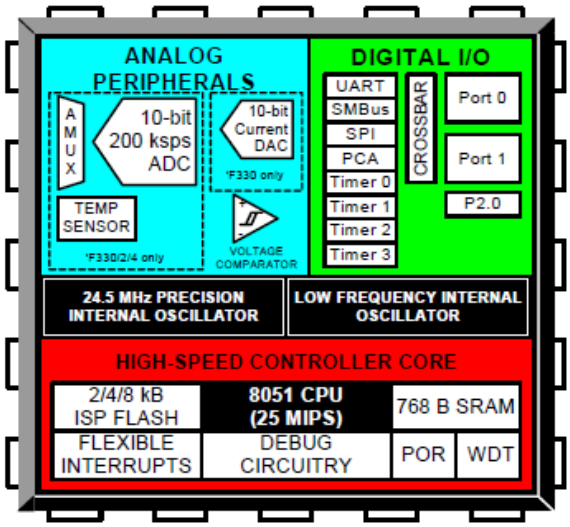
# Mastering the SiLabs C8051 Microcontroller

In a world where upgrades and advancements are constant, it is easy to overlook older technology in favour of newer, more advanced options. However, the case of 8051 microcontrollers defies this trend. Despite being considered relics of the past, there is still a significant demand for these microcontrollers. Manufacturers have revitalized the proven 8051 architecture by incorporating modern features such as ADCs and communication modules, transforming them into powerful, reliable and versatile devices.




Silicon Laboratories (SiLabs) is an American semiconductor-manufacturing company, similar to Microchip and STMicroelectronics. They are renowned for producing a wide range of semiconductor components, including both 8 and 32-bit microcontrollers. Notably, SiLabs is highly regarded for its RF chips and USB-Serial converters such as CP2102.

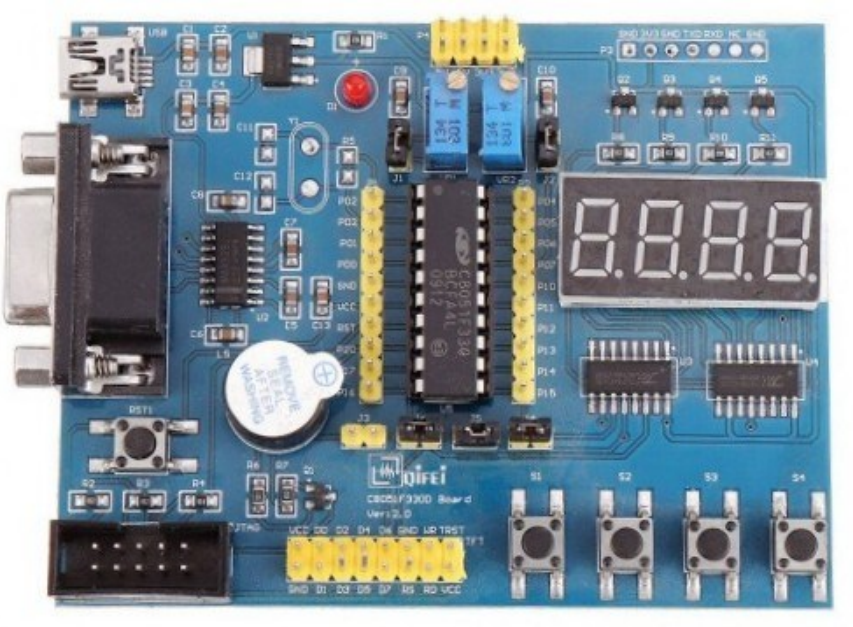
In terms of their 8-bit MCU product line-up, SiLabs offers microcontrollers based on the well-established 8051 architecture. However, their MCUs go beyond being simple, traditional 8051 devices. Like other manufacturers like Nuvoton and STC, SiLabs enhances their MCUs with additional modern hardware components such as DACs and communication peripherals.

SiLabs C8051 microcontrollers are recognized for their good performance, reliability, and scalability. They cater to the evolving needs of the embedded systems industry, whether it's in the realm of consumer electronics, industrial automation, or smart home applications. These microcontrollers serve as a solid foundation for various projects, providing developers with a dependable and flexible platform.

## Hardware

The major hardware items I used for this work are as follows:

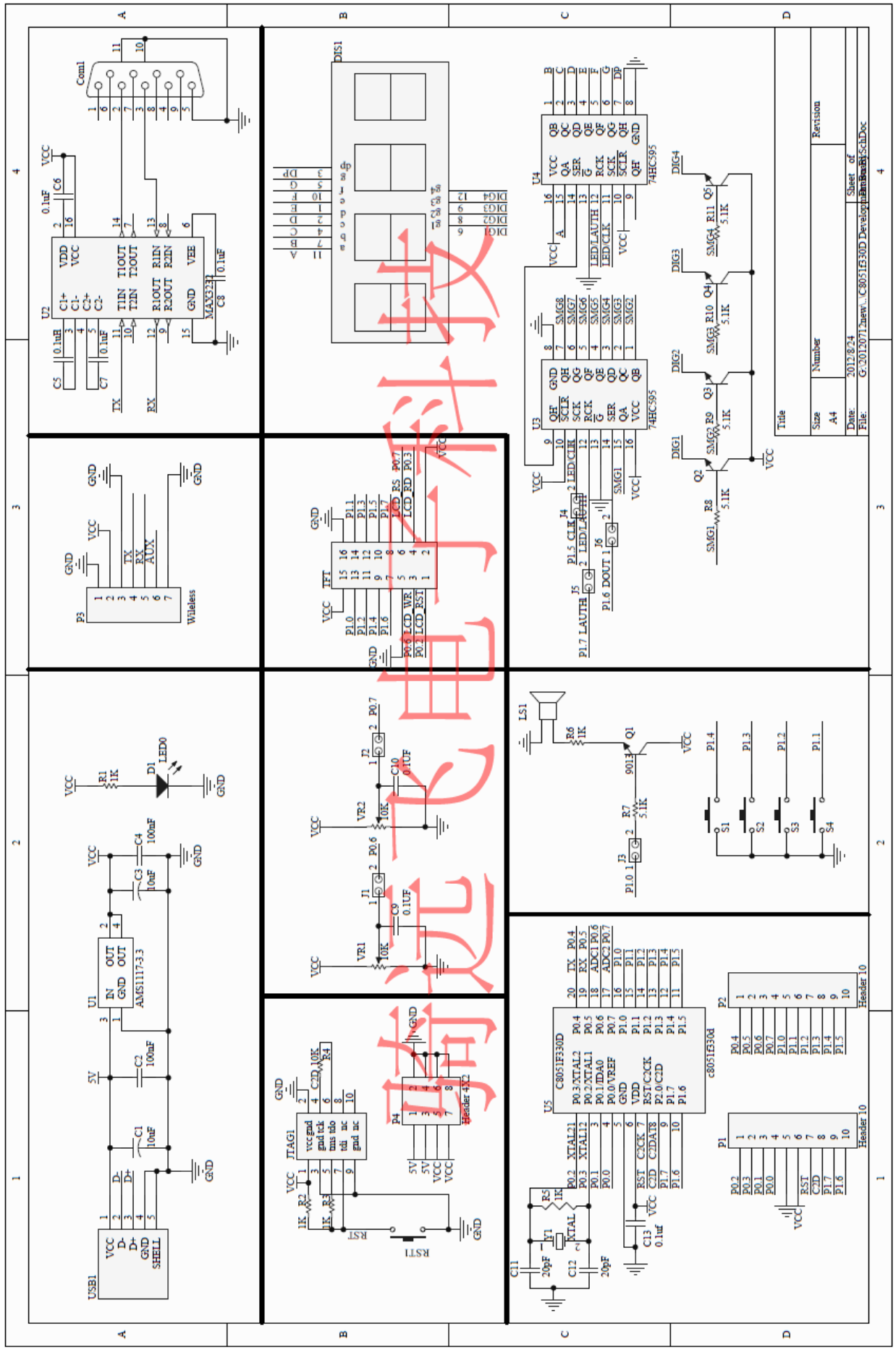
- SiLabs C8051F330D Microcontroller Development Board. This board features a 74HC595 seven-segment display, four input buttons, a programmer-debugger interface, GPIO pin headers, a buzzer, UART to RS232 converter and power pins.



