Firmware for Single Board Heater System

Rupak Mohan Rokade

February 22, 2012

Contents

1 Introduction **3**

2 Initializing microcontroller ports **4**

3 Establishing serial comunication between µC and computer **5**
- 3.1 Receiving operation 8
- 3.2 Transmitting operation 8

4 Accepting Heater and Fan values from the computer and its execution **9**
- 4.1 PWM generation 12
- 4.2 Timer Operation 13

5 Reading the temperature value and sending it to the computer **15**
- 5.1 Analog to Digital conversion 15
- 5.2 Sending valid temperature value to the computer 17

6 Displaying various parameters on on-board display **20**

7 The Preprocessor Directives **27**

8 The Complete C code **28**

1 Introduction

The 'Single Board Heater System'uses microcontroller ATmega16. This document is intented to explain the firmware for the same. The programming is exclusively done in C language. The code is written with respect to WINAVR, which is a free and opensource C compiler and is used here for the generation of hex file along with burning the code in to the microcontroller. The code is divided in to parts for better understanding. It is however recomended that one should study the architecture of ATmega16 before proceeding to read this document. One can find its datasheet on `www.atmel.com`. One should know the availability and usage of various ports and registers. Although some necessary information will be provided wherever necessary, it is still wise to study ATmega16 first[**?**].

The microcontroller is aimed at performing the following functions

- Establishing serial comunication between itself and computer
- Accepting Heater and Fan values from the computer and its execution
- Reading the temperature value and sending it to the computer
- Displaying Temperature, Heater, Fan and SER value on the on-board display

It must be noted that a hex value is always preceeded by a 0x. Otherwise it is a decimal value. For example, 0x33 means 33 is a hex value.

Well, we would now begin with the programming of microcontroller.

2 Initializing microcontroller ports

ATmega16 has four ports, PortA, PortB, PortC and PortD. Apart from being used for normal data transfer each of these have alternate functions too. PortA is been used for accepting the analog temperature value at bit PA0 (pin no. 40). PortB is configured as output port whose SCK, MISO and MOSI pins are used for In-System programming. PortC is again configured as output port to which LCD Display is been connected. And finally PortD is configured as output port to which heater and fan drivers are connected. A buzzer is also connected to this port. The program for port initialization is given below.

```
1  //Initializing microcontroller ports
2  void port_init(void)
3  {
4   PORTA = 0x00;
5   DDRA  = 0x00;
6   PORTC = 0x00; //m103 output only
7   DDRC  = 0xFF;
8   PORTD = 0x7F; //-----------7F
9   DDRD  = 0xF0;
10 }
```

The operation of every port is goverened by three 8-bit registers, PORTx (PORTx Data register), DDRx (PORTx Data Direction register), PINx (PORTx Input Pins Address register) where x stands for port name A, B, C or D. Each port pin therefore consist of three register bits, PORTxn, DDRxn and PINxn where n stands for bit number(0 to 7).

DDRxn value decides the corresponding port pin direction. Logic 1 means output and logic 0 means input. PORTxn is used in conjunction with DDRxn. If DDRxn=0(I/P) and PORTxn=1, pull-up resistor is activated. To disable the pull-up resistor PORTxn should be made 0. If DDRxn = 1(O/P) and PORTxn=1, the corresponding port pin is driven high(1). If DDRxn = 1(O/P) and PORTxn=0, the corresponding port pin is driven low(0).
The values shown in the above code are just for the initialization purpose. We may change the values later in the remaining code as and when required.

3 Establishing serial comunication between µC and computer

Microntroller ATmega16 has inbuilt Programmable serial USART. TXD and RXD pins of µC are used to carry out the serial communicatio. The program dedicated to use this facility is given below.

```
//USART initialization for serial communication
void uart0_init(void)
{
 UCSRB = 0x00; //disable while setting baud rate
 UCSRA = 0x00;
 UCSRC = 0x86;
 UBRRL = 0x33; //set baud rate lo----------0x19--------0
    x33
 UBRRH = 0x00; //set baud rate hi
 UCSRB = 0x98; //0x98--------------
}
```

The code shown above is an initialization for serial communication. Various parameters related to it like baud rate, character size, parity etc are defined here. The USART is operated using three registers, UCSRA,UCSRB and UCSRC.

1. UCSRA register (USART Control and Status register A)

Bit	7	6	5	4	3	2	1	0	
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	UCSRA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

Figure 1: UCSRA register

2. UCSRB register (USART Control and Status register B)

Bit	7	6	5	4	3	2	1	0	
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	UCSRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 2: UCSRB register

It can be seen that the UCSRA and UCSRB registers are written 0x00. They actually ensure that transmission and reception operation is disabled during the setting of baud rate and other parameters.

3. UCSRC register (USART Control and Status register C)

Bit	7	6	5	4	3	2	1	0	
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	UCSRC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	0	0	0	0	1	1	0	

Figure 3: UCSRC register

The UCSRC register is loaded with the value 0x86. It says that the operation would be assynchronous, parity is disabled, No. of stop bits = 1 and character size is 8-bit. It should be noted that the value of D7th bit is made high. This indicates that UCSRC register is to be read. If it was to be zero then it would mean that UBRRH register is to be read. This is because the UCSRC and UBRRH share the same I/O location.

