ECE 510: REAL TIME CONCEPTS
SPRING 2016
PROJECT REPORT

Title : TEMPERATURE CONTROL USING PC COOLING FAN
Author : VENKATA SASIKIRAN VEERAMACHANENI (G00977751)

Problem Statement:

The main idea of the project is to vary speed of a cooling fan based upon the temperature reading from a temperature sensor and at the same time to display the temperature reading and the fan speed on a mobile device connected through a Bluetooth module.

In RTOS point of view, the idea is to split the entire application into 5 different tasks and use Co-operative scheduling algorithm to schedule the tasks. Moreover, the tasks should be synchronized and should communicate with one other. Data should be exchanged between the tasks.

Architecture:

Co-operative Scheduler has been used to schedule the tasks. Many architectural components, like semaphores, global variables etc., have been used in order to make it functional.

The following are the different architectural components of the Co-operative scheduler:

- ❖ Regular tasks
- ❖ ISR tasks
- ❖ Scheduler task
- ❖ Library functions
- ❖ Registers
- ❖ Stack utilization
- ❖ Global variables
- ❖ Synchronization objects
- ❖ Communication objects
- ❖ Other scheduler objects

1. Regular tasks: The total application has been divided into total 5 tasks and of them three are regular tasks
   - i. Task 1 (Data transmission to mobile application)
   - ii. Task 2 (Measurement of temperature)
   - iii. Task 3 (Controlling the speed of the fan)
2. ISR tasks: Out of 5 tasks two are ISR tasks
   - i. Task 4 (Measurement of speed of fan)
   - ii. Task 5 (Push Button)

3. Scheduler task: This is subroutine which is called at the leave points in task 1. This can be called with the function call "scheduler()". In the subroutine all the semaphores are checked and the appropriate task would be chosen for the execution.
4. Library functions: Libraries have been created by writing different Header files and C files to implement functionalities of each task. The following are the various library functions I have used to implement each task:
   i. Task 1 (Data transmission to mobile application)
      - Uart_init()
      - Transmit_digit()
      - Transmit_char()
      - Transmit_value()
      - Transmit_space()
   ii. Task 2 (Measurement of temperature)
      - Adc_init()
      - Read_adc()
      - Calculate_temp()
   iii. Task 3 (Controlling the speed of fan)
      - Pwm_init()
      - Pwm_set()
   iv. Task 4 (Measurement of speed of fan)
      - Measurement_init()
      - Timer_interupt_init()
   v. Task 5 (Push button)
      - Push_button_init()
5. Registers: stack pointer has been used and manipulated to save the context and restore the context. Stack pointer and general purpose registers (R4-R15) are saved on to the stack in context save and are restored in context restore.
6. Stack utilization: By default compilers allots 160 Bytes for the stack on the RAM and the stack starts from 0x0023FC. The context would be saved from 0x00239C address on the stack.
7. Global variables: The following are the global variables used in programming:
   - Temp[]
   - Speed[]
   - Old_duty_cycle
   - Stack_pointer_TCB_start
   - Push
8. Synchronization objects: Semaphores have been used for the task synchronization. The following are the semaphores used in coding:
   - Temp_sem
   - Speed_sem
9. Communication objects: Few integer variables have been used as communication objects between task 1 and the other tasks. The other tasks compute the speed of fan and temperature reading of sensor and assign them to "temp_value" and "rpm" respectively. The task 1 reads the variables and transmit them to the mobile application.
10. Other scheduler objects: The other scheduler objects used are Hard Timers and Counters.

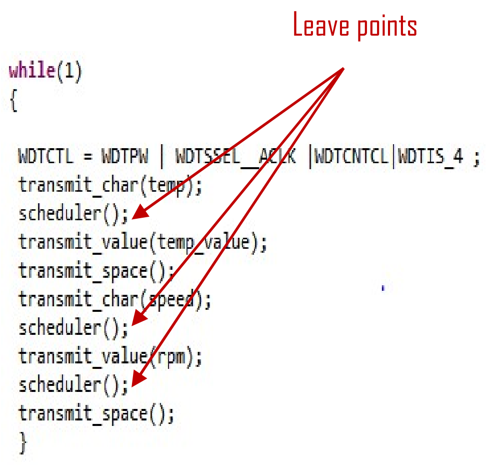
Tasks & their priorities:

There are total 5 tasks in the application and they are classified into Regular and ISR tasks. Among the regular tasks, all of them have the same priority and in the same way among the ISR tasks all of them have the same priority. When compared to Regular tasks, ISR tasks have highest priority. It is the Scheduler's duty to check the conditions on semaphores and choose an appropriate task when it comes to same priority tasks.