The table below summarizes the feature of C8051F33x microcontrollers. Note only C8051F330 comes in PDIP package and this makes it breadboard-friendly.

| Ordering Part Number | MIPS (Peak) | Flash Memory (kB) | RAM (bytes) | Calibrated Internal 24.5 MHz Oscillator | Internal 80 kHz Oscillator | SMBus/$I^2C$ | Enhanced SPI | UART | Timers (16-bit) | Programmable Counter Array | Digital Port I/Os | 10-bit 200ksps ADC | 10-bit Current Output DAC | Internal Voltage Reference | Temperature Sensor | Analog Comparator | Lead-free (RoHS Compliant) | Package |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C8051F330 | 25 | 8 | 768 | ✓ | ✓ | ✓ | ✓ | ✓ | 4 | ✓ | 17 | ✓ | ✓ | ✓ | ✓ | ✓ | — | QFN-20 |
| C8051F330-GM | 25 | 8 | 768 | ✓ | ✓ | ✓ | ✓ | ✓ | 4 | ✓ | 17 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | QFN-20 |
| C8051F330D | 25 | 8 | 768 | ✓ | ✓ | ✓ | ✓ | ✓ | 4 | ✓ | 17 | ✓ | ✓ | ✓ | ✓ | ✓ | — | PDIP-20 |
| C8051F330-GP | 25 | 8 | 768 | ✓ | ✓ | ✓ | ✓ | ✓ | 4 | ✓ | 17 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | PDIP-20 |
| C8051F331 | 25 | 8 | 768 | ✓ | ✓ | ✓ | ✓ | ✓ | 4 | ✓ | 17 | — | — | — | — | ✓ | — | QFN-20 |
| C8051F331-GM | 25 | 8 | 768 | ✓ | ✓ | ✓ | ✓ | ✓ | 4 | ✓ | 17 | — | — | — | — | ✓ | ✓ | QFN-20 |
| C8051F332-GM | 25 | 4 | 768 | ✓ | ✓ | ✓ | ✓ | ✓ | 4 | ✓ | 17 | ✓ | — | ✓ | ✓ | ✓ | ✓ | QFN-20 |
| C8051F333-GM | 25 | 4 | 768 | ✓ | ✓ | ✓ | ✓ | ✓ | 4 | ✓ | 17 | — | — | — | — | ✓ | ✓ | QFN-20 |
| C8051F334-GM | 25 | 2 | 768 | ✓ | ✓ | ✓ | ✓ | ✓ | 4 | ✓ | 17 | ✓ | — | ✓ | ✓ | ✓ | ✓ | QFN-20 |
| C8051F335-GM | 25 | 2 | 768 | ✓ | ✓ | ✓ | ✓ | ✓ | 4 | ✓ | 17 | — | — | — | — | ✓ | ✓ | QFN-20 |

The board has the schematic as shown below.



- Silicon Labs USB-programmer-debugger. This will be needed to download codes into C8051 microcontrollers.


- Apart from these I have used various breakout boards and sensors. All of the items I used are common and inexpensive.

- I also recommend downloading the latest datasheet, app notes and other tools of the C8051F330D microcontroller from the SiLabs' official website.

## Software Suite

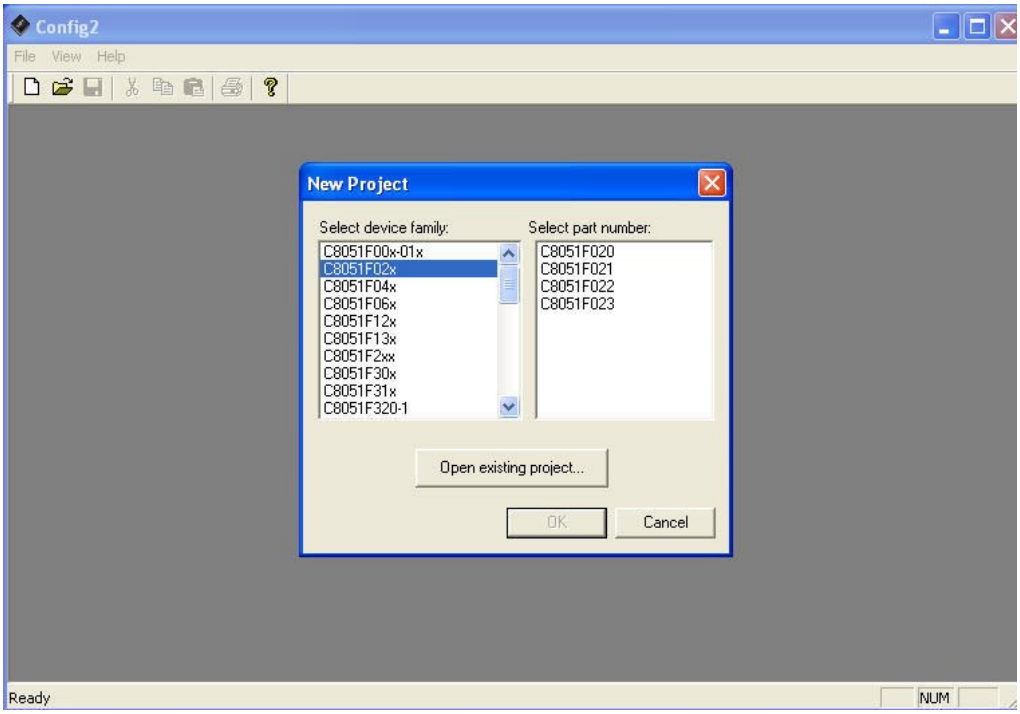
For coding, I used the following software:

- **MikroC Pro for 8051 Compiler**

  This is a C compiler and IDE from MikroElektronika. I chose MikroC Pro for 8051 for quick development and other facilities like their rich collection of libraries and utilities. MikroElektronika also has several hardware and software solutions for rapid prototyping and DIY. I have been using MikroC compilers for 8051, AVR and ARM microcontrollers for quite a long time and so far, personally I am satisfied with their products and services.

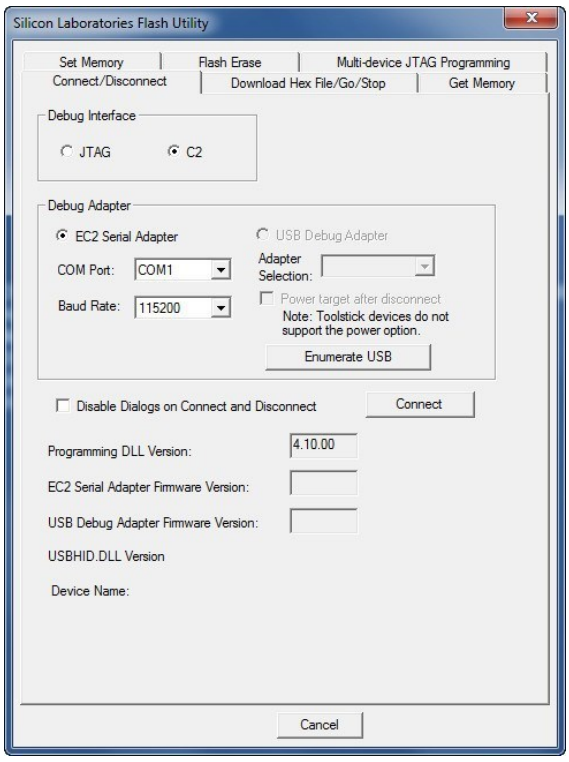

- **Si Labs Code Configuration Wizard 2**

  When it comes to working with a new microcontroller, the most tedious job is going through the functionalities of its internal registers, and without setting them up, we would never get the desired outcome. Any coder would need a good understanding of internal peripherals and something to speed up code development. Surely, no coder would like to waste time setting internal registers manually. To address this issue, most present-day embedded-hardware manufacturers have opted for software solutions like code generators, software libraries and board support packages. In the case of SiLabs' microcontrollers, we have the luck of using their free code generator software called ***Si Lab Code Configuration Wizard 2***. This simple-to-use software can generate register initialization codes for all hardware peripherals in either C or ASM (assembly). Thus, a lot of coding effort is reduced significantly.




- **Si Labs Flash Program Utility**

  Lastly, we would need software and hardware to flash code into our target microcontroller, and ***SiLabs Flash Program Utility*** is just that tool.

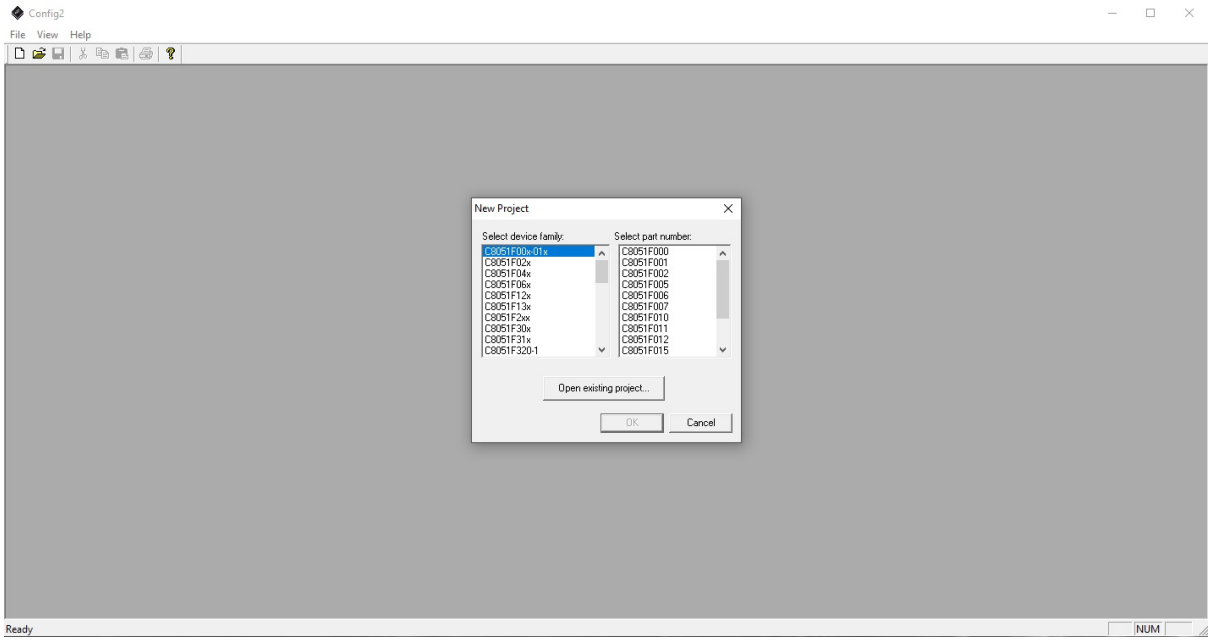



Both Si Labs' software can be downloaded from here.