4. UBRR register (USART Baud Rate register)

Bit	15	14	13	12	11	10	9	8	
	URSEL	–	–	–	UBRR[11:8]				UBRRH
	UBRR[7:0]								UBRRL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Figure 4: UBRR register

UBRR is a 16-bit register used as two 8-bit registers, UBRRH and UBRRL. Bits D0 to D11 contains the baud rate value. The UBRR register along with the Down counter connected to it function as a baud rate generator. The down counter which runs at the system clock (f_{osc}) is loaded with the value in UBRR. This happens every time the counter counts to zero or when UBRRL is written. A clock is generated every time the

counter reaches zero. The value of UBRR can be found out using the given formula.

$$UBRR = \frac{f_{osc}}{16 * BAUD} - 1 \tag{1}$$

In our case the value of f_{osc} is 8MHz and BAUD is 9600. Hence, the value of UBRR is

$$UBRRL = 0x33 \tag{2}$$
$$UBRRH = 0x00 \tag{3}$$

Data gets corrupted if the baud rate value is changed during ongoing transmission or reception.

After setting the communication parameters the serial transmission and reception is enabled by loading the UCSRB register with value 0x98.

5. UDR register (USART I/O Data register)

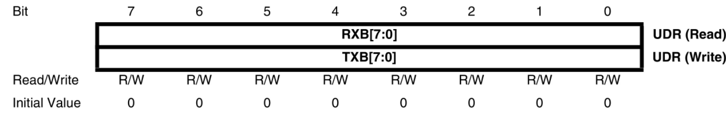

Figure 5: UDR register

The data to be transmitted is written in to TXD register and the data to be received is buffered in the RXD register. But they both share the same I/O address refered to as UDR register. TXD stands for Transmit data buffer and RXD stands for Receive data buffer.It is evident that both the data to be written and to be read uses the UDR register. An obvious question would be as to how the µC would come to know wheather the data is to be accepted or transmitted. This issue is addressed using the above mentioned UCSRA and UCSRB registers.

3.1 Receiving operation

The RXD is actually a two level FIFO (First In First Out) register. The receiver is enabled by setting the RXEN bit in the UCSRB register. Data coming at the RXD pin is accepted only if a valid start bit is identified. When the RXD register is written with the received data the RXC flag in the UCSRA register is set indicating that there is unread data in UDR. The RXC flag is automatically cleared when the UDR (RXD) register is empty. The RXC flag can also generate a receive complete interrupt if the RXCIE bit of UCSRB register is set. This feature is actually used here by setting UCSRB=0x98.

3.2 Transmitting operation

The data to be transmitted can be written to UDR(TXD) register only if the UDRE bit in the UCSRA register is set. UDRE stands for USART Data Register Empty. The transmitter is enabled by setting the TXEN bit in the UCSRB reg. With UDRE=TXEN=1, the data present in the UDR is shifted to a Transmit Shift Register. The TXC bit in the UCSRA reg is set when the UDR becomes empty indicating that it is ready to be written. The TXC bit can also generate a transmit complete interrupt if the TXCIE bit of UCSRB register is set. TXC bit is automatically cleared when the transmit complete interrupt is executed.

4 Accepting Heater and Fan values from the computer and its execution

If you are reading this document from top to bottom, we have uptill now initialised the ports, established serial communication between computer and µC and also enabled the receive complete interrupt. We would now write the corresponding interrupt subroutine. Since the program uses some predefined variables we would first see the initialization of these variables.

```
1  //Serial ISR variables
       ------------------------------------
2  int value=0;
3  int temperature_c = 0;
4  int test1 = 0;
5  int temp_val = 0;
6  unsigned char next_byte_light = 0;
7  unsigned char next_byte_fan = 0;
8  unsigned char temp_upper_byte_send;
9  unsigned char temp_lower_byte_send;
10 unsigned char ser_data = 0;
11 unsigned char light = 0;
12 unsigned char fan = 0;
13 int temperature_c_upper_byte = 0;
14 int temperature_c_lower_byte = 0;
15 int temperature = 0;
16 int r=1;
```

All variables are assigned the value 0. We need not worry about some variables which we are not using right now. Their usage will become clear as the document proceeds.

After initialization, the code for serial receive interrupt subroutine is given below.

```
1  //USART receive complete interrupt subroutine
2  ISR(USART_RXC_vect)
3
4  {
5   //uart has received a character in UDR
6
7   ser_data = UDR;
8
9   if(ser_data < 253)
10  {
11   if (next_byte_light == 1)
12   {
```

```
13     light = ser_data;
14     if (light > 40)
15     {
16     light = 40;
17     }
18     OCR1AL = light; //heating element input
19     next_byte_light = 0;
20    }
21
22    if (next_byte_fan == 1)
23    {
24     fan = ser_data;
25     OCR1BL = fan; //fan speed input
26     next_byte_fan = 0;
27    }
28   }
29
30   if (ser_data == 254) //command for heater
       --------------
31   {
32    next_byte_light = 1;
33    next_byte_fan = 0;
34   }
35
36   if (ser_data == 253) //command for fan -------------
37   {
38    next_byte_light = 0;
39    next_byte_fan = 1;
40   }
41
42   if(ser_data == 255)
43   {
44    temperature = test1;
45    temperature_c_lower_byte = temperature%10;
46    temperature_c_upper_byte = temperature/10;
47    temp_upper_byte_send = (unsigned char)
         temperature_c_upper_byte;
48    temp_lower_byte_send = (unsigned char)
         temperature_c_lower_byte;
49    UDR = temp_upper_byte_send;
50    UDR = temp_lower_byte_send;
51    UDR = temp_upper_byte_send;
52    UDR = temp_lower_byte_send;
53    UDR = temp_upper_byte_send;
54    UDR = temp_lower_byte_send;
55   }
```

```
56
57
58
59 }
```

The interrupt subroutine beigns with the line $ISR(USART_RXC_vect)$. This is a default ISR name for serial receive complete interrupt in WINAVR. Any thing written below it will be executed as the subroutine. This removes the need of checking the flag registers, etc.

