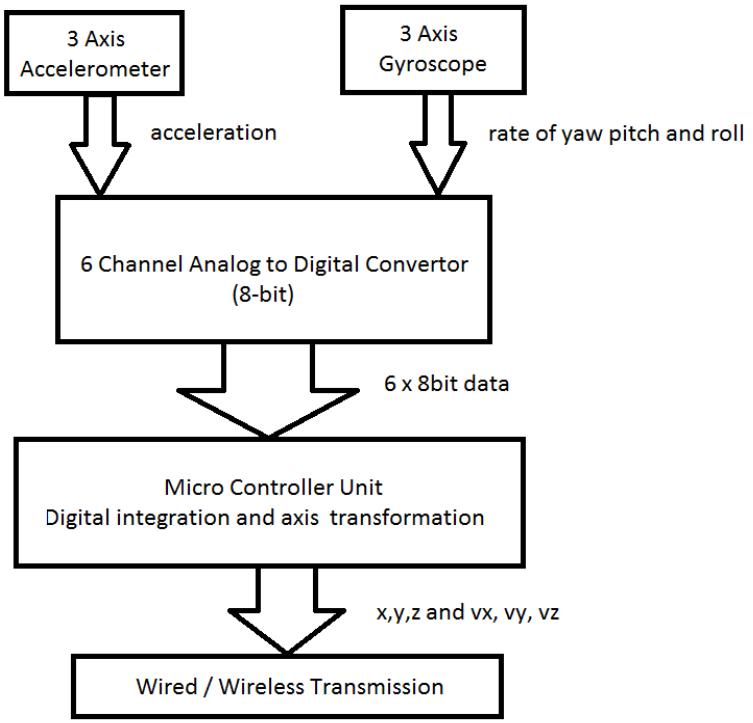
Project: Head Tracking system

Final Report by:
Uttam Sikaria (08D07042)
Kumar Akarsh (08D07043)
Palash Jhabak (08D07044)

Midsemester report at a glance

Block diagram



Digital Integration and Axis Transformation

We had to integrate the 3-axis acceleration data (gyroscope transformed) to get the 3-axis velocity values and again do that to get the 3-axis displacement values for tracking the head of the person, Its current state being defined by the above 6 state variables.

We had a lot of algorithms available for this problem and we studied all of them to reach the conclusion that we are going to employ the Runge-Kutta integration method. Here is a brief history of what we methods we considered and thus comparatively studied about their advantages and problems to reach to our final decision.

- **Euler Integration Scheme:**

  The first algorithm that naturally comes to mind and we try to implement was Euler Integration method. Here we planned to get integrated values as follows

  $\frac{\Delta x}{\Delta t} = f(x)$ ….. Where f(x) is the function to be integrated

  Or, $x_n - x_{n-1} = f\ (x_{n-1})\ \Delta t$ ….. Where Δt is sufficiently small

  But there's a big disadvantage involved in this method is that even a slight error (such as truncation error) gets accumulated by a exponent of 2 .i.e. error α h^2. Thus even very small truncation errors become unbounded really fast. Thus we discarded this method and decided to go on with some different one.

- **Verlet Integration Scheme:**

  Here our calculations will be as according

  $$x_n = x_{n-1} + v_{n-1}\,\Delta t + a_{n-1}\,\Delta t^2$$

  $$v_n = v_{n-1} + 0.5(a_n + a_{n-1}).\ \Delta t$$

  But here also the integrated error is quite high and thus we have to implement some sort of powerful enough low pass filter to remove the noise. Then we decided to go for the Runge-Kutta method which turned out to be the most efficient of all of those we tried.

- **Runge Kutta method:**

  The governing equations for the discrete application of Runge-Kutta method are are

  $$I_n = I_{n-1} + X_n + 2x_{n-1} + 2x_{n-2} + x_{n-3}$$

  The advantage with this method is the error here is an order proportional to h5. Therefore the total accumulated error is an order of h4. Moreover, It smoothens the curve considerably which in other methods is quite 'edgy' due to discrete samples. Thus we decided to go on with this scheme of integration as it was sufficiently efficient as far as our specifications were concerned.