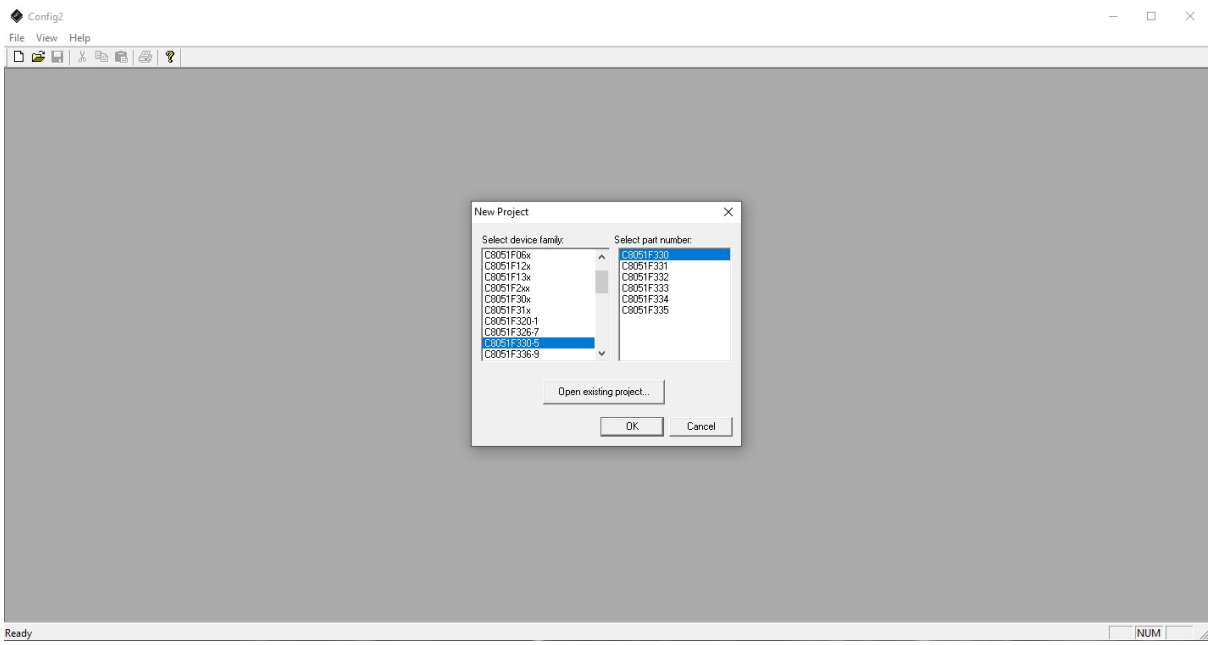
MikroC for 8051 can be downloaded from here.

# How to use Configuration Wizard 2 Code Generator?

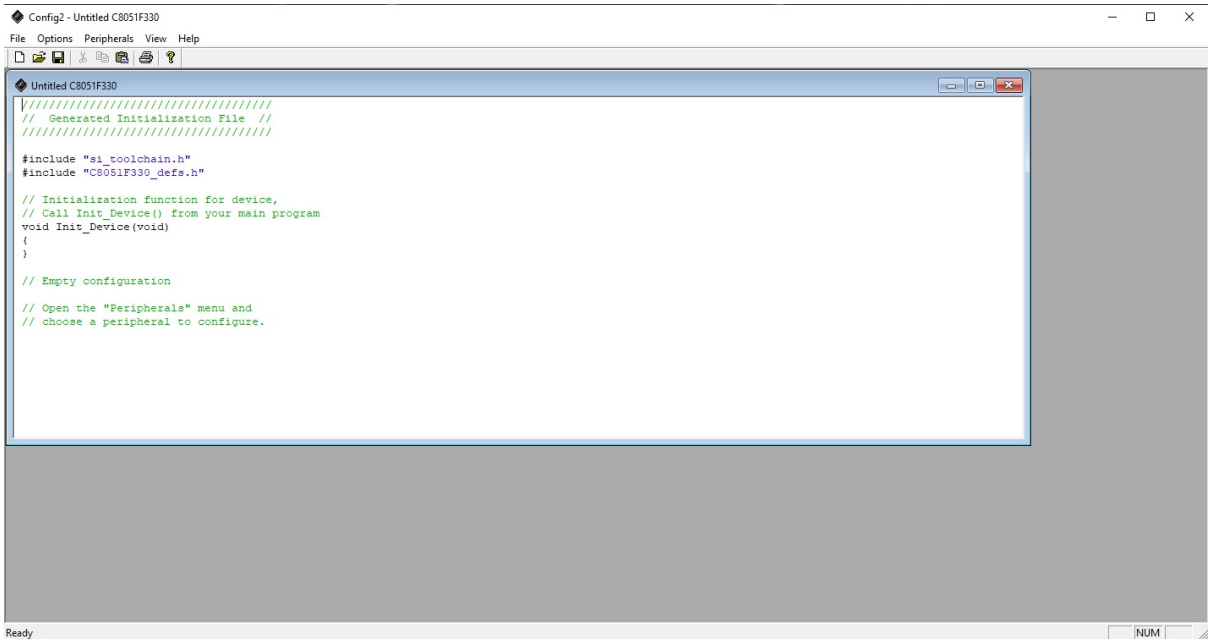
After running the software, the following screen would show up first.



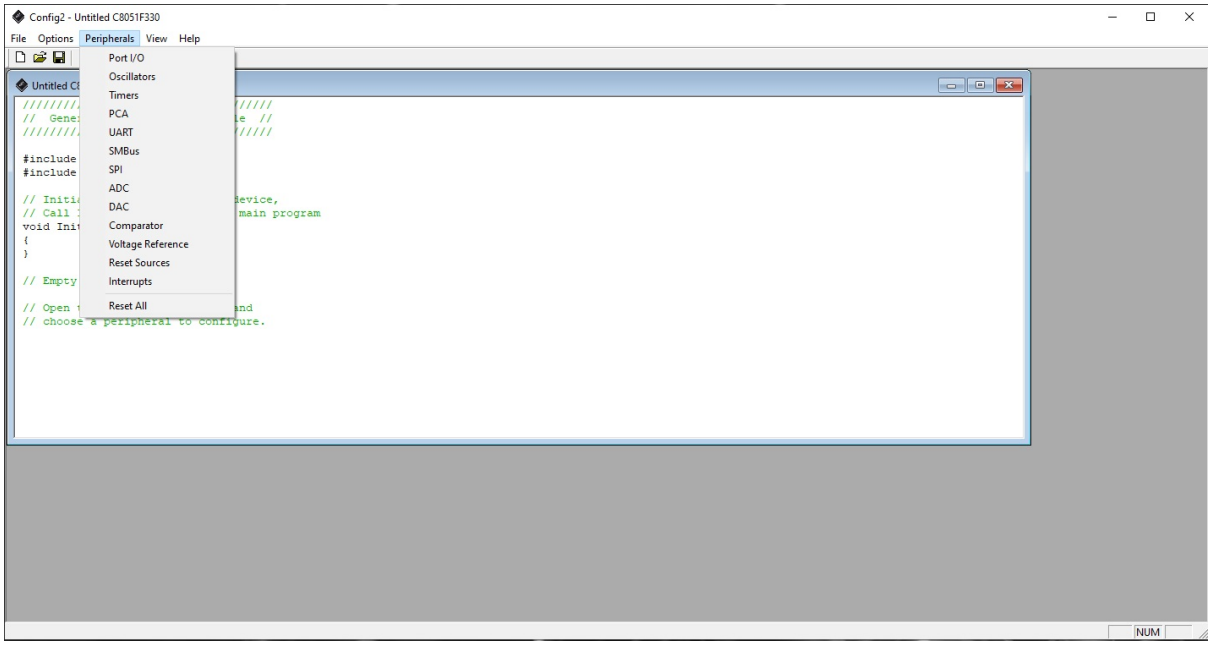

From here we have to choose the target microcontroller from family and part number.