Since the size of the UDR register is limited to 8-bits, only 2^8=256 possibilities exist. Here we have allocated unique values to particular tasks. This means the Fan speed algorithm has the label 253, heater current has the label 254 and 255 is for temperature. The first time the µC would receive data in UDR would be anyone among the three values mentioned above.

As soon as a byte is received in the UDR register it will generate an interrupt. This will execute the serial recieve complete ISR.

If the value is 253 or 254, it gets copied to a variable *ser_data*. The value of next_byte_light and next_byte_fan are also defined. Now the next byte µC would receive in the UDR will be the value of the fan speed or heater current, ranging from 0 to 252. With the help of the values of next_byte_fan and next_byte_light the program would jump to appropriate loop. If its a fan speed then its value will be loaded in to the OCR1BL register and if its a heater current value then it would be loaded in to the OCR1AL register. For heater current, its value is limited to 40.
If the value received in the UDR is 255, it means that the µC has to send the value of temperature to the computer. This requires some extra operations like analog to digital conversion and so on. This would be dealt on later.

4.1 PWM generation

Now the question is how actually the value of heater current and fan speed would be altered. The answer is Pulse Width Modulation (PWM). PWM technique is used to alter the power delivered to heater and fan. Here we are not concerned about hardware requirements, we just want the µC to generate appropriate PWM waveform. ATmega16 has four PWM channels, OC0, OC1A, OC1B and OC2. The µC also has 3 Timers/Counters, Timer/Counter0, Timer/Counter1 and Timer/Counter2. Of these, Timer/Counter0 and Timer/Counter2 are 8-bit. Whereas Timer/Counter1 is 16-bit. Here Timer/Counter1 is used to generate PWM for heater and Fan speed. The Timer/counter1 is a 16-bit register used as two 8-bit registers TCNT1H and TCNT1L together known as TCNT1. It is configured using two control registers, TCCR1A and TCCR1B.

```
1  //Timer/Counter1 initialization for PWM generation
2  void timer1_init(void)
3  {
4   TCCR1B=0x00; //stop timer during configuration
5   TCCR1A=0xA1;
6   OCR1AH=0x00;
7   OCR1AL=0x00; //setting heater 0
8   OCR1BH=0x00;
9   OCR1BL=0xFC; //setting fan full
10  TCCR1B=0x0B; //start timer
11 }
```

1. TCCR1B (Timer/Counter Control Register 1B)

Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 6: TCCR1B register

This register is essentially loaded with value 0x00 to disable the TIMER/-COUNTER1 during initialization. It also says that the Noise Canceler facility is not being used.

2. TCCR1A (Timer/Counter Control Register 1A)

Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 7: TCCR1A register

This register is loaded with value A1. Bits D7 to D4 say that OC1A/B will override the normal port functionality. D3 and D2 say that we are using the PWM mode of operation. Finaly, Bits D1-D0 are used to configure the PWM mode of operation. Their usage is in conjuntion with the D3 and D4 bits of TCCR1B. The TCCR1B register is again loaded with value 0B. Hence, both these registers together specify that the PWM mode is ‘PWM 8bit fast, TOP=0x00FF ’. The timer is started.

4.2 Timer Operation

Until now we have configured the timer registers TCCR1B and TCCR1A. Also, earlier we had seen that the heater and fan values are loaded in OCR1A and

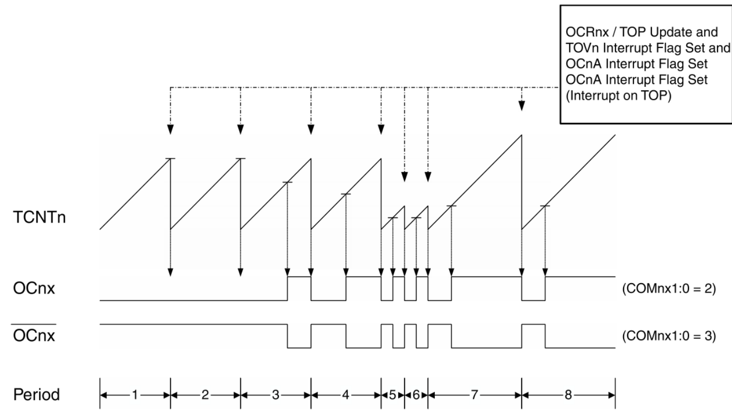

Figure 8: Fast PWM mode

OCR1B registers respectively. Timer now starts counting till the top value (here 0xFF) after which it starts again from 0x00. At the same time the OCR1A/B registers are compared with the timer value. It is been configured such that as soon as there is a match, the respective pin OC1A or OC1B goes low. Also, as the timer reaches the top value the OC1A and OC1B go high. In this way a square wave of some duty cycle, decided by the value of OCR1A/B registers, is generated. The PWM waveform generation process is depicted in figure 8.The PWM frequency can be calculated using the following formula.

$$f_{OCnPWM} = \frac{f_{osc}}{N * 256} \tag{4}$$

Where, f_{osc} is the crystal frequency and N is the prescale factor(1,8,64,256, or 1024). Therefore, in our case f_{osc} is 8MHz and prescale is 64. Hence, the PWM frequency is around 488Hz.

5 Reading the temperature value and sending it to the computer

The temperature is a physical parameter. It is first converted in to electronic signal using a temperature sensor. We have used AD590 for this purpose. The output of AD590 is in analog form. It is actualy in µA/°K. Hence, after some necessary signal conditioning it is converted to digital data using an Analog to Digital converter. ATmega16 features 8 Channel, 10-bit ADC. The two distinct processes of reading the temperature value and sending it to the computer are explained below.

