LAB TITLE AND CODE : EMBEDDED COMPUTING LAB (19CLE283)
EXPERIMENT NUMBER : 1
DATE : 12/04/2022 (TUESDAY)

GPIO INTERFACING USING MSP432 (LED BLINKING & SEVEN SEGMENT)

* AIM:
To interface external peripherals, LED and seven segment, using MSP432 microcontroller.

* SOFTWARE REQUIRED:
Keil µVision 5 IDE (32 bit)
Publisher - ARM Ltd.
Version - 5.30.0.0

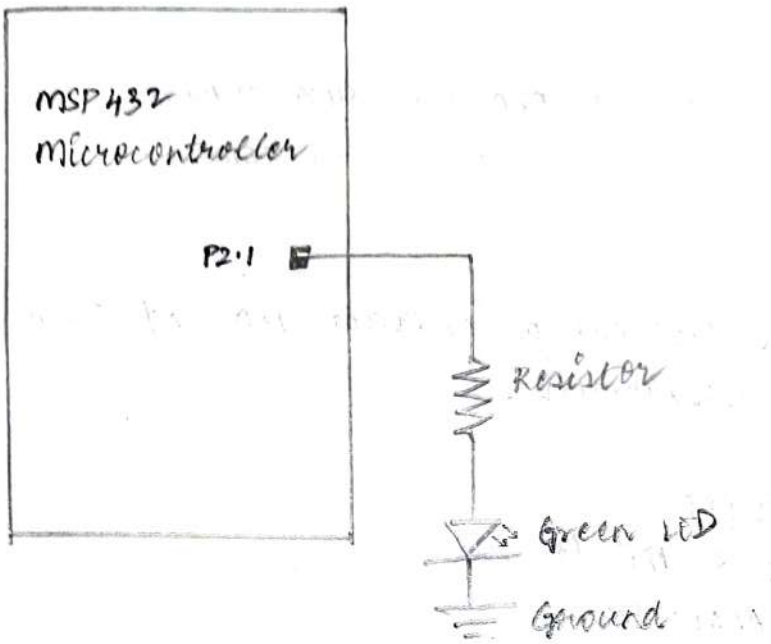
* ALGORITHM (LED BLINKING):
① Configure functionality of P2.1 as GPIO (General Purpose I/O) Port
② Configure Direction of P2.1 as Output Port
③ Switch on LED on P2.1 (configure Data as Logic High)
④ Delay
⑤ Switch off LED on P2.1 (configure Datas as Logic Low)
⑥ Delay

* CODE (LED BLINKING):

```
/*
 P2_1.c Toggling green LED in C using header file register definitions.
 This program toggles green LED for 0.5 second ON and 0.5 second OFF.
 The green LED is connected to P2.1.
 The LEDs are high active (a '1' turns ON the LED).

 Tested with Keil 5.30 and MSP432 Device Family Pack v2.2.0 on
 XMS432P401R Rev C.
*/
```

```
#include "msp.h"

void delayms (int n); // Delay Function

// Main Function:
int main (void)
{
  P2->SEL1 &= ~2; // Configure P2.1 as Simple I/O
  P2->SEL0 &= ~2;
  P2->DIR  |= 2;  // P2.1 set as output pin

  // Infinite Loop (An embedded program does not stop):
  while (1)
  {
    P2->OUT |= 2;  // Turn ON P2.1 Green LED
    delayms (500); // Delay for 500 ms
    P2->OUT &= ~2; // Turn OFF P2.1 Green LED
    delayms (500); // Delay for 500 ms
  }
}

// Delay milliseconds when system clock is at 3MHz for Rev C MCU:
void delayms (int n)
{
  int i, j;

  for (j=0; j<n; j++)
    for (i=750; i>0; i--); // Delay of 1ms
}
```

* CIRCUIT DIAGRAM (LED BLINKING) :



The green LED will turn ON for half a second and turn OFF for half a second repeatedly forever.

* ALGORITHM (SEVEN SEGMENT) :

① Define the hexadecimal values in an unsigned character array.
② Configure functionality of P4, P5.0 and P5.1 as simple I/O.
③ Configure direction of P4, P5.0 and P5.1 as output pins.
④ Display and select tens digit for each of the hexadecimal value repeatedly.
⑤ Delay

* CODE (SEVEN SEGMENT) :

```
#include "msp.h"

void delayms(int n);

// Main Function:
int main(void)
{
  const unsigned char digitPattern[] = {0x3F, 0x06, 0x5B, 0x4F,
  0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F, 0x77, 0x7C, 0x39, 0x5E, 0x79,
  0x71 }; // Hexadecimal Values

  P4 -> SEL1 &= ~0xFF; // configure P4 as simple I/O
  P4 -> SEL0 &= ~0xFF;
  P4 -> DIR |= 0xFF; // P4 set as output pins

  P5 -> SEL1 &= ~2; // configure P5.0, P5.1 as simple I/O
  P5 -> SEL0 &= ~2;
  P5 -> DIR |= 2; // P5.0, P5.1 set as output pins

  // Infinite Loop (An embedded program does not stop):
  while(1)
  {
    int i;
```

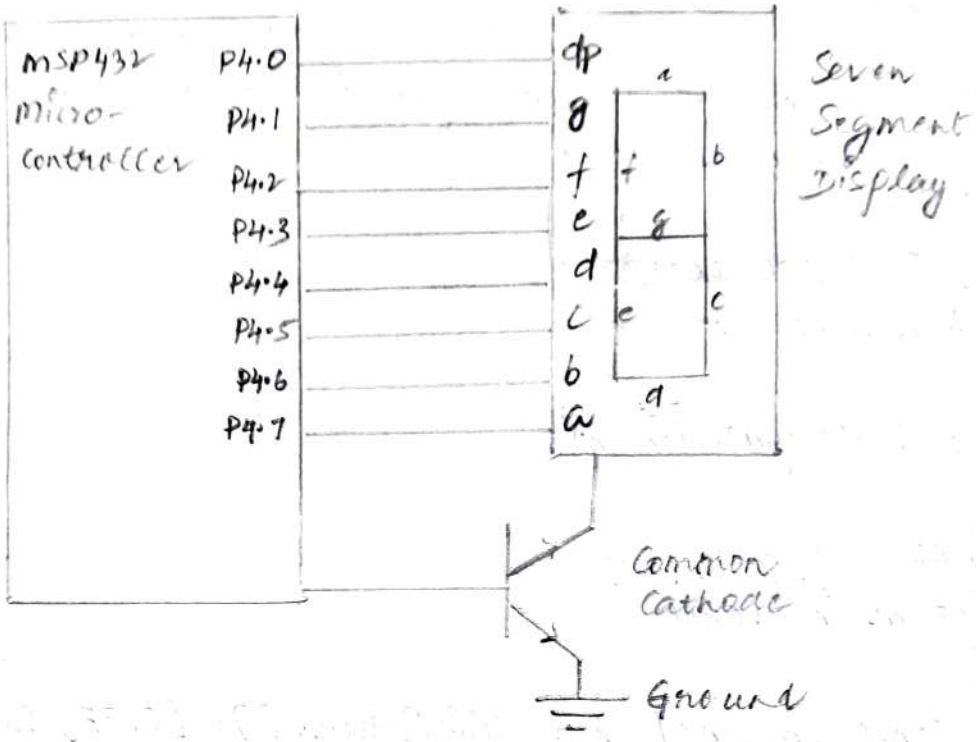
→ P11

+ SEVEN SEGMENT INTERFACING (DISPLAY)

	dp	g	f	e	d	c	b	a	Hex
	P4.0	P4.1	P4.2	P4.3	P4.4	P4.5	P4.6	P4.7	
0	0	0	1	1	1	1	1	1	0x3F
1	0	0	0	0	0	1	1	0	0x06
2	0	1	0	1	1	0	1	1	0x5B
3	0	1	0	0	1	1	1	1	0x4F
4	0	1	1	0	0	1	1	0	0x66
5	0	1	1	0	1	1	0	1	0x6D
6	0	1	1	1	1	1	0	1	0x7D
7	0	0	0	0	0	1	1	1	0x07
8	0	1	1	1	1	1	1	1	0x7F
9	0	1	1	0	1	1	1	1	0x6F
10(A)	0	1	1	1	0	1	1	1	0x77
11(b)	0	1	1	1	1	1	0	0	0x7C
12(C)	0	0	1	1	1	0	0	1	0x39
13(d)	0	1	0	1	1	1	1	0	0x5E
14(E)	0	1	1	1	1	0	0	1	0x79
15(F)	0	1	1	1	0	0	0	1	0x71

Common Cathode Type

+ CIRCUIT DIAGRAM :



All the positive terminals (cathode) of all the 8 LEDs are connected together and all the negative terminals are left alone (ground).

```
    for (i=0; i< 16; i++)
    {
      P4->OUT = digitPattern[i]; // Display Tens Digit
      P5->OUT |= 2; // Select Tens Digit
      delayMs(5000);
    }
  }
}
```

```
// Delay milliseconds when systems clock is at 3MHz for Rev C MCUs
void delayMs(int n)
{
  int i, j;
  for (j=0; j<n; j++)
    for (i=150; i>0; i--); // Delay of 1 ms
}
```

+ INFERENCE:
Interface external peripherals, LED and Seven Segment, using MSP432 microcontroller and all simulation results were verified successfully.

GPIO INTERFACING USING MSP432 (LED - SWITCH & SEVEN SEGMENT)

LAB TITLE AND CODE: EMBEDDED COMPUTING LAB (19CCE283)
EXPERIMENT NUMBER: 2
DATE: 19/04/2022 (TUESDAY)

* AIM:

To interface external peripherals, LED and Seven Segment, using MSP432 microcontroller.

* SOFTWARE REQUIRED:

Keil µVision 5 IDE (32 bit)
Publisher - ARM Ltd
Version - 5.30.0.0

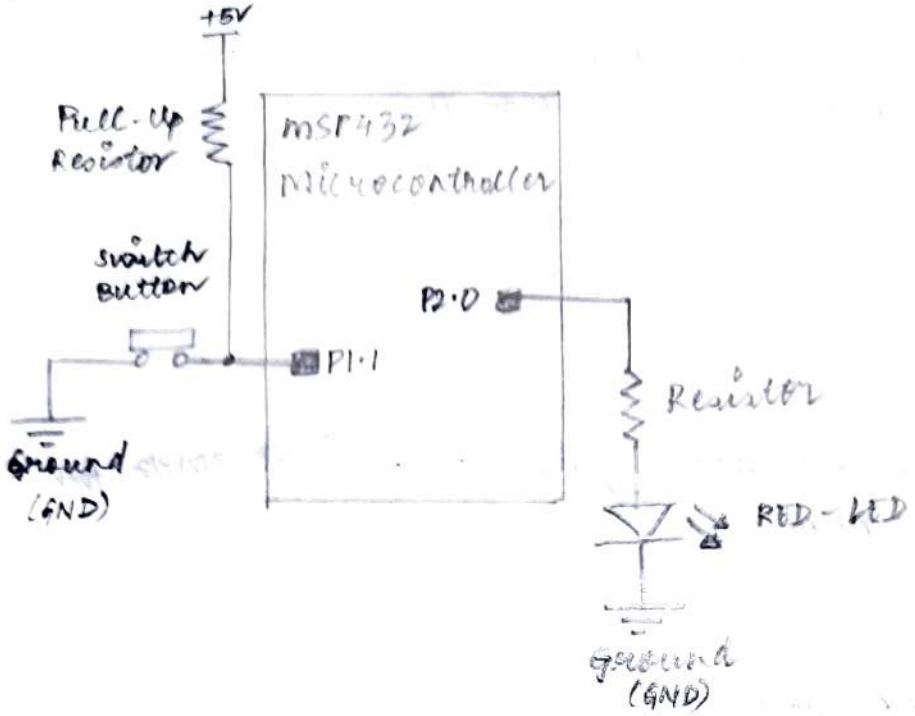
* ALGORITHM (LED-SWITCH): — (1)

(1) Configure functionality of P1.1 and P2.0 as simple GPIO Port.
(2) Configure direction of P1.1 as input pin.
(3) Enable P1.1 pull resistar; Pull up/down is selected by Px→OUT register.
(4) Configure direction of P2.0 as output pin.
(5) Use switch 1 to control the RED-LED.
(6) If not pressed, switch OFF RED-LED connected to P2.0
(7) Else, switch ON RED-LED connected to P2.0

* SOURCE CODE (LED-SWITCH):

```
/*
  p2_5.c Read a switch and write it to the LED.

  This program reads an external switch connected to P1.1 and
  writes the value to the RED-LED on P2.0. When switch is pressed,
  it connects P1.1 to ground and bit 1 of P1IN reads as '0'.

  P1.1 pin pull-up is enabled so that it is high when the switch is
  not pressed and bit 1 of P1IN reads as '1'. The LEDs are high
  active (a '1' turns ON the LED).
```

→ CIRCUIT DIAGRAM (LED · SWITCH):



Read a switch and write it to the LED.

```
  Tested with Keil 5.20 and MSP432 Device Family Pack V3.2.0 on
  XMS432P401R Rev C.
*/

#include "msp.h"

// main Function :
int main (void)
{
  P1 -> SEL1 &= ~2; // Configure P1.1 as simple GPIO Port
  P1 -> SEL0 &= ~2;

  P1 -> DIR &= ~2; // Configure direction of P1.1 as input pin
  P1 -> REN |= 2; // P1.1 pull resistor enabled.
  P1 -> OUT |= 2; // Pull up/down is selected by Px -> OUT register

  P2 -> SEL1 &= ~1; // Configure P2.0 as simple GPIO Port
  P2 -> SEL0 &= ~1;
  P2 -> DIR |= 1; // Configure direction of P2.0 as output pin

  // Infinite loop (An embedded program does not stop);
  while (1)
  {
    if (P1 -> IN & 2) // use switch 1 to control the RED-LED
      P2 -> OUT &= ~1; // If not pressed, switch OFF RED-LED
                       connected to P2.0
    else
      P2 -> OUT |= 1; // Else, switch ON RED-LED connected to P2.0
  }
}
```

PTO

* ALGORITHM (LED - SWITCH - COUNTER) : - (2)

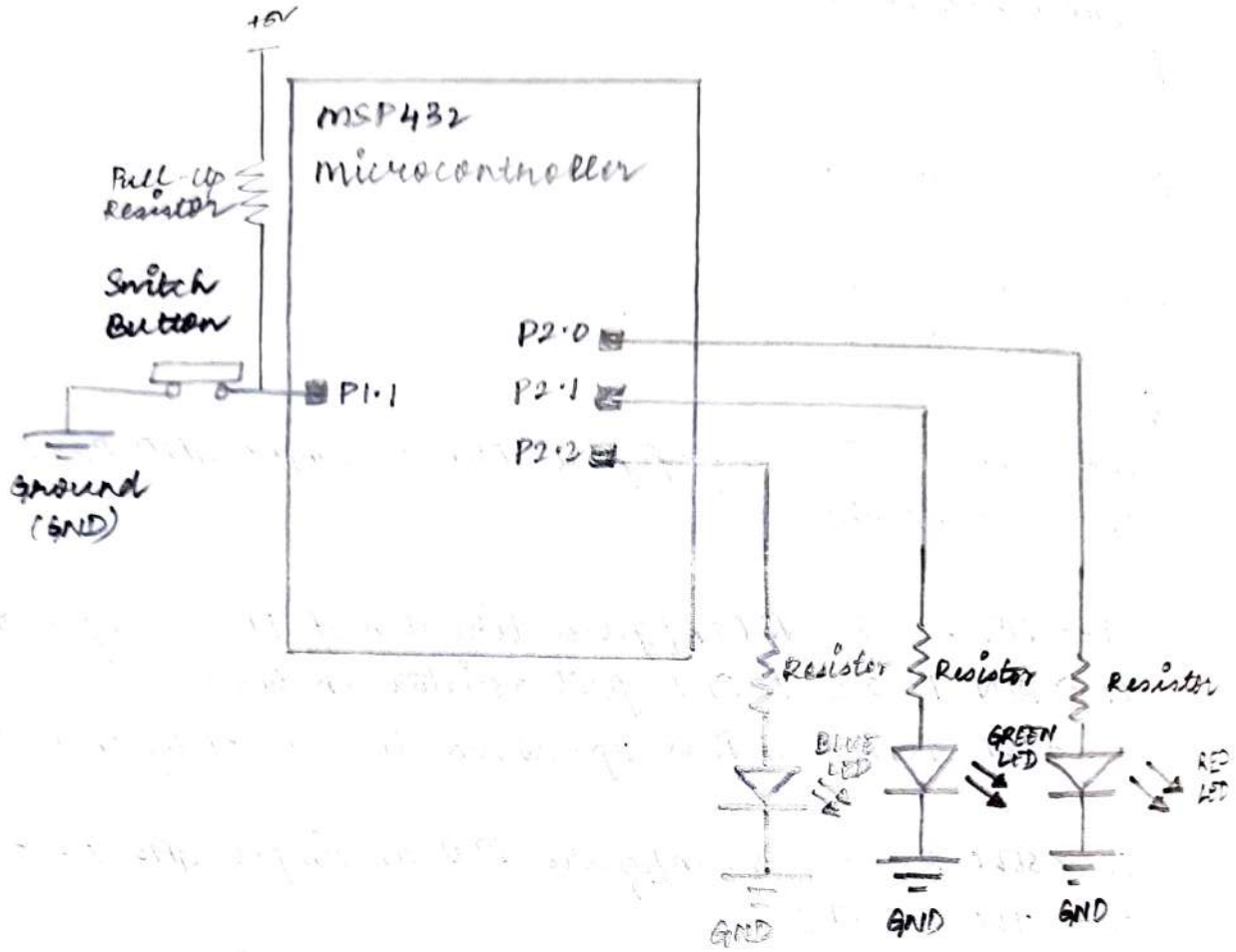
① Configure P1.1, P2.0, P2.1, P2.2 as simple GPIO Port.
② Configure direction of P1.1 as input pin.
③ Enable P1.1 pull resistor ; Pull up/down is selected by Px→OUT register.
④ Configure direction of P2.0, P2.1, P2.2 as output pin.
⑤ Use switch 1 to control the LEDs.
⑥ If pressed, switch ON the LED connected to the corresponding port.
⑦ Delay

* CODE (LED - SWITCH - COUNTER) :

```
#include "msp.h"
void delayMs(int n);
unsigned char i;

// main Function :
int main(void)
{
  P1->SEL1 &= ~2;  // Configure functionality of P1.1 as simple GPIO port
  P1->SEL0 &= ~2;
  P1->DIR  &= ~2;  // Configure direction of P1.1 as input pin

  P1->REN |= 2;  // P1.1 pull resistor enabled
  P1->OUT |= 2;  // Pull up/down is selected by Px->OUT register

  P2->SEL1 &= ~7;  // Configure functionality of P2.0, P2.1, P2.2
                   CRED, GREEN and BLUE LEDs) as simple GPIO port

  P2->SEL0 &= ~7;
  P2->DIR |= 7;  // Configure direction of P2.0, P2.1, P2.2 as
                    output pin
```

P.T.O

* CIRCUIT DIAGRAM (LED- SWITCH- COUNTER):



```
// Infinite loop (An embedded program does not stop):
while (1)
{
  if (P1 -> IN & 2) // Use switch 1 to control the LEDs
  {
    for (i = 0; i <= 7; i++)
    {
      P2 -> OUT |= i; // If pressed, switch ON the LED connected
                      to the corresponding port
      delayms (500); // Delay of 500 ms
    }
  }
}
}

// Delay milliseconds when system clock is at 3MHz for Rev C MCU:
void delayms (int n)
{
  int i, j;
  for (j=0; j<n; j++)
    for (i= 750; i>0; i--); // Delay of 1 ms
}
```

* ALGORITHM (SWITCH – SEVEN SEGMENT) : - (3)

① Define the hexadecimal values in an unsigned character array.
② Configure functionality of P1.1 as simple GPIO Port.
③ Configure direction of P1.1 as input pin.
④ Enable P1.1 pull resistor; Pull up/down is selected by Px -> OUT register.
⑤ Configure functionality of P4 as simple GPIO Port.
⑥ Configure direction of Port 4 as output pins.
⑦ Use switch 1 to control the external peripheral.
⑧ Display and Select Tens Digit of the Seven Segment.
⑨ Delay

* SEVEN SEGMENT INTERFACING (DISPLAY) :

