateValue(int * _variable)
{

 int current_id = -1;
 for(int i = 0; i < guino_item_counter; i++)
```

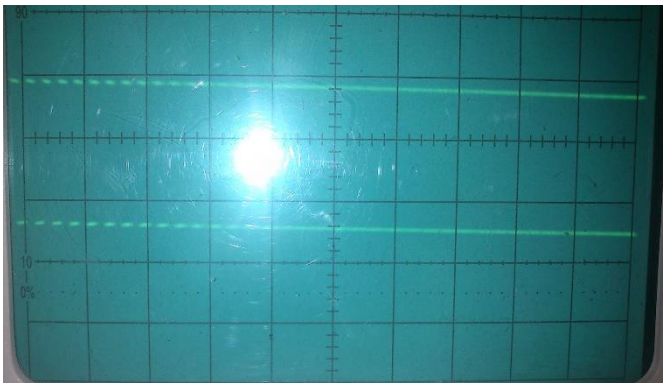
```
  {

    if(guino_item_values[i] == _variable)
    {

      current_id = i;
      gUpdateValue(current_id);
    }
  }
  // if(current_id != -1)

}



void gSendCommand(byte _cmd, byte _item, int _value)
{
  if(!internalInit && (guidino_initialized || guino_executed || _cmd == guino_iamhere)
)
  {
    guino_data.cmd = _cmd;
    guino_data.item = _item;
    guino_data.value = _value;
    ET.sendData();
  }

}
```

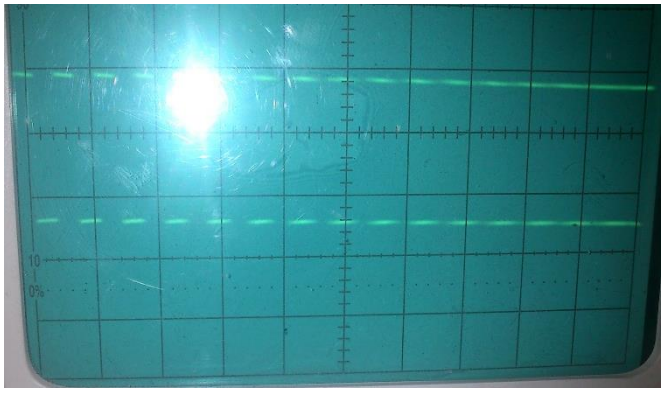