5.1 Analog to Digital conversion

```
1 //Initializing ADC
2 void adc_init(void)
3 {
4  ADCSRA = 0x00; //disable adc
5  ADMUX = 0x00; //select adc input 0
6  ACSR  = 0x80;
7  ADCSRA = 0xEF; //----------------0xEE-----------0xEF
8  SREG|=0x80;
9  }
```

The above code is used for ADC initialization. As mentioned earlier, the microcontroller ports have multiple functions. The eight pins of PORTA also function as eight input channels of ADC. A multiplexer is used to choose between these channels. Two registers ADMUX and ADCSRA are used to operate the ADC.

1. ADCSRA (ADC Control and Status register A)
   Here, the value of this register is set to be 0x00. It ensures that the ADC is disabled during initialization. Bit D6 is used for this purpose. Thus this register can be used to switch the ADC On and OFF. Later the ADCSRA register is again loaded with value 0xEF. It specifies that interrupt on conversion completion is enabled. Also bits D2 to D0 selects the prescale value. Here the prescale value is selected to be 128. It is actually the division factor between crystal frequency and the frequency of the clock input to the ADC. Also bit D6 says that the ADC should begin the conversion.

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 9: ADCSRA register

2. ADMUX (ADC Multiplexer selection register)
   The ADC has 8 input channels. The ADMUX register is used to operate the analog multiplexer used for channel selection. The ADMUXvalue is loaded to be 0x00. This means that channel 0 (PA0) is used for A to D conversion. It also says that the voltage reference for the ADC is applied externally on pin32.

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 10: ADMUX register

The ACSR register is loaded with value 0x80 just to specify that the Analog Comparator is switched of. After the conversion is completed the digital equivalent data is stored in ADC Data register known as ADCH and ADCL. Lower 8

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Figure 11: ADC Data register

bits are stored in ADCL (ADC lower) and upper 2 bits in ADCH (ADC higher).

The data is by default right adjusted as shown in figure 11. After the conversion is complete the decimal equivalent of the binary data present in ADC data register can be calculated from the following formula.

$$ADC = \frac{Vin * 1024}{Vref} \tag{5}$$

Here, Vin means the incoming analog signal at the appropriate ADC channel. Vref means the reference signal for the A to D conversion. In our case the reference signal is applied as 5V at pin32. Since the ADC resolution is 10 bits, 2n = 1024 possibilities exist. For example, if Vin = 5V then ADC = 1024 (for fixed Vref=5V). It must be noted that the ADC can maximum accomodate 03FF value which represents the reference voltage minus one LSB. This means that the ADC value is actually 1024th possibility i.e.1023, whose hex equivalent comes out to be 0x03FF. Now we would see how this converted data is transmitted to the computer.

5.2 Sending valid temperature value to the computer

```
1 //ADC conversion complete interrupt subroutine
2 ISR(ADC_vect)
3 {
4   //conversion complete, read value (int) using...
5   temperature_c=ADCL;              //Read 8 low bits first
        (important)
6   temperature_c|=(int)ADCH << 8;  //read 2 high bits and
        shift into top byte
7   value = temperature_c;
8  MCUCR = 0x00;
9 }
```

As soon as the conversion is completed, an interrupt is generated. The code shown above is an interrupt subroutine for the same. It could be seen that the routine is written under the name ISR(ADC vect). It is a default name for ADC conversion complete ISR. Any thing written below it will be executed as the subroutine. This removes the need of checking the ag registers, etc. First the ADCL register is copied to a varialbe, *temperature_c*. Now the ADCH register is read in such a way that it is left shifted by 8-bits, bit wise ORed with *temperature_c* and then the integer value is stored in *temperature_c* itself. Another variable value is then defined with this data. This can be explained with an example. Suppose that a signal of 2V is fed to the ADC. Then according to the equation 5 the value contained in the ADC data register would be 409th possibility i.e. 408 in decimal. ADCH and ADCL would contain the following values.

$$ADCL = 10011000$$
$$ADCH = 00000001$$

After left shifting ADCH by 8 bits, bitwise OR operation is performed between ADCH and ADCL.

$$\begin{array}{r l} 010011000 & < -ADCL \\ \underline{+100000000} & < -ADCH \\ 110011000 & \end{array}$$

The result is type casted to integer and is stored in variable value. Thus,

$$value = 408(decimal)$$

```
val[2] = val[1];
val[1] = 1.163*value;

for(i=2;i<11;i++)
{
      if(val[i]==0)
          {
          val[i] = val[1];
           }
}

for(i=15;i>2;i--)
{
 val[i] = val[i-1];
}

temp_val = (int) val[1];

temperature_c = temp_val;
temp1 = temp_val;
test1 = temp_val;
```