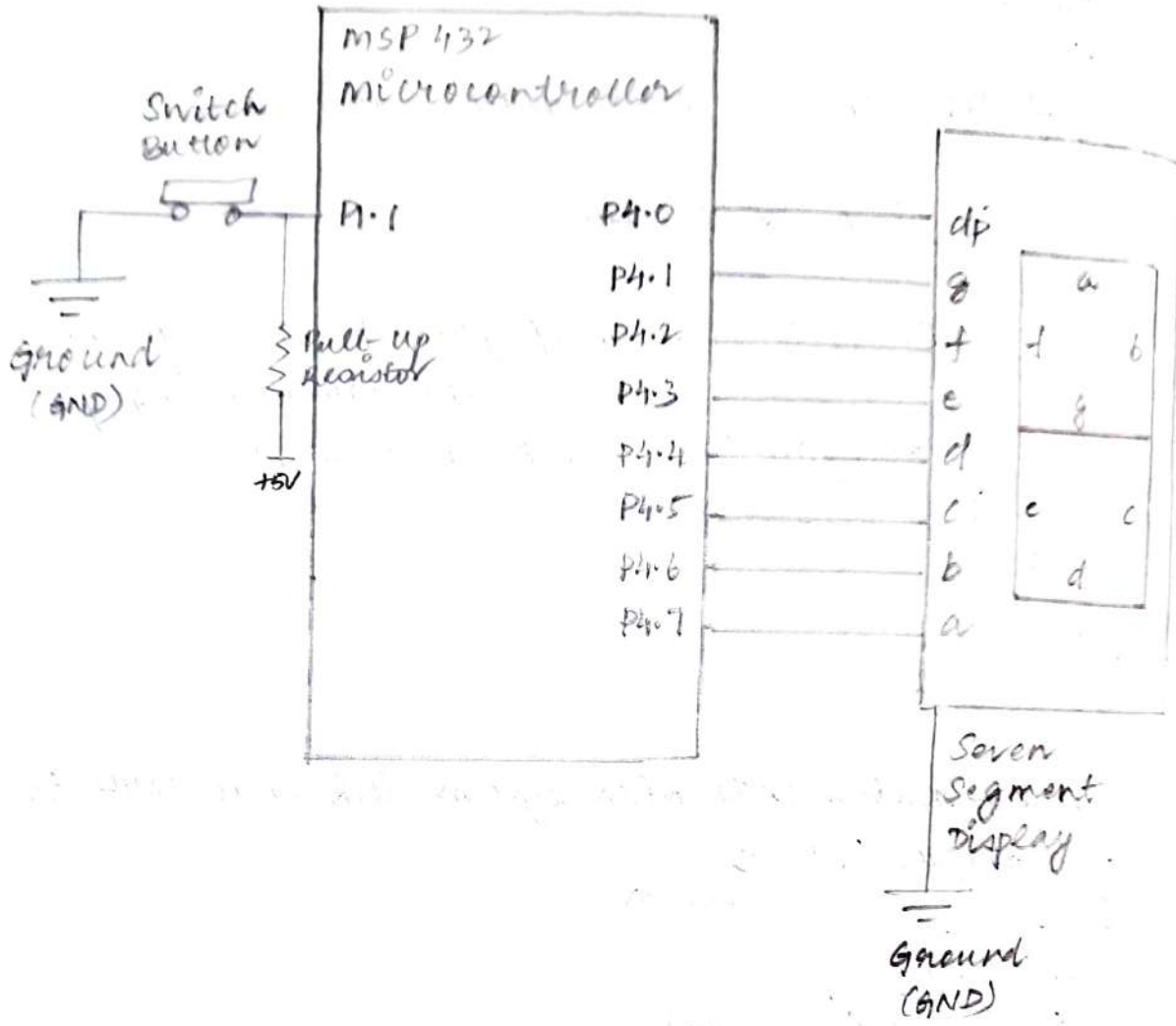
Decimal	dp (Decimal Point) P4.0	g P4.1	f P4.2	e P4.3	d P4.4	c P4.5	b P4.6	a P4.7	Hex
0	0	0	1	1	1	1	1	1	0x3F
1	0	0	0	0	0	1	1	0	0x06
2	0	1	0	1	1	0	1	1	0x5B
3	0	1	0	0	1	1	1	1	0x4F
4	0	1	1	0	0	1	1	0	0x66
5	0	1	1	0	1	1	0	1	0x6D
6	0	1	1	1	1	1	0	1	0x7D
7	0	0	0	0	0	1	1	1	0x07
8	0	1	1	1	1	1	1	1	0x7F
9	0	1	1	0	1	1	1	1	0x6F
10 (H)	0	1	1	1	0	1	1	1	0x77
11 (b)	0	1	1	1	1	1	0	0	0x7C
12 (C)	0	0	1	1	1	0	0	1	0x39
13 (d)	0	1	0	1	1	1	1	0	0x5E
14 (E)	0	1	1	1	1	0	0	1	0x79
15 (F)	0	1	1	1	0	0	0	1	0x71

Common Cathode Type

* SOURCE CODE (SWITCH - SEVEN SEGMENT) :

```
#include "msp.h"
void delayMs (int n); // Delay Function
unsigned char i;

// main Function :
int main (void)
{
 const unsigned char digitPattern [] = {0x3F, 0x06, 0x5B, 0x4F, 0x66,
 0x6D, 0x7D, 0x07, 0x7F, 0x6F, 0x77, 0x7C, 0x39, 0x5E, 0x79,
 0x71}; // Hexadecimal values

 P1 -> SEL1 &= ~2; // Configure functionality of P1.1 as simple
                      GPIO Port
 P1 -> SEL0 &= ~2;
 P1 -> DIR &= ~2; // Configure direction of P1.1 as input pin

 P1 -> REN |= 2; // P1.1 pull resistor enabled
 P1 -> OUT |= 2; // Pull up/down is selected by Px -> OUT register

 P4 -> SEL1 &= ~0xFF; // Configure functionality of P4 as simple
                         GPIO Port
 P4 -> SEL0 &= ~0xFF;
 P4 -> DIR |= 0xFF; // Configure direction of Port 4 as output pins

 // Infinite Loop (An embedded program does not stop):
 while (1)
 {
  if (P1 -> IN & 2) // Use switch 1 to control the external peripher
   for (i= 0; i< 16; i++)
   {
     P4 -> OUT |= digitPattern [i]; // Display Tens Digit
     P5 -> OUT |= 2; // Select Tens Digit
     delayMs (500); // Delay of 500 ms
   }
```

CIRCUIT DIAGRAM (SWITCH - SEVEN SEGMENT) :



All the positive terminals (cathode) of all the 8 LEDs are connected together. All the negative terminals (anode) are left alone (ground).

```
    }
}
```

```
// Delay milliseconds when system clock is at 8MHz for Rev C MCU:
void delayMs(int n)
{
  int i, j;
  for (j = 0; j < n; j++)
    for (i = 750; i > 0; i--); // Delay of 1ms
}
```

* **INFERENCE:**

Interface external peripherals, LED and seven segment, using MSP32 microcontroller and all simulation results were verified successfully.

LAB TITLE AND CODES EMBEDDED COMPUTING LAB (19CLE213)
EXPERIMENT NUMBER : 3
DATE : 26/04/2022 (TUESDAY)

* AIM :
To configure MSP43x on-chip peripherals and establish serial communication - transmission and reception.

* ALGORITHM : (CONTINUOUS TRANSMISSION OF A CHARACTER USING UART)
① Put in reset mode and disable oversampling.
② Set 8-bits data - NO Parity - 1 stop bit for Transmission (LSB)
③ In asynchronous mode, first LSB and then MSB should be set, followed by SWRST.
④ Enabled EUSCI_A0 logic is held in reset state (SW).
⑤ Configure functionality of P1.7, P1.3 as UART pins.
⑥ Take UART out of reset mode.
⑦ In the main function after calling the UART transmission function, perform the following steps in an infinite loop -
(i) Wait until transmitter buffer is empty
(ii) Send a character
(iii) Delay

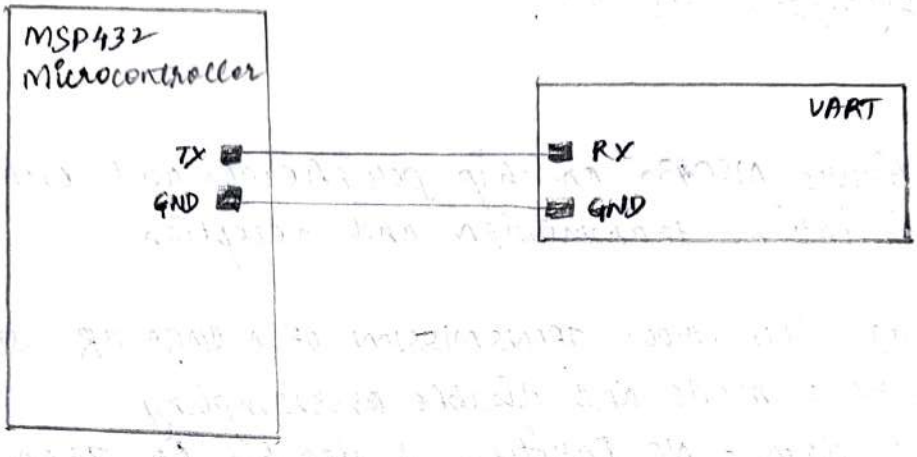
* SOURCE CODE : (CONTINUOUS TRANSMISSION OF A CHARACTER USING UART)

```
#include "msp.h"
void UART0_init(void); // Function for UART Transmission
void delayms(int n); // Delay Function

// main Function :
int main(void)
{
  UART0_init(); // Call UART Transmission Function
```

→ P1

```
// Infinite Loop (An embedded program does not stop):
while (1)
{
  while (!(EUSCI_A0 -> IFG & 0x02)) {} // Wait until transmission
                                          buffer is empty
  EUSCI_A0 -> TXBUF = 'Y'; // Send a character

  while (!(EUSCI_A0 -> IFG & 0x02)) {} // Wait until transmission
                                          buffer is empty
  EUSCI_A0 -> TXBUF = 'E'; // Send a character

  while (!(EUSCI_A0 -> IFG & 0x02)) {} // Wait until transmission
                                          buffer is empty
  EUSCI_A0 -> TXBUF = 'S'; // Send a character

  delayms(2); // Delay by 2 ms
}
}

// Function for UART Transmission:
void UART0_init(void)
{
  EUSCI_A0 -> CTLW0 |= 1; // Put in reset mode to configure UART
  EUSCI_A0 -> MCTLW = 0; // Disable oversampling
  EUSCI_A0 -> CTLW0 = 0x0081; // 00 - 1 stop bit for Transmission,
  No Parity, 8-bits data, Asynchronous mode, First LSB Then MSB,
  SMCLK, 81 - Enabled EUSCI_A0 Logic is held in reset state

  EUSCI_A0 -> BRW = 26; // 3000000/115200 = 26
  P1 -> SEL0 |= 0x0C; // Configure functionality of P1.2, P1.3 as
                         UART pins
  P1 -> SEL1 &= ~0x0C;
  EUSCI_A0 -> CTLW0 &= ~1; // Take UART out of reset mode
}
```

* OUTPUT (CONTINUOUS TRANSMISSION OF A CHARACTER USING UART) :



→ SERIAL WINDOW :-

YES YES YES YES...

```
// Delay milliseconds when system clock is at 3 MHz for Rev C MCU:
void delayMs (int n)
{
  int i, j;
  for (j=0; j<n; j++)
     for (i=750; i>0; i--); // Delay of 1ms
}
```

* **ALGORITHM: (RECEPTION AND TRANSMISSION OF A CHARACTER USING UART)**

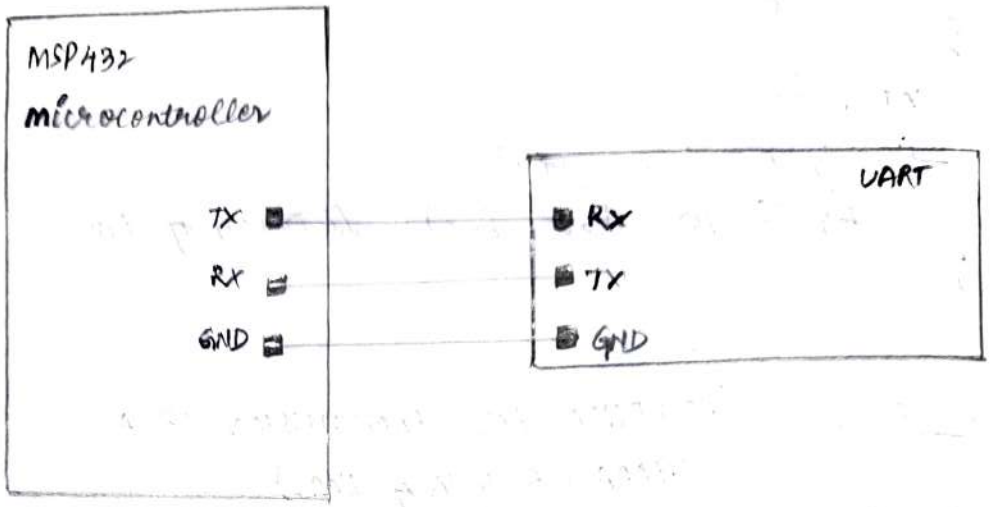
① Put in reset mode and disable oversampling.
② set 8-bits data - NO Parity - 1 stop bit for transmission. In asynchronous mode, first LSB and then MSB should be set, followed by SMCLK (00).
③ Enabled EUSCI_A0 logic is held in reset state (01).
④ Configure functionality of P1.2, P1.3 as UART pins.
⑤ Take UART out of reset mode.
⑥ In the main function after calling the UART transmission function, wait until transmitter buffer is empty and then, send the required number of characters bit-by-bit in a for loop.
⑦ In an infinite loop, perform the following steps -
(i) Wait until transmitter buffer is empty
(ii) Receive a character
(iii) Wait until transmitter buffer is empty
(iv) Display the character

* **SOURCE CODE: (RECEPTION AND TRANSMISSION OF A CHARACTER USING UART)**

```
#include "msp.h"
void UART0_init (void); // Function for UART Transmission
void delayMs (int n); // Delay Function

unsigned char c;
unsigned char message[] = "Ready\n";
int i;
```

```
// main function:
int main (void)
{
  UART0_init(); // call UART Transmission Function

  for (i=0; i<7; i++)
  {
    while (!(EUSCI_A0 -> IFG & 0x02)) {} // Wait until transmitter
                                            buffer is empty
    EUSCI_A0 -> TXBUF = message[i]; // Send a character
  }

  // Infinite Loop (An embedded program does not stop):
  while (1)
  {
    while (!(EUSCI_A0 -> IFG & 0x02)) {} // Wait until transmitter
                                            buffer is empty
    c = EUSCI_A0 -> RXBUF; // Receive a character

    while (!(EUSCI_A0 -> IFG & 0x02)) {} // Wait until transmitter
                                            buffer is empty
    EUSCI_A0 -> RXBUF = c; // Display the character
  }
}

// Function for UART Transmission:
void UART0_init (void)
{
  EUSCI_A0 -> CTLW0 |= 1; // Put in reset mode to configure UART
  EUSCI_A0 -> MCTLW = 0; // Disable oversampling
  EUSCI_A0 -> CTLW0 = 0x0081; // 00 - 1 stop bit, NO Parity, 8-bits data,
  Asynchronous Mode, First LSB Then MSB, SMCLK (00), Enabled
  EUSCI_A0 logic is held in reset state (81)
```

* OUTPUT : (RECEPTION AND TRANSMISSION OF A CHARACTER USING UART)



→ SERIAL WINDOW :-

Ready

Receive a character : Amrita

Diplay the character : Amrita

```
    EUSCI_A0 → BRW = 26;  // 3000000 / 115200 = 26
    P1 → SEL0 |= 0x0C;  // Configure functionality of P1.2, P1.3 as UART pins
    P1 → SEL1 &= ~0x0C;
    EUSCI_A0 → CTLW0 &= ~1;  // Take UART out of reset mode
}

// Delay milliseconds when system clock is at 3MHz for Rev C MCU:
void delayMs (int n);
{
  int i, j;
  for (j=0; j<n; j++)
    for (i=750; i>0; i--); // Delay of 1 ms
}
```

* ALGORITHM: (DEVICE CONTROL USING UART)

① Put in reset mode and disable oversampling.

② Set 8-bits data - NO Parity - 1 stop bit for transmission. In asynchronous mode, first LSB and then MSB should be set, followed by SMCLK (00).

③ Enabled EUSCI_A0 logic is held in reset state (01).

④ Configure functionality of P1.2, P1.3 as UART pins.

⑤ Take UART out of reset mode.

⑥ In the main function after calling the UART transmission function, configure functionality of P2.1 as simple GPIO Port and configure direction of P2.1 as LED output.

⑦ In an infinite loop, perform the following steps -

(i) Wait until transmitter buffer is empty.

(ii) Receive a character

(iii) If character is A/a, turn ON P2.1 Green LED.

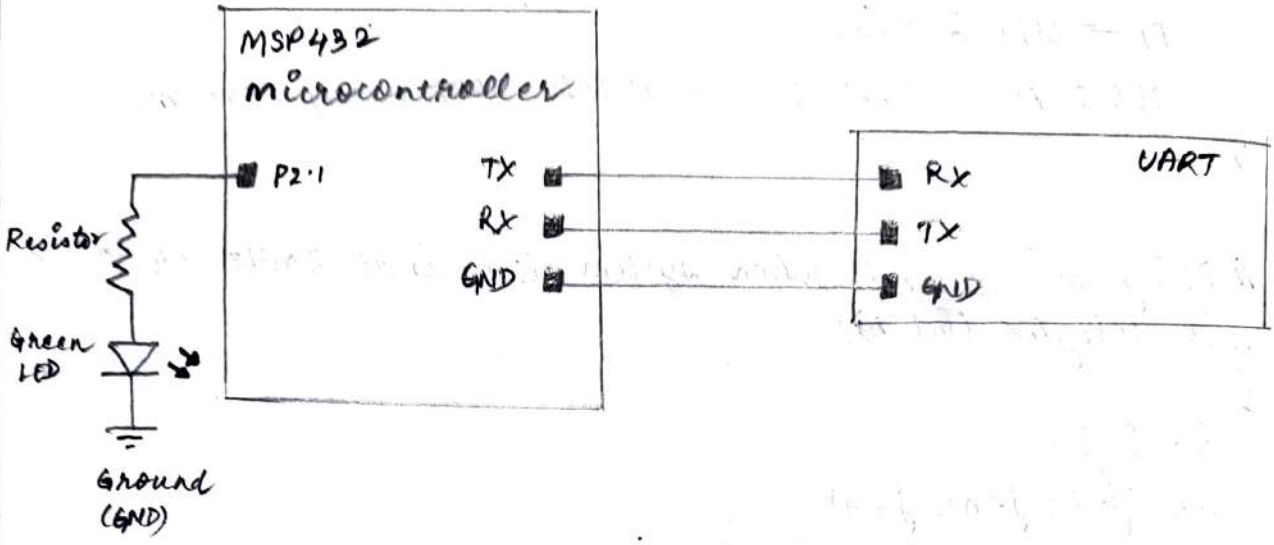
(iv) Else if character is B/b, turn OFF P2.1 Green LED.

PTO

* SOURCE CODE : (DEVICE CONTROL USING UART) :

```
#include "msp.h"
void UART0_init(void); //Function for UART Transmission
unsigned char c;

//Main Function:
int main(void)
{
  UART0_init(); // Call UART Transmission Function

  P2->SEL1 &= ~2; // Configure functionality of P2.1 as simple GPIO Port
  P2->SEL0 &= ~2;
  P2->DIR |= 2; // Configure direction of P2.1 as LED output

  // Infinite Loop (An embedded program does not stop):
  while(1)
  {
   while(!(EUSCI_A0->IFG & 0x01)) {} // wait until transmitter buffer is empty
   c = EUSCI_A0->RXBUF; // Receive a character

   if(c == 'A' || c == 'a')
   {
    P2->OUT |= 2; // TURN ON P2.1 Green LED
   }
   else if(c == 'B' || c == 'b')
   {
    P2->OUT &= ~2; // TURN OFF P2.1 Green LED
   }
  }
}
```

* OUTPUT : (DEVICE CONTROL USING UART)



→ SERIAL WINDOW :-

Receive a character: ABABAB

```
// Function for UART Transmission :
void UART0_ init (void)
{
  EUSCI_A0 → CTLW0 |= 1 ; // Put in reset mode to configure UART
  EUSCI_A0 → MCTLW = 0 ; // Disable oversampling
  EUSCI_A0 → CTLW0 = 0x0081 ; // 00- 1 stop bit, NO Parity, 8-bit data,
Asynchronous Mode, First LSB Then MSB, SMCLK ; 81- Enabled
EUSCI_A0 logic held in reset state
  EUSCI_A0 → BRW = 26 ; // 3000000 / 115200 = 26
  P1 → SEL0 |= 0x0C ; // Configure functionality of P1.2, P1.3 as
UART pins
  P1 → SEL1 &= ~0x0C ;
  EUSCI_A0 → CTLW0 &= ~1 ; // Take UART out of reset mode
}
```

* **RESULT :**

Configured MSP432 on-chip peripherals and established serial communications (transmission and reception). All simulation results were verified successfully.

EXPERIMENT 4 : ANALOG TO DIGITAL CONVERSION

LAB TITLE AND CODE : EMBEDDED COMPUTING LAB (19CLE283)
EXPERIMENT NUMBER : 4
DATE : 10/04/2022 , TUESDAY

* AIM :
Perform analog to digital conversion using MSP43x by configuring its on-chip peripherals.

* ALGORITHM : (ADC PERIPHERAL PROGRAMMING AND DIGITAL OUTPUT DISPLAY ON LED)

① Configure functionality of P2.0, P2.1 and P2.2 as simple GPIO pins.
② Configure direction of P2.0, P2.1 and P2.2 as output for tri-color LEDs.
③ Power ON should be disabled during configuration.
④ Sample-and-hold pulse-mode select, sysclk, 32 sample clocks and software trigger.
⑤ Set resolution as 12-bit (14 clock cycle conversion time)
⑥ A6 input, single-ended and $V_{ref} = AV_{CC}$.
⑦ Configure functionality of P4.7 for ADC pin A6 and its direction as input.
⑧ To convert for memory register 5, repeat single channel (sequence sample as active) and enable ADC after configuration.
⑨ In an infinite while-loop, perform the following operations -
(i) start the conversion.
(ii) Wait for ADC conversion to complete
(iii) Read ADC data register.
(iv) Display bits 10, 9, 8 on tri-color LEDs.