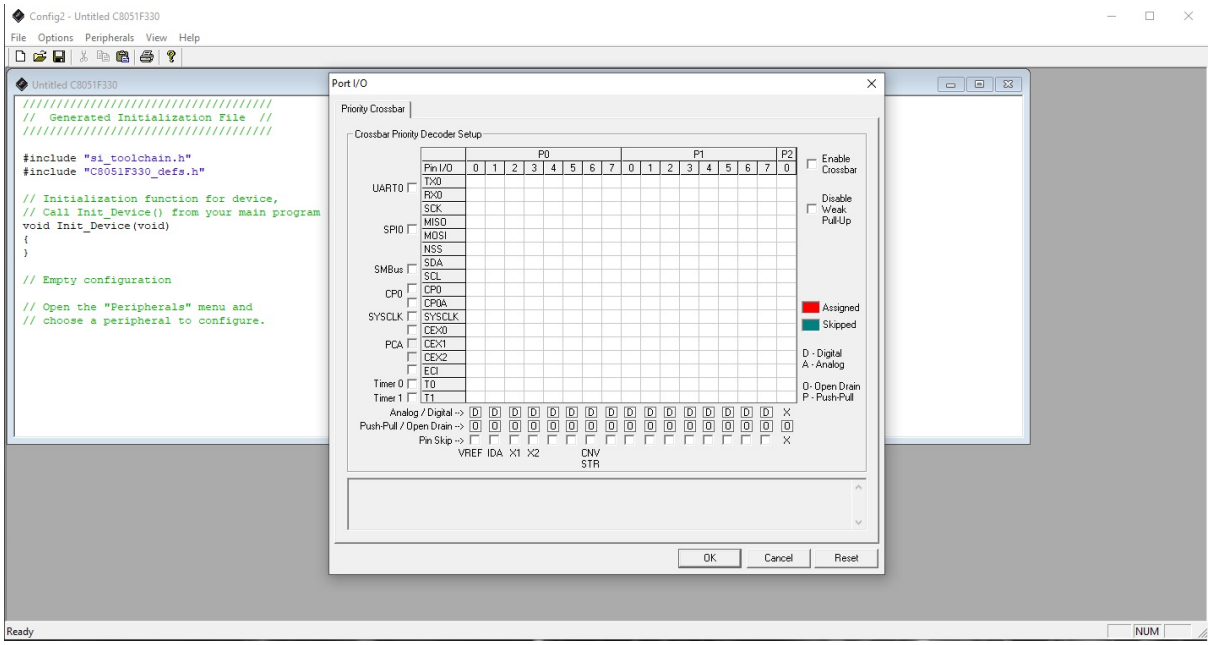
The following screen pops up after chip selection. Note that it is empty as no peripheral has been initialized.



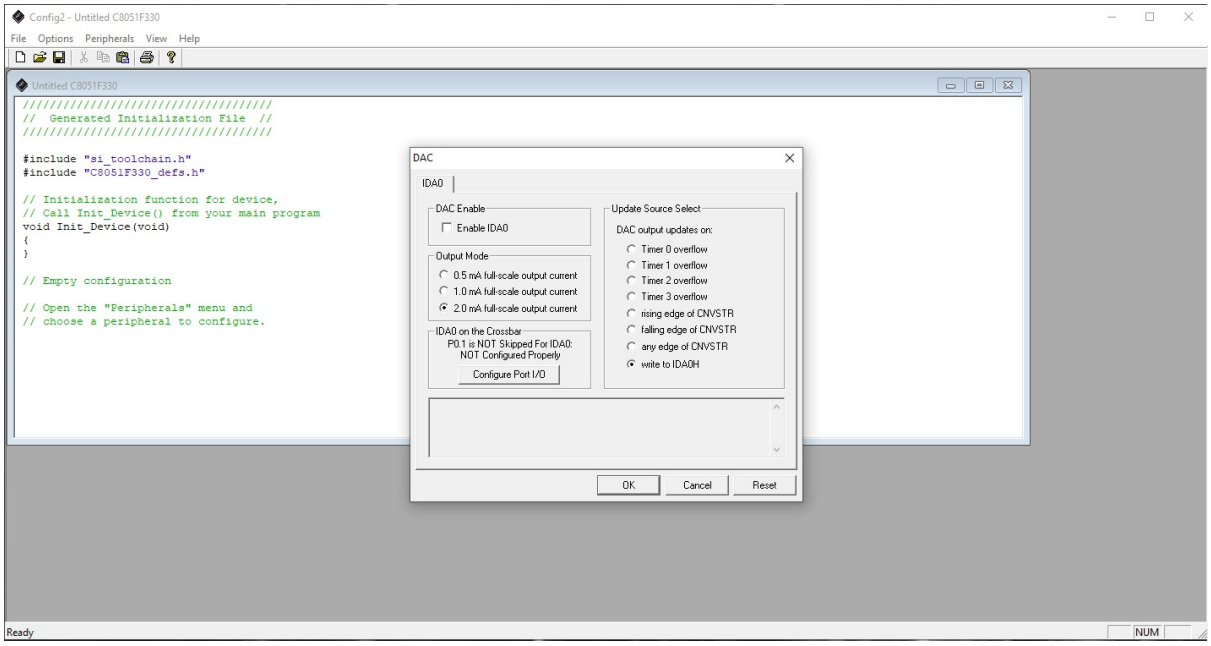

Now we have to click the *"Peripherals"* tab and explore what peripherals the selected chip offers.



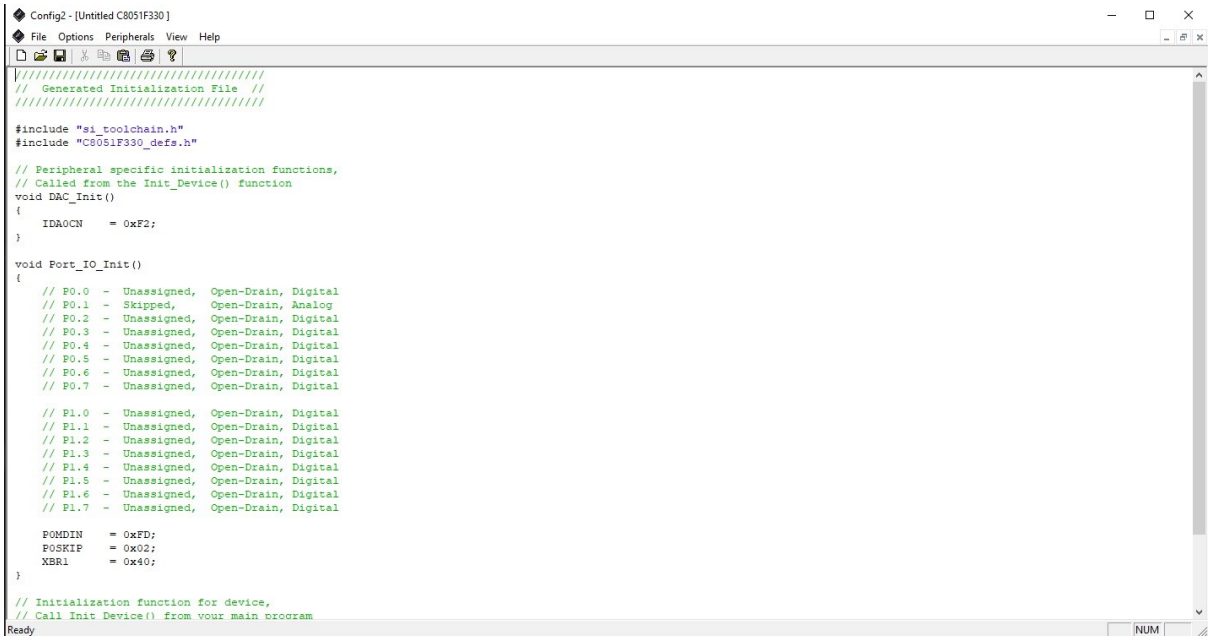
As an example, let's see the Port I/O peripheral. On selecting it, the following window pops up and it shows all the GPIOs and the options we have for them.




The same applies to other peripherals such as the internal DAC.



After setting up the hardware peripherals as per need, the software automatically generates a temporary initialization code with comments. The comments give some overview of our initialization. We can save it for future reference or just copy it to the MikroC compiler. In case of copying, we do not need to copy the *#include* directives. All we need are the register values.