A float type array named *val* containing 15 elements is being declared (refer section 4). Now, it could be seen that the variable value is been multiplied by 1.163. This is because the signal which was been digitized was not the direct output of the temperature sensor but was a conditioned signal. This signal conditioning is necessary to push the signal within the input range of the ADC. The instrumentation amplifer provides a gain of 42 to the ADC input signal. Also, resolution of ADC too should be considered. The 10 bit ADC has a reference signal of 5V. Hence its resolution is

$$ADCResolution = \frac{Vref}{2n} = \frac{5}{1024} = 4.883mV$$

Dividing the resolution by gain we get

$$\frac{4.883}{42} = 0.1163$$

Now multiplying the variable value with 0.1163 yields 47.45. Since the result will be later type casted to integer type, the decimal will be lost. Hence to retain the decimal value it is multiplied by 10. Therefore the formula now becomes

$$\frac{ADC_Resolution}{Gain} * 10 = 1.163 \tag{6}$$

Hence, the first element of val array now contains 474(decimal). This is nothing but the actual temperature which is read as 47.4. It is then copied to the integer type variables $temp_val$, $temperature_c$, $temp1$ and $test1$. Referering to section 4 in the code for serial receive interrupt subroutine, the $test1$ value is copied to an integer type variable $temperature$. Next, it's modulo 10 is copied to an integer variable $temperature_c_lower_byte$. This means that only the remainder is copied. In this case it would be 4. After that the temperature value is divided by 10 and the quotient is copied to another integer variable $temperature_c_upperbyte$. In this case it would be 47. Then after type casting these bytes as unsigned variables they are copied to UDR for transmission. It could be seen that these bytes are sent thrice. This is done to ensure that correct value of temperature is transmitted.

6 Displaying various parameters on on-board display

```
1  void main(void)
2  {
3   init_devices();
4   lcd_init();
5   lcd_print_sensor_data();
6
7   while(r=1)
8   {
9    refresh_lcd();
10   lcd_print_sensor_data();
11  }
12 }
```

As soon as the setup is turned ON, the code execution begins from here. The program for initiallizing the devices connected to it is called first. It is given below.

```
1  //Initialization routine
2  void init_devices(void)
3  {
4   //stop errant interrupts until set up
5   cli(); //disable all interrupts
6   port_init();
7   timer1_init();
8   uart0_init();
9   adc_init();
10  GICR  = 0xC0;
11  TIMSK = 0x40; //timer interrupt sources
12  sei(); //re-enable interrupts
13  //all peripherals are now initialised
14 }
```

The port, timer, serial communication and ADC initiallization is been already explained and the respective codes can be found in the respective section. The GICR register is set to 0xC0. This register controls the placement of interrupt vector table for external interrupts. The external interrupts (INT0/1/2) are not used and hence the register is not of much importance right now. The TIMSK register is loaded with value 0x40. It disables the interrupt generation on compare match for timer/counter0. The SEI command is a microcontroller instruction to globally enable the interrupts. It sets the D7th bit in the SERG register.
Next we would see the code for *lcd_init*.

```
1  //Initializing LCD
2  void lcd_init()
3  {
4
5   unsigned char data,upper,lower;
6   PORTC&=0xFB;
7   PORTC&=0x0F;
8
9
10   _delay_ms(10);
11
12   data=0x28;
13   upper_lower(&data,&lower,&upper);
14   data_transfer(upper,lower);
15
16   data=0x0E;
17   upper_lower(&data,&lower,&upper);
18   data_transfer(upper,lower);
19
20
21   data=0x06;
22   upper_lower(&data,&lower,&upper);
23   data_transfer(upper,lower);
24
25
26  }
```

The code lcd init is used for initializing the LCD Display. This requires some knowledge about how to establish communication between LCD and C. Let us first see how this is done.

LCD JHD162A is being used in SBHS. It is a 16×2 Liquid Crystal Dis- play. This means that it can display 16 characters in one line. Likewise, it has capacity to display two lines. The display basically has three control lines and eight data lines. There are also other pins available for other purposes like contrast adjust but these are not of concern now. EN, RS and RW are the three control lines and DB0 to DB7 are the three data lines.

The EN signal stands for ENABLE and it tells the LCD that data has been written onto the data pins or the microcontroller is ready to read the data from LCD. For this purpose, the first the EN pin is held low and the other contol lines are configured according to the need. Also mean while the data is put (if it is a write operation) onto the data lines. Then the EN pin is made high and held there for some time and then again made zero. This is done because the LCD understands only high to low transition at the EN pin.

The RS pin stands for REGISTER SELECT. It is used to tell the LCD wheather the data on the data lines is the data to be displayed or it is an instruction to the LCD. Logic 1 means data and logic 0 means command. The standard hex code for the commands is as shown in the figure 12

No.	Instruction	Hex	Decimal
1	Function Set: 8-bit, 1 Line, 5x7 Dots	0x30	48
2	Function Set: 8-bit, 2 Line, 5x7 Dots	0x38	56
3	Function Set: 4-bit, 1 Line, 5x7 Dots	0x20	32
4	Function Set: 4-bit, 2 Line, 5x7 Dots	0x28	40
5	Entry Mode	0x06	6
6	Display off Cursor off (clearing display without clearing DDRAM content)	0x08	8
7	Display on Cursor on	0x0E	14
8	Display on Cursor off	0x0C	12
9	Display on Cursor blinking	0x0F	15
10	Shift entire display left	0x18	24
12	Shift entire display right	0x1C	30
13	Move cursor left by one character	0x10	16
14	Move cursor right by one character	0x14	20
15	Clear Display (also clear DDRAM content)	0x01	1
16	Set DDRAM address or cursor position on display	0x80+add	128+add
17	Set CGRAM address or set pointer to CGRAM location	0x40+add	64+add

Figure 12: LCD Commands

The RW pin stands for READ/WRITE. When RW is logic 0 it means that data is to be written to the LCD. Whereas when RW is logic 1, data is read from the LCD.

The LCD is connected to the PORTC of microcontroller. Out of the eight pins of PORTC, PCO to PC2 is used to generate contol signals for LCD and PC4 to PC7 is used for $data_transfer$. PC0 is connected to RS, PC1 to RW and PC2 to EN pin. PC4 to PC7 is connected to DB4 to DB7 pin of LCD respectively.