* CODE : (ADC PERIPHERAL PROGRAMMING AND DIGITAL OUTPUT DISPLAY ON LED)

```
#include "msp.h"

//Main Function:
int main(void)
{
```

```
int result;

P2 -> SEL0 &= ~7; // Configure functionality of P2.0, P2.1 and P2.2 as simple GPIO pins
P2 -> SEL1 &= ~7;
P2 -> DIR |= 7; // Configure direction of P2.0, P2.1 and P2.2 as output for tri-color LEDs

ADC14 -> CTL0 = 0x00000010; // Power ON should be disabled during configuration
ADC14 -> CTL0 = 0x04080300; // Sample-and-hold pulse-mode select, sysclk, 32 sample clocks, software trigger
ADC14 -> CTL1 = 0x00000020; // Set resolution as 12-bit (14 clock cycle conversion time)
ADC14 -> MCTL[5] = 6; // A6 input, single-ended, vref = AVcc

// Configure functionality of P4.7 for ADC pin A6 and direction as input :-
P4 -> SEL1 |= 0x80;
P4 -> SEL0 |= 0x80;

// Convert for memory register 5:-
ADC14 -> CTL1 |= 0x00050000; // Repeat single channel, sequence sample is active
ADC14 -> CTL0 |= 2; // Enable ADC after configuration

// Infinite Loop (An embedded program does not stop):
while (1)
{
  ADC14 -> CTL0 |= 1; // start the conversion now
  while (!ADC14 -> IFGR0); // Wait for ADC conversion to complete
  result = ADC14 -> MEM[5]; // Read ADC data register
  P2 -> OUT = result >> 8; // Display bits 10:8 on tri-color LEDs
}
```

CIRCUIT DIAGRAM :

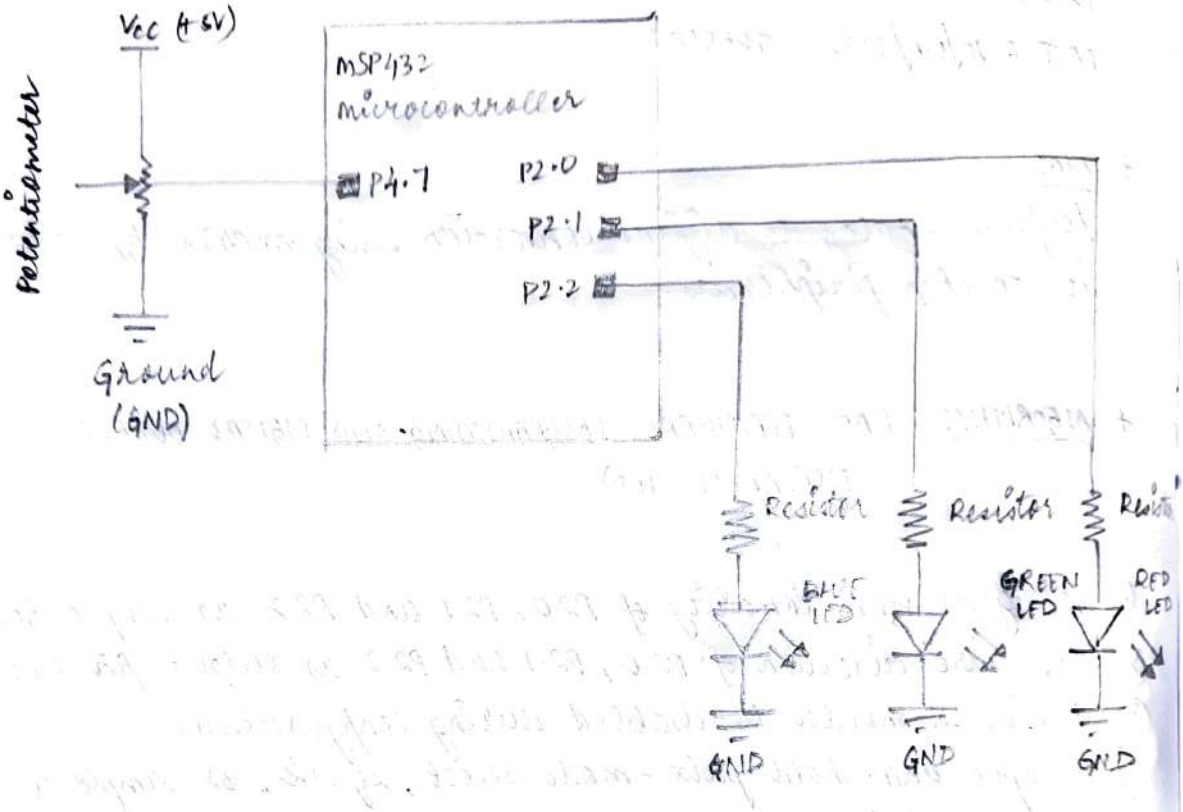


Figure - ADC Peripheral Programming and Digital Output Display on LED

* ALGORITHM : (TEMPERATURE CONTROLLER IMPLEMENTATION USING ADC)

① Configure functionality of P2.1 as simple GPIO pins.
② Configure direction of P2.1 as output for GREEN-LED
③ Power ON should be disabled during configuration.
④ sample-and-hold pulse-mode select, sysclk, 32 sample clocks and software trigger.
⑤ Set resolution as 8-bit (9 clock cycle conversion time)
⑥ Set A6 input, single-ended, $V_{REF} = AV_{CC}$
⑦ Configure functionality of P4.7 as ADC pin and it's direction as input.
⑧ To convert for memory register 5, repeat single channel (sequence sample is active) and enable ADC after conversion.
⑨ In an infinite while-loop, perform the following operations -
(i) start the conversion now
(ii) Wait for ADC conversion to complete
(iii) Read ADC data register
(iv) Display bit for GREEN-LED
(v) If temperature of the room goes above 28°, turn ON the GREEN-LED; else OFF.

* CODE: (TEMPERATURE CONTROLLER IMPLEMENTATION USING ADC)

```
#include "msp.h"

// main Function :
int main (void)
{
  int result;

  P2->SEL0 &= ~2; // Configure functionality of P2.1 as simple GPIO pins
  P2->SEL1 &= ~2;
  P2->DIR |= 2; // Configure direction of P2.1 as output for GREEN-LED
```

```
ADC14 -> CTL0 = 0x00000010; // Power ON should be disabled during configuration
ADC14 -> CTL0 = 0x04080300; // sample-and-hold pulse-mode select, sysclk, 32 sample clocks, software trigger
ADC14 -> CTL1 = 0x00000000; // set resolution as 8-bit (9 clock cycle conversion time)
ADC14 -> MCTL[5] = 6; // A6 input, single-ended, Vref = AVcc

// Configure functionality of P4.7 as ADC pin and it's direction as input :-
    P4 -> SEL1 |= 0x80;
    P4 -> SEL0 |= 0x80;

// Convert for memory register 5 :-
ADC14 -> CTL1 |= 0x00050000; // Repeat single channel, sequence sample is active
ADC14 -> CTL0 |= 2; // Enable ADC after conversion

// Infinite Loop (An embedded program does not stop)
while(1)
{
  ADC14 -> CTL0 |= 1; // start the conversion now
  while(!(ADC14 -> IFGR0)); // wait for ADC conversion to complete
  result = ADC14 -> MEM[5]; // Read ADC data register
  P2 -> OUT = result >> 8; // display bit for GREEN-LED

  /*
  Temperature Range - 0 degrees (0x00) to 60 degrees (0x11)
  If temperature goes above 28 degrees, turn ON the LED; else OFF.
  60 degrees = 256 bits (2^8 bit-resolution)
  Therefore, 28 degrees = ??? = (256 * 28) / 60 = 119.4667 ~ 120 bits
  In hexadecimal representation, (120)10 = (78)16
  */
```

* CIRCUIT DIAGRAM :

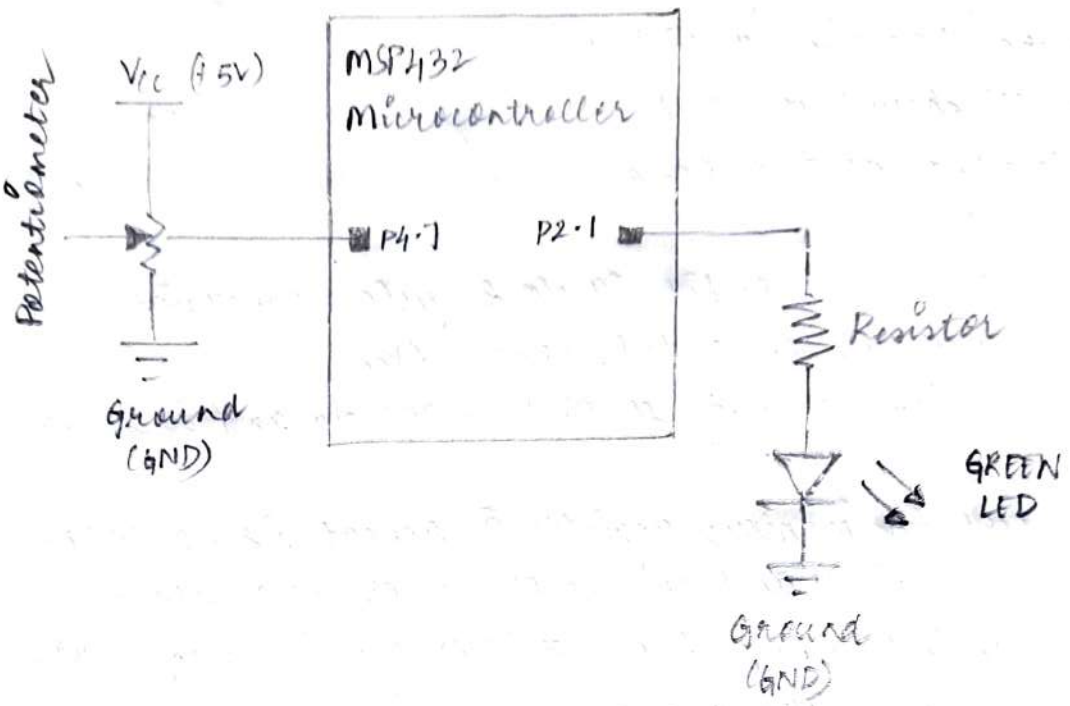


Figure. Temperature controller implementation using ADC

```
        if (result >= 0x18)
        {
          P2->OUT |= 2; // TURN ON P2.1 GREEN-LED
        }
        else
        {
          P2->OUT &= ~2; // TURN OFF P2.1 GREEN-LED
        }
    }
}
```

* **RESULT:**

Implemented analog to digital conversion using MSP432 by configuring its on-chip peripherals and all simulation results were verified successfully

EXPERIMENT 5 – TIMER PROGRAMMING USING MSP432R

LAB TITLE AND CODE : EMBEDDED COMPUTING LAB (MCCE283)
EXPERIMENT NUMBER : 5
DATE : 17/05/2022, TUESDAY

* **AIM :**
Control an LED by developing a timer program using System Tick, Timer 32 and Timer A timers that can be interfaced with an MSP43x microcontroller.

a) **LED CONTROL USING SYSTICK TIMER :**

① Configure functionality of P2.1 as simple GPIO pins.
② Configure direction of P2.1 as output for GREEN-LED.
③ Reload register value for generating 1 Hz or 1 sec delay ; MCLK = 30,00,000 Hz
④ Clear STCurrent Value Register.
⑤ Enable SysTick to begin counting down and disable interrupt generation. For the clock source, system clock (MCLK) is only implemented.
⑥ In an infinite while-loop, perform the following operations-
(i) check whether the SysTick has counted down to zero and if COUNTFLAG is set.
(ii) If yes, trigger the GREEN-LED.

→ **C PROGRAM CODE –**

```
# include "msp.h"

// Main Function:
int main (void)
{
  P2 → SEL0 &= ~2; // configure functionality of P2.1 as simple GPIO pins
  P2 → SEL1 &= ~2;
  P2 → DIR |= 2; // configure direction of P2.1 as output for GREEN-LED
```

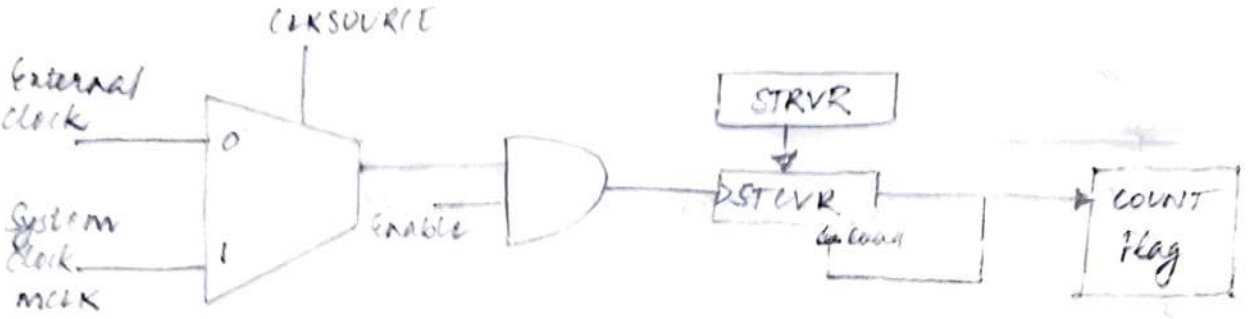


Figure 1 - System Tick Timer Internal Structure

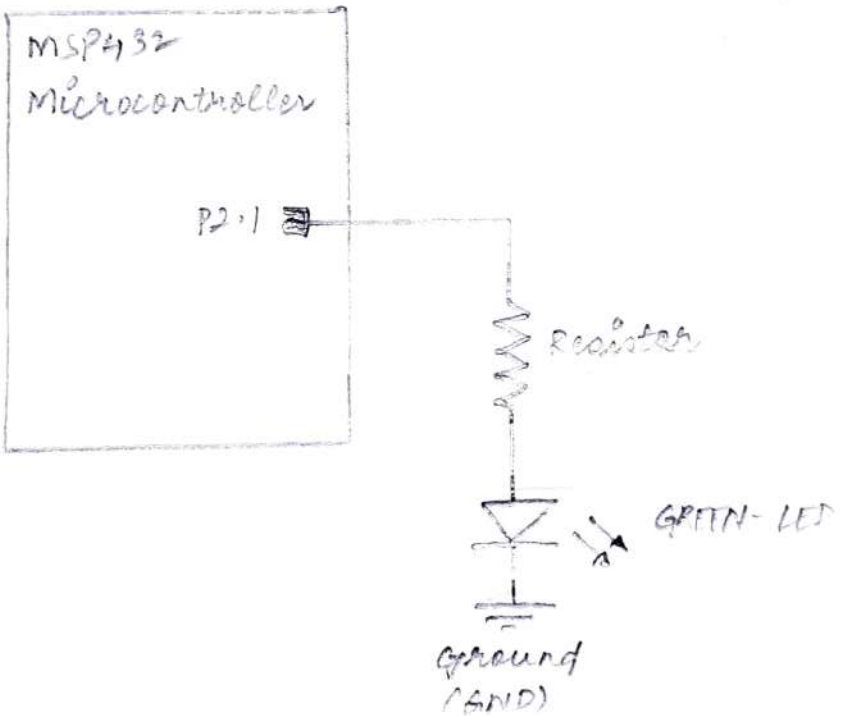


Figure 2 - LED Control using System Tick Timer

```
    SysTick → LOAD = 3000000-1; // Reload register value for
generating 1 Hz or 1 sec delay; MCLK = 30,00,000 Hz
    SysTick → CTRL = 5; // Enables SysTick to begin counting down;
Interrupt generation is disabled; System clock MCLK.
    SysTick → VAL = 0; // Clear ST Current Value Register

    // Infinite Loop (An embedded program does not stop):
    while (1)
    {
      // The SysTick has counted down to zero :-
      if (SysTick → CTRL & 0x10000) // If COUNTFLAG is set
        P2 → OUT ^= 2; // Trigger the GREEN-LED
    }
}
```

5) LED CONTROL USING TIMER 32 TIMER :

① Configure functionality of P2.1 as simple GPIO pins.
② Configure direction of P2.1 as output for GREEN-LED.
③ Reload register value for generating 1 Hz or 1 sec delay; Assume prescale unit to be equal to 1; Set MCLK = 30,00,000 Hz
④ Enable timer to begin counting down and mode bit in periodic mode. The counter generates an interrupt at a constant interval, reloading the value from T32LOADn register.
⑤ Disable timer interrupt enable bit and set prescale bits to 00. clock is divided by 1. Select 32-bit counter operation.
⑥ Select wrapping mode, i.e., the timer continues counting when it reaches to zero.
⑦ In an infinite while-loop, perform the following operations -
(i) Wait till timer is completed.
(ii) Clear the raw interrupt.
(iii) Trigger the GREEN-LED.

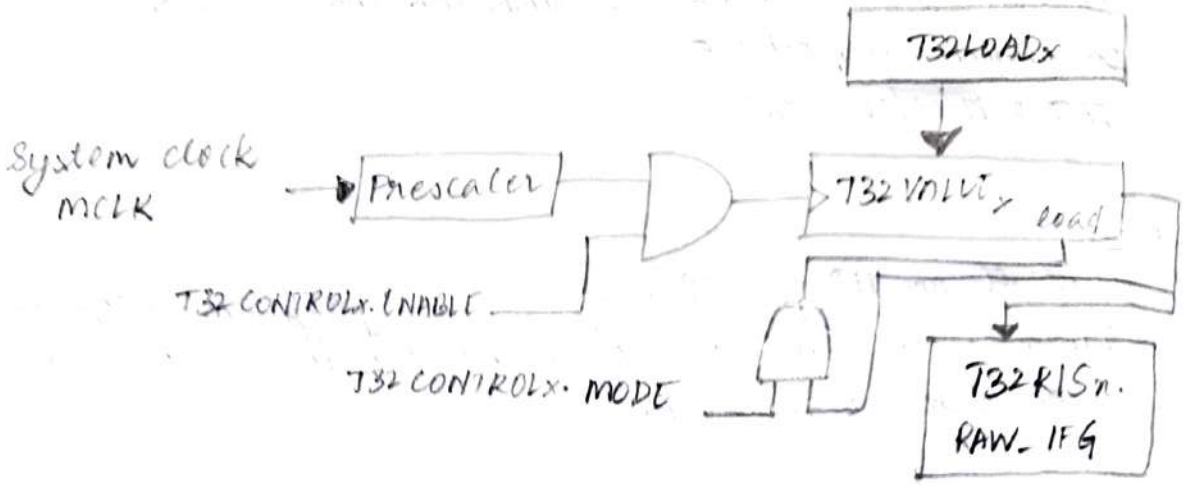


Figure 3- Timer32 Timer Internal structure

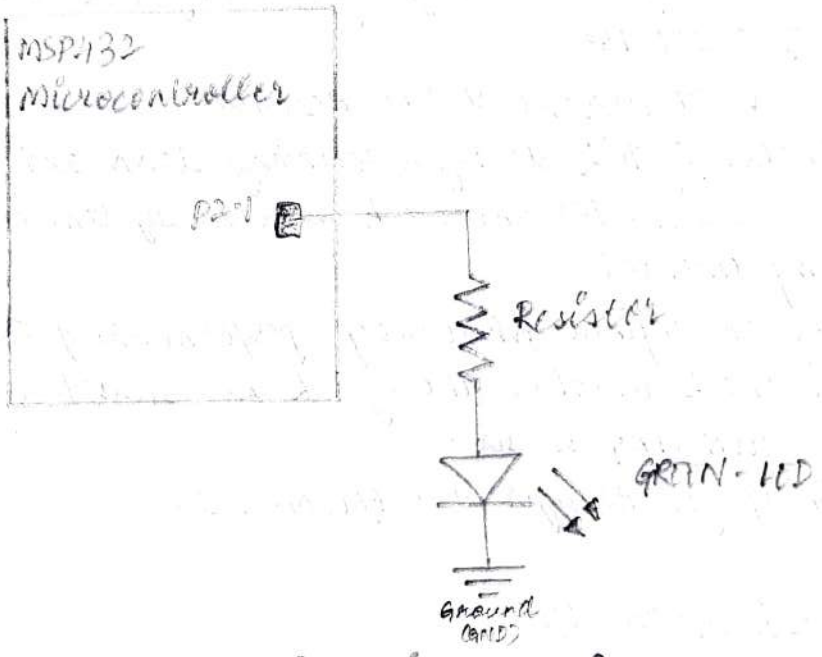


Figure 4- LED Control using Timer32 Timer

→ C PROGRAM CODE -

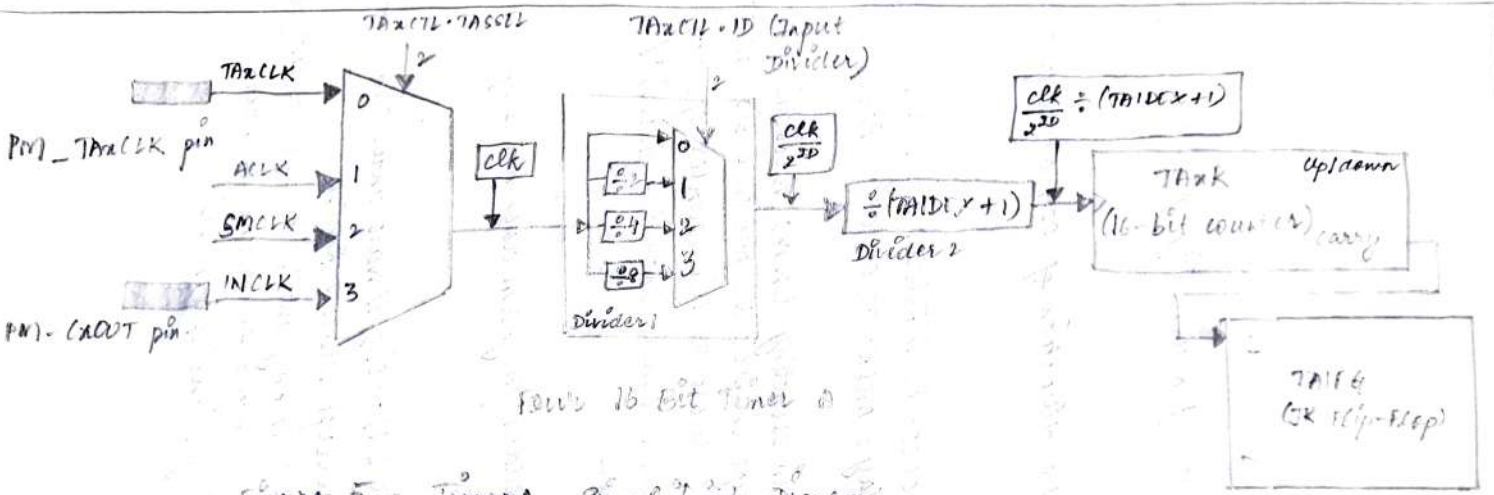
```
#include "msp.h"

// Main Function:
int main(void)
{
    P2->SEL0 &= ~2; // Configure functionality of P2.1 as simple GPIO pins
    P2->SEL1 &= ~2;
    P2->DIR |= 2; // Configure direction of P2.1 as output for GREEN-LED

    TIMER32_1->LOAD = 3000000-1; // Reload register value for generating 1 Hz or 1 sec delay; MCLK = 30,00,000; Prescale unit = 1

    /* Enables timer to begin counting down; Periodic mode; Disable timer interrupt enable; set prescale bits to 00; clock is divided by 1; select 32-bit counter operation; Wrapping mode (Timer 1 Timer Control Register)*/
    TIMER32_1->CONTROL = 0xC2;

    // Infinite Loop (An embedded program does not stop):
    while(1)
    {
      while((TIMER32_1->RIS & 1)==0); // wait until timer is completed (Timer 1 Raw Interrupt status Register)
      TIMER32_1->INTCLR=0; // Any write to the T32INTCLR1 register clears the interrupt output from the counter.
      P2->OUT ^=2; // Trigger the GREEN-LED
    }
}
```

c) LED CONTROL USING TIMER A :

① Configure functionality of P2.1 as simple GPIO pins.
② Configure direction of P2.1 as output for GREEN - LED.
③ Set TimerA interrupt flag to 1 (timer overflowed). TAIFG will be cleared on writing 1 to this bit. Disable interrupt.
④ Set mode control bits to 01, up mode. The timer counts up to TAXCCR0. Set input divider bits to 11, divide by 8. These bits select the divider for the input clock.
⑤ Set clock source select bits to 10, SMCLK (internal clock).
⑥ For generating 1 Hz or 1 sec delay, set CCR[0] = 46875 - 1. Assume $2^{ID} = 8$ and TAIDEX = 7. Set system clock (SMCLK) equal to 30,00,000 Hz
⑦ In an infinite while - loop, perform the following operations -
(i) Wait until the CCIFG is set.
(ii) Clear interrupt flag.
(iii) Trigger the GREEN - LED.