  For axis transformation, standard matrix equations for Cartesian coordinate axis rotation are used

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix} \begin{bmatrix} X'' \\ Y'' \\ Z'' \end{bmatrix}$$

ADC conversion

We chose the microcontroller ATMEGA 16 for our application because it had 10 bit resolution .i.e. a precision of an order of 2-10 i.e. 0.00097656 which was more than enough for our case. Moreover the processing speed of atmega 16 is decent enough for our application.

The steps involved for A/D conversion are as follows –

- Initialising the ADC registers ADMUX with the value 0b00000000 and the register ADSCRA with 0b10000111
- Put the channel value in ADMUX.
- Start conversion by setting ADSC bit for conversion.
- Monitor the ADIF bit for checking if the conversion is complete.
- Clear the ADIF bit by writing 1 to it.
- A/D converted result is now available in ADCH and ADCL registers with alignment as decided during the initialization

Hardware:

- ATMEGA16 Microcontroller alongwith MMA7341LC accelerometers and ITG-3200 gyroscope
- Data communication with serial port

Accelerometers and Gyroscope

- Turned out to be the rate determining part of the progress of the project
- Noise and sensitivity of the accelerometer was being characterized but it broke down and could not be used further
- Halted the project a crucial step
- Pin diagram of MMA7341LC accelerometers

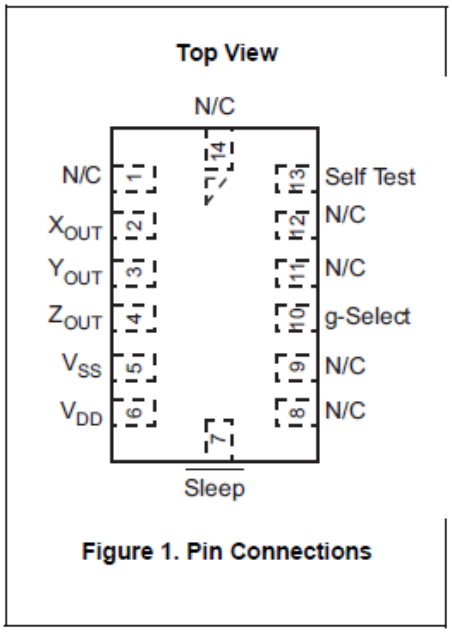


Figure 1. Pin Connections

Range of acceleration in each of the three directions = 3g

Sensitivity: 440mv/g

Output capacitors of 3.3nF at each output for low pass characteristic so as to filter the noise. This corresponds to output bandwidth of 1507Hz and is minimum requirement to filter out clock noise

0.1uF capacitor between V_{ss} and V_{dd}
g-select low refers to 3g range – desired range (high refers to 11g)

- What we learnt:
    - Accelerometers of cheap range are noisy and need to mounted in specific format to reduce jitters. One of the requirements is to provide a ground plane at the base of the accelerometer
    - Accelerometers and Gyroscope pose two major problems:
        - **Offset in output:** This is easy to handle as mean offset at each value can be experimentally determined and subtracted from the output during the use
        - **Ramp in offset:** The above solution would be ideal if the offset remained constant but as it happens, the devices suffer from a ramp offset which is difficult to easy to characterize through experiments but robustness in eliminating the same is a little difficult

    - Solution to both the above problems lie in initial calibration of the devices at the start of every run. The following tests were devised for the accelerometer:
        - **Rest:** The device was kept at rest and the outputs were observed. This helps is finding out the ramping offset
        - **Motion:** The device is moved one by one along each axis, steadily in the front and backward direction. From the data obtained, ramp in first test is subtracted. Owing to steady movement, a constant acceleration should result and output should so be more or less constant
        - Rest test was completed succesfully:
            X and Y: ~1.6V offset, ramp ~1mv/sec
            Z: ~1.6V offset with ramp ~1mv/sec
        - For the purpose of the test, a setup of Arduino UNO board for ADC and serial communication and MATLAB for monitoring the data was used. This way, live observation of the digital data could be done.