Refering to the code for $lcd_init()$, three variables namely data, upper and lower of unsigned character type are declared. Then PORTC is been bitwise anded with 0xFB. This makes the enable pin to go low. Immediately after, the data line is made 0 and RS bit is also made 0 to indicate that the following data is a command for LCD. A delay of around 10 millisecond is put. After that comes the task of LCD configuration. The hex value 28,0E and 06 is sent to the LCD and the routines $upper_lower()$ and $data_transfer()$ are called every time. This is done because the LCD is operated in 4-bit mode. Hence the 8-bit

data is sent in nibbles. The routine *upper_lower*() does the task of bifurcating the 8-bit data in to two 4-bit data. The routine *data_transfer*() is used to send the 4-bit data. The data is actually a command for the LCD, for this routine. Both the routines are shown below.

```
1  //Routine for extarcting upper and lower nibble
2  unsigned char upper_lower(unsigned char *dataptr,unsigned
       char *lowerptr,unsigned char *upperptr)
3  {
4   *upperptr=(*dataptr) & 0xF0;
5   *lowerptr=(*dataptr) & 0x0F;
6   *lowerptr=(*lowerptr<<4);
7   return (0);
8  }
9
10 //Routine for transferring command to LCD
11 unsigned char data_transfer(unsigned char upper,unsigned
      char lower)
12 {
13  PORTC&=0x0F;
14  PORTC|=upper;
15  strobe();
16
17  PORTC&=0x0F;
18  PORTC|=lower;
19  strobe();
20  _delay_ms(10);
21  }
```

The routine *data_transfer*() calls the subroutine *strobe*() which makes the necessary high to low transition of the Enable pin of LCD. The code is shown below.

```
1  void enable()// To make Enable pin high
2  {
3   PORTC|=0x04;
4  }
5
6  void disable()//To make Enable pin low
7  {
8   PORTC&=0xFB;
9  }
10
11 void strobe()
12 {
13  enable();
```

```
14  disable();
15  }
```

Uptill now we have just configured the LCD. Now we would look at the transmission of the data to be displayed on the LCD.
After completing the initialization of the LCD, the main routine calls the next routine, *lcd_print_sensor_data*(). The code for the same is given below.

```
1  //Routine to print the values of Temp, Fan, Hea and Ser
2  void lcd_print_sensor_data (void)
3  {
4   unsigned char data,upper,lower,i;
5   unsigned char upper_lower();
6   int temp1 = 0;
7
8
9  val[2] = val[1];
10 val[1] = 1.163*value;
11
12 for(i=2;i<11;i++)
13 {
14      if(val[i]==0)
15         {
16         val[i] = val[1];
17          }
18 }
19
20 for(i=15;i>2;i--)
21 {
22  val[i] = val[i-1];
23 }
24
25 temp_val = (int) val[1];
26
27 temperature_c = temp_val;
28 temp1 = temp_val;
29 test1 = temp_val;
30 lcd_data2[0] = (temp1/1000) + 48;
31 lcd_data2[1] = (temp1/100)%10 + 48;
32 lcd_data2[2] = (temp1/10)%10 + 48;
33 lcd_data2[3] = (temp1%10) + 48;
34
35  //Serial data________________
36  temp1 = ser_data;
37  lcd_data2[15] = (temp1 % 10) + 48;
38  lcd_data2[14] = (temp1 / 10);
```

```
39    lcd_data2[14] = (lcd_data2[14] % 10) + 48;
40    lcd_data2[13] = (temp1 / 100) + 48;
41
42    //Fan----------------
43    temp1 = fan;//---------------------
44    lcd_data2[7] = (temp1 % 10) + 48;
45    lcd_data2[6] = (temp1 / 10);
46    lcd_data2[6] = (lcd_data2[6] % 10) + 48;
47    lcd_data2[5] = (temp1 / 100) + 48;
48
49    //Heater----------------
50    temp1 = light; //-----------------
51    lcd_data2[11] = (temp1 % 10) + 48;
52    lcd_data2[10] = (temp1 / 10);
53    lcd_data2[10] = (lcd_data2[10] % 10) + 48;
54    lcd_data2[9] = (temp1 / 100) + 48;
55
56
57
58    refresh_lcd();
59
60
61    for(i=0; i<16; i++)
62
63    {
64     data=lcd_data1[i];
65     upper_lower(&data,&lower,&upper);
66     lcd_data_transfer(upper,lower);
67    }
68   goto_nextline();
69
70     for(i=0; i<16; i++)
71
72    {
73     data=lcd_data2[i];
74     upper_lower(&data,&lower,&upper);
75     lcd_data_transfer(upper,lower);
76    }
77
78   }
```

The temperature value to be displayed is a four digit number. Every digit is extracted and is sent as a single LCD character. *lcddata*1[] contains the information to be displayed on the upper line. Whereas *lcddata*2[] contains the information to be displayed on the lower line. Since now the data transmitted to the LCD is the one to be displayed, a different routine *lcd_data_transfer*()

is called. It makes the RS pin high. The routine is given below.

```
unsigned char lcd_data_transfer(unsigned char upper,
    unsigned char lower)
 {

  PORTC|=0x01;
  PORTC&=0x0F;

  PORTC|=upper;
   strobe();


  PORTC&=0x0F;
  PORTC|=lower;
   strobe();
   _delay_ms(10);
 }
```

To make the LCD print on the next line, *gotonextline*() routine is called. It must have been noted that before the data transmission process the *refreshlcd*() routine is called. It commands the LCD to clear its memory as well as the display pannel.

```
 void refresh_lcd()
{

unsigned char data,lower,upper;

_delay_us(10);
data=0x01;
upper_lower(&data,&lower,&upper);
data_transfer(upper,lower);
}
```

The task of refresing the LCD and printing the data on it is repeated continuously.