5. TESTING AND ANALYSIS

a. CRO input and output:

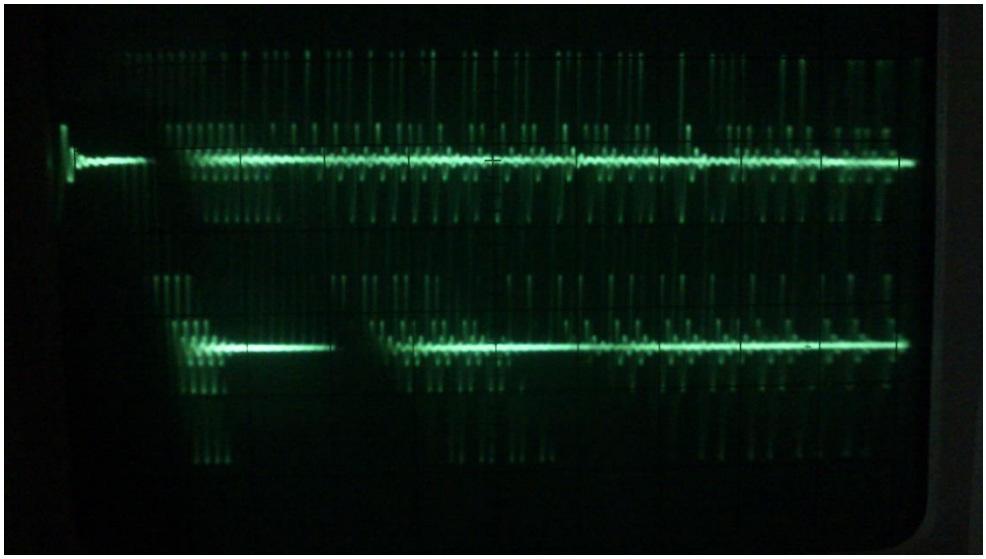
PWM Input:



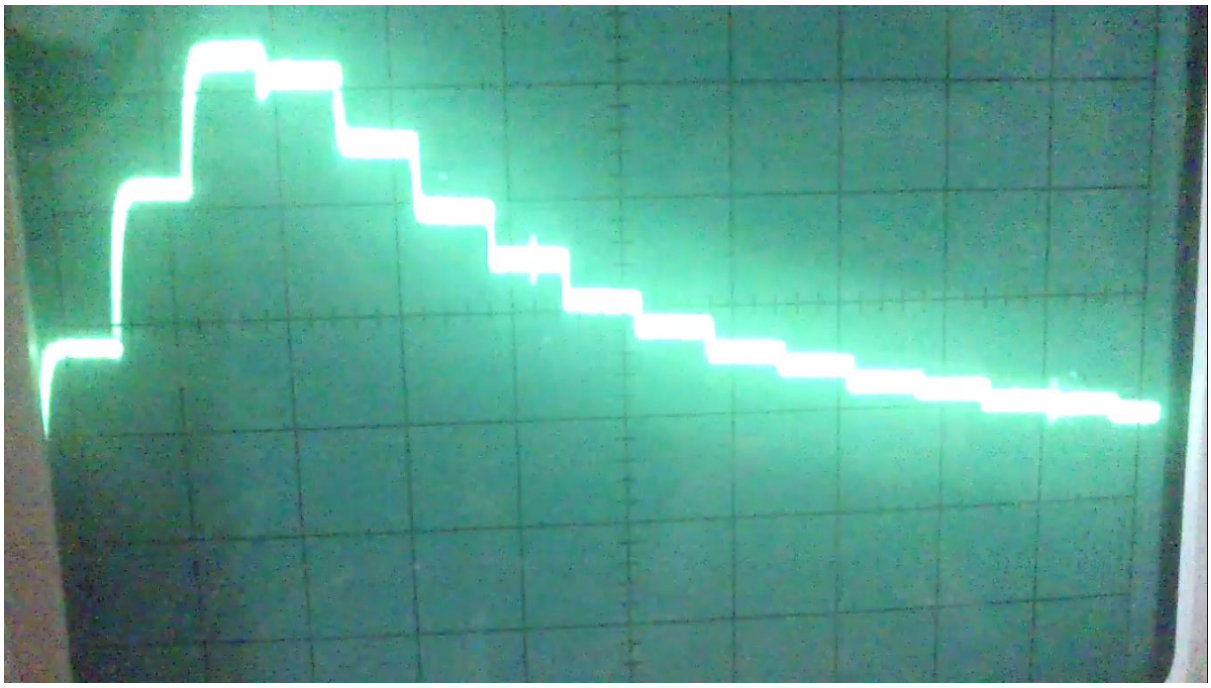




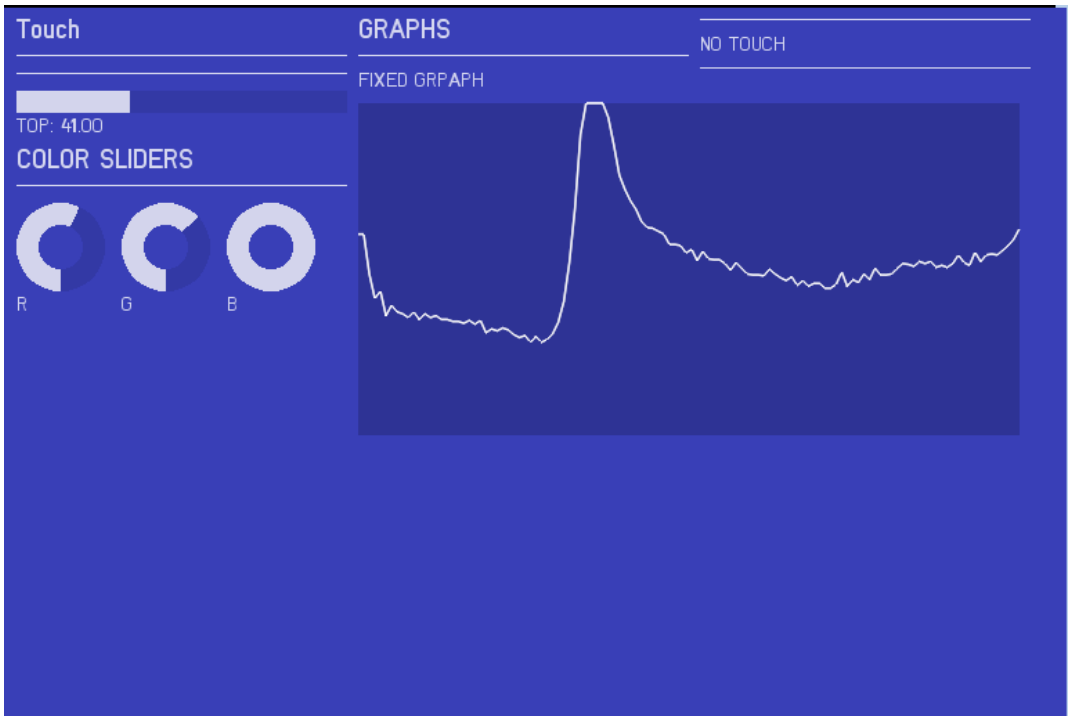
Distorted PWM after touching the object:

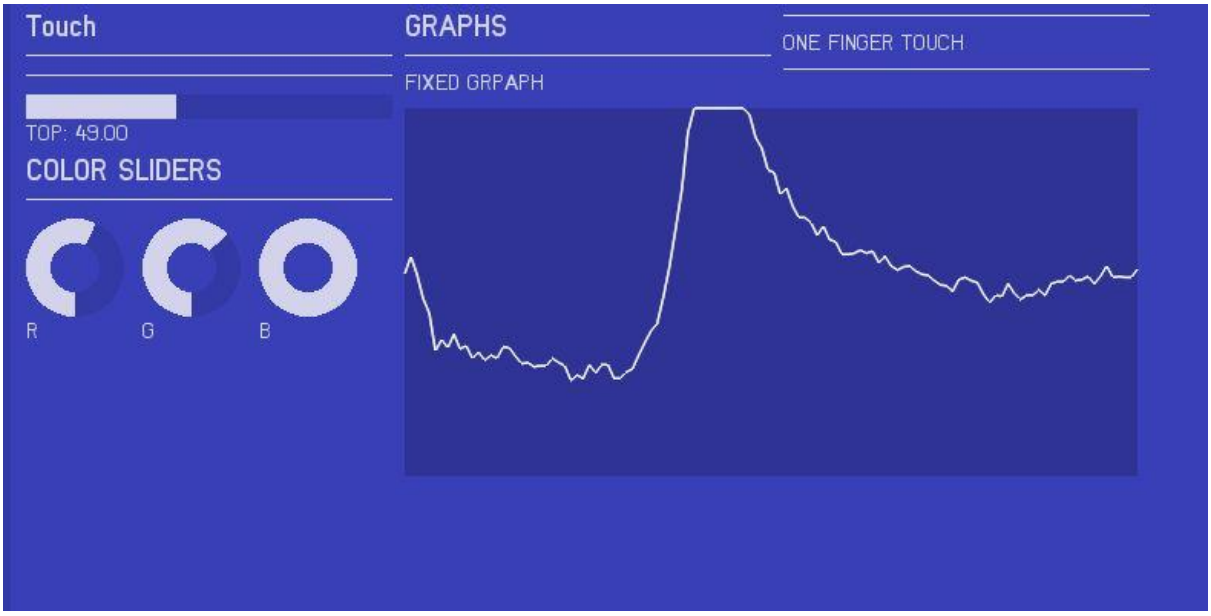


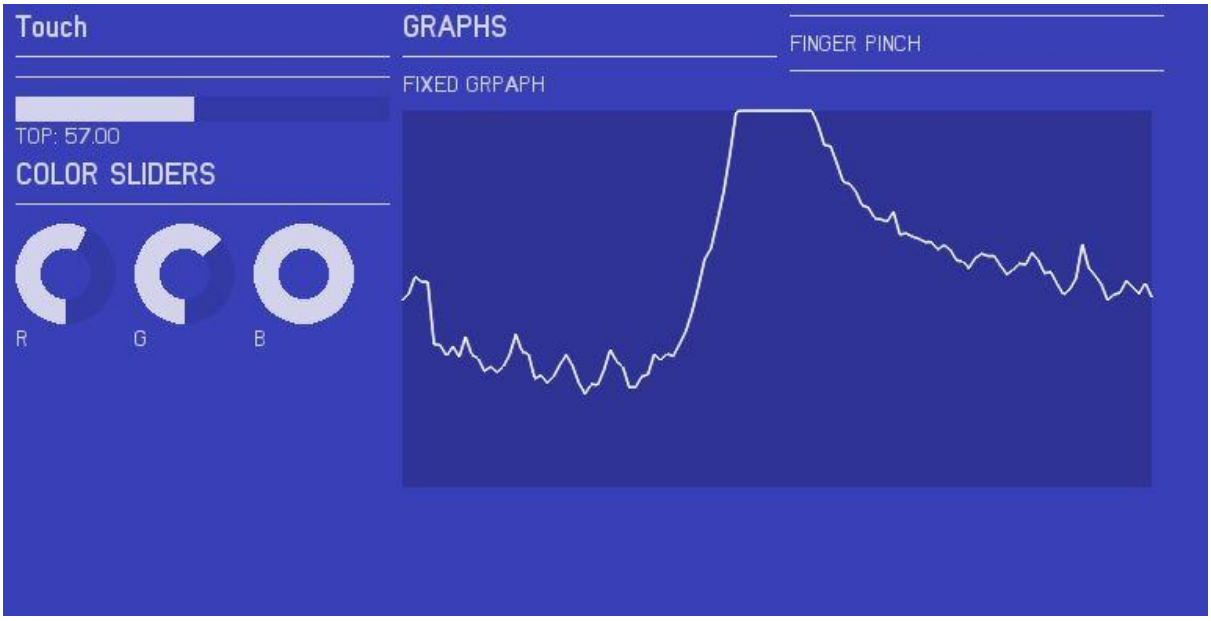
OUTPUT after Envelope Detector:

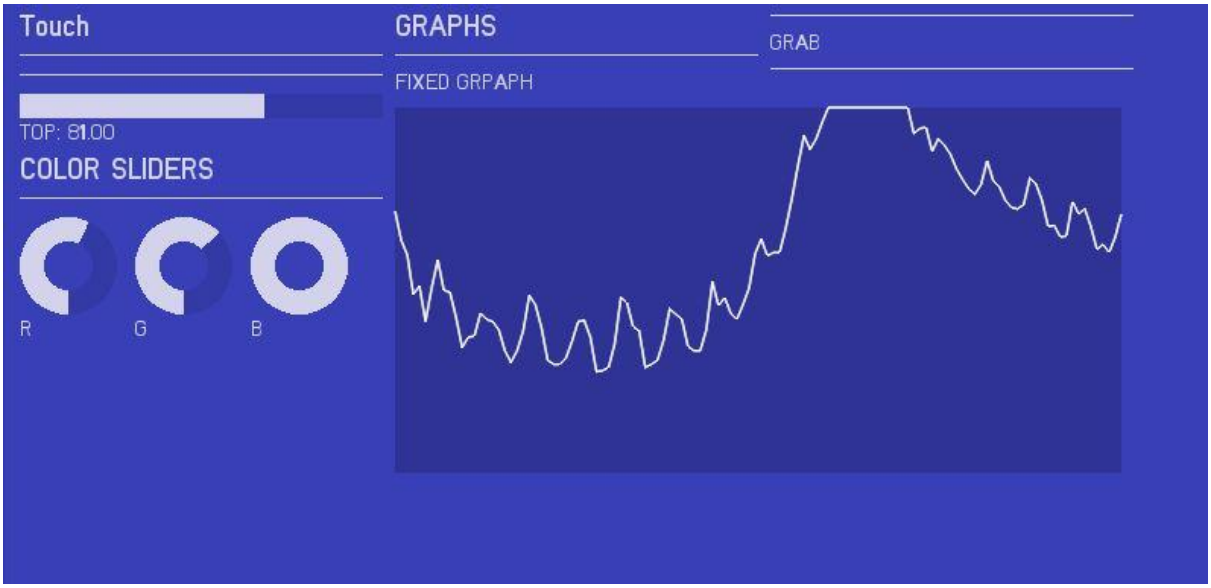


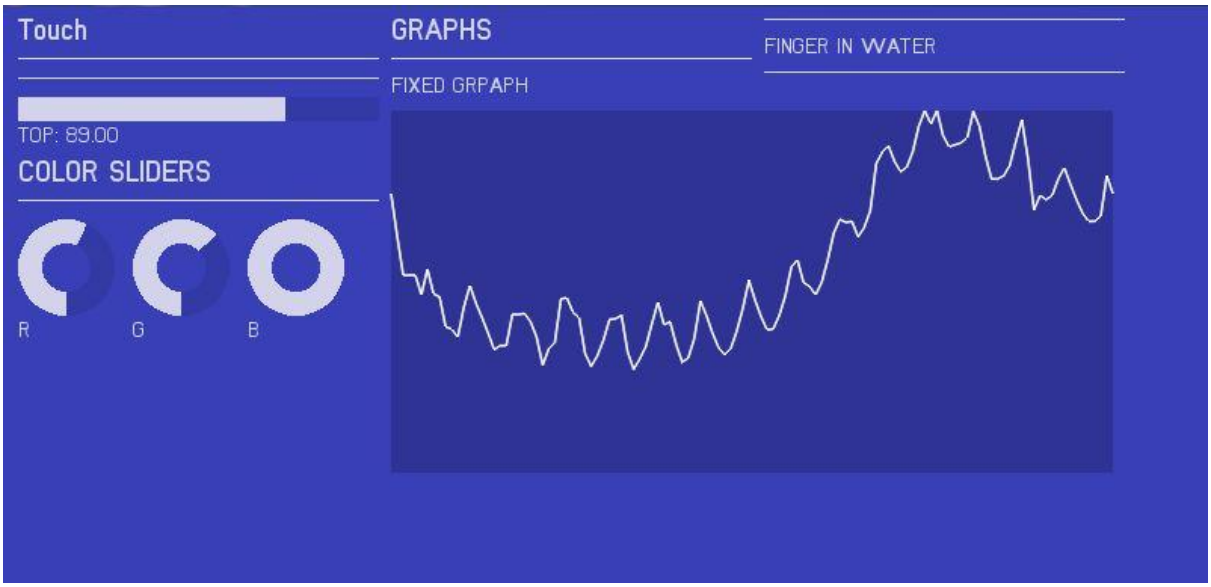
b. Various Output Configurations:




Touch
GRAPHS
ONE FINGER TOUCH
FIXED GRPAPH
TOP: 49.00
COLOR SLIDERS
R
G
B


Touch
GRAPHS
FINGER PINCH