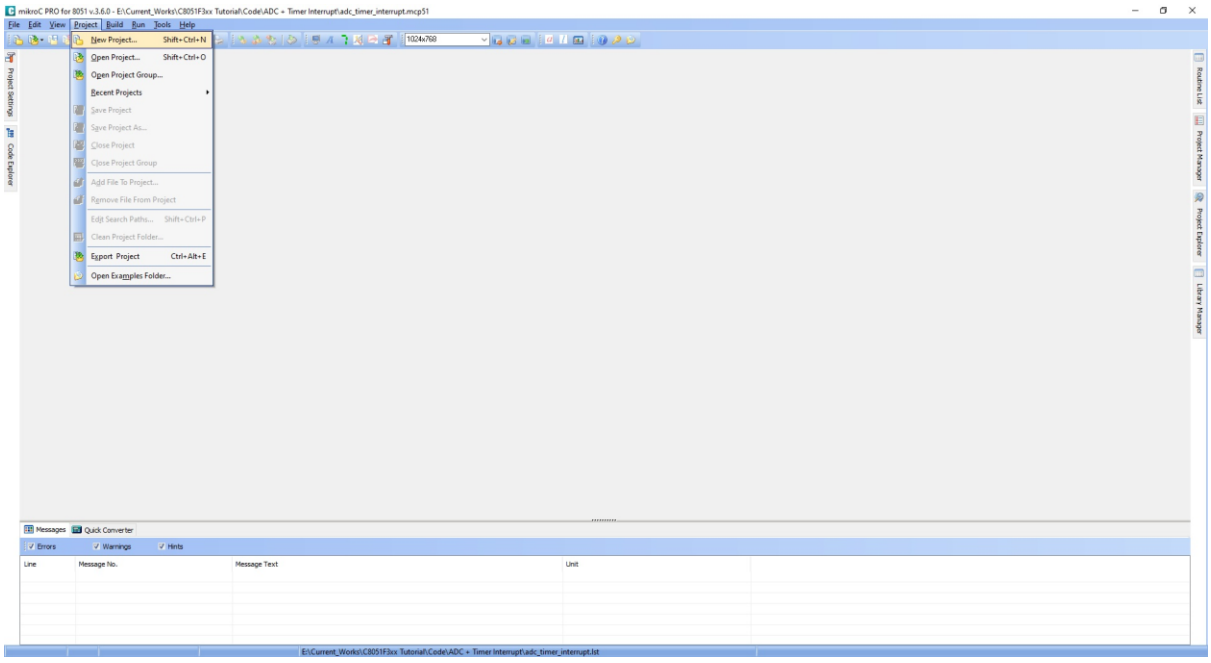



When changes are made in hardware peripheral settings, the generated code is automatically regenerated.

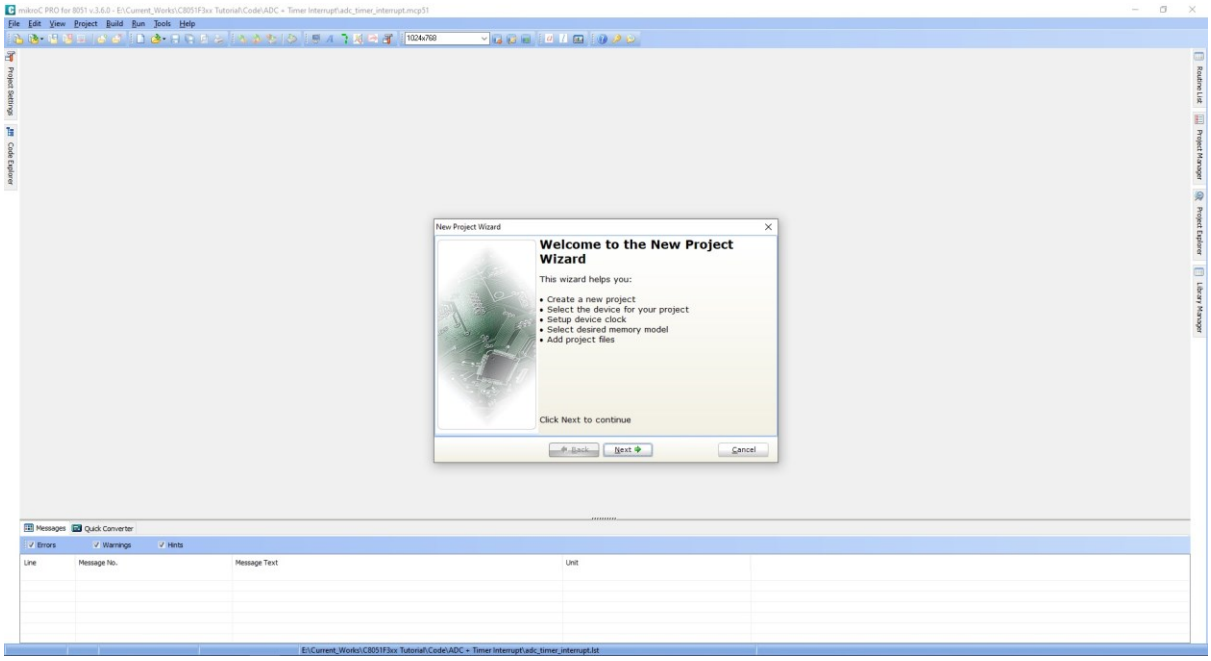
# How to use MikroC Pro for 8051?

MikroC Pro for 8051 is very simple to use. Like many C compilers, it uses a projected-based file-folder structural approach.

Thus, after running it, we should start by opening a new project unless we already have created one before.

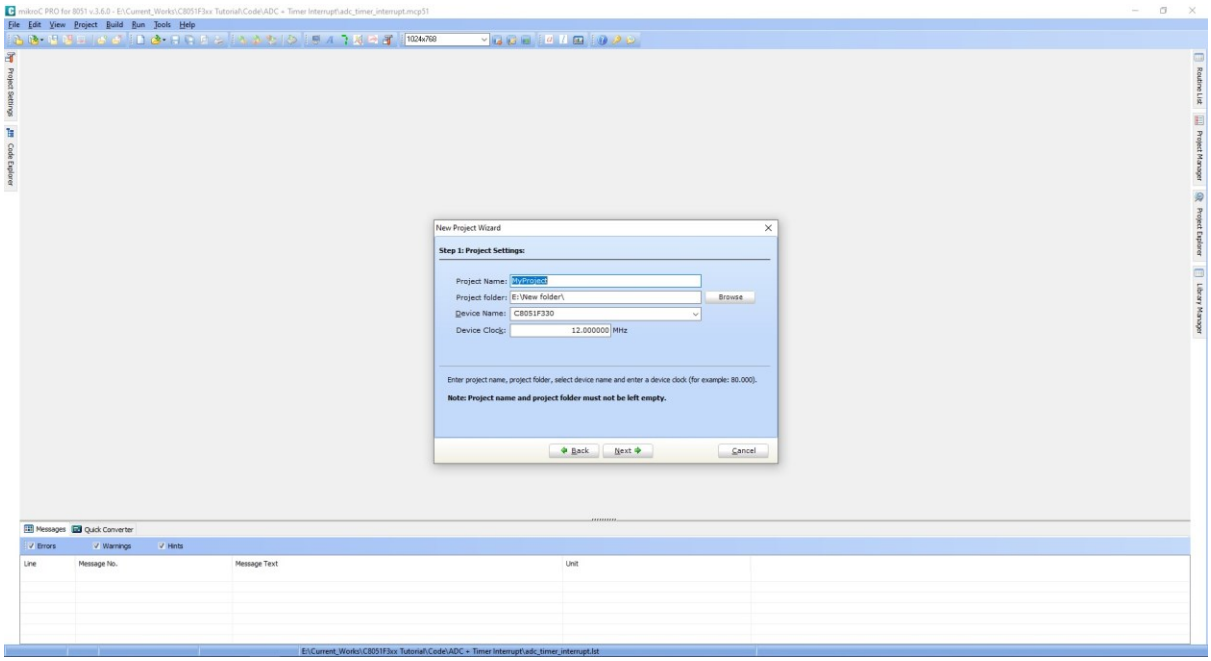


Once we click “*New Project*”, a project wizard pops up and it guides us in creating a new project in a step-by-step approach.

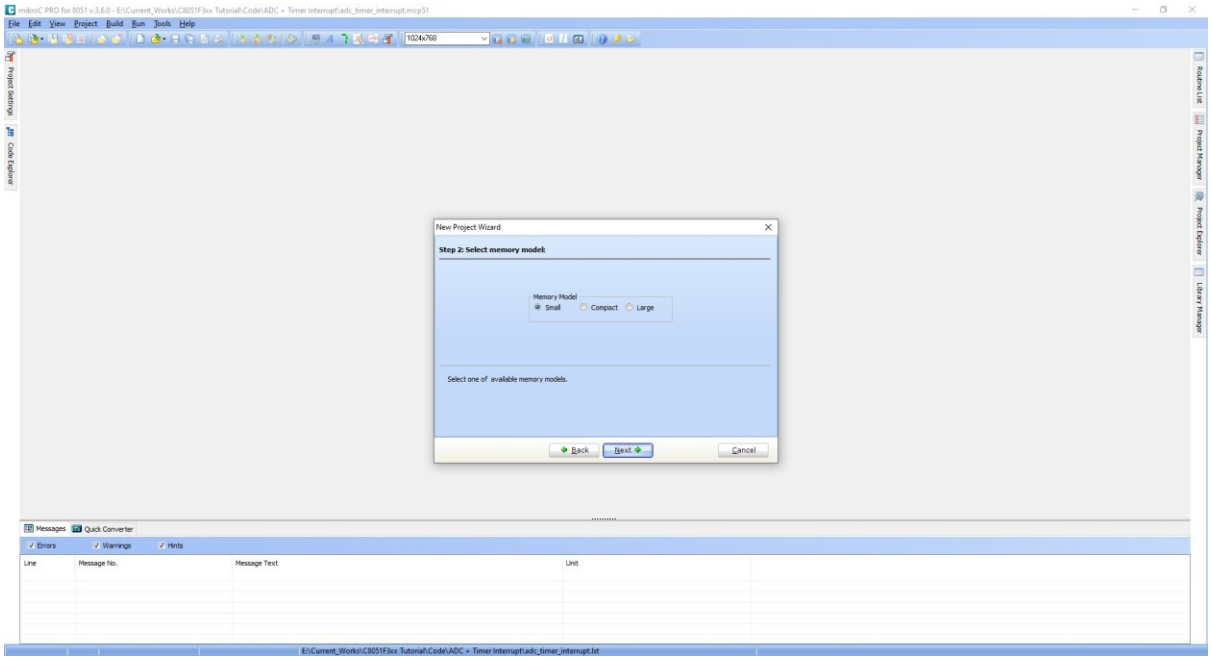



The important things that we need to define when creating a new project are:

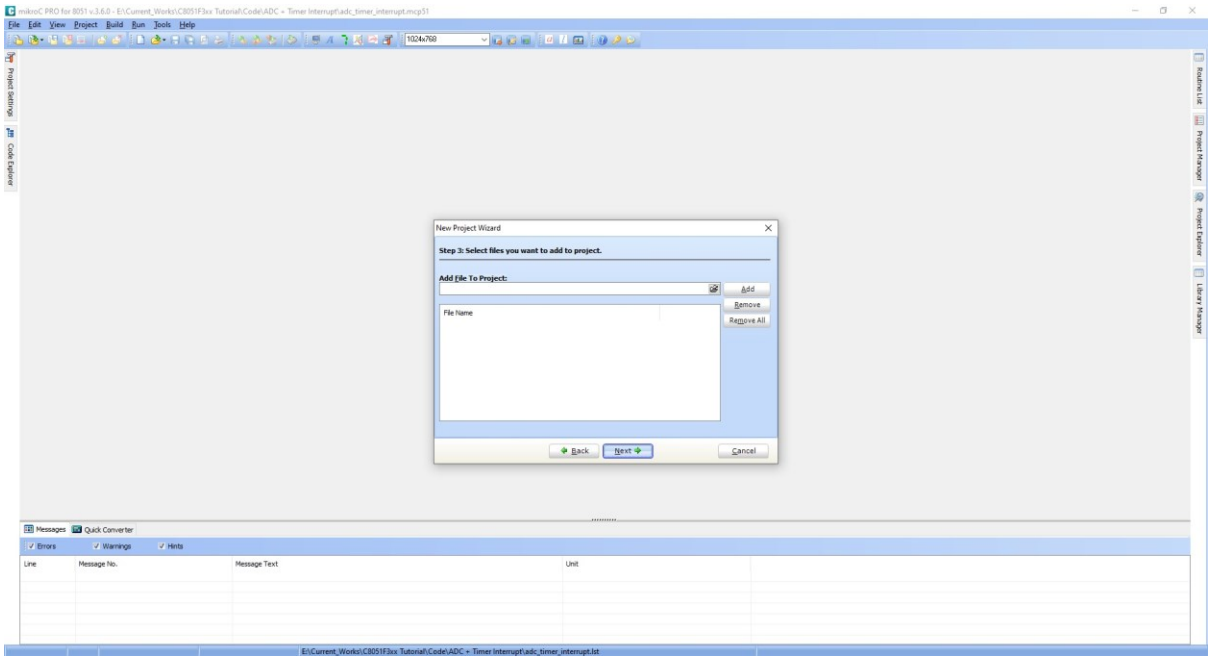
- Project Name
- Project Location in our computer
- Part Number or MCU
- Clock Speed



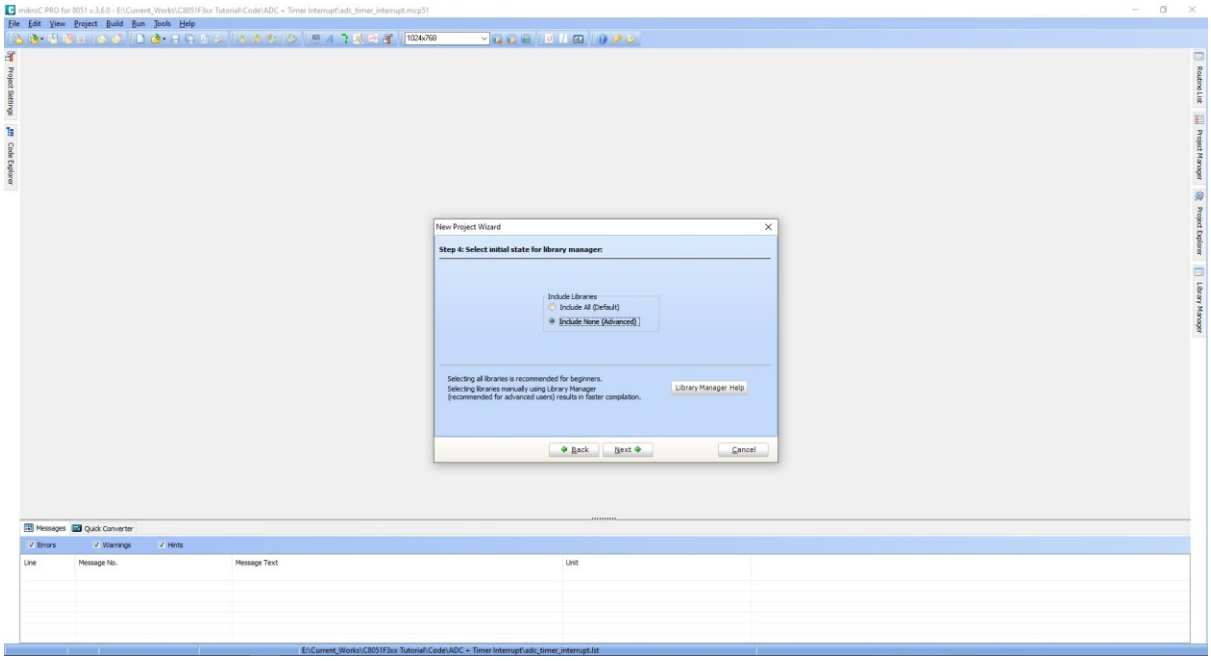
Next, we have to define the memory model. In most cases, the *"Small"* memory model is chosen. Details about the memory model are discussed here.



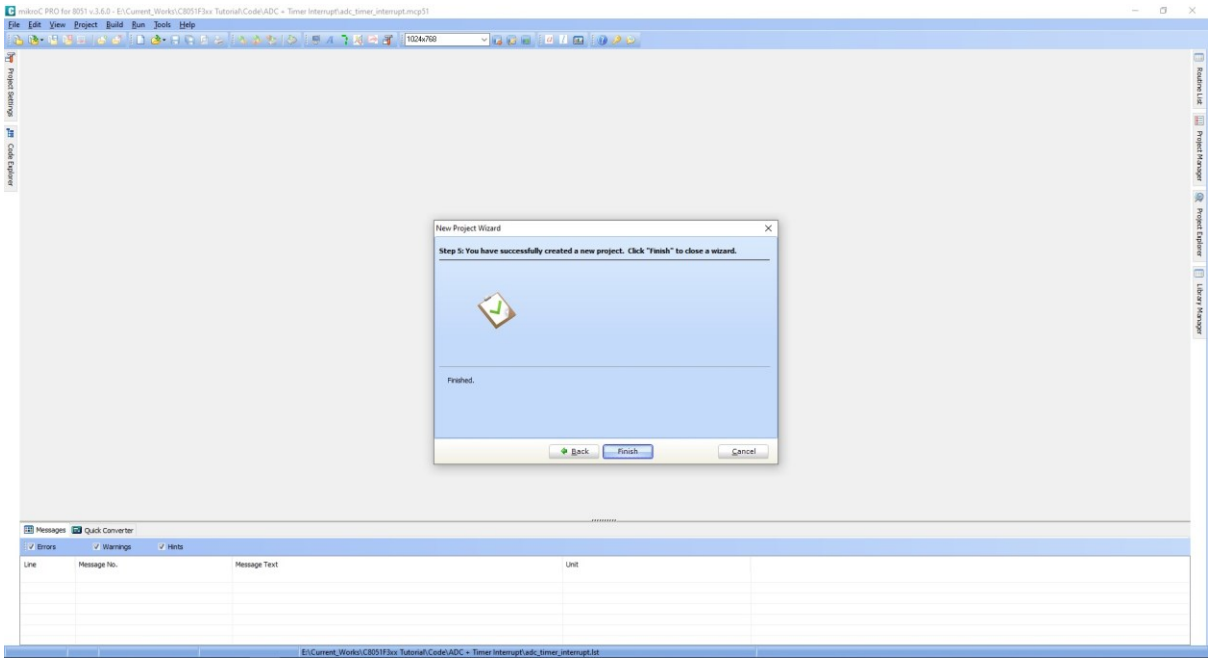

There is an option to add additional user files such as custom libraries and files such as images, notes, etc. that are needed to be associated with the project under construction. Usually, I keep it empty and add my own fabricated libraries manually.