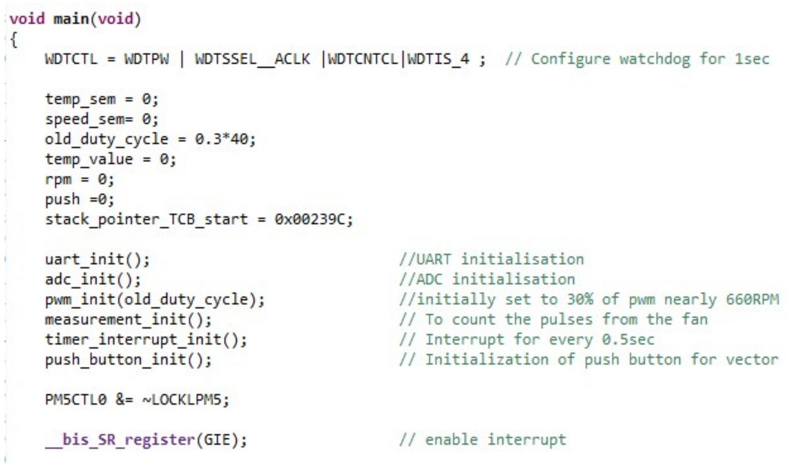
The following is the explanation of each task and its working:

1. Task 1 (Data Transmission):

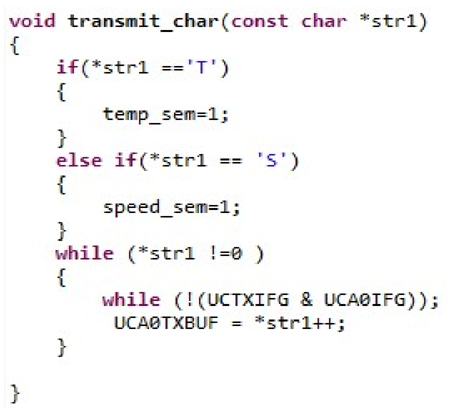
This is the main task. It transmits temperature reading and RPM of fan using a Bluetooth module (HC-06) to a mobile application. This task has been divided into initialization part and infinite loop part. In the initialization section it initializes all the registers of hardware modules, global variables, semaphores and communication objects. This is shown in below figure. The infinite loop part has leave points to the scheduler. In the below figure you can see this. At the leave points scheduler sub routine is called using function call "scheduler()". In the while loop when the transmit_char() function is called, based on the string transmitted, the semaphore is set. If, for suppose, 'TEMP::' string is transmitted, temp_sem is set. Next scheduler is called and the scheduler checks the semaphores and executes respective task. The below are the screenshots of code of various functions used in the task



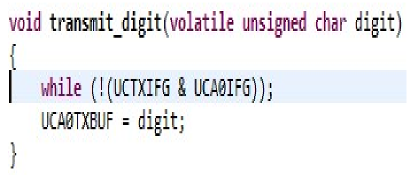
```
while(1)
{

 WDTCTL = WDTPW | WDTSSEL__ACLK |WDTCNTCL|WDTIS_4 ;
 transmit_char(temp);
 scheduler();
 transmit_value(temp_value);
 transmit_space();
 transmit_char(speed);
 scheduler();
 transmit_value(rpm);
 scheduler();
 transmit_space();
 }
```



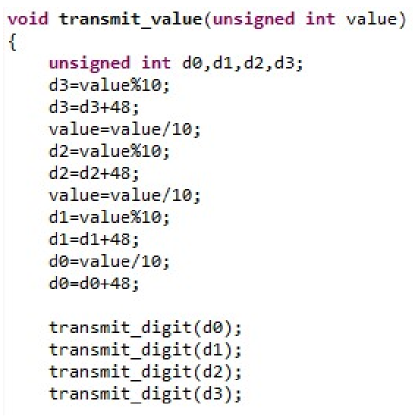
```
void main(void)
{
    WDTCTL = WDTPW | WDTSSEL__ACLK |WDTCNTCL|WDTIS_4 ;  // Configure watchdog for 1sec

    temp_sem = 0;
    speed_sem= 0;
    old_duty_cycle = 0.3*40;
    temp_value = 0;
    rpm = 0;
    push =0;
    stack_pointer_TCB_start = 0x00239C;

    uart_init();                         //UART initialisation
    adc_init();                          //ADC initialisation
    pwm_init(old_duty_cycle);            //initially set to 30% of pwm nearly 660RPM
    measurement_init();                  // To count the pulses from the fan
    timer_interrupt_init();              // Interrupt for every 0.5sec
    push_button_init();                  // Initialization of push button for vector

    PM5CTL0 &= ~LOCKLPM5;

    __bis_SR_register(GIE);              // enable interrupt
```



```
void transmit_char(const char *str1)
{
    if(*str1 =='T')
    {
        temp_sem=1;
    }
    else if(*str1 == 'S')
    {
        speed_sem=1;
    }
    while (*str1 !=0 )
    {
        while (!(UCTXIFG & UCA0IFG));
         UCA0TXBUF = *str1++;
    }

}
```



```
void transmit_digit(volatile unsigned char digit)
{
  while (!(UCTXIFG & UCA0IFG));
  UCA0TXBUF = digit;
}
```