FIXED GRPAPH
TOP: 57.00
COLOR SLIDERS
R
G
B


Touch
GRAPHS
GRAB
FIXED GRPAPH
TOP: 81.00
COLOR SLIDERS
R
G
B


Touch
GRAPHS
FINGER IN WATER
FIXED GRPAPH
TOP: 89.00
COLOR SLIDERS
R
G
B

6. CONCLUSIONS

The purpose of this project was that to extend the touch interactions to the objects which we use in day to day life. We use the property of effective capacitance i.e. the flow of AC current is through the path with least impedance (as combination of capacitor and resistor act as impedance in AC circuits). When we connect the object to our sensor the object becomes the path of flow of charge when the person touches the object. Various configurations can be obtained by just analysis of the position of the highest amplitude of AC component through the envelope detector.

7. FUTURE WORK

We found that it was difficult, if not impossible, to determine a-priori which frequency bands are most characteristic for specific interactions, applications, users, materials and contexts.

In place of Simple Micro-controller, Digital Signal Processor should be used. This increases the range of frequency from mere KHz to MHz.

We can use ADC with higher conversion rate in MHz rather than the ADC which we used with highest rate of conversion of 200 KHz.

We can use simple sine wave in place of PWM of the Micro-controller to get a more precise and detailed output.

8. REFERENCES

- http://www.instructables.com/id/Touche-for-Arduino-Advanced-touch-sensing/?ALLSTEPS
- http://www.instructables.com/id/Guino-Dashboard-for-your-Arduino/?ALLSTEPS
- **Touché: Enhancing Touch Interaction on Humans, Screens, Liquids, and Everyday Objects** – Disney Research Lab Paper published in May, 2012
- http://arduino.cc/en/Tutorial/HomePage
- Datasheet of 1N4148
- Datasheet of ATmega324