Experimental values (Testing done using ADXL 335)

- To get the feel of the for of the output we get from the accelerometer.
- Noise characterstics and hence implement noise reduction filter.

Testing done using ADXL 335

Typical Frequency response.

Xout, Yout – 1600Hz
Zout – 550Hz

Cutoff

Z-1000Hz - .0047 uF = 4.7pF
X,Y – 2000Hz - .0027uF = 2.7pF

Vss = 3V

maximise rating from -0.3V to 3.6V

ST pin

Self-test – if given Vs, to internal electrostatic force.

Xout - -3.25mV (-1.088)
Yout - +325mV (+1.088)
Zout - +550mV (1.838)

If not to be self-tested , leave it open or connect it to GND.

Test

ST pin high

Xout – 1.2V
Yout – 1.8V
Zout – 2.2V

ST low (0g)

Xout – 1.5V
Yout -1.5V
Zout-1.8V

Now, we plan to implement

-subtractor circuit
-integrator twice to set point
-after moving again to same place , it should come put to be close to zero.

Codes Used

```
#include <avr/io.h>
#include <inttypes.h>
#include <math.h>

typedef unsigned int type; // we can change from unsigned int to float
or whatever
type h = 0.01; // h = t(n)-t(n-1) i.e the time difference between
two consecutive samples
type fun(type t, type y);
type runge_kutta(type yn, type tn);
void ADC_init(void);
type ADC_read(unsigned char ch);

void USARTInit(uint16_t ubrr_value);
type USARTReadInt();
void USARTWriteChar(type data);


float transform (type alpha, type beta, type gamma);
void matrix_multiply ( float *matrix1, float *matrix2, float
*matrix, int m, int n, int p) ;



void main()
{
    type I_current[9]={0,0,0,0,0,0,0,0,0};
    type I_previous[9]={0,0,0,0,0,0,0,0,0};
    type buffer[9][4]={0,0,0,0,
    `      0,0,0,0,
           0,0,0,0,
           0,0,0,0,
           0,0,0,0,
           0,0,0,0,
           0,0,0,0,
           0,0,0,0,
           0,0,0,0};

     float * temp1 ;//[3][3] for holding the transformation matrix
     type * temp2 ;//[3][1] for holding the pre-transformed values
     type * temp3 ;//[3][1] for holding the post-transformed values

    unsigned char i=0;
```

```
    unsigned char j=0;
    int k=0;
    ADC_init();


     while (1)
    {
          for (j=0;j<=5;j++)
        {
            for (i=3;i<=1;i--)
            {
                buffer[j][i]=buffer[j][i-1];
            }

            buffer[j][0]=ADC_read(j);

               if (j<3)
               {
                    temp2[j]=buffer[j][0];
               }
          }

          for (m=3;m<=5;m++)
        {
               I_current[m]=runge_kutta(I_previous, buffer,m);
               I_previous[m]=I_current[m];
        }

          temp1 = transform(I_current[3], I_current[4],
I_current[5]);
          matrix_multiply (temp1, temp2, temp3, 3, 3, 1); // type
conversion to be taken care of
          // now the temp3 array contains the gyro transformed
acceleration data
          for (m=0; m<3; m++)
          {
               buffer[m][0]= temp3[m];
          }

          for (k=0;k<=2;k++)
            {
                I_current[k] = runge_kutta(I_previous, buffer,k);
                I_previous[k]=I_current[k];
                for (i=3;i<=1;i--)
                {
                    buffer[k+6][i]=buffer[k+6][i-1];
                }
                buffer[k+6][0]=I_current[k];
            }
```

```
          for (m=6; m<=8; m++)
            {
                I_current[m]=runge_kutta(I_previous, buffer,m);