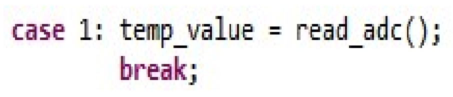


```
void transmit_value(unsigned int value)
{
    unsigned int d0,d1,d2,d3;
    d3=value%10;
    d3=d3+48;
    value=value/10;
    d2=value%10;
    d2=d2+48;
    value=value/10;
    d1=value%10;
    d1=d1+48;
    d0=value/10;
    d0=d0+48;

    transmit_digit(d0);
    transmit_digit(d1);
    transmit_digit(d2);
    transmit_digit(d3);
```

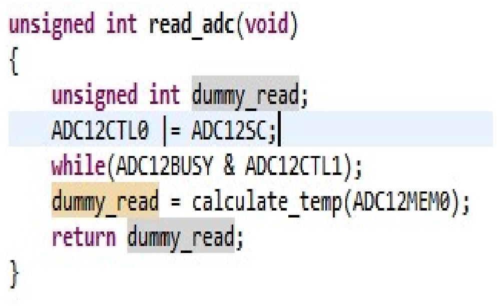
The “temp_value” and “rpm” are the variables used as communication objects through which other tasks communicate with the task 1. Task 2, when executed, sets variable “temp” and similarly Task 4, when executed, sets “rpm”. When the control is back to task 1 it reads these variables and transmit them.

2. <u>Task 2 (Measurement of Temperature):</u>

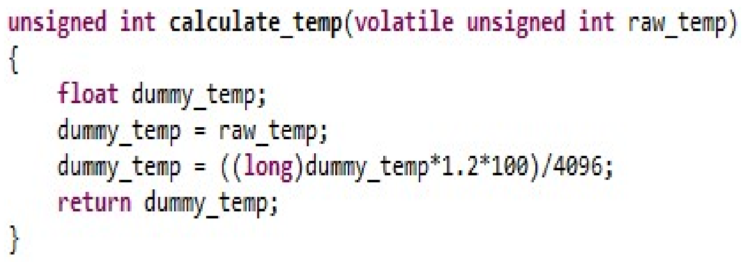
This task is to use 12bit ADC and take voltage reading from the temperature sensor (LM35) and convert that into Centigrade scale. After the conversion is done, it assigns that value to communication object “temp_value”. The below is the screen shots of code of library function used for this task. This task is executed by the scheduler only when “temp_sem” semaphore has been set.



```
case 1: temp_value = read_adc();
        break;
```



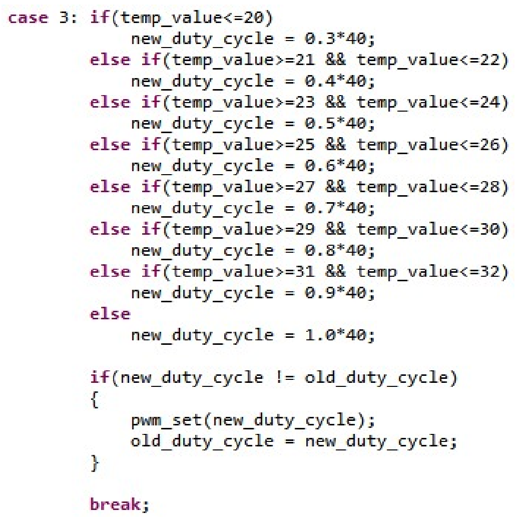
```
unsigned int read_adc(void)
{
    unsigned int dummy_read;
    ADC12CTL0 |= ADC12SC;
    while(ADC12BUSY & ADC12CTL1);
    dummy_read = calculate_temp(ADC12MEM0);
    return dummy_read;
}
```



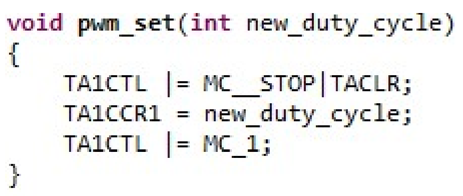
```
unsigned int calculate_temp(volatile unsigned int raw_temp)
{
    float dummy_temp;
    dummy_temp = raw_temp;
    dummy_temp = ((long)dummy_temp*1.2*100)/4096;
    return dummy_temp;
}
```

3. <u>Task 3 (Controlling speed of fan):</u>

This task is to set the duty cycle of the PWM output based on the temperature reading stored in “temp_value”, which is an intercommunication object. The “temp_value” is compared using if loop and corresponding value of duty cycle is selected. The below are the screen shots code for task 3.