7 The Preprocessor Directives

The C code to work fine as far as WINAVR is concerned, the necessary header files must be included. The following header files must be included.

```
#include <avr/io.h>
#include<util/delay.h>
#include <avr/interrupt.h>
```

The header file *avr/io.h* enables the direct addressing of the microcontroller ports with names like PORTA PORTB, etc.

The header file *util/delay.h* enables the use of internal routines of WINAVR for generating microsecond and millisecond delays. Its usage is obvious because one cannot generate a delay using a for loop in WINAVR.

The header file *avr/interrupt.h* is used to use the available interrupt subroutine calls in WINAVR. We have used this function for serial communication and ADC interrupt handling.

8 The Complete C code

```
1
2 #include <avr/io.h>
3 #include<util/delay.h>
4 #include <avr/interrupt.h>
5
6
7 //LCD Variables
    -------------------------------------------
8 int lcd_data1[16]={'T','E','M','P',0X20,'F','A','N',0X20,
    'H','E','A',0X20,'S','E','R' };
9 int lcd_data2[16]={0X20,0X20,0X20,0X20,0X20,0X20,0X20,0
    X20,0X20,0X20,0X20,0X20,0X20,0X20,0X20,0X20};
10 float val[15] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
11
12 //Serial ISR variables
    ------------------------------------
13 int value=0;
14 int temperature_c = 0;
15 int test1 = 0;
16 int temp_val = 0;
17 unsigned char next_byte_light = 0;
18 unsigned char next_byte_fan = 0;
19 unsigned char temp_upper_byte_send;
20 unsigned char temp_lower_byte_send;
21 unsigned char ser_data = 0;
22 unsigned char light = 0;
23 unsigned char fan = 0;
24 int temperature_c_upper_byte = 0;
25 int temperature_c_lower_byte = 0;
26 int temperature = 0;
27 int r=1;
28
29
30 //Initializing microcontroller ports
31 void port_init(void)
32 {
33  PORTA = 0x00;
34  DDRA  = 0x00;
35  PORTC = 0x00; //m103 output only
36  DDRC  = 0xFF;
37  PORTD = 0x7F; //-----------7F
38  DDRD  = 0xF0;
39 }
```

```
40
41
42
43 //Initializing ADC
44 void adc_init(void)
45 {
46  ADCSRA = 0x00; //disable adc
47  ADMUX = 0x00; //select adc input 0
48  ACSR  = 0x80;
49  ADCSRA = 0xEF; //---------------0xEE----------0xEF
50  SREG|=0x80;
51   }
52
53
54 //ADC conversion complete interrupt subroutine
55 ISR(ADC_vect)
56 {
57    //conversion complete, read value (int) using...
58    temperature_c=ADCL;              //Read 8 low bits first
       (important)
59    temperature_c|=(int)ADCH << 8; //read 2 high bits and
       shift into top byte
60    value = temperature_c;
61   MCUCR = 0x00;
62 }
63
64
65 //Timer/Counter1 initialization for PWM generation
66 void timer1_init(void)
67 {
68  TCCR1B=0x00;//stop timer during configuration
69  TCCR1A=0xA1;
70  OCR1AH=0x00;
71  OCR1AL=0x00;//setting heater 0
72  OCR1BH=0x00;
73  OCR1BL=0xFC;//setting fan full
74  TCCR1B=0x0B;//start timer
75 }
76
77
78 //USART initialization for serial communication
79 void uart0_init(void)
80 {
81  UCSRB = 0x00; //disable while setting baud rate
82  UCSRA = 0x00;
83  UCSRC = 0x86;
```

```
84   UBRRL = 0x33; //set baud rate lo----------0x19--------0
        x33
85   UBRRH = 0x00; //set baud rate hi
86   UCSRB = 0x98; //0x98--------------
87  }
88
89
90  //USART receive complete interrupt subroutine
91  ISR(USART_RXC_vect)
92
93  {
94   //uart has received a character in UDR
95
96   ser_data = UDR;
97
98   if(ser_data < 253)
99   {
100   if (next_byte_light == 1)
101   {
102    light = ser_data;
103    if (light > 40)
104    {
105    light = 40;
106    }
107    OCR1AL = light; //heating element input
108    next_byte_light = 0;
109   }
110
111   if (next_byte_fan == 1)
112   {
113    fan = ser_data;
114    OCR1BL = fan; //fan speed input
115    next_byte_fan = 0;
116   }
117  }
118
119  if (ser_data == 254) //command for heater
        --------------
120  {
121   next_byte_light = 1;
122   next_byte_fan = 0;
123  }
124
125  if (ser_data == 253) //command for fan -------------
126  {
127   next_byte_light = 0;
```

```
128     next_byte_fan = 1;
129    }
130
131    if(ser_data == 255)
132    {
133     temperature = test1;
134     temperature_c_lower_byte = temperature%10;
135     temperature_c_upper_byte = temperature/10;
136     temp_upper_byte_send = (unsigned char)
          temperature_c_upper_byte;
137     temp_lower_byte_send = (unsigned char)
          temperature_c_lower_byte;
138     UDR = temp_upper_byte_send;
139     UDR = temp_lower_byte_send;
140     UDR = temp_upper_byte_send;
141     UDR = temp_lower_byte_send;
142     UDR = temp_upper_byte_send;
143     UDR = temp_lower_byte_send;
144    }
145
146
147