→ C PROGRAM CODE -

```
#include "msp.h"

// Main Function :
int main (void)
{
  P2->SEL0 &= ~2; // Configure functionality of P2.1 as simple GPIO pins.
  P2->SEL1 &= ~2;
  P2->DIR |= 2; // Configure direction of P2.1 as output for GREEN-LED

  /*
    Timer overflowed; Interrupt disabled; Not clear; Up mode;
    Interrupt divider; Select clock source as SMCLK
  */
  TIMER_A1->CTL = 0x02D1;
```



Four 16 Bit Timer_A

Figure 5 - Timer_A Simplified Diagram

```
TIMER_A1 → EX0 = 7; // Divider 2 = TAIDEX + 1 = 7+1 = 8
TIMER_A1 → CCR[0] = 47685-1 ; // For generating 1 Hz or 1 sec delay ; Assume 2^ID = 8, TAIDEX = 7; SMCLK = 30,00,000

// Infinite Loop (An embedded program does not stop):
while (1)
{
  while ((TIMER_A1 → CCTL[0] & 1) == 0); // Wait until the CCIFG flag is set
  TIMER_A1 → CCTL[0] &= ~1; // Clear interrupt flag
  P2 → OUT ^= 2; // Trigger the GREEN-LED
}
}
```

d) LED CONTROL USING TIMER PROGRAMMING :

(TURN LED ON for 30 ms and LED OFF for 40 ms)

① Initialize a variable "temp" for iteration, to turn LED ON and OFF consecutively.

② Configure functionality of P2.1 as simple GPIO pins.

③ Configure direction of P2.1 as output for GREEN-LED.

④ Reload register value for generating 1 Hz or 1 sec delay : MCLK = 30,00,000 Hz

⑤ Clear STCurrent Value Register.

⑥ Enable SysTick to begin counting down and disable interrupt generation. For the clock source, system clock (MCLK) is only implemented.

⑦ Turn ON P2.1 GREEN-LED.

⑧ In an infinite while-loop, perform the following operations.

(i) Check whether the SysTick has counted down to zero and if COUNTFLAG is set.

(ii) If yes, check whether (temp % 2 == 0), i.e. even number -
- If yes, turn OFF P2.1 GREEN-LED.
- else, turn ON P2.1 GREEN-LED

(iii) Increment 'temp' by +1.

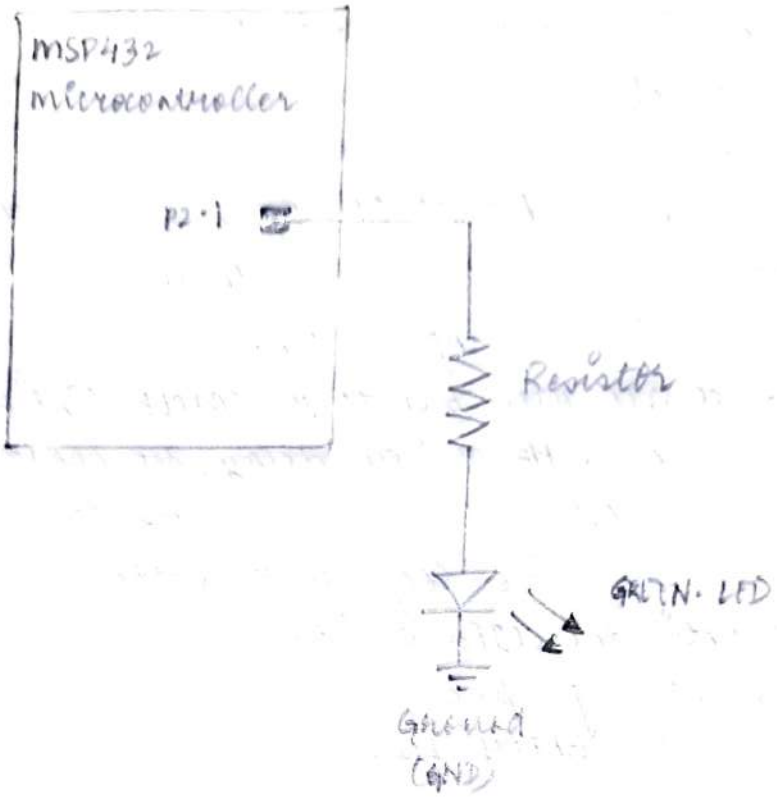


Figure 6- LED Control using Timer A

→ C PROGRAM CODE –

```
#include "msp.h"
int temp = 0; // Variable for iteration to turn LED ON and OFF consecutively.

// Main Function :
int main (void)
{
  P2 → SEL0 &= ~2; // Configure functionality of P2.1 as simple GPIO pins
  P2 → SEL1 &= ~2;
  P2 → DIR |= 2; // Configure direction of P2.1 as output for GREEN-LED

  SysTick → LOAD = 90000-1; // Reload register value for generating 30 milliseconds delay; MCLK = 30,00,000 HZ
  SysTick → VAL = 0; // Clear ST current value Register
  SysTick → CTRL = 5; // Enables SysTick to begin counting down; Interrupt generation is disabled; System clock MCLK
  P2 → OUT |= 2; // Turn ON P2.1 GREEN-LED

  // Infinite Loop (An embedded program does not stop):
  while (1)
  {
    // The SysTick has counted down to zero :-
    if (SysTick → CTRL & 0x10000) // If COUNTFLAG is set

      // Generate 40 milliseconds delay :-
      if (temp % 2 == 0)
      {
        SysTick → LOAD = 120000-1;
        SysTick → VAL = 0;
        SysTick → CTRL = 5;
        P2 → OUT &= ~2; // Turn OFF P2.1 GREEN-LED
```

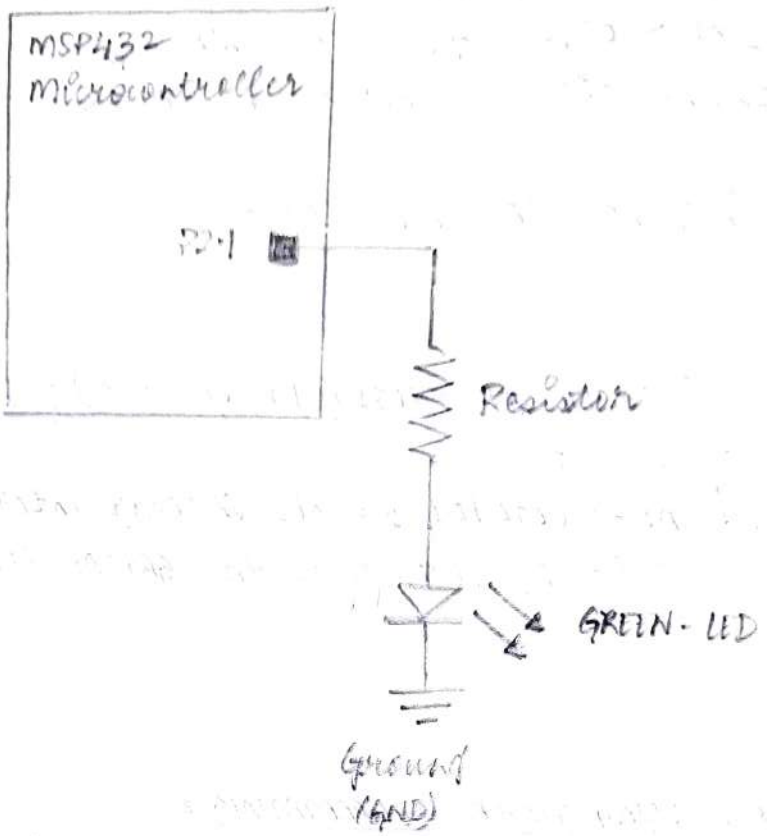


Figure 7 - LED control using Timer Programming
(Turn LED ON for 80 ms and LED OFF for 40 ms)

```
// Generate 30 milliseconds delay :-
else
{
  SysTick -> LOAD = 90000 - 1;
  SysTick -> VAL = 0;
  SysTick -> CTRL = 5;
  P2 -> OUT |= 2; // TURN ON P2.1 GREEN LED
}

  temp++ );
 }
}
```

* **RESULT:**

Thus, an LED was controlled by developing a timer program using System Tick, Timer32 and TimerA timers that can be interfaced with an MSP43x microcontroller. All simulation results were verified successfully.

PAGE : 30

EXPERIMENT 6 - PWM GENERATION USING MSP43x

LAB TITLE AND CODE: EMBEDDED COMPUTING LAB (MCCE283)
EXPERIMENT NUMBER : 6
DATE : 24/05/2022, TUESDAY

★ **AIM:**

Develop a program for edge and centre-aligned pulse width modulation (PWM) generation using an MSP43x microcontroller. Also generate PWM based on ADC input.

a) **EDGE ALIGNED PWM GENERATION:**

① Configure functionality of P2.7 as simple GPIO pin.
② Configure direction of P2.7 as output for timer.
③ Configure timer A0.4 as PWM pin. Set PWM period and duty cycle.
④ Set the following bits of TAxCCTLn Register as follows-
(i) Bit 8 - Capture mode to 0: Compare mode
(ii) Bit 7-5 - Output mode to 111: Reset/set
(iii) Bit 4 - Capture/compare interrupt enable to 0: Interrupt disabled
(iv) Bit 0 - capture/compare interrupt flag to 0: No interrupt pending
⑤ Set the following bits of TAxCTL (TimerA Control Register) as follows-
(i) Bit 0 - TimerA Interrupt Flag to 0: Timer did not overflow
(ii) Bit 1 - TimerA Interrupt Enable to 0: Disabled
(iii) Bit 2 - TimerA clear (Clear TAxR)
(iv) Bit 5-4 - Mode control to 01: Up mode: Timer counts up to TAxCCR0
(v) Bit 6-7 - Input divider to 00: divide by 1.
(vi) Bit 8-9 - TimerA clock source select to 10: SMCLK (internal clock)
⑥ Define an infinite while-loop since an embedded program never stops executing.

PTO

→ C PROGRAM CODE –

```
#include "msp.h"

// Main Function :
int main (void)
{
    // P2.7/PM_TA0.4 :-
    P2 → SEL0 |= 0x80; // Configure functionality of P2.7 as simple GPIO pin
    P2 → SEL1 &= ~0x80;
    P2 → DIR |= 0x80; // Configure direction of P2.7 as output for timer

    // Configure timer A0.4 as PWM pin :-
    TIMER_A0 → CCR[0] = 999-1; // PWM Period
    TIMER_A0 → CCR[4] = 250; // PWM Duty Cycle
    TIMER_A0 → CCTL[4] = 0x00E0; // Compare mode; Reset/set output mode; Interrupt disabled; No interrupt pending

    /* Timer did not overflow; Disable interrupt; clear TimerA; Up mode; Input divider - Divide by 1; clock source select - SMCLK (internal clock) */
    TIMER_A0 → CTL = 0x0214;

    // Infinite Loop (An embedded program does not stop):
    while (1) {}
}
```

B) Centre Aligned Pwm Generation

① Configure functionality of P2.7 as simple GPIO pin.
② Configure direction of P2.7 as output for timer.
③ Configure timer A0.4 as PWM pin. Set PWM period and duty cycle.

a) Edge aligned PWM Generation :

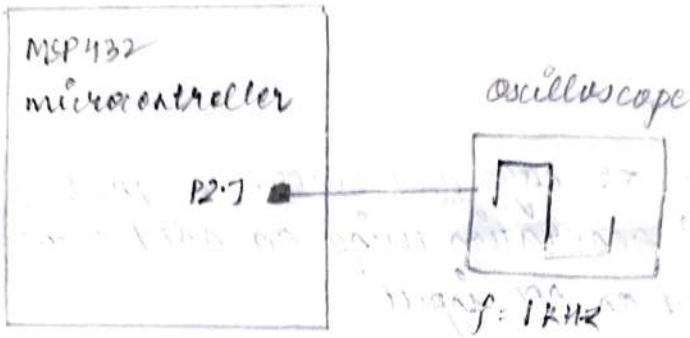


Figure 1- Circuit Diagram

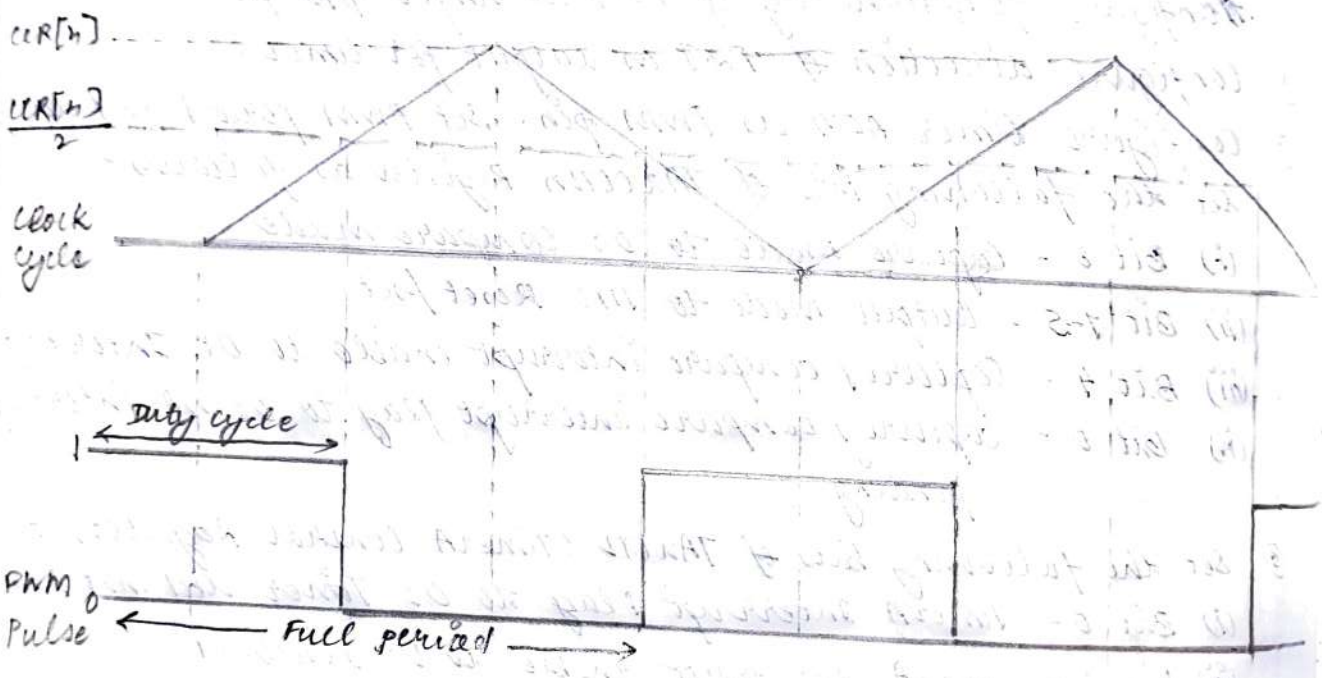


Figure 2- PWM Pulse and Timing Diagram

④ Set the following bits of TAxCCTLn Register as follows -
(i) Bit 8 - Capture mode to 0 : compare mode
(ii) Bit 7-5 - output mode to 010 : Toggle / reset
(iii) Bit 4 - Capture / compare interrupt enable to 0 : Interrupt disabled.
(iv) Bit 0 - Capture / compare interrupt flag to 0 : No interrupt pending

⑤ Set the following bits of TAxCTL (Timer A control Register) as follows -
(i) Bit 0 - TimerA Interrupt Flag to 0 : Timer did not overflow.
(ii) Bit 1 - TimerA Interrupt enable to 0 : Disabled
(iii) Bit 2 - TimerA clear (clear TAxR).
(iv) Bit 5-4 - Mode control to 11 : Up/down mode : Timer counts up to TAxCCR0 then down to 0.
(v) Bit 6-7 - Input divider to 00 : divide by 1.
(vi) Bit 8-9 - TimerA clock source select to 10 : SMCLK (internal clock)

⑥ Define an infinite while-loop since an embedded program never stops executing.

→ C PROGRAM CODE -

```
#include "msp.h"

// Main Function :
int main (void)
{
    // P2.7 / PWM_TA0.4 :-
    P2 → SEL0 |= 0x80; // configure functionality of P2.7 as simple GPIO pin
    P2 → SEL1 &= ~0x80;
    P2 → DIR |= 0x80; // Configure direction of P2.7 as output for timer

    // configure timer A0.4 as PWM pin :-
    TIMER_A0 → CCR[0] = 400 - 1; // PWM period
    TIMER_A0 → CCR[4] = 250; // PWM Duty Cycle
```

b) Center edge PWM generation :

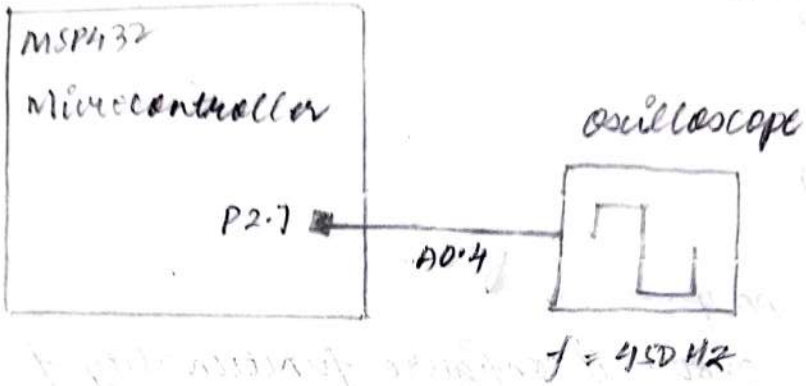


Figure 1- Circuit Diagram

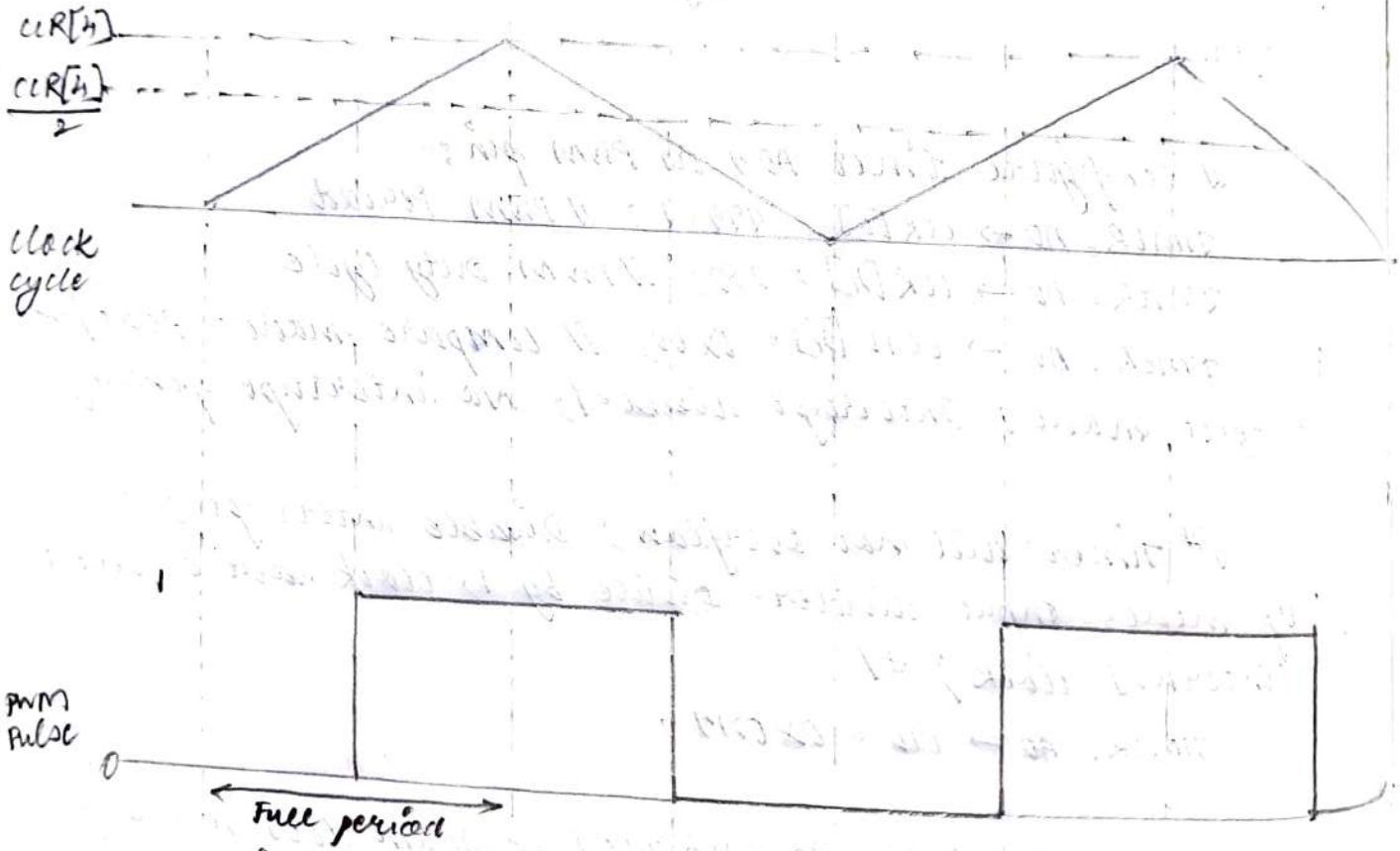


Figure 2 - Pulse and Timing Diagram

TIMER_A0 → CCTL[4] = 0x40; // compare mode; Toggle/reset output mode; Interrupt disabled; No interrupt pending

/* Timer did not overflow; Disable interrupt; clear TimerA; up/down mode; Input divider - Divide by 1; clock source select - SMCLK (internal clock) */

TIMER_A0 → CTL = 0x0234;

// Infinite Loop (An embedded program does not stop):
while(1) {}
}

c) PWM GENERATION BASED ON ADC INPUT:

(1) Configure functionality of P2.5 as simple GPIO pin. Configure direction of P2.5 as output for timer.
(2) Configure functionality of P4.7 as simple GPIO pin. Configure direction of P4.7 as input for ADC.
(3) Power ON should be disabled during configuration. Sample-and-hold pulse-mode select, sysclk, 32 sample clocks and software trigger.
(4) Set resolution as 12-bit (14 clock cycle conversion time).
(5) To convert for memory register 5, repeat single channel (sequence sample is active) and enable ADC after conversion.
(6) In an infinite while-loop, perform the following operations-
(i) Wait for ADC conversion to complete.
(ii) Read ADC data register.
(iii) Set PWM period and duty cycle.
(iv) Set the following bits of TAxCCTLn register as follows -
a. Bit 8 - Capture mode to 0: Compare mode
b. Bit 7-5 - Output mode to 010: Toggle/reset
c. Bit 4 - Capture/compare interrupt enable to 0: Interrupt disabled
d. Bit 0 - Capture/compare interrupt flag to 0: No interrupt pending

P.T.O.