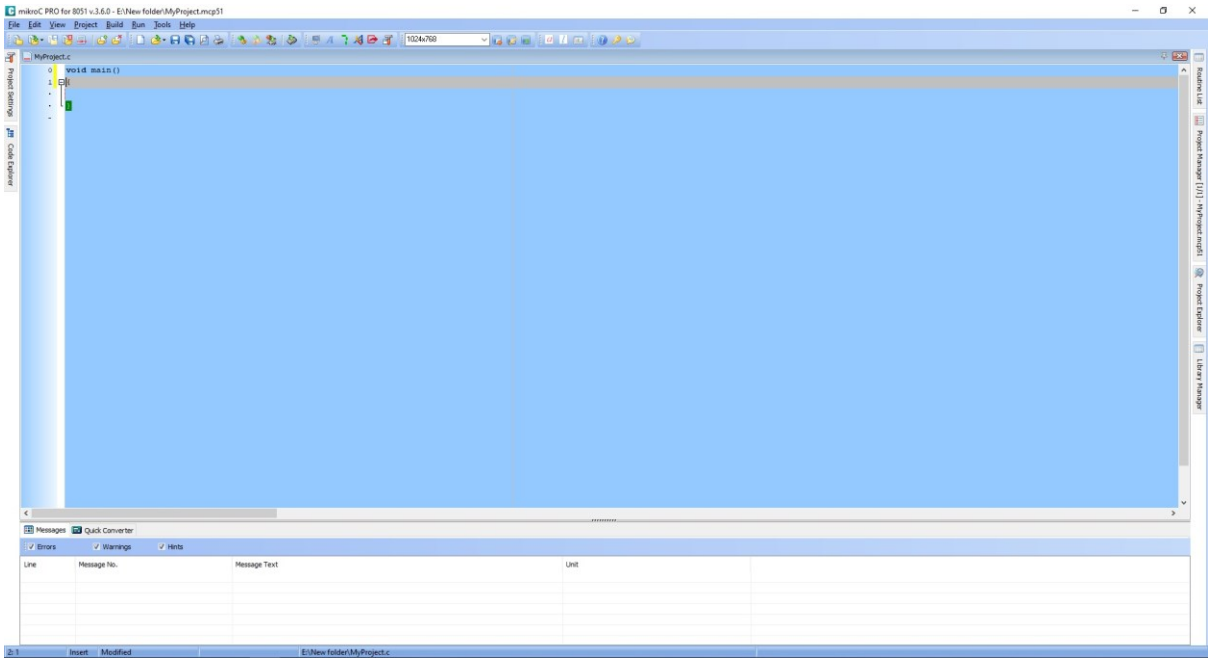
Lastly, the MikroC compiler comes with many prebuilt libraries. We should not include them at this stage as code conflicts are likely the other way.



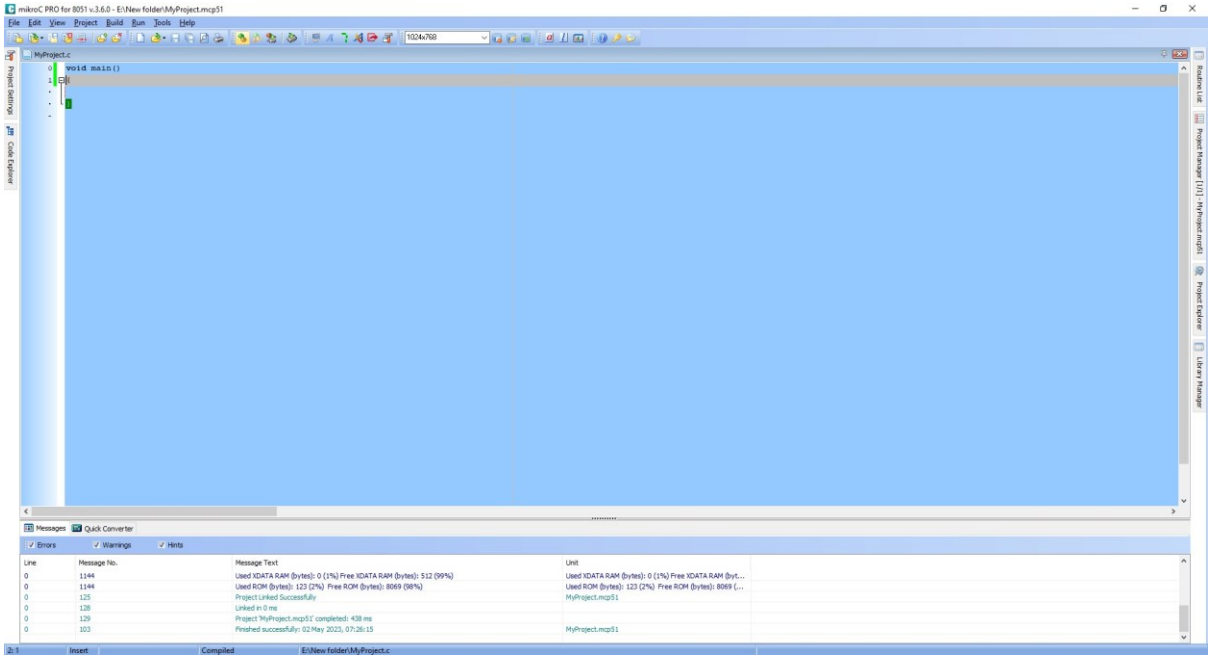

This completes a new project construction.



Once the wizard is completed and closed, we are good to go for coding.



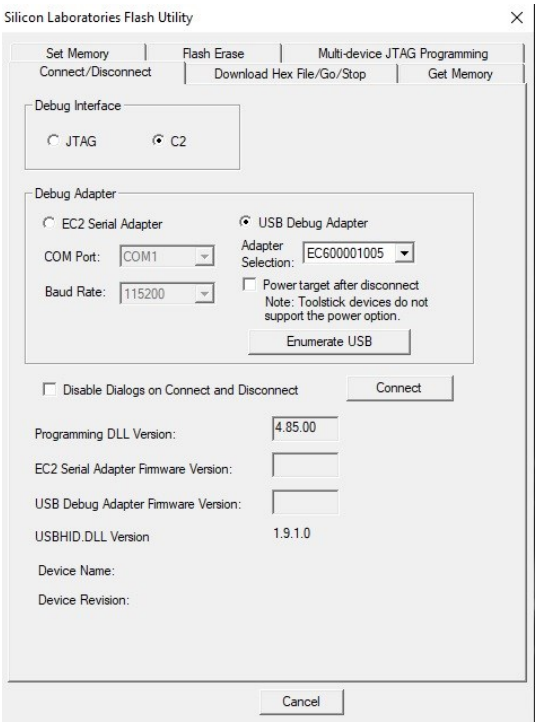

We can begin coding by copying code generated by Configuration Wizard 2 and adding necessary built-in and personally-coded libraries.



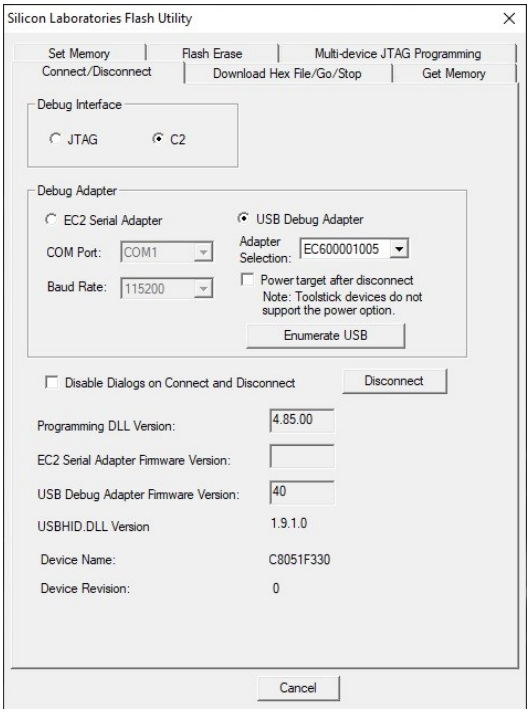

On successful code compilation, we get messages regarding memory usage (RAM and ROM) and generated files. Likewise, when there is an error, we get error messages, stating the source and cause of that error.