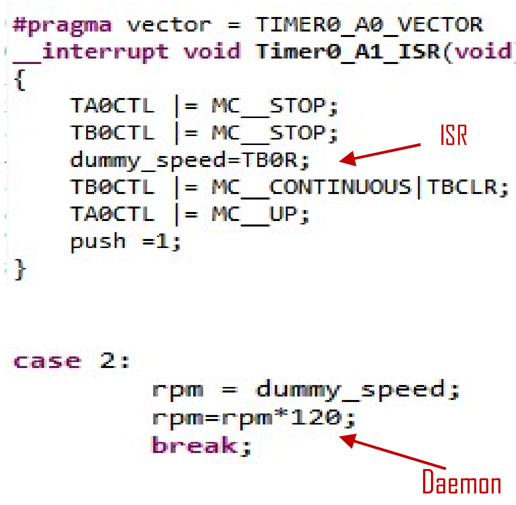
```
case 3: if(temp_value<=20)
            new_duty_cycle = 0.3*40;
        else if(temp_value>=21 && temp_value<=22)
            new_duty_cycle = 0.4*40;
        else if(temp_value>=23 && temp_value<=24)
            new_duty_cycle = 0.5*40;
        else if(temp_value>=25 && temp_value<=26)
            new_duty_cycle = 0.6*40;
        else if(temp_value>=27 && temp_value<=28)
            new_duty_cycle = 0.7*40;
        else if(temp_value>=29 && temp_value<=30)
            new_duty_cycle = 0.8*40;
        else if(temp_value>=31 && temp_value<=32)
            new_duty_cycle = 0.9*40;
        else
            new_duty_cycle = 1.0*40;

        if(new_duty_cycle != old_duty_cycle)
        {
            pwm_set(new_duty_cycle);
            old_duty_cycle = new_duty_cycle;
        }

        break;
```



```
void pwm_set(int new_duty_cycle)
{
    TA1CTL |= MC__STOP|TACLR;
    TA1CCR1 = new_duty_cycle;
    TA1CTL |= MC_1;
}
```

4. Task 4 (Measurement of speed of fan):

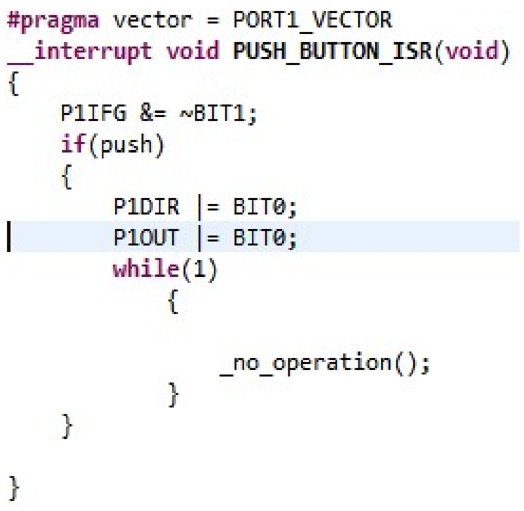
The purpose of this task is to measure the speed of revolving fan. The cooling fan used has an inbuilt hall sensor and it generates 2 pulses for every revolution of shaft. These pulses are fed as external clock to TIMER B0 of the micro controller and the pulses are counted. TIMER A0 has been used to generate interrupt for every 0.5sec to generate interrupt and measure the speed. This task has been divided into ISR + Daemon task in order to make the ISR as short as possible. In ISR the counter value is assigned to a dummy variable. In the daemon task, the dummy variable is transformed into RPM by performing necessary computations on the variable and value is assigned to "rpm", an intercommunication object. The daemon task gets executed by the scheduler only when semaphore "speed_sem" is set. The below are the screen shots of code used for this task.


```
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer0_A1_ISR(void)
{
    TA0CTL |= MC__STOP;
    TB0CTL |= MC__STOP;
    dummy_speed=TB0R;
    TB0CTL |= MC__CONTINUOUS|TBCLR;
    TA0CTL |= MC__UP;
    push =1;
}
```

```
case 2:
        rpm = dummy_speed;
        rpm=rpm*120;
        break;
```

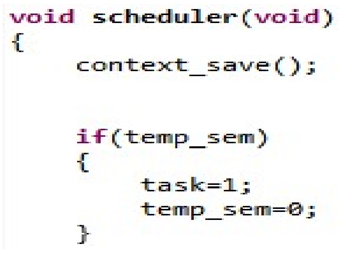
5. Task 5 (Push Button):

This task is to simply induce some disturbance into the system in order to analyze how system responds and rejuvenates from the disturbances. This is simply an empty infinite while loop ISR. When S1 push button on Launchpad is pressed, it generates interrupt and make the system to run in an empty while loop. This is simulated in order to show the resilience of system for unexpected disturbances.