(V) Set the following bits of TAxCTL (TimerA control register) as follows
a. Bit 0 - TimerA Interrupt flag to 0 : Timer did not overflow.
b. Bit 1 - TimerA Interrupt enable to 0 : Disabled
c. Bit 2 - TimerA clear (clear TAxR).
d. Bit 5-4 - Mode control to 11 : up/down mode : Timer counts up to TAxCCR0 then down to 0.
e. Bit 6-7 - Input divider to 00 : divide by 1.
f. Bit 8-9 - TimerA clock source select to 10 : SMCLK (internal clock)

→ C PROGRAM CODE -

```
#include "msp.h"

// Main Function :-
int main (void)
{
  int result;

  P2->SEL0 |= 0x20; // Configure functionality of P2.5 as simple GPIO pin
  P2->SEL1 &= ~0x20;
  P2->DIR |= 0x20; // Configure direction of P2.5 as output for timer

  P4->SEL1 |= 0x80; // Configure functionality of P4.7 as simple GPIO pin
  P4->SEL0 |= 0x80; // Configure direction of P4.7 as input for ADC

  ADC14->CTL0 = 0x00000010; // Power ON should be disabled during configuration
  ADC14->CTL0 |= 0x04080300; // sample-and-hold pulse-mode select, sysclk, 32 sample clocks, software trigger
```

1) PWM GENERATION USING ADC INPUT :

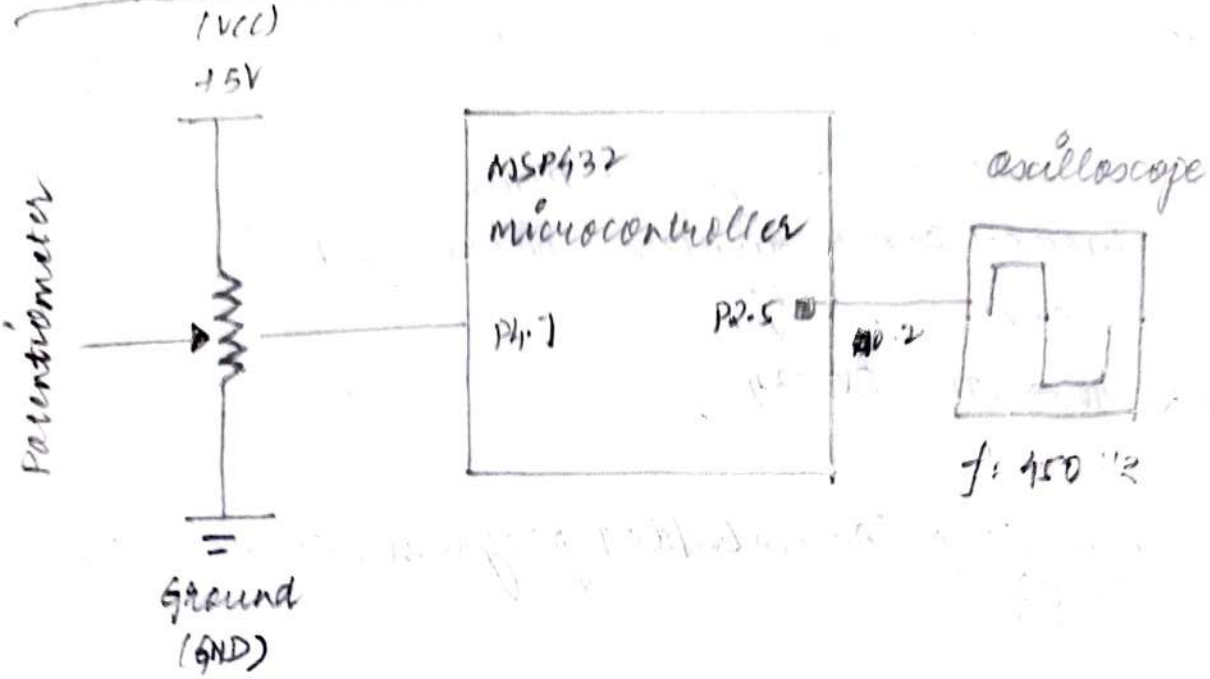


Figure - Circuit Diagram

```
ADC14 → CTL1 = 0x00000020; // Set resolution as 12-bit (14 clock cycle conversion time)

// Convert for memory register 10:-
ADC14 → CTL1 |= 0x000A0000; // Repeat single channel, sequence sample is active
ADC14 → CTL0 |= 2; // Enable ADC after configuration

// Infinite loop (An embedded program does not stop):-
while (1)
{
  while (!ADC14 → IFGR0) // Wait for ADC conversion to complete
  result = ADC14 → MEM[10]; // Read ADC data register

  TIMER_A0 → CCR[0] = 1000-1; // PWM period
  result = result >> 8;
  TIMER_A0 → CCR[3] = result; // PWM duty cycle
  TIMER_A0 → CCTL[3] = 0x40; // Compare mode; Toggle/reset output mode; Interrupt disabled; No interrupt pending

  /* Timer did not overflow, Disable interrupt; clear TimerA, Up/Down mode; Input divider - Divide by 1, clock source select- SMCLK (Internal clock) */

  TIMER_A1 → CTL = 0x0234;
}
}
```

d) SINUSOIDAL SIGNAL PWM GENERATION:

① Initialize a variable "flag" equal to zero.
② Configure functionality of P2.7 as simple GPIO pin.
③ Configure direction of P2.7 as output for timer.

④ Configure timer A0.4 as PWM pin. Set PWM period and duty cycle.

⑤ Set the following bits of TAxCCTLn Register as follows-
(i) Bit 8 - Capture mode to 0: compare mode
(ii) Bit 7-5 - Output mode to 111: Reset/set
(iii) Bit 4 - Capture/compare interrupt enable to 1: Interrupt enabled
(iv) Bit 0 - Capture/compare interrupt flag to 0: No interrupt pending

⑥ Set the following bits of TAxCTL (TimerA control Register) as follows-
(i) Bit 0 - TimerA Interrupt flag to 0: Timer did not overflow
(ii) Bit 1 - TimerA Interrupt Enable to 0: Disabled
(iii) Bit 2 - TimerA Clear (Clear TAxR)
(iv) Bit 5-4 - Mode control to 01: Up mode: Timer counts up to TAxCCR0
(v) Bit 6-7 - Input divider to 00: divide by 1.
(vi) Bit 8-9 - TimerA clock source select to 10: SMCLK (Internal clock)

⑦ In an infinite while-loop, perform the following operations -
(i) If interrupt flag is set, assign TAxCCTL[n] to 0xF0.
(ii) If TAxCCR register is set to full time period, decrement duty cycle by step size.
(iii) Else if TAxCCR register is set to 0, increment duty cycle by step size after.
(iv) Else, increment or decrement according to "flag".

① SINUSOIDAL SIGNAL - PWM GENERATION :

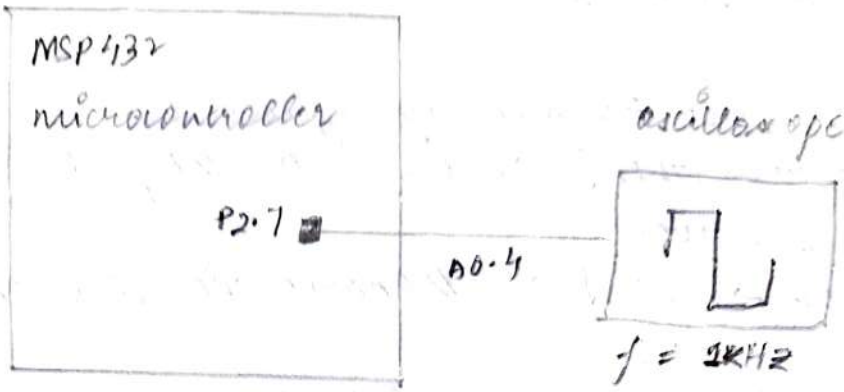


Figure 1 - Circuit Diagram.

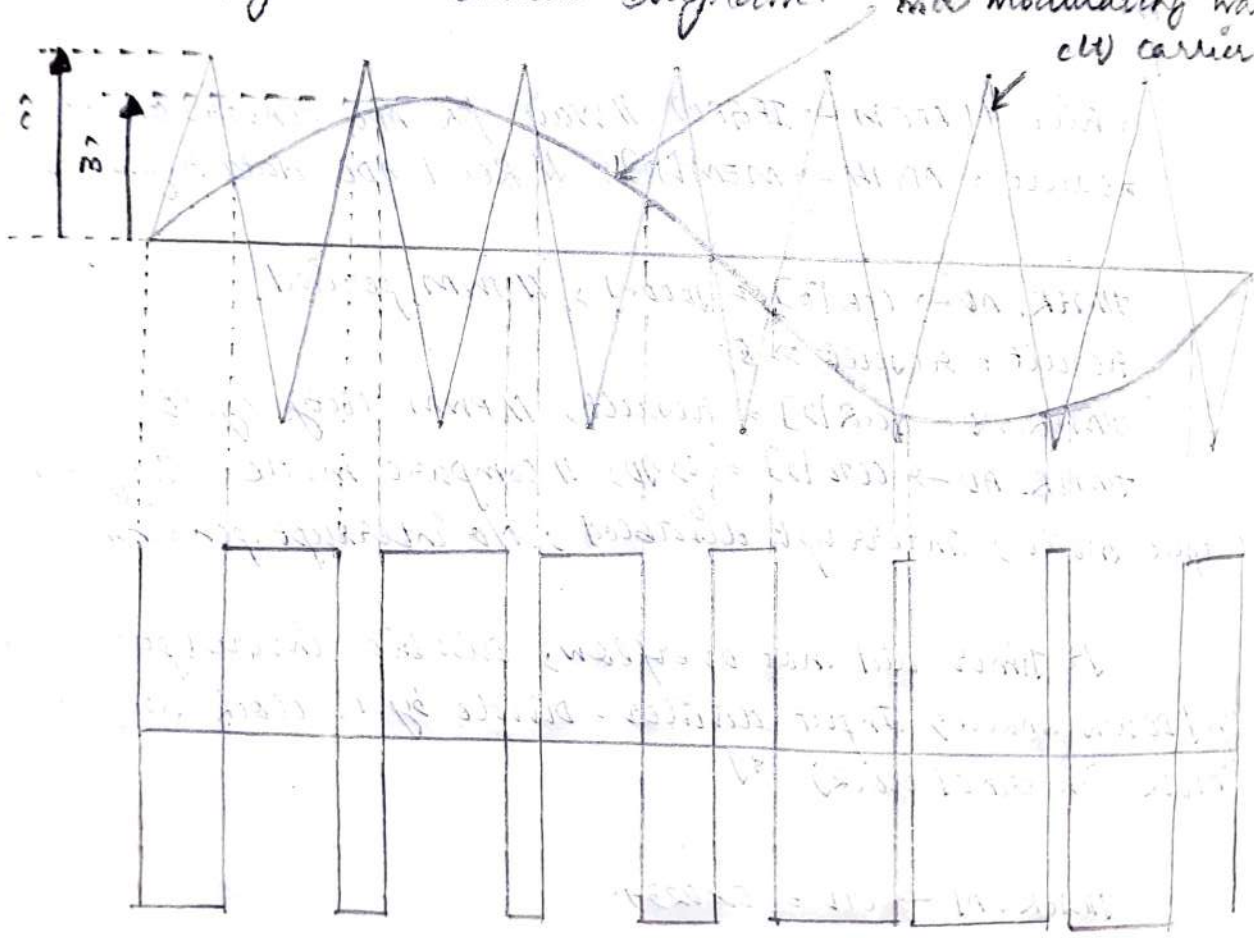


Figure 2 - Pulse and Timing Diagram

→ C PROGRAM CODE -

```
#include "msp.h"
int flag = 0;

// Main Function :
int main (void)
{
  P2 -> SEL1 |= 0x80; // Configure functionality of P2.7 as simple GPIO pin
  P2 -> SEL0 &= ~0x80;
  P2 -> DIR |= 0x80; // Configure direction of P2.7 as output for timer

  // Configure timer A0.4 as PWM pin :-
  TIMER_A0 -> CCR[0] = 999 - 1; // PWM Period
  TIMER_A0 -> CCR[4] = 250; // PWM Duty Cycle
  TIMER_A0 -> CCTL[4] = 0xF0; // Compare mode, Reset/set output mode; Interrupt enabled; No interrupt pending

  /* Timer did not overflow; Disable interrupt; Clear TimerA; Up mode; Input divider - Divide by 1; clock source select - SMCLK (internal clock) */

  // Infinite Loop (An embedded program does not stop):
  while (1)
  {
    while (TIMER_A0 -> CCTL[4] & 0xF1);
    TIMER_A0 -> CCTL[4] = 0xF0;

    if (TIMER_A0 -> CCR[4] == 1000)
    {
      flag = -1;
      TIMER_A0 -> CCR[4] += flag * 200;
    }
```

```
    else if (TIMER_A0->CCR[4] == 0)
    {
      flag = 1;
      TIMER_A0->CCR[4] += flag * 200;
    }

    else
      TIMER_A0->CCR[4] += flag * 200;
  }
}
```

* **RESULT:**

Thus, developed a program for edge and centre-aligned pulse width modulation (PWM) generation using an MSP43x microcontroller. Also, generated PWM based on ADC input. All simulation results were verified successfully.

EXPERIMENT 7 - LCD INTERFACING USING MSP432

LAB CODE AND TITLE : 19CCE283 EMBEDDED COMPUTING LAB

EXPERIMENT NUMBER : 7

DATE - 31/05/2022, TUESDAY

* **AIM:**

Develop a program to interface LCD module using an MSP432 microcontroller. Print characters on the LCD. Also, by supplying varying voltage to an analog pin, read the voltage value and display whether it is low, medium or high.

a) **Hello world display using LCD:**

① Interface the LCD in 8-bit mode.
② Make the required function declarations for LCD module and delay
③ In the main program, call function to initialize LCD.
④ In an infinite loop (while), perform the following operations-
(i) clear display along with DDRAM contents.
(ii) Set DDRAM address or cursor position on display.
(iii) Print "HELLO" on the LCD module.
(iv) Delay.

I. **Function to configure LCD:-**

① Set port 4 for LCD data lines (D0 to D7) and P3.5, 6, 7 as output for RS, RW and EN respectively.
② Function Set: 8-bit, 1 Line, 5x7 Dots
Function Set: 8-bit, 2 Lines, 5x7 Dots
③ Set the Entry mode - Cursor Increment (Right Side), Display Shift OFF
④ Clear display (also clear DDRAM contents)
⑤ Initialize cursor to home position
⑥ Display ON, Cursor OFF, Blink OFF

PTO

II. Display character using Data Write Function:-

① Set RS = 1, R/W = 0 and move data to P4.

② Give LCD Enable Signal-
(i) Pulse E High
(ii) Clear E
(iii) Wait for controller to do the display

III. Display character using command Write Function :-

① Set RS = 0 and R/W = 0

② Move command to P4 - Put command on the data bus.

③ Give LCD Enable Signal-
(i) Pulse E High
(ii) Clear E
(iii) Wait for controller to do the display

→ C PROGRAM CODE :-

```
#include "msp.h"

// LCD Interfacing - 8-bit mode:-
#define RS 0x20 // Mask P3.5
#define RW 0x40 // Mask P3.6
#define EN 0x80 // Mask P3.7

void LCD_init(void); // Function to configure LCD
void LCD_data(unsigned char data); // Display character using
Data Write Function
void LCD_command(unsigned char command); // Display character
using command write Function
void delayMs(int n); // Delay milliseconds when system clock is at
3MHz for Rev C MCU
```

P10 →

```
// Main Function :-
int main (void)
{
  LCD_init ();  // Call function to initialise LCD

  // Infinite Loop (An embedded program does not stop) :-
  while (1)
  {
    LCD_command (1) : // Clear display (also clear DDRAM content)
    delay Ms (500);
    LCD_command (0x80); // Set DDRAM address or cursor position on display

    // Print "HELLO" on the LCD module :-
    LCD_data ('H'); LCD_data ('E'); LCD_data ('L');
    LCD_data ('L'), LCD_data ('O');
    delay Ms (500);
  }
}

// Function to configure LCD :-
void LCD_init (void)
{
    // Set P4 LCD Data (D0 to D7) and P3.5, 6, 7 as output for
RS, RW, EN respectively :-
    P3 → DIR |= RS|RW|EN;
    P4 → DIR = 0xFF;
    delayMs (30);

    LCD_command (0x30); // Function Set : 8-bit, 1 Line, 5x7 Dots
    delayMs (10);
    LCD_command (0x30); // Function Set : 8-bit, 1 Line, 5x7 Dots
    delayMs (1);
    LCD_command (0x30); // Function set : 8-bit, 1 Line, 5x7 Dots
```

```
    LCD_command (0x38); // Function set: 8-bit, 2 Lines, 5x7 Dots
    LCD_command (0x01); // Set the entry Mode - Cursor Increment
(Right side), Display shift OFF
    LCD_command (0x01); // clear display (also clear DDRAM contents)
    LCD_command (0x02); // Initialize Cursor to Home Position
    LCD_command (0x0C); // Display ON, Cursor OFF, Blink OFF
}

// Display Character Using Data Write function:-
void LCD_data (unsigned char data)
{
    P3→OUT |= RS; // RS = 1
    P3→OUT &= ~RW; // R/W = 0
    P4→OUT |= data; // Move data to P4;

    // Give LCD Enable Signal:-
    P3→OUT |= EN; // Pulse E HIGH
    delayMs(5);
    P3→OUT &= ~EN; // Clear E
    delayMs(4); // Wait for microcontroller to do the display
}

// Display character using command write function:-
void LCD_command (unsigned char command)
{
  P3→OUT |= ~(RS|RW); // RS = 0, R/W = 0
  P4→OUT |= command; // Move command to P4 - Put command on
the data bus

    // Give LCD Enable Signal:-
    P3→OUT |= EN // Pulse E HIGH
    delayMs(5);
```

```
    P3→OUT &= ~EN; // Clear E
    delayMs(4); // Wait for microcontroller to do the display
}

// Delay milliseconds when the system clock is at 3 MHz for
Rev C MCU:-
void delayMs(int n)
{
    int i, j;
    for (j=0; j<n; j++)
      for (i=750; i>0; i--); // Delay of 1ms
}
```

b) Voltmeter implementation using ADC and LCD :

① Interface the LCD in 8-bit mode.
② Make the required function declarations for LCD module and delay.
③ In the main program, declare the variables required to print the voltage value on the LCD. Call LCD initialization function.
④ Power ON should be disabled during configuration. sample-and-hold pulse-mode select, sysclk, 32 sample clocks and software trigger.
⑤ Set resolution as 12-bit (14 clock cycle conversion time). A6 input, single-ended and $V_{ref} = AV_{CC}$.
⑥ Configure functionality of P6.1 for ADC pin A6 and its direction as input. To convert for memory register 5, repeat single channel (sequence sample is active) and enable ADC after configuration.
⑦ In an infinite while-loop, perform the following operations -
(i) Start the conversion. Wait for ADC conversion to complete.
(ii) Read ADC data register and calculate the voltage value from the ADC value.
(iii) clear display (also clear DDRAM content) and set DDRAM address or cursor position on display. Delay.
(iv) Based on the result value, print low, high or medium. Delay.

I. Function to configure LCD :-

(1) Set port 4 for LCD data lines (D0 to D7) and P3.5, 6, 7 as output for RS, RW and EN respectively.

(2) Function Set : 8-bit, 1 Line, 5X7 Dots
Function Set : 8-bit, 2 Lines, 5X7 Dots

(3) Set the entry Mode - Cursor Increment (Right Side), Display shift OFF

(4) Clear display (also clear DDRAM contents)

(5) Initialize cursor to home position.

(6) Display ON, Cursor OFF, Blink OFF

II. Display Character Using Data Write Function :-

(1) Set RS=1, R/W=0 and move data to P4.

(2) Give LCD Enable Signal -
(i) Pulse E High
(ii) clear E
(iii) wait for controller to do the display.

III. Display character Using Command Write Function :-

(1) Set RS=0 and R/W=0

(2) Move command to P4 - Put command on the data bus.

(3) Give LCD Enable Signal -
(i) Pula E High
(ii) Clear E
(iii) Wait for controller to do the display.