## How to use Silicon Laboratories Flash Utility?

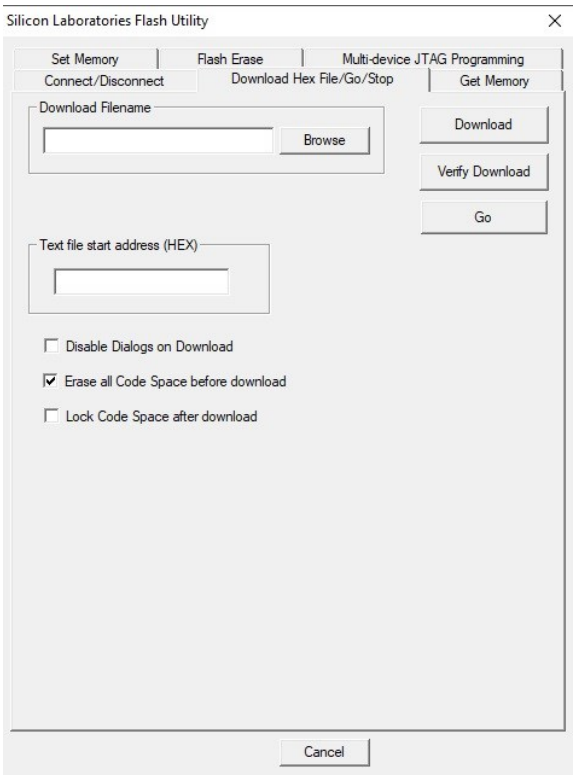
The first screen that appears after running the software is shown below. Note at this stage no chip or programmer is connected. We would always be using C2 debug interface.



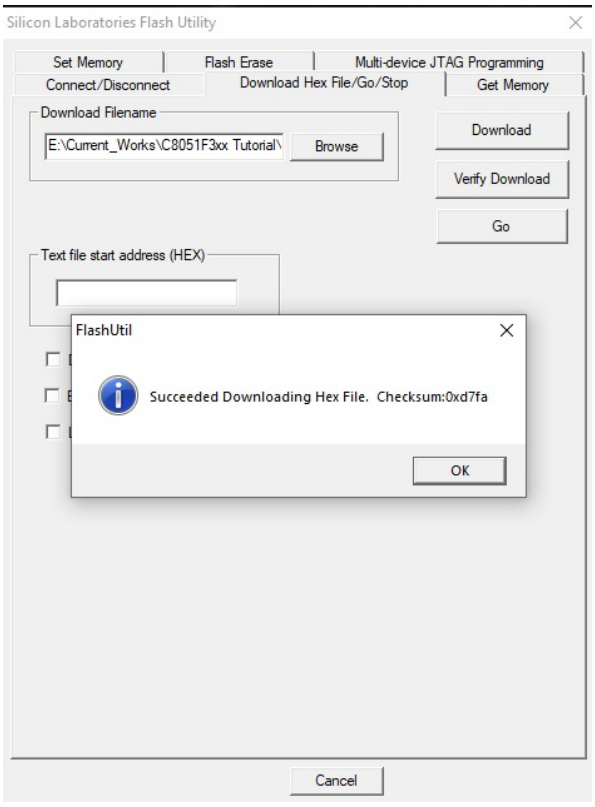

After physically connecting a programmer and a chip, we have to click the connect button. Also note that after a successful connection, both the part number and programmer firmware version show up.



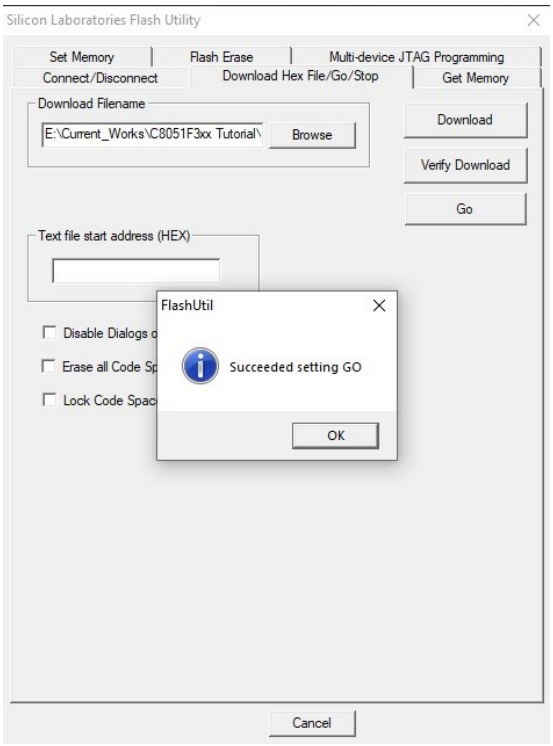
Once connected, we have to select the *"Download Hex File/Go/Stop"* tab. Check the "*Erase all Code Space before download"* checkbox.



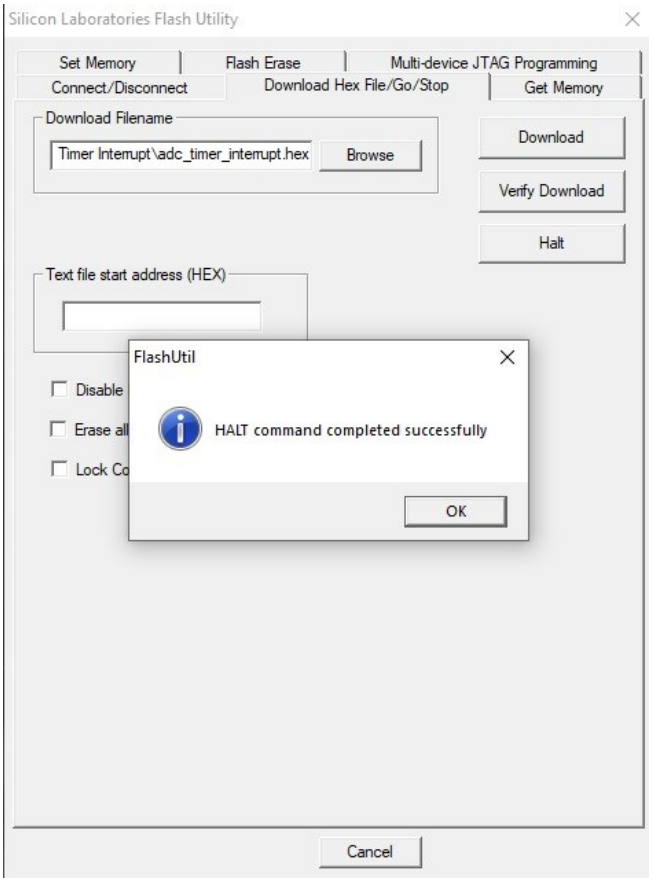

We have to set the target hex file location by browsing its location in our computer. When the *"Download"* button is clicked, the hex file is downloaded to the target chip.



On successful download, a popup message box appears with checksum value. Now the downloaded code is ready to be run and so clicking the *"Go"* button, runs the downloaded code.

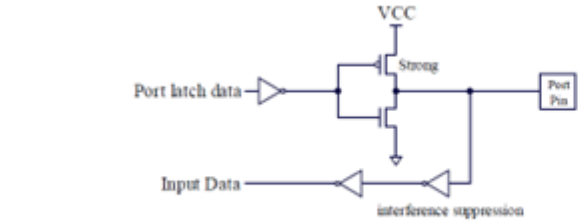



To stop the code from execution, we can click the *"Halt"* button and the code stops working.

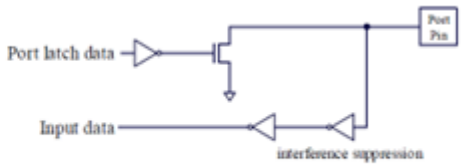


## General Purpose Input-Output (GPIO) and Clock System

Playing with the GPIOs of C8051 microcontrollers is very easy. GPIOs can be either open-drain inputs and push-pull type outputs. The arrangements are simply as shown below:

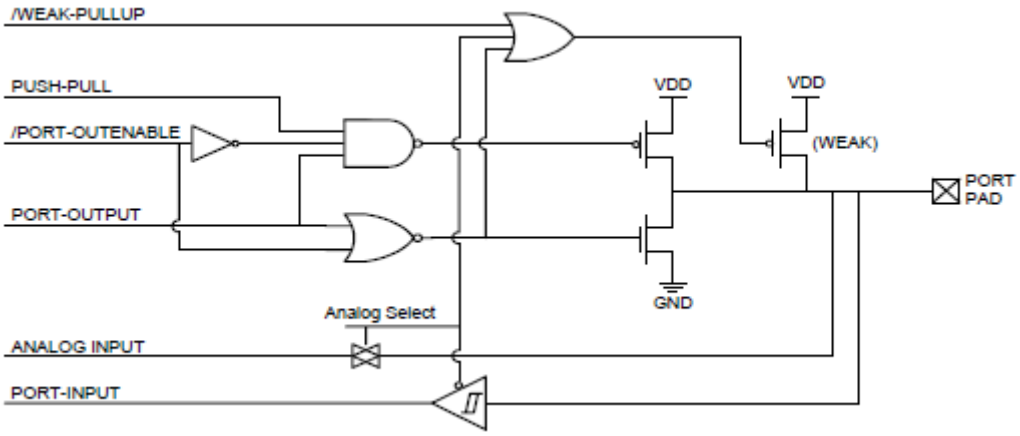



Push-Pull Mode