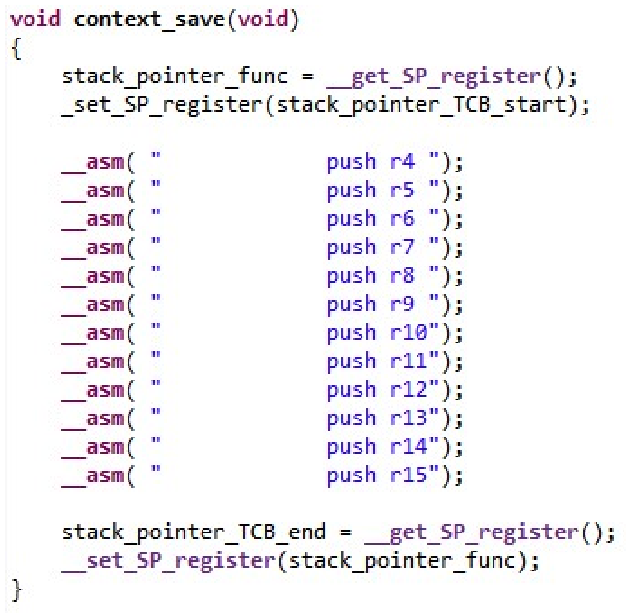

```
#pragma vector = PORT1_VECTOR
__interrupt void PUSH_BUTTON_ISR(void)
{
    P1IFG &= ~BIT1;
    if(push)
    {
        P1DIR |= BIT0;
        P1OUT |= BIT0;
        while(1)
            {

                _no_operation();
            }
    }

}
```

Context Switching:

In the main task, Task 1, we will have a leave points to the scheduler. When the leave point is executed, the control goes to scheduler. As soon as control is in hands of scheduler, it first duty is to do context save. So, it calls context_save function, which stores the present stack pointer value and sets the stack pointer to 0x00239C. It starts saving all the general purpose registers from that address top to down. At last, it also saves the end address of stack. Finally it reverts the stack pointer to the value it had before starting context save.

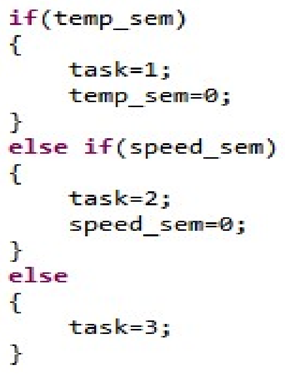


```
void scheduler(void)
{
    context_save();


    if(temp_sem)
    {
        task=1;
        temp_sem=0;
    }
```

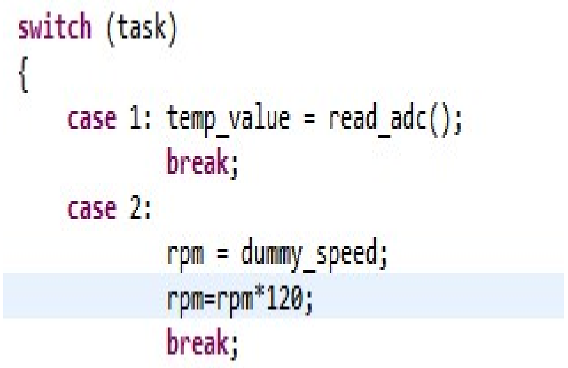


```
void context_save(void)
{
    stack_pointer_func = __get_SP_register();
    _set_SP_register(stack_pointer_TCB_start);

    __asm( "            push r4 ");
    __asm( "            push r5 ");
    __asm( "            push r6 ");
    __asm( "            push r7 ");
    __asm( "            push r8 ");
    __asm( "            push r9 ");
    __asm( "            push r10");
    __asm( "            push r11");
    __asm( "            push r12");
    __asm( "            push r13");
    __asm( "            push r14");
    __asm( "            push r15");

    stack_pointer_TCB_end = __get_SP_register();
    __set_SP_register(stack_pointer_func);
}
```

Next scheduler check the conditions on the semaphores and chooses appropriate task.

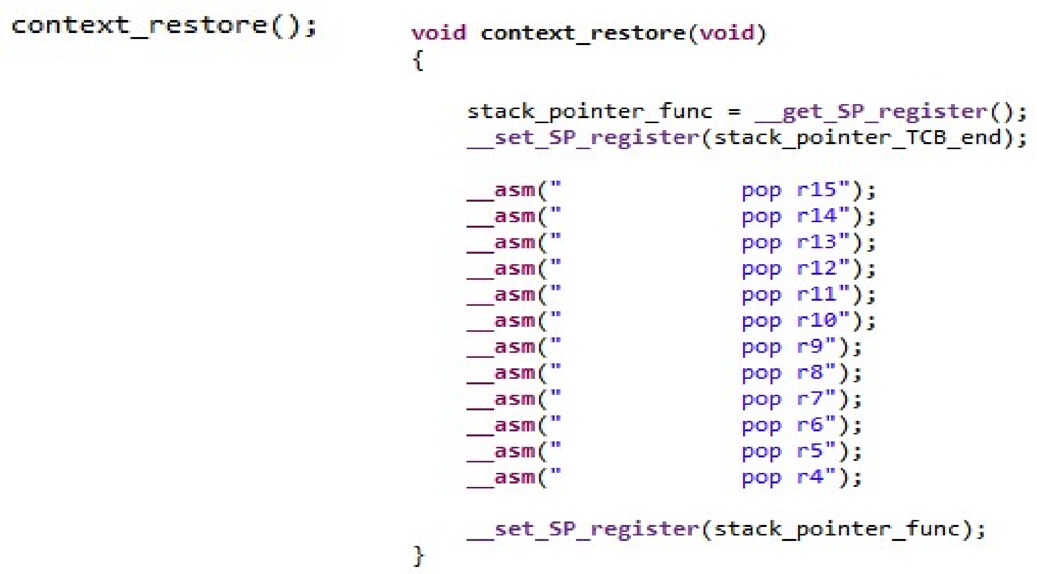


```
if(temp_sem)
{
    task=1;
    temp_sem=0;
}
else if(speed_sem)
{
    task=2;
    speed_sem=0;
}
else
{
    task=3;
}
```



```
switch (task)
{
    case 1: temp_value = read_adc();
            break;
    case 2:
            rpm = dummy_speed;
            rpm=rpm*120;
            break;
```