→ C PROGRAM CODE :

```
#include "MSP.R"
#include <stdio.R>

// LCD Interfacing - 8 bit Mode :-
#define RS 0x20 // Mask P3.5
#define RW 0x40 // mask P3.6
#define EN 0x80 // Mask P3.7
```

```
void LCD_init(void); // Function to configure LCD
void LCD_data(unsigned char data); // Display character using Data Write Function
void LCD_command(unsigned char command); // Display character using Command Write function
void LCD_string(char word[]); // Display string using Data Write Function
void delayMs(int n); // Delay milliseconds when system clock is at 3 MHz for Rev C MCU

// Main Function:
int main(void)
{
    int result; float new_result; char output[4];
    LCD_init(); // call function to initialize LCD

    ADC14->CTL0 = 0x00000010; // Power ON should be disabled during configuration
    ADC14->CTL0 = 0x04080300; // sample-and-hold pulse-mode select, sysclk, 32 sample clocks, software trigger
    ADC14->CTL1 = 0x00000020; // Set resolution as 8-bit (14 clock cycle conversion time)
    ADC14->MCTL[5] = 6; // A6 input, single-ended, Vref = AVcc

    // configure functionality of 5.1 for ADC pin A6 and direction as input :-
    P5->SEL1 |= 0x80;
    P5->SEL0 |= 0x80;

    // convert for memory register 5:-
    ADC14->CTL1 |= 0x00050000; // Repeat single channel, sequence sample is active
    ADC14->CTL0 |= 2; // Enable ADC after configuration
```

```
// Infinite loop (An embedded program does not stop):
while (1)
{
  ADC14 -> CTL0 |= 1;  // start the conversion now
  while (!ADC14 -> IFGR0);  // Wait for ADC conversion to complete
  result = ADC14 -> MEM[5];  // Read ADC data register.

  new_result = (result/2^12) * 5;  // Calculate the voltage value from the ADC value
   /*
     sprintf stands for "string print".
     Instead of printing on console, it stores output on char buffer which are specified in sprintf.
     That is, returns a formatted string.
   */
   sprintf (output, "%f", new_result);

   LCD_command (1);  // Clear display (also clear DDRAM content)
   delayMs (500);
   LCD_command (0x80);  // Set DDRAM address or cursor position on display (First Line)
   delayMs (500);
   LCD_string (output);
   delayMs (500);
   LCD_command (0xC0);  // Set DDRAM address or cursor position on display (second line)
   delayMs (500);

   if (result > 0 && result <= ((2^12) * (1/3)))
   {
     LCD_data ('L'); LCD_data ('O'); LCD_data ('W');
     delayMs (500);
   }
```

```
    else if (result > ((2^12) * (1/3)) && result <= ((2^12) * (2/3)))
    {
      LCD_data('M'); LCD_data('E'); LCD_data('D');
      LCD_data('I'); LCD_data('U'); LCD_data('M');
      delayMs(500);
    }

    else
    {
      LCD_data('H'); LCD_data('I'); LCD_data('G'); LCD_data('H');
      delayMs(500);
    }
  }
}

// Function to configure LCD:-
void LCD_init(void)
{
    // set P4 LCD Data (D0 to D7) and P3.5, 6, 7 as output for
RS, RW, EN respectively :-
    P3->DIR |= RS|RW|EN;
    P4->DIR = 0xFF;
    delayMs(30);

    LCD_command(0x30); // Function set: 8-bit, 1 Line, 5x7 Dots
    delayMs(10);
    LCD_command(0x30); // Function set: 8-bit, 1 Line, 5x7 Dots
    delayMs(1);
    LCD_command(0x30); // Function set: 8-bit, 1 Line, 5x7 Dots
    LCD_command(0x38); // Function set: 8-bit, 2 Lines, 5x7 Dots
    LCD_command(0x06); // set the Entry Mode - Cursor Increment
(Right side), Display shift OFF
```

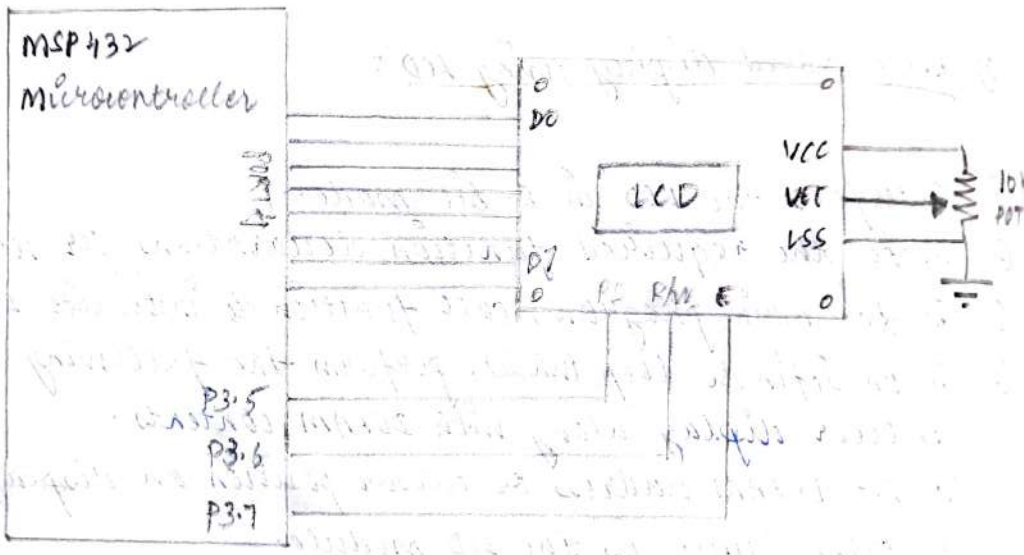
```
    LCD_command (0x01); // Clear display (also clear DDRAM contents)
    LCD_command (0x02); // Initialize Cursor to Home Position
    LCD_command (0x0C); // Display ON, Cursor OFF, Blink OFF
}

// Display character using Data Write Function :-
void LCD_data (unsigned char data)
{
    P3->OUT |= RS;  // RS = 1
    P3->OUT &= ~RW; // R/W = 0
    P4->OUT |= data; // Move data to P4;

    // Give LCD Enable Signal :-
    P3->OUT |= EN; // Pulse E High
    delayMs (5);
    P3->OUT &= ~EN; // Clear E
    delayMs (5); // wait for microcontroller to do the display
}

// Display character using command Write Function :-
void LCD_command (unsigned char command)
{
    P3->OUT |= ~(RS|RW); // RS = 0, R/W = 0
    P4->OUT |= command; // Move command to P4 - Put
command on the data bus

    // Give LCD Enable signal :-
    P3->OUT |= EN; // Pulse E High
    delayMs (5);
    P3->OUT &= ~EN; // Clear E
    delayMs (4); // wait for microcontroller to do the display
}
```

Figure - LCD Pin Diagram

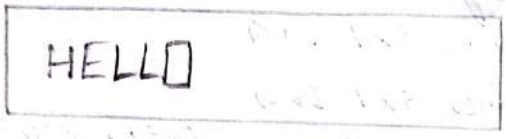
VSS	Power supply (GND)
VCC	Power supply (+5V)
VEE	Contrast adjust control
RS	Register Select { 0 = Instruction input, 1 = Data input
R/W	Read/Write { 0 = Write to LCD module, 1 = Read from LCD module
EN	Enable signal
D0	Data bus line 0 (LSB)
D1	Data bus line 1
D2	Data bus line 2
D3	Data bus line 3
D4	Data bus line 4
D5	Data bus line 5
D6	Data bus line 6
D7	Data bus line 7 (MSB)

* Components Required -

① MSP432 Microcontroller (1)
② LCD module (16x2)
③ Bread Board
④ 10K Potentiometer
⑤ Power Supply / Battery
⑥ Connecting Wires, as required
⑦ Resistor



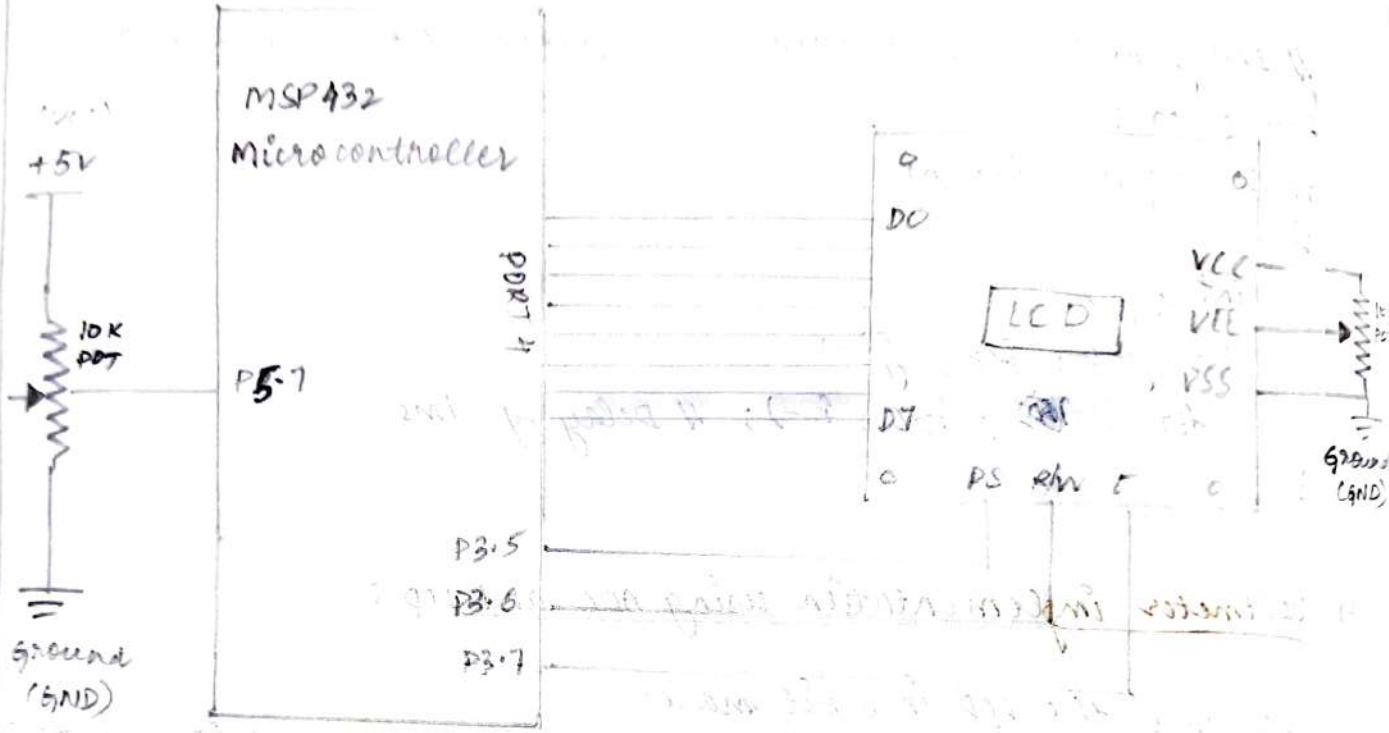
(a) Hello World display using LCD



LCD module display

* Components required -

① MSP432 microcontroller
② LCD Module (16×2)
③ Bread Board
④ Two 10 K Potentiometer
⑤ Power Supply / Battery
⑥ Connecting wires, as required.
⑦ Resistor



(b) Voltmeter implementation using ADC and LCD

LCD module display - All three cases.

```
// Display string using Data Write function:-
void LCD_string (char word[])
{
  int i;
  for (i=0; word[i] != '\0'; i++)
  {
    LCD_data (word[i]);
  }
}

// Delay milliseconds when system clock is at 3 MHZ for Rev C MCU:
void delayMs (int n)
{
  int i, j;
  for (j=0; j< n; j++)
    for (i= 750; i>0; i--); // Delay of 1 ms
}
```

+ **RESULT:**

Thus, developed a program to interface LCD module using an MSP432 microcontroller. All simulation results were verified successfully.

EXPERIMENT 8 - INTERRUPT PROGRAMMING USING MSP43x

LAB CODE AND TITLE - 19CCE283 EMBEDDED COMPUTING LAB
EXPERIMENT NUMBER - 8
DATE - 14/06/2022

* AIMS
Design and develop an embedded computing platform to perform GPIO port and UART serial port interrupt programming using an MSP43x microcontroller.

A) GPIO port Interrupt Programming:
1. Disable global IRQs in the main function.
2. Configure functionality of P1.1 and P1.4 as simple I/O and direction as input for switch.
3. Enable P1.1 and P1.4 pull resistor; Pull up/down is selected by P1→ OUT.
4. Make interrupt trigger on high-to-low transition and clear pending interrupt flags. Enable interrupt from P1.1 and P1.4.
5. Configure functionality of P2.0 – P2.2 as simple I/O and direction as output for tri-color LEDs. Turn all three LEDs OFF.
6. Set priority to 3 in NVIC, enable interrupt in NVIC and global enable IRQs. Toggle the red LED (2.0) continuously.
7. Switch 1 is connected to P1.1 pin and switch 2 is connected to P1.4 pin. Both of them trigger PORT1 interrupt. Toggle green LED (P2.1) three times. clear the interrupt flag before return.
8. Delay n milliseconds (3 MHz CPU clock). DO nothing for 1 ms.

→ C PROGRAM CODE:

```
/* www.microDigitalEd.com
 * P6_1 : Toggle red LED on P2.0 continuously. Upon pressing either
 * SW1 or SW2, the green LED of P2.1 should toggle for three times.
 * main program toggles red LED while waiting for interrupt for
 * SW1 or SW2. When green LED is blinking, the red LED ceases to
 * blink.
```

```
* Tested with Keil 5.20 and MSP432 Device Family Pack V2.2.0
* on XMS432P401R Rev C.
*/

#include "msp.h"

void delayMs(int n);

int main(void) {
    __disable_irq();  /* global disable IRQs */

    /* configure P1.1, P1.4 for switch inputs */
    P1->SEL1 &= ~0x12; /* configure P1.1, P1.4 as simple I/O */
    P1->SEL0 &= ~0x12;
    P1->DIR  &= ~0x12; /* P1.1, P1.4 set as input */
    P1->REN  |= 0x12;  /* P1.1, P1.4 pull resistor enabled */
    P1->OUT  |= 0x12;  /* Pull up/down is selected by P1->OUT */
    P1->IES  |= 0x12;  /* make interrupt trigger on high-to-low
                          transition */
    P1->IFG = 0;       /* clear pending interrupt flags */
    P1->IE |= 0x12;    /* enable interrupt from P1.1, P1.4 */

    /* configure P2.2-P2.0 for tri-color LEDs */
    P2->SEL1 &= ~7;  /* configure P2.2-P2.0 as simple I/O */
    P2->SEL0 &= ~7;

    P2->DIR |= 7;  /* P2.2-2.0 set as output */
    P2->OUT &= ~7; /* turn all three LEDs off */

    NVIC_SetPriority(PORT1_IRQn, 3); /* set priority to 3 in NVIC */
    NVIC_EnableIRQ(PORT1_IRQn);      /* enable interrupt in NVIC */
    __enable_irq();                  /* global enable IRQs */
```

```
    /* toggle the red LED (P2.0) continuously */
    while (1) {
      P2->OUT |= 0x01;
      delayMs(500);
      P2->OUT &= ~0x01;
      delayMs(500);
    }
}

/* SW1 is connected to P1.1 pin, SW2 is connected to P1.4
 * Both of them trigger PORT1 interrupt */
void PORT1_IRQHandler(void) {
    int i;
    volatile int readback;
    /* toggle green LED (P2.1) three times */
    for (i=0; i<3; i++) {
      P2->OUT |= 0x02;
      delayMs(500);
      P2->OUT &= ~0x02;
      delayMs(500);
    }
    P1->IFG &= ~0x12; /* clear the interrupt flag before return */
}

/* delay n milliseconds (3 MHz CPU clock) */
void delayMs(int n) {
    int i, j;

    for (j=0; j<n; j++)
      for (i=750; i>0; i--); /* do nothing for 1 ms */
}
```

b) UART Serial Port Interrupt Programming :

I. Main Function -
① Disable global IRQs and call UART0 initialization function.
② Configure P2.2 - P2.0 as simple I/O and set direction as output for tri-color LEDs.
③ Set priority to 4 in NVIC, enable interrupt in NVIC and global enable IRQs.
④ Declare an empty infinitely running while loop.

II. Continuous Transmission of a character Using UART -
① Put in reset mode and disable oversampling.
② Set 8-bits data - No Parity - 1 stop bit for transmission. In asynchronous mode, first LSB and then MSB should be set, followed by SMCLK (00).
③ Enabled EUSCI_A0 logic is held in reset state (81).
④ Configure functionality of P1.2, P1.3 as UART pins.
⑤ Take UART out of reset mode.
⑥ Enable receive interrupt.

III. Reception of a character Using UART -
① Read the received character and set the LEDs.
② Interrupt flag is cleared by reading RXBUF.

→ C PROGRAM CODE :-

```
/* www.microDigitalEd.com
 * P4_3.c UART0 Receive Interrupt Handling
 *
 * This program modifies p4_2.c to use interrupt to handle the UART0
 * receive. It receives any key from terminal emulator (TeraTerm) of
 * the host PC to the UART0 on MSP432 LaunchPad. The UART0 is
 * connected to the XDS110 debug interface on the LaunchPad and it
 * has a virtual connection to the host PC COM port.
```

```
* Launch a terminal emulator (Teraterm) on a PC and hit any key.
* The tri-color LEDs are turned on or off according to the key
* received.
*
* By default the subsystem master clock is 3MHz.
* Setting EUSCI_A0 -> BRW = 26 with oversampling disabled yields
* 115200 Baud.
*
* Tested with Keil 5.20 and MSP432 Device Family Pack V2.2.0
* on XMS432P401R Rev C.
*/
#include "msp.h"

void UART0_init(void);

int main(void) {
  __disable_irq();

  UART0_init();

  /* initialize P2.2-P2.0 for tri-color LEDs */
  P2->SEL1 &= ~7;  /* configure P2.2-P2.0 as simple I/O */
  P2->SEL0 &= ~7;
  P2->DIR |= 7;    /* P2.2-P2.0 set as output */

  NVIC_SetPriority(EUSCIA0_IRQn, 4); /* set priority to 4 in NVIC */
  NVIC_EnableIRQ(EUSCIA0_IRQn);      /* enable interrupt in NVIC */
  __enable_irq();                    /* global enable IRQs */

  while(1) {
  }
}
```

```
void UART0_init(void) {
    EUSCI_A0 -> CTLW0 |= 1;      /* put in reset mode for config */
    EUSCI_A0 -> MCTLW = 0;       /* disable oversampling */
    EUSCI_A0 -> CTLW0 = 0x0081;  /* 1 stop bit, no parity, SMCLK,
                                    8-bit data */
    EUSCI_A0 -> BRW = 26;        /* 3000000/115200 = 26 */
    P1 -> SEL0 |= 0x0C;          /* P1.3, P1.2 for UART */
    P1 -> SEL1 &= ~0x0C;
    EUSCI_A0 -> CTLW0 &= ~1;     /* take UART out of reset mode */
    EUSCI_A0 -> IE |= 1;         /* enable receive interrupt */
}

void EUSCIA0_IRQHandler(void) {
    P2 -> OUT = EUSCI_A0 -> RXBUF;  /* Read the receive char and
                                       set the LEDs */
                                    /* Interrupt flag is cleared by
                                       reading RXBUF */
}
```

4) GPIO - UART SERIAL PORT INTERRUPT PROGRAMMING :

I. Main Function -

① Disable global IRQs and call UART0 initialization function.
② Configure P2.2 - P2.0 as simple I/O and set direction as output for tri-color LEDs.
③ Set priority to 4 in NVIC, enable interrupt in NVIC and global enable IRQs.
④ Declare an empty infinitely running while loop.

II. Continuous Transmission of a character using UART -

① Put in reset mode and disable oversampling.
② Set 8-bits data - NO Parity - 1 stop bit for transmission. In asynchronous mode, first LSB and then MSB should be set, followed by SMCLK (00).

(3) Enabled EUSCI_A0 logic is held in reset state (S1). Configure functionality of P1.2, P1.3 as UART pins.
(4) Take UART out of reset mode. Enable receive interrupt.

10. Reception of a character using UART -
(1) Read the received character and set the LEDs.
(2) Interrupt flag is cleared by reading RXBUF.
(3) Toggle green LED (P2.1) two times, if input = aaa; else,
(4) Toggle red LED (P2.0) three times, if input = bbbbb; else
(5) Toggle the red and green LED continuously.