148    }
149
150
151
152
153
154  //Initialization routine
155  void init_devices(void)
156  {
157   //stop errant interrupts until set up
158   cli(); //disable all interrupts
159   port_init();
160   timer1_init();
161   uart0_init();
162   adc_init();
163   GICR  = 0xC0;
164   TIMSK = 0x40; //timer interrupt sources
165   sei(); //re-enable interrupts
166   //all peripherals are now initialised
167  }
168
169
170
171  void enable() // To make Enable pin high
```

```
172 {
173  PORTC|=0x04;
174 }
175
176 void disable()//To make Enable pin low
177 {
178  PORTC&=0xFB;
179 }
180
181 void strobe()
182 {
183  enable();
184  disable();
185  }
186
187
188
189
190
191 //Routine for extarcting upper and lower nibble
192 unsigned char upper_lower(unsigned char *dataptr,unsigned
      char *lowerptr,unsigned char *upperptr)
193 {
194  *upperptr=(*dataptr) & 0xF0;
195  *lowerptr=(*dataptr) & 0x0F;
196  *lowerptr=(*lowerptr<<4);
197  return (0);
198 }
199
200 //Routine for transferring command to LCD
201 unsigned char data_transfer(unsigned char upper,unsigned
     char lower)
202 {
203  PORTC&=0x0F;
204  PORTC|=upper;
205  strobe();
206
207  PORTC&=0x0F;
208  PORTC|=lower;
209  strobe();
210  _delay_ms(10);
211  }
212
213
214
215
```

```
216  //Routine for transferring data to LCD
217  unsigned char lcd_data_transfer(unsigned char upper,
       unsigned char lower)
218   {
219
220     PORTC|=0x01;
221     PORTC&=0x0F;
222
223     PORTC|=upper;
224     strobe();
225
226
227     PORTC&=0x0F;
228     PORTC|=lower;
229     strobe();
230     _delay_ms(10);
231   }
232
233
234   //Switching to next line of LCD
235 void goto_nextline()
236  {
237   unsigned char data,lower,upper;
238   _delay_ms(10);
239
240   PORTC&=0xFE;
241   data=0xC0;
242   upper_lower(&data,&lower,&upper);
243   data_transfer(upper,lower);
244  }
245
246
247   void refresh_lcd()
248  {
249
250  unsigned char data,lower,upper;
251
252  _delay_us(10);
253  data=0x01;
254  upper_lower(&data,&lower,&upper);
255  data_transfer(upper,lower);
256 }
257
258
259 //Initializing LCD
260 void lcd_init()
```

```
261 {
262
263  unsigned char data, upper, lower;
264  PORTC&=0xFB;
265  PORTC&=0x0F;
266
267
268  _delay_ms(10);
269
270  data=0x28;
271  upper_lower(&data,&lower,&upper);
272  data_transfer(upper, lower);
273
274  data=0x0E;
275  upper_lower(&data,&lower,&upper);
276  data_transfer(upper, lower);
277
278
279  data=0x06;
280  upper_lower(&data,&lower,&upper);
281  data_transfer(upper, lower);
282
283
284 }
285
286
287 //Routine to print the values of Temp, Fan, Hea and Ser
288 void lcd_print_sensor_data (void)
289 {
290  unsigned char data, upper, lower, i;
291  unsigned char upper_lower();
292  int temp1 = 0;
293
294
295 val[2] = val[1];
296 val[1] = 1.163*value;
297
298 for(i=2;i<11;i++)
299 {
300      if(val[i]==0)
301          {
302          val[i] = val[1];
303           }
304 }
305
306 for(i=15;i>2;i--)
```

```
307 {
308  val[i] = val[i-1];
309 }
310
311 temp_val = (int) val[1];
312
313 temperature_c = temp_val;
314 temp1 = temp_val;
315 test1 = temp_val;
316 lcd_data2[0] = (temp1/1000) + 48;
317 lcd_data2[1] = (temp1/100)%10 + 48;
318 lcd_data2[2] = (temp1/10)%10 + 48;
319 lcd_data2[3] = (temp1%10) + 48;
320
321  //Serial data________________
322  temp1 = ser_data;
323  lcd_data2[15] = (temp1 % 10) + 48;
324  lcd_data2[14] = (temp1 / 10);
325  lcd_data2[14] = (lcd_data2[14] % 10) + 48;
326  lcd_data2[13] = (temp1 / 100) + 48;
327
328  //Fan________________
329  temp1 = fan; //---------------------
330  lcd_data2[7] = (temp1 % 10) + 48;
331  lcd_data2[6] = (temp1 / 10);
332  lcd_data2[6] = (lcd_data2[6] % 10) + 48;
333  lcd_data2[5] = (temp1 / 100) + 48;
334
335  //Heater________________
336  temp1 = light; //-----------------
337  lcd_data2[11] = (temp1 % 10) + 48;
338  lcd_data2[10] = (temp1 / 10);
339  lcd_data2[10] = (lcd_data2[10] % 10) + 48;
340  lcd_data2[9] = (temp1 / 100) + 48;
341
342
343
344  refresh_lcd();
345
346
347  for(i=0; i<16; i++)
348
349  {
350   data=lcd_data1[i];
351   upper_lower(&data,&lower,&upper);
352   lcd_data_transfer(upper,lower);
```

```
353      }
354    goto_nextline();
355
356        for(i=0;  i<16;  i++)
357
358      {
359        data=lcd_data2[i];
360        upper_lower(&data,&lower,&upper);
361        lcd_data_transfer(upper,lower);
362      }
363
364    }
365
366
367    void  main(void)
368    {
369     init_devices();
370     lcd_init();
371     lcd_print_sensor_data();
372
373     while(r=1)
374     {
375       refresh_lcd();
376       lcd_print_sensor_data();
377     }
378    }
```