After the execution of chosen task is done, before leaving, it should perform context restore. So, it executes context_restore() subroutine before relinquishing its control to task 1. The context_restore() subroutine saves the present stack pointer and sets the pointer to the end of task control block by using the end address saved previously while running context_save() subroutine. It pops back all the stored values into respective general purpose registers. At last it reverts the stack pointer to the value it had before starting context restore. The below is the screenshot of context_restore() subroutine.

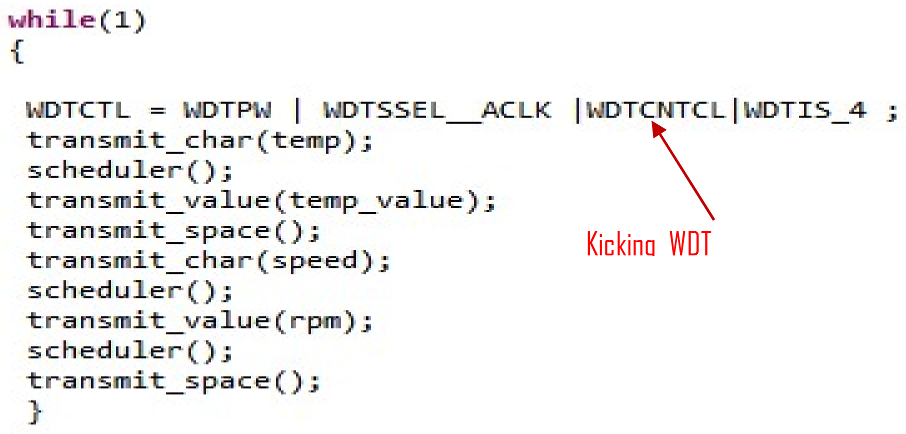


```
context_restore();          void context_restore(void)
                            {

                                stack_pointer_func = __get_SP_register();
                                __set_SP_register(stack_pointer_TCB_end);

                                __asm("          pop r15");
                                __asm("          pop r14");
                                __asm("          pop r13");
                                __asm("          pop r12");
                                __asm("          pop r11");
                                __asm("          pop r10");
                                __asm("          pop r9");
                                __asm("          pop r8");
                                __asm("          pop r7");
                                __asm("          pop r6");
                                __asm("          pop r5");
                                __asm("          pop r4");

                                __set_SP_register(stack_pointer_func);
                            }
```

Now the control moves back to task 1 and it continues until reaches another leave point. If a leave point is reached it follows the same procedure what we discussed above.

Resilience:

Watch Dog Timer has been used in the application in order to make it resilient to any unexpected disturbances. WDT is kicked at the start of every loop of Task 1. The timer has been set to 1sec. If any unexpected disturbance occurs, it prevents the appli8cation from WDT. This results in expiration of Timer, which in turn results in reset of system. This can be tested by push button S1, which is simulated to induce some disturbance into the system. The below is the code of Task 1 where first line of the loop is a command line which kicks the WDT at start of every loop.



```
while(1)
{

 WDTCTL = WDTPW | WDTSSEL__ACLK |WDTCNTCL|WDTIS_4 ;
 transmit_char(temp);
 scheduler();
 transmit_value(temp_value);
 transmit_space();
 transmit_char(speed);
 scheduler();
 transmit_value(rpm);
 scheduler();
 transmit_space();
 }
```

Micro-controller and its resources:

MSP430FR6989 Micro-controller has been used in the project. The following are its features:

- 1.8V – 3.6V operation
- 16-bit RISC architecture up to 16MHZ system clock and 8MHz FRAM process.
- 16 channel 12 bit ADC
- Five timers
- 83 GPIOs
- DMA