                I_previous[m]=I_current[m];
            }

          // send x,y,z,vx,vy,vz to output port one by one
          USARTInit(103);
          for(m=0; m<9; m++)
          {
               if(m==3||m==4||m==5)      continue;
               USARTWriteChar(I_current[m]);
               USARTWriteChar(0x00);
          }
     }
}

type runge_kutta(type *I_previous, type *signal, int j)
{
    type next;
    next= I_previous[j] +
(signal[j][0]+2*signal[j][1]+2*signal[j][2]+signal[j][3])/6.0;
    return next;
}

void ADC_init(void) // Initialization of ADC
{
    //ADMUX=(1<<REFS0); // AVcc with external capacitor at AREF
    // no change in ADMUX for external Vref
    ADCSRA=(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
    // Enable ADC and set Prescaler division factor as 128
}

type ADC_read(unsigned char ch)
{
    ch= ch & 0b00000111; // channel must be b/w 0 to 7
    ADMUX |= ch; // selecting channel

    ADCSRA|=(1<<ADSC); // start conversion
    while(!(ADCSRA & (1<<ADIF))); // waiting for ADIF, conversion
complete
    ADCSRA|=(1<<ADIF); // clearing of ADIF, it is done by writing
1 to it

    return (ADC);
}
```

```
float* transform (type alpha, type beta, type gamma)
{
     float result [3][3], result_updated[3][3];

     float matrix_alpha[3][3]={ cos(alpha), -sin(aplha),  0,
                                sin(alpha),  cos(aplha),  0,
                                   0  ,      0     ,  1 };

     float matrix_beta[3][3]={  cos(beta),    0    , sin(beta),
                                   0   ,    1    ,     0    ,
                                -sin(beta),    0    , cos(beta) };

     float matrix_gamma[3][3]={    1  ,    0     ,     0
                                   0   , cos(gamma) , -sin(gamma),
                                   0    , sin(gamma) , cos(gamma)};

     matrix_multiply(matrix_alpha,matrix_beta,result,3,3,3);
     matrix_multiply(result,matrix_gamma,result_updated,3,3,3);

     return(result_updated);
}

void matrix_multiply ( float *matrix1, float *matrix2, float
*matrix, int m,int n, int p)

{
     for (int i=0;i<m;i++)
     {
          for (int j=0;j<p;j++)
          {
               matrix[i][j] = 0;
          }
     }

     for (int i=0;i<m;i++)
     {
          for (int j=o;j<p;j++)
          {
               for (int k=0;k<n;k++)
               {

     matrix[i][j]=matrix[i][j]+matrix1[i][k]*matrix2[k][j];
               }
          }

     }
```

```
}

void USARTInit(uint16_t ubrr_value)
{

   //Set Baud rate
   //UBRR|= ubrr_value;
   UBRRL = ubrr_value;
   UBRRH = (ubrr_value>>8);

   /*Set Frame Format


   >> Asynchronous mode
   >> No Parity
   >> 1 StopBit
   >> char size 8

   */

   UCSRC=(1<<URSEL)|(3<<UCSZ0);


   //Enable The receiver and transmitter
   UCSRB=(1<<RXEN)|(1<<TXEN);


}

type USARTReadInt()

{
   //Wait untill a data is available
   while(!(UCSRA & (1<<RXC)))
   {
    ;//Do nothing
   }

   //Now USART has got data from host
   //and is available in buffer

   return UDR;
}

//This fuction writes the given "data" to
//the USART which then transmit it via TX line
```

```
void USARTWriteChar(type data)

{
   //Wait untill the transmitter is ready

   while(!(UCSRA & (1<<UDRE)))
   {
      //Do nothing
   }

   //Now write the data to USART buffer

   UDR=data;

}
```