→ C PROGRAM CODE -

```
#include "msp.h"
#include <string.h>

void UART0_init(void);
void delayMs(int n);
char ch, str[5], case1[3] = "aaa", case2[5] = "bbbbb";
int i=0;

int main(void) {
    __disable_irq();

    UART0_init();

    /* initialize P2.2 - P2.0 for tri-color LEDs */
    P2->SEL1 &= ~7;  /* configure P2.2-P2.0 as simple I/O */
    P2->SEL0 &= ~7;
    P2->DIR |= 7;  /* P2.2-2.0 set as output */

    NVIC_SetPriority(EUSCIA0_IRQn, 4);  /* set priority to 4 in NVIC */
    NVIC_EnableIRQ(EUSCIA0_IRQn);  /* enable interrupt in NVIC */
    __enable_irq();  /* global enable IRQs */
```

```
    while (1) {
    }
}

void UART0_init (void) {
    EUSCI_A0 -> CTLW0 |= 1;      /* put in reset mode for config */
    EUSCI_A0 -> MCTLW = 0;       /* disable oversampling */
    EUSCI_A0 -> CTLW0 = 0x0081;  /* 1 stop bit, no parity, SMCLK, 8-bit data */
    EUSCI_A0 -> BRW = 26;        /* 3000000/115200 = 26 */
    P1 -> SEL0 |= 0x0C;          /* P1.3, P1.2 for UART */
    P1 -> SEL1 &= ~0x0C;
    EUSCI_A0 -> CTLW0 &= ~1;     /* take UART out of reset mode */
    EUSCI_A0 -> IE |= 1;         /* enable receive interrupt
}

void EUSCIA0_IRQHandler (void) {
    while (ch != '\n')
    {
      ch = EUSCI_A0 -> RXBUF;  /* read the receive char and set the LEDs */
                               /* interrupt flag is cleared by reading RXBUF */
      str[i] = ch;
      i++;
    }

    if (strcmp (str, case1) == 0) {
     /* taggle green LED (P2.1) two times */
     for (i=0; i<2; i++)
     {
       P2 -> OUT |= 0x02;
       delayms (500);
       P2 -> OUT &= ~0x02;
       delayms (500);
     }
}
```

```
else if (strcmp (str, cmd2) == 0) {
  /* toggle red LED (P2.0) three times */
  for (i=0; i<3; i++)
  {
    P2->OUT |= 0x01;
    delayMs (500);
    P2->OUT &= ~0x01;
    delayMs (500);
  }
}

else {
  /* toggle the red LED (P2.0) and green LED (P2.1) continuously */
  P2->OUT |= 0x01;
  delayMs (500);
  P2->OUT &= ~0x01;
  delayMs (500);

  P2->OUT |= 0x02;
  delayMs (500);
  P2->OUT &= ~0x02;
  delayMs (500);
}
}

/* delay n milliseconds (3 MHz CPU clock) */
void delayMs (int n) {
  int i, j;

  for (j=0; j<n; j++)
    for (i=750; i>0; i--); /* do nothing for 1 ms */
}
```

- CIRCUIT DIAGRAM :-

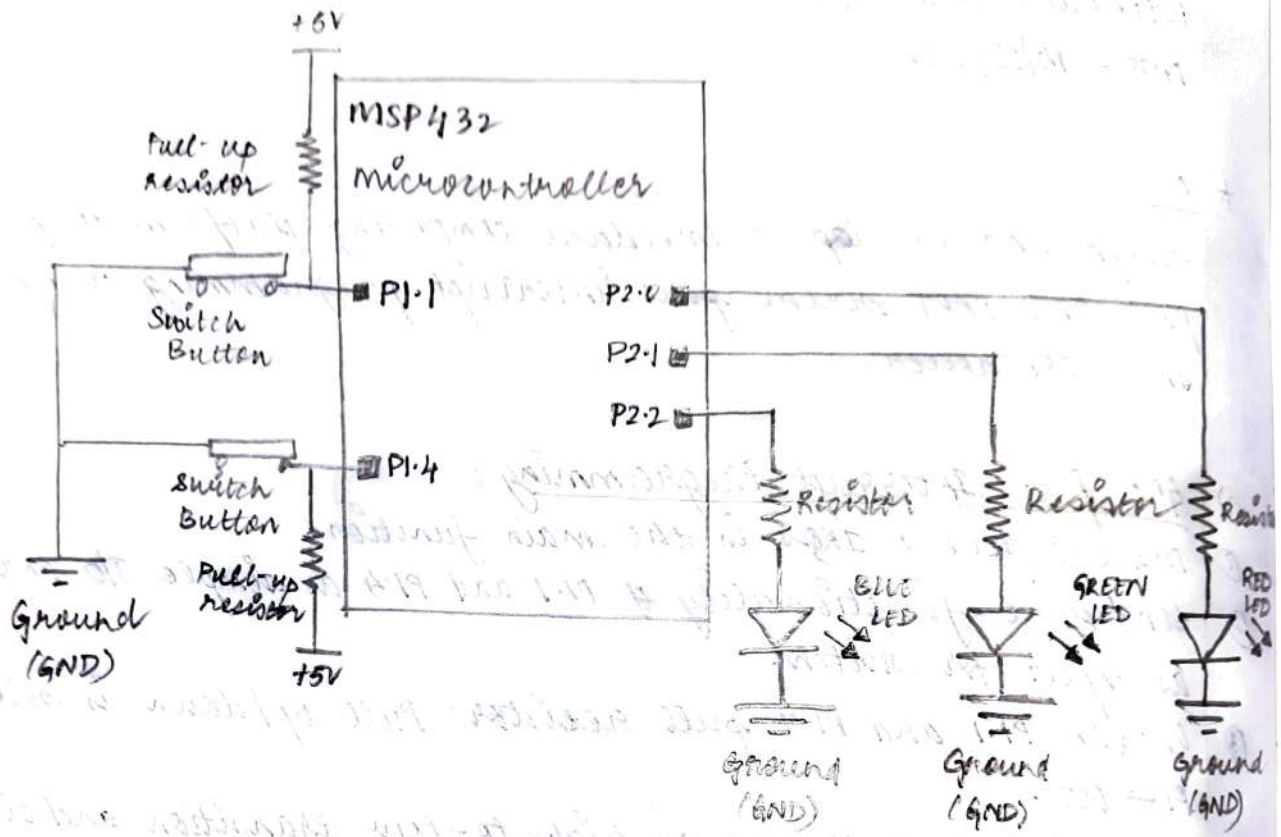


Figure 1 - GPIO Port Interrupt Programming

→ CIRCUIT DIAGRAM :-

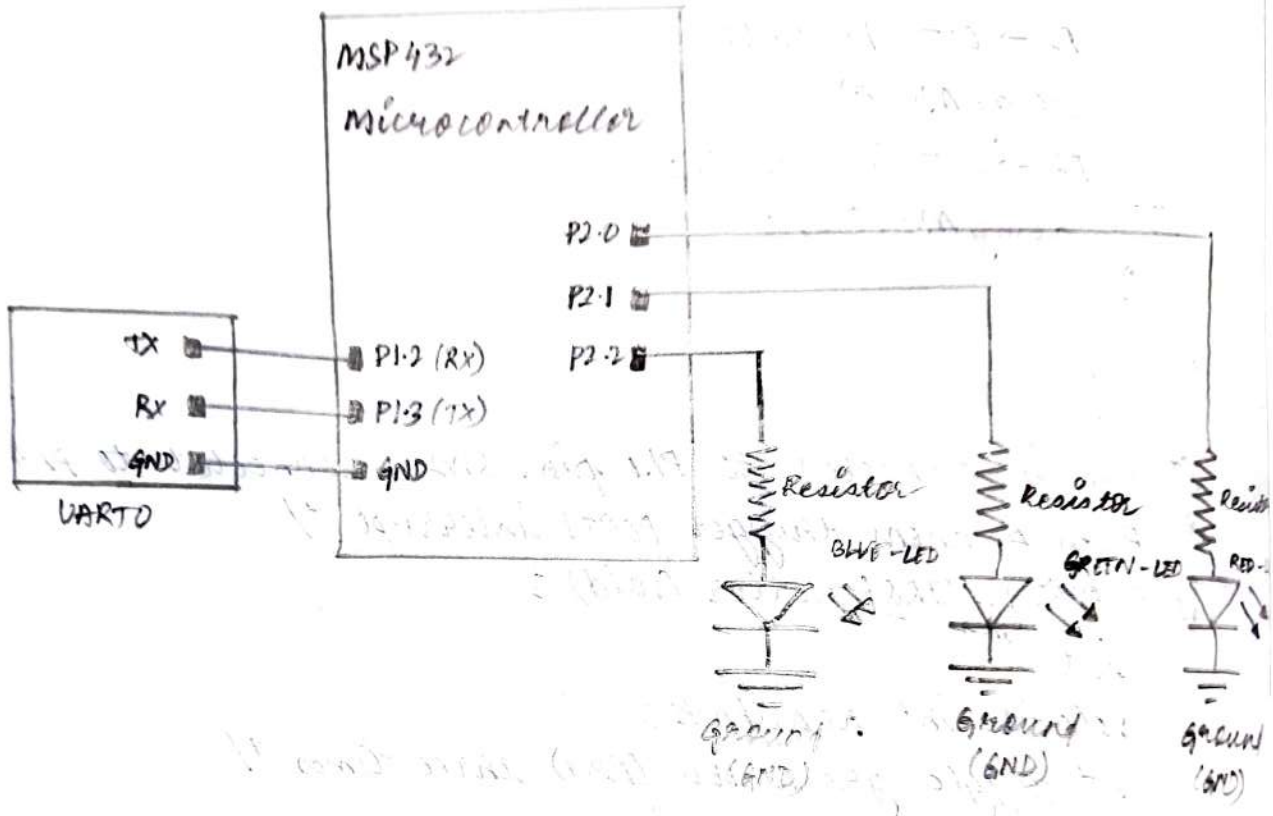


Figure 2- UART Serial Port Interrupt Programming

→ CIRCUIT DIAGRAM :-

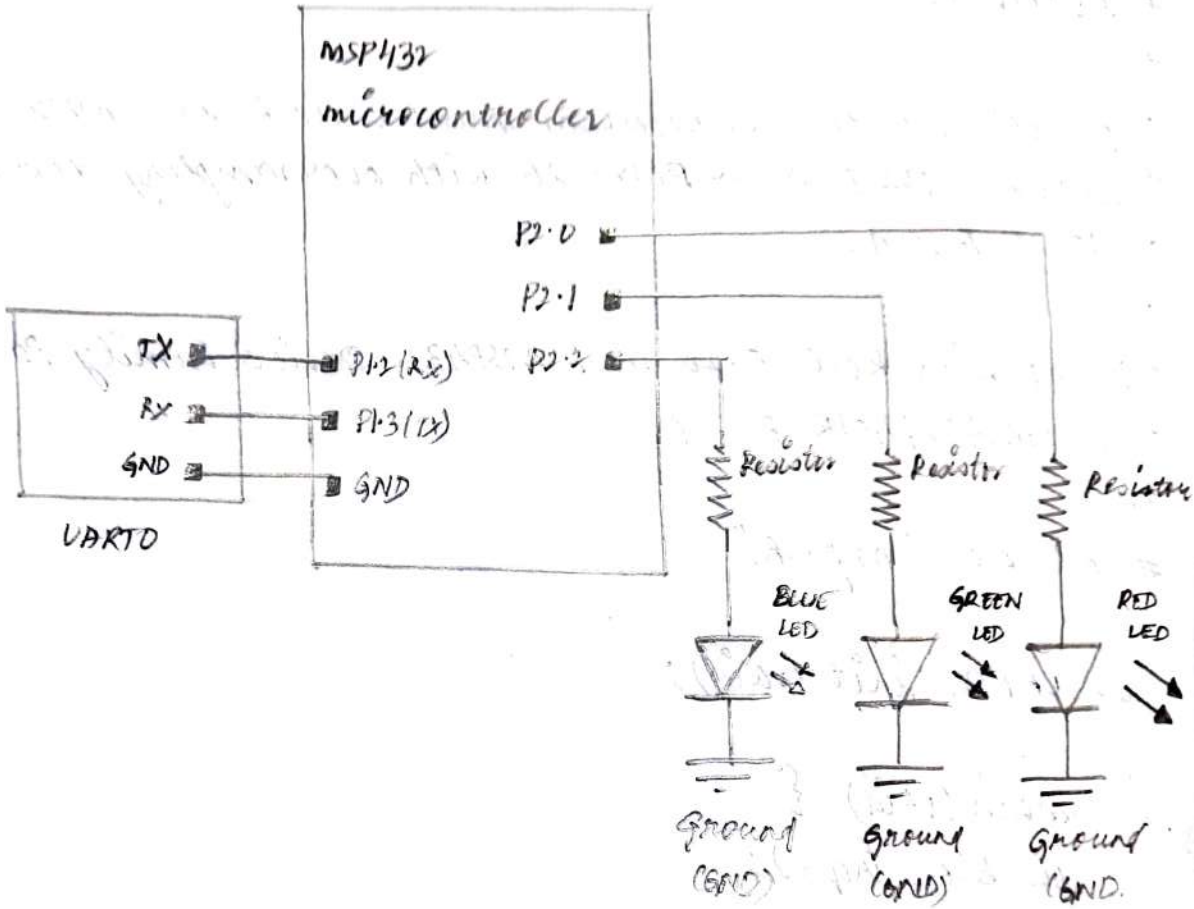


Figure 3 - SPI0- UART Serial Port Interrupt Programming

RESULT:

Thus, designed and developed an embedded computing platform to perform GPIO port and UART serial port interrupt programming using an MSP432 microcontroller. All simulation results were verified successfully.

EXPERIMENT 9 - DEMONSTRATION OF FREERTOS

LAB CODE AND TITLE - 19CCE283 EMBEDDED COMPUTING LAB
EXPERIMENT NUMBER - 9
DATE - 05/07/2022 (TUESDAY)

* **AIM:**
Implement task management in a multi-tasking system using FreeRTOS.

① **EXAMPLE 7. DEFINING AN IDLE TASK HOOK FUNCTION:**

① Declare a variable that will be incremented by the hook function.
② Idle hook functions MUST be called vApplicationIdleHook(), take no parameters, and return void.
③ This hook function does nothing but increment a counter.
④ The string to print out is passed in via the parameter. Cast this to a character pointer.
⑤ As per most tasks, this task is implemented in an infinite loop.
⑥ Print out the number of this task AND the number of times ulIdleCycleCount has been incremented.
⑦ Delay for a period of 250 milliseconds.

→ **C PROGRAM CODE:-**

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"

/* Stellaris library includes. */
// #include "hw_types.h"
// #include "hw_memmap.h"
// #include "sysctl.h"

/* Demo includes. */
#include "basic_io.h"
```

```
/* The task function. */
void vTaskFunction( void *pvParameters );

/* A variable that is incremented by the idle task hook function. */
static unsigned long ulIdleCycleCount = 0UL;

/* Define the strings that will be passed in as the task parameters.
These are defined const and off the stack to ensure they remain
valid when the tasks are executing. */
const char *pcTextForTask1 = "Task 1 is running, ulIdleCycleCount = ";
const char *pcTextForTask2 = "Task 2 is running, ulIdleCycleCount = ";

/*-----------------------------------------------------------*/

int main(void)
{
  /* Create the first task at priority 1... */
  xTaskCreate( vTaskFunction, "Task 1", 240, (void*)pcTextForTask1, 1,
NULL);

  /* ... and the second task at priority 2. The priority is the second to
  last parameter. */
  xTaskCreate( vTaskFunction, "Task 2", 240, (void*)pcTextForTask2, 2,
  NULL);

  /* start the scheduler so our tasks start executing. */
  vTaskStartScheduler();

  for(;;);
}

/*-----------------------------------------------------------*/
```

```
void vTaskFunction (void *pvParameters)
{
  char *pcTaskName;

  /* The string to print out is passed in via the parameter. Cast this
  to a character pointer. */
  pcTaskName = (char *) pvParameters;

  /* As per most tasks, this task is implemented in an infinite loop.*/
  for (;;)
  {
    /* Print out the name of this task AND the number of times
       ulIdleCycleCount has been incremented. */
    vPrintStringAndNumber (pcTaskName, ulIdleCycleCount);

    /* Delay for a period. This time we use a call to vTaskDelay()
    which puts the task into the Blocked state until the delay
    period has expired. The delay period is specified in 'ticks'. */
    vTaskDelay (250/ portTICK_RATE_MS);
  }
}
/*-----------------------------------------------------------*/

/* Idle hook functions MUST be called vApplicationIdleHook(), take no
parameters, and return void. */
void vApplicationIdleHook (void)
{
  /* This hook function does nothing but increment a counter. */
  ulIdleCycleCount ++;
}
/*-----------------------------------------------------------*/
```

```
void vApplicationMallocFailedHook (void)
{
  /* This function will only be called if an API call to create a
  task, queue or semaphore fails because there is too little heap
  RAM remaining - and config USE_MALLOC_FAILED_HOOK is set to 1
  in FreeRTOSConfig.h. */
  for (;;);
}
/*-----------------------------------------------------------*/

void vApplicationStackOverflowHook (xTaskHandle *pxTask,
signed char *pcTaskName)
{
  /* This function will only be called if a task overflow its stack. Note
  that stack overflow checking does slow down the context switch
  implementation and will be only performed if config CHECK_FOR_STACK_
  OVERFLOW is set to either 1 or 2 in FreeRTOSConfig.h. */
  for (;;);
}
/*-----------------------------------------------------------*/

void vApplicationTickHook (void)
{
  /* This example does not use the tick hook to perform any
  processing. The tick hook will only if config USE_TICK_HOOK is set
  to 1 in FreeRTOSConfig.h. */
}
```

(2) EXAMPLE 8 - CHANGING TASK PRIORITIES:

(1) Task 1 is created with the highest priority, so is guaranteed to run first. Task 1 prints out a couple of strings before raising the priority of Task 2 to above its own priority.

(2) Task 2 starts to run (enters the Running state) as soon as it has the highest relative priority. Only one task can be in the Running state at any one time; so, when Task 2 is in the Running state, Task 1 is in the Ready state.

(3) Task 2 prints out a message before setting its own priority back to below that of Task 1.

(4) Task 2 setting its priority back down means Task 1 is once again the highest priority task, so Task 1 re-enters the Running state forcing Task 2 back into the Ready state.

→ C PROGRAM CODE :-

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"

/* stellaris library includes. */
/#include "hw_types.h"
// #include "hw_memmap.h"
// #include "sysctl.h"

/* Demo includes. */
#include "basic_io.h"

/* The two task functions. */
void vTask1 (void *pvParameters);
void vTask2 (void *pvParameters);

/* Used to hold the handle of Task 2. */
xTaskHandle xTask2Handle;

/*-----------*/
```

```
int main (void)
{
/* Create the first task at priority 2. This time the task parameter
is not used and is set to NULL. The task handle is also not used so
likewise is also set to NULL. */
xTaskCreate (vTask1, "Task1", 240, NULL, 2, NULL);
    /* The task is created at priority 2^. */

/* Create the second task at priority 1 - which is lower than the
priority given to Task 1. Again the task parameter is not used so is
set to NULL. But this time we want to obtain a handle to the task
so pass in the address of the xTask2Handle variable. */
xTaskCreate (vTask2, "Task2", 240, NULL, 1, &xTask2Handle);
    /* The task handle is the last parameter ^^^^^^^^ */

/* start the scheduler so our tasks start executing. */
vTaskStartScheduler();

for(;;);
}
/*-----------------------------------------------------*/
void vTask1 (void *pvParameters)
{
unsigned portBASE_TYPE uxPriority;

/* This task will always run before Task2 as it has the higher
priority. Neither Task1 nor Task2 ever block so both will always be
in either the Running or the Ready state.

Query the priority at which this task is running - passing in
NULL means "return our own priority". */
uxPriority = uxTaskPriorityGet (NULL);
```

```
for (;;)
{
    /* Print out the name of this task. */
    vPrintString ("Task1 is running\n");

    /* Setting the Task2 priority above the Task1 priority will
    cause Task2 to immediately start running (as then Task2 will
    have the higher priority of the two created tasks). */
    vPrintString ("About to raise the Task2 priority\n");
    vTaskPrioritySet (xTask2Handle, (uxPriority + 1));

    /* Task1 will only run when it has a priority higher than
    Task2. Therefore, for this task to reach this point Task2 must
    already have executed and set its priority back down to 0. */
}
}

/*-----------------------------------------------------------*/

void vTask2 (void *pvParameters)
{
  unsigned portBASE_TYPE uxPriority;

  /* Task1 will always run before this task as Task1 has the
  higher priority. Neither Task1 nor Task2 ever block so will
  always be in either the Running or the Ready state.

  Query the priority at which this task is running - passing in
  NULL means "return our own priority". */
  uxPriority = uxTaskPriorityGet (NULL);
```

P.T.O.

```
    for (;;)
    {
        /* For this task to reach this point Task1 must have
        already run and set the priority of this task higher than its
        own. */

        /* Print out the name of this task. */
        vPrintString ("Task2 is running\r\n");

        /* set our priority back down to its original value. Passing
        in NULL as the task handle means "change our own priority".
        Setting the priority below that of Task1 will cause Task1 to
        immediately start running again. */
        vPrintString ("About to lower the Task2 priority\r\n");
        vTaskPrioritySet (NULL, (uxPriority - 2));
    }
}
/*-----------------------------------------------------------*/

void vApplicationMallocFailedHook (void)
{
    /* This function will only be called if an API call to create a
    task, queue or semaphore fails because there is too little heap
    RAM remaining - and configUSE_MALLOC_FAILED_HOOK is set to 1 in
    FreeRTOSConfig.h. */
    for (;;);
}
/*-----------------------------------------------------------*/

void vApplicationStackOverflowHook (xTaskHandle *pxTask,
signed char *pcTaskName)
{
```

```
    /* This function will only be called if a task overflows its
    stack. Note that stack overflow checking does slow down the
    context switch implementation and will only be performed if
    configCHECK_FOR_STACK_OVERFLOW is set to either 1 or 2 in
    FreeRTOSConfig.h. */
    for(;;);
}
/*-----------------------------------------------------------*/

void
vApplicationIdleHook(void)
{
    /* This example does not use idle hook to perform any
    processing. The idle hook will only be called if configUSE_IDLE_HOOK
    is set to 1 in FreeRTOSConfig.h. */
}
/*-----------------------------------------------------------*/


void vApplicationTickHook(void)
{
    /* This example does not use the tick hook to perform any processing.
    The tick hook will only be called if configUSE_TICK_HOOK is set to 1 in
    FreeRTOSConfig.h. */
}
```

(3) **EXAMPLE 9. DELETING TASKS :**

(1) Task 1 is created by main() with priority 1. When it runs, it creates Task 2 at priority 2. Task 2 is now the highest priority task, so it starts to execute immediately. ~~The source for main()~~

(2) Task 2 does nothing but delete itself. It could delete itself by passing NULL to vTaskDelete() but instead, for demonstration purposes, it uses its own task handle. ~~The source~~

③ When Task 2 has been deleted, Task1 is again the highest priority task, so continues executing - at which point it calls vTaskDelay() to block for a short period.

④ The Idle task executes while Task 1 is in the blocked state and frees the memory that was allocated to the now deleted Task 2.

⑤ When Task1 leaves the blocked state it again becomes the highest priority Ready state task and so pre-empts the Idle task. When it enters the Running state it creates Task 2 again, and so it goes on.

→ C PROGRAM CODE :-

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"

/* Stellaris library includes. */
// #include "hw_types.h"
// #include "hw_memmap.h"
// #include "sysctl.h"

/* Demo includes. */
#include "basic_io.h"

/* The two task functions. */
void vTask1 (void *pvParameters);
void vTask2 (void *pvParameters);

/* Used to hold the handle of Task 2. */
xTaskHandle xTask2Handle;

/*-----------------------------------------------------*/
```

PTO →

```
int main (void)
{
    /* Note that this project uses the heap_2.c sample memory
    allocator, as this implements vPortFree() as well as pvPortMalloc().
    The free function is required to release heap memory when a task
    is deleted.