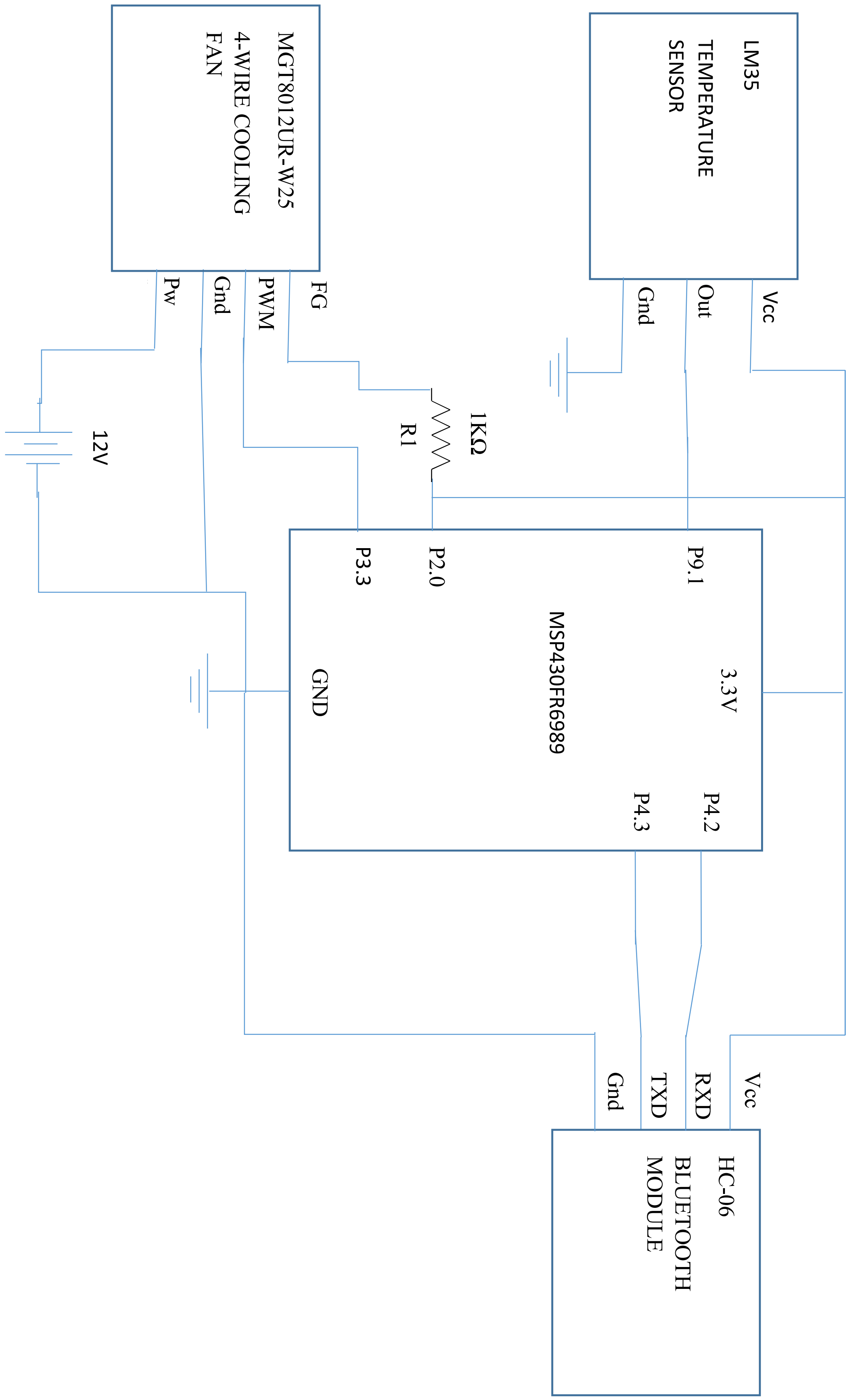
The following are the resources of MCU I have utilized I the project:

- Timer A1 for PWM generation
- Timer A0 for interrupt generation at every 0.5 sec
- Timer B0 for counting of pulses emanating from the cooling fan terminal
- eUSCI_A module in UART mode to transmit data to Bluetooth module (HC-06)
- One channel of 12 bit ADC to measure the voltage on output terminal of LM35 temperature sensor.

Experimentation:

The experimentation setup has been shown on the next page. It depicts all the components used and the connections between them. The following is the description of each component used:

1. LM35 (Temperature sensor):
   - It operates on 3.3V.
   - Its output is in millivolts and gives 10mV/℃.
   - The output terminal is connected to one of the ADC channels.
2. HC-06(Bluetooth Module):
   - It operates on 3.3V.
   - Its TXD is connected to RXD of MCU and its RXD is connected to TXD of MCU.
3. MGT8012UR-W25(4-Wire cooling fan):
   - It operates on 12V
   - PWM input can be given on one terminal.
   - For 0% duty cycle it runs at 650 RPM and for 100% duty cycle it runs at 4500RPM if rated voltage is applied.
   - It has hall sensor which gives out two pulses for each revolution. The output can be collected from one of the terminals.
   - 1KΩ resistor has been used to pull up the value fed to the MCU
   - The pulses are counted using a counter and speed can be determined.

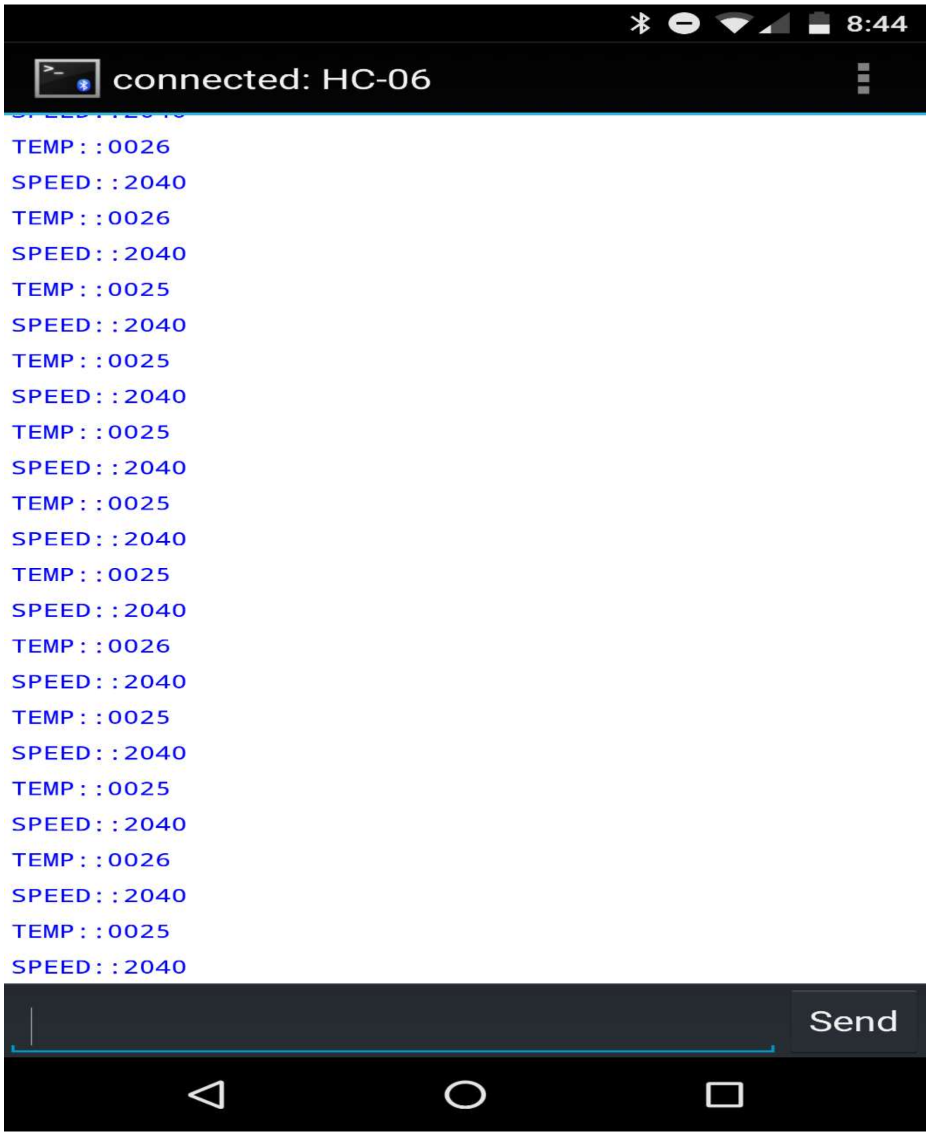

LM35
TEMPERATURE
SENSOR
Vcc
Out
Gnd
MGT8012UR-W25
4-WIRE COOLING
FAN
FG
PWM
Gnd
Pw
1KΩ
R1
12V
MSP430FR6989
P9.1
P2.0
P3.3
3.3V
GND
P4.2
P4.3
Vcc
RXD
TXD
Gnd
HC-06
BLUETOOTH
MODULE

Testing:

The connections are made as per the above setup and it is tested. I paired my mobile phone to the Bluetooth module and then values started displaying on my mobile screen. I increased the temperature of the sensor, then automatically updated values started displaying on my mobile screen. Moreover, due to increased temperature, the cooling also started to revolve at higher RPM. The updated speed is also displayed on the mobile screen. Next I wish to induce some disturbance into the system by pressing push button S1. As soon as I push the button, I stop seeing the updated values on my mobile screen. But because of WDT, my system jumps back to normal state after 1sec.

Results:

The below is the screen shot of the results displayed on my mobile phone.



```
TEMP::0026
SPEED::2040
TEMP::0026
SPEED::2040
TEMP::0025
SPEED::2040
TEMP::0025
SPEED::2040
TEMP::0025
SPEED::2040
TEMP::0025
SPEED::2040
TEMP::0025
SPEED::2040
TEMP::0026
SPEED::2040
TEMP::0025
SPEED::2040
TEMP::0025
SPEED::2040
TEMP::0026
SPEED::2040
TEMP::0025
SPEED::2040
```