    Create the first task to priority 1. This time the task parameter
    is not used and is set to NULL. The task handle is also not used
    so likewise is also set to NULL. */
    xTaskCreate (vTask1, "Task1", 240, NULL, 1, NULL);
        /* The task is created at priority 1 ^. */

    /* Start the scheduler so our tasks start executing. */
    vTaskStartScheduler();

    for(;;);
}
/*-----------------------------------------------------------*/

void vTask1 (void *pvParameters)
{
    const portTickType xDelay100ms = 100 / portTICK_RATE_MS;

    for(;;)
    {
        /* Print out the name of this task. */
        vPrintString("Task1 is running\n");

        /* Create task 2 at a higher priority. Again the task perimeter
        is not used so is set to NULL - BUT this time we want to
        obtain a handle to the task so pass in the address of the
        xTask2Handle variable. */
```

```
    xTaskCreate (vTask2, "Task 2", 240, NULL, 2, &xTask2Handle);
        /* The task handle is the last parameter ^^^^^^^^^^^ */

    /* Task2 has/had the higher priority, so for Task1 to reach here
    Task2 must have already executed and deleted itself. Delay for
    100 milliseconds. */
    vTaskDelay (xDelay100ms);
  }
}
/*-----------------------------------------------------------*/

void vTask2 (void *pvParameters)
{
  /* Task 2 does nothing but delete itself. To do this it would call
  vTaskDelete() using a NULL parameter, but instead and purely for
  demonstration purposes it instead calls vTaskDelete() with its
  own task handle. */
  vPrintString ("Task2 is running and about to delete itself\n");
  vTaskDelete (xTask2Handle);
}
/*-----------------------------------------------------------*/

void vApplicationMallocFailedHook (void)
{
  /* This function will only be called if an API call to create a task,
  queue or semaphore fails because there is too little heap RAM
  remaining - and configUSE_MALLOC_FAILED_HOOK is set to 1 in
  FreeRTOSConfig.h. */
}
/*-----------------------------------------------------------*/
```

```
void vApplicationStackOverflowHook (xTaskHandle *pxTask, signed
char *pcTaskName)
{
   /* This function will only be called if a task overflows its stack.
   Note that stack overflow checking does slow down the context
   switch implementation and will only be performed if
   configCHECK_FOR_STACK_OVERFLOW is set to either 1 or 2 in
   FreeRTOSConfig.h. */
   for(;;);
}
/*-----------------------------------------------------------*/

void vApplicationIdleHook (void)
{
   /* This example does not use the idle hook to perform any
   processing. The idle hook will only be called if configUSE_IDLE_HOOK
   is set to 1 in FreeRTOSConfig.h. */
}
/*-----------------------------------------------------------*/

void vApplicationTickHook (void)
{
   /* This example does not use the tick hook to perform any
   processing. The tick hook will only be called if configUSE_TICK_HOOK
   is set to 1 in FreeRTOSConfig.h. */
}
```

① **EXAMPLE 10. BLOCKING WHEN RECEIVING FROM A QUEUE:**

I. Implementation of the task that writes to the queue -

① Two instances of this task are created, one that writes continuously the value 100 to the queue, and another that writes continuously the value 200 to the same queue.

② The task parameter is used to pass these values into each task instance.

III. Implementation of the task that receives data from the queue -

(1) The receiving task specifies a block time of 100 milliseconds, so will enter the Blocked state to wait for data to become available.

(2) It will leave the Blocked state when either data is available on the queue or 100 milliseconds passes without data becoming available.

(3) In this example, the 100 milliseconds timeout should never expire, as there are two tasks writing continuously to the queue.

IV. Definition of the main() function -

(1) This simply creates the queue and the three tasks before starting the scheduler.

(2) The queue is created to hold a maximum of five long values, even though the priorities of the tasks are set such that the queue will never contain more than one item at a time.

→ C PROGRAM CODE :-

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* Stellaris library includes. */
// #include "hw_types.h"
// #include "hw_memmap.h"
// #include "sysctl.h"

/* Demo includes. */
#include "basic_io.h"

/* The tasks to be created. Two instances are created of the sender
task while only a single instance is created of the receiver task. */
static void vSenderTask (void *pvParameters);
static void vReceiverTask (void *pvParameters);
```

```
/* Declare a variable of type xQueueHandle. This is used to store
   the queue that is accessed by all three tasks. */
xQueueHandle xQueue;

int main (void)
{
  /* The queue is created to hold a maximum of 5 long values. */
  xQueue = xQueueCreate (5, sizeof (long));

  if (xQueue != NULL)
  {
    /* Create two instances of the task that will write to the
       queue. The parameter is used to pass the value that the
       task should write to the queue, so one task will
       continuously write 200 to the queue. Both tasks are
       created at priority 1. */
    xTaskCreate (vSenderTask, "Sender 1", 240, (void *) 100, 1, NULL);
    xTaskCreate (vSenderTask, "Sender 2", 240, (void *) 200, 1, NULL);

    /* Create the task that will read from the queue. The task
       is created with priority 2, so above the priority of the
       sender tasks. */
    xTaskCreate (vReceiverTask, "Receiver", 240, NULL, 2, NULL);

    /* start the scheduler so the created tasks start executing. */
    vTaskStartScheduler();
  }
  else
  {
    /* The queue could not be created. */
  }
```

```
    /* If all is well we will never reach here as the scheduler will
    now be running the tasks. If we do reach here then it is likely
    that there was insufficient heap memory available for a
    resource to be created. */
    for (;;);
}
/*-----------------------------------------------------------*/

static void vSenderTask (void *pvParameters)
{
  long lValueToSend;
  portBASE_TYPE xStatus;

  /* Two instances are created of this task so the value that is
  sent to the queue is passed in via the task parameter rather
  than be hard coded. This way each instance can use a different
  value. Cast the parameter to the required type. */
  lValueToSend = (long) pvParameters;

  /* As per most tasks, this task is implemented within an
  infinite loop. */
  for (;;)
  {
    /* The first parameter is the queue to which data is being
    sent. The queue was created before the scheduler was started,
    so before this task started to execute.

    The second parameter is the address of the data to be sent.

    The third parameter is the Block time - the time the task should be
    kept in the Blocked state to wait for space to become available on
    the queue should the queue already be full. In this case we
    don't specify a block time because there should always be space in the
    queue. */
```

```
xStatus = xQueueSendToBack (xQueue, &lValueToSend, 0);

    if (xStatus != pdPass)
    {
      /* We could not write to the queue because it was full - this
      must be an error as the queue should never contain more than
      one item! */
      vPrintString ("Could not send to the queue.\r\n");
    }

    /* Allow the other sender task to execute. */
    taskYIELD();
  }
}
/*-----------------------------------------------------------*/

static void vReceiverTask (void *pvParameters)
{
/* Declare the variable that will hold the values received from
 the queue. */
long lReceivedValue;
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    /* This task is also defined within an infinite loop */
    for (;;)
    {
      /* As this task unblocks immediately that data is written to
      the queue this call should always find the queue empty. */
      if (uxQueueMessagesWaiting (xQueue) != 0)
      {
        vPrintString ("Queue should have been empty!\r\n");
      }
```

```
/* The first parameter is the queue from which data is to be
received. The queue is created before the scheduler is started,
and therefore before this task runs for the first time.

The second parameter is the buffer into which the received data
will be placed. In this case the buffer is simply the address of a
variable that has the required size to hold the received data.

The last parameter is the block time - the maximum amount of time
that the task should remain in the Blocked state to wait for
data to be available should the queue already be empty. */
xStatus = xQueueReceive ( xQueue, &lReceivedValue, xTicksToWait);

if (xStatus == pdPASS)
{
  /* Data was successfully received from the queue, print out
  the received value. */
  vPrintStringAndNumber ("Received = ", lReceivedValue);
}

else
{
  /* We did not receive anything from the queue even after
  waiting for 100 ms. This must be an error as the sending tasks
  are free running and will be continuously writing to the queue. */
  vPrintString ("Could not receive from the queue.\r\n");
}
}
}
/*-----------------------------------------------------------*/


void vApplicationMallocFailedHook (void)
{
  /* This function will only be called if an API call to create a task,
  queue or semaphore fails because there is too little heap RAM
```

```
    remaining - and configUSE_MALLOC_FAILED_HOOK is set to 1 in
    FreeRTOSConfig.h. */
    for(;;);
}
/*-----------------------------------------------------------*/

void vApplicationStackOverflowHook(xTaskHandle *pxTask, signed
char *pcTaskName)
{
    /* This function will only be called if a task overflows its stack.
    Note that stack overflow checking does slow down the context
    switch implementation and will only be performed if
    configCHECK_FOR_STACK_OVERFLOW is set to either 1 or 2 in
    FreeRTOSConfig.h. */
    for(;;);
}
/*-----------------------------------------------------------*/

void vApplicationIdleHook(void)
{
    /* This example does not use the idle hook to perform any
    processing. The idle hook will only be called if configUSE_IDLE_HOOK
    is set to 1 in FreeRTOSConfig.h. */
}
/*-----------------------------------------------------------*/

void vApplicationTickHook(void)
{
    /* This example does not use the tick hook to perform any
    processing. The tick hook will only be called if configUSE_TICK_HOOK
    is set to 1 in FreeRTOSConfig.h. */
}
```

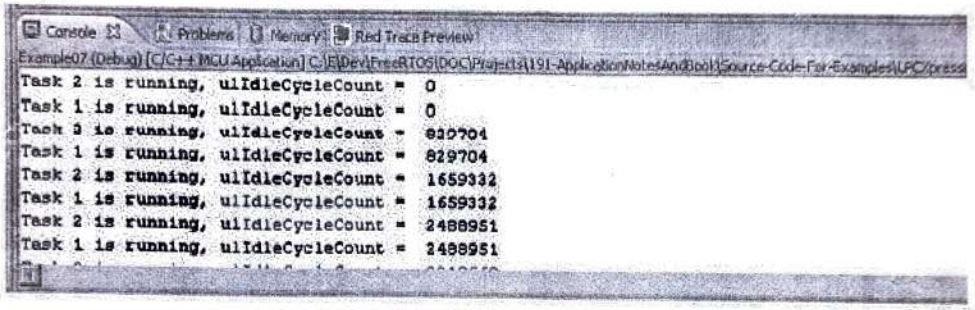


Figure 1 - The output produced when example 7 is executed
It shows that the idle task hook function is called approximately 83000-times between each iteration of the application tasks.

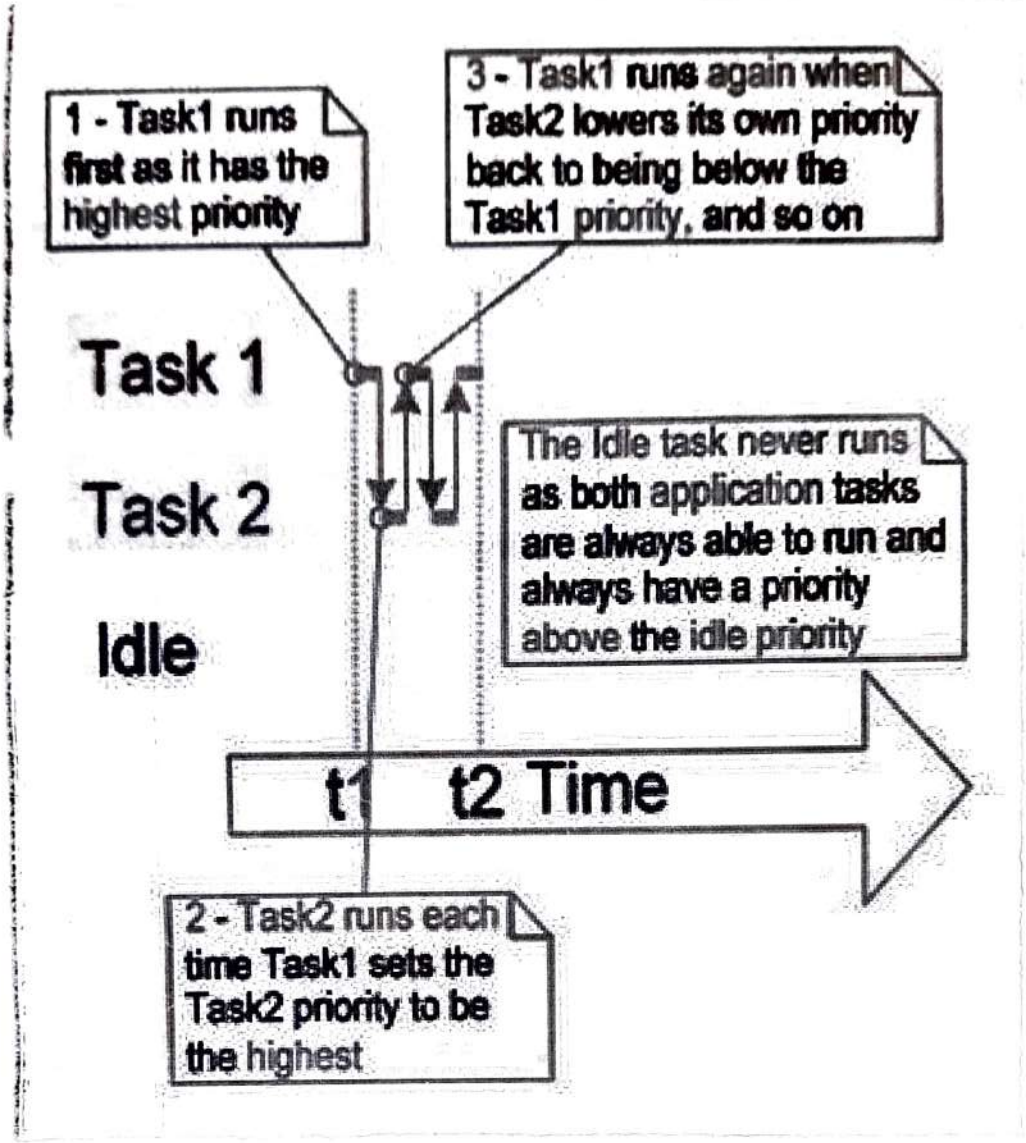


Figure 2 - The sequence of task execution when running Example 2.

It demonstrates the sequence in which the Example 2 tasks execute, with the associated output shown in Figure 3

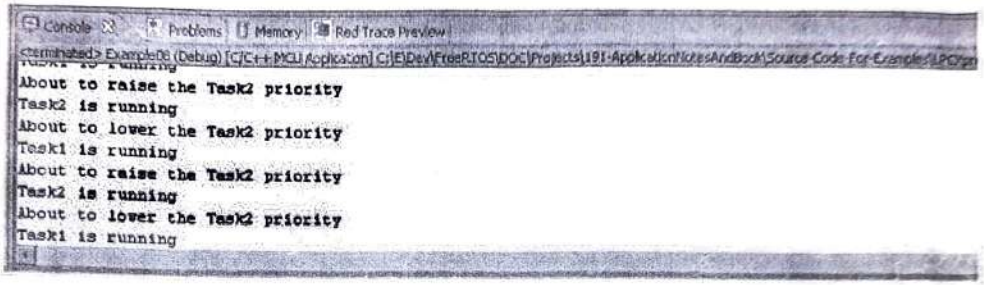


Figure 3- The output produced when Example 6 is executed

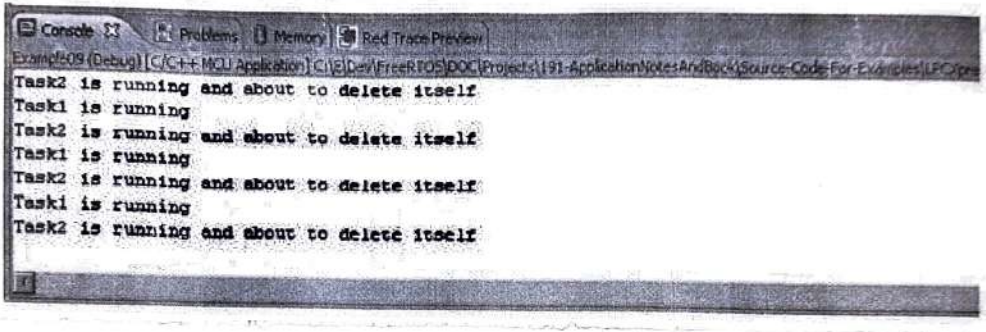


Figure 4- The output produced when Example 9 is executed

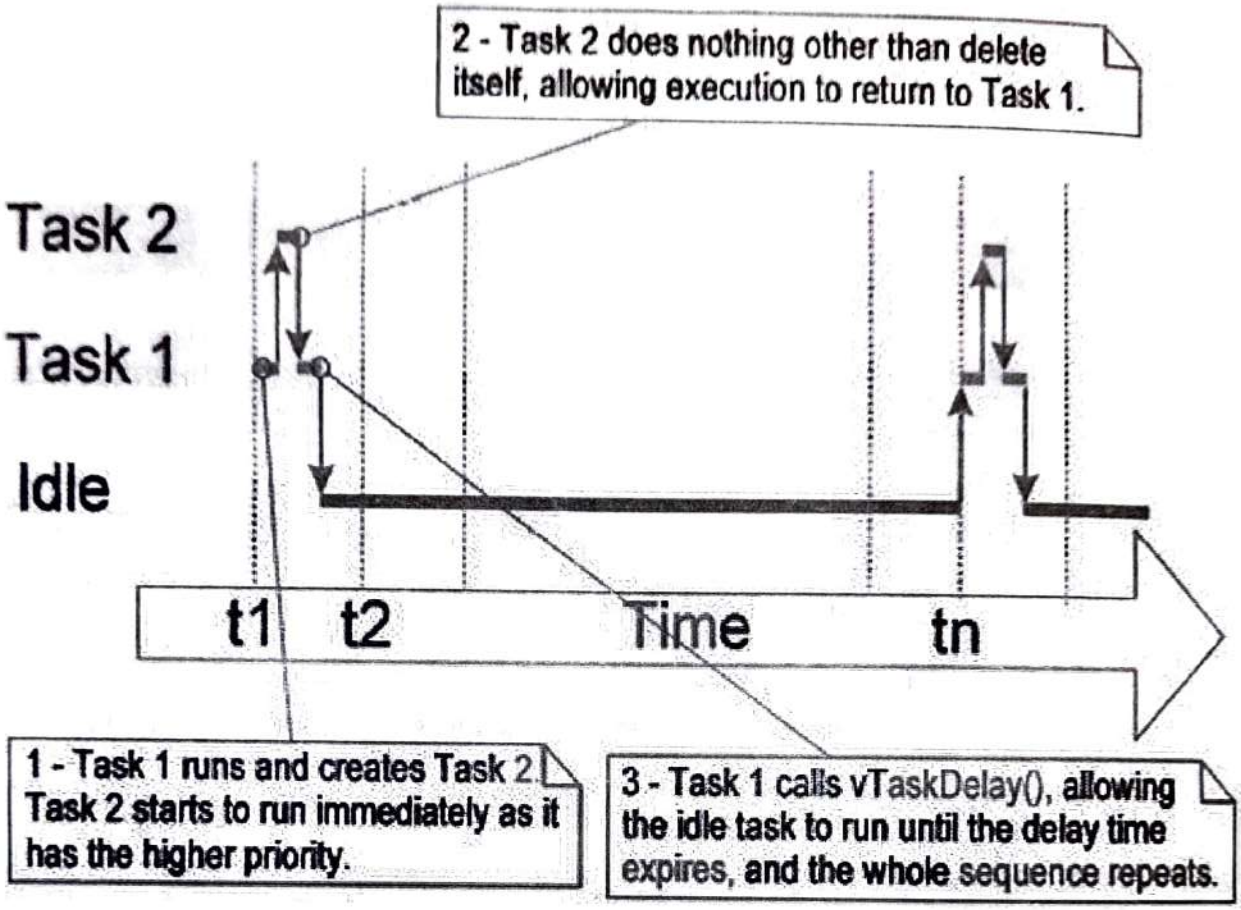


Figure 5- The execution sequence for Example 2

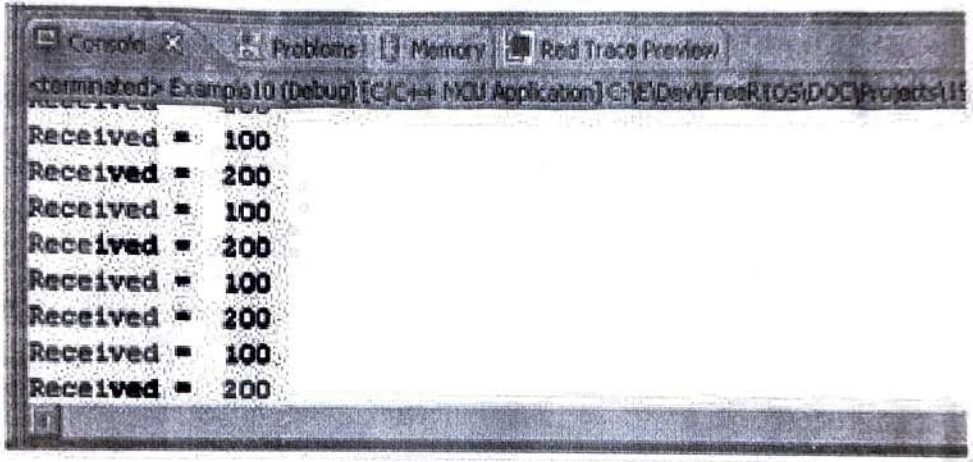


Figure 5- The output produced when Example 10 is executed.

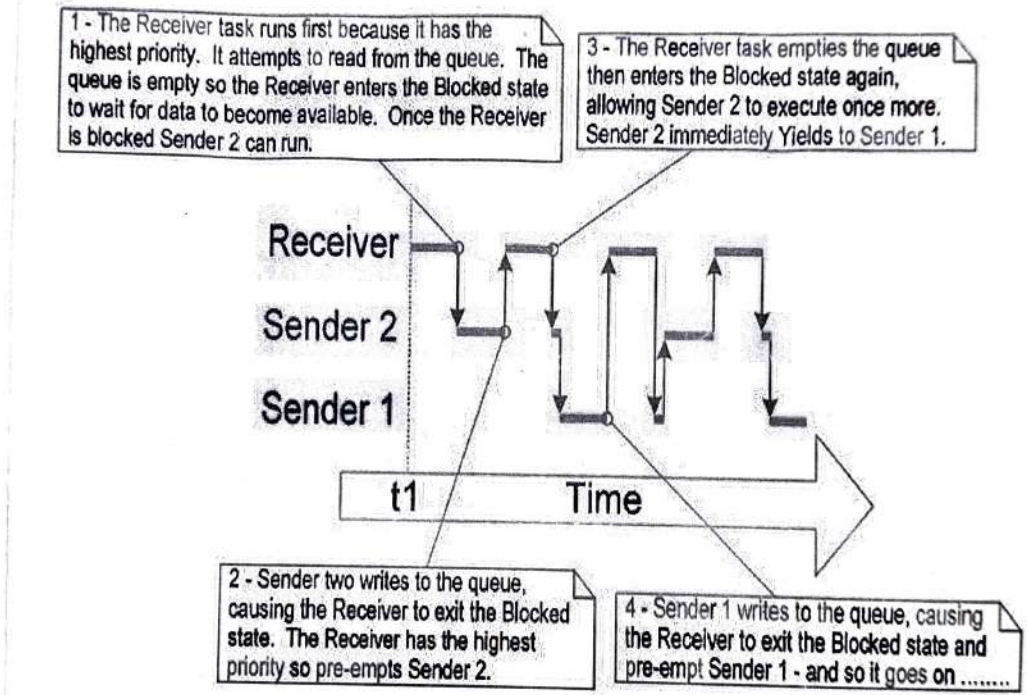


Figure 7 - The Execution sequence for Example 10

* **RESULT:**

Thus, implemented task management in a multi-tasking system using FreeRTOS. All the simulation results were verified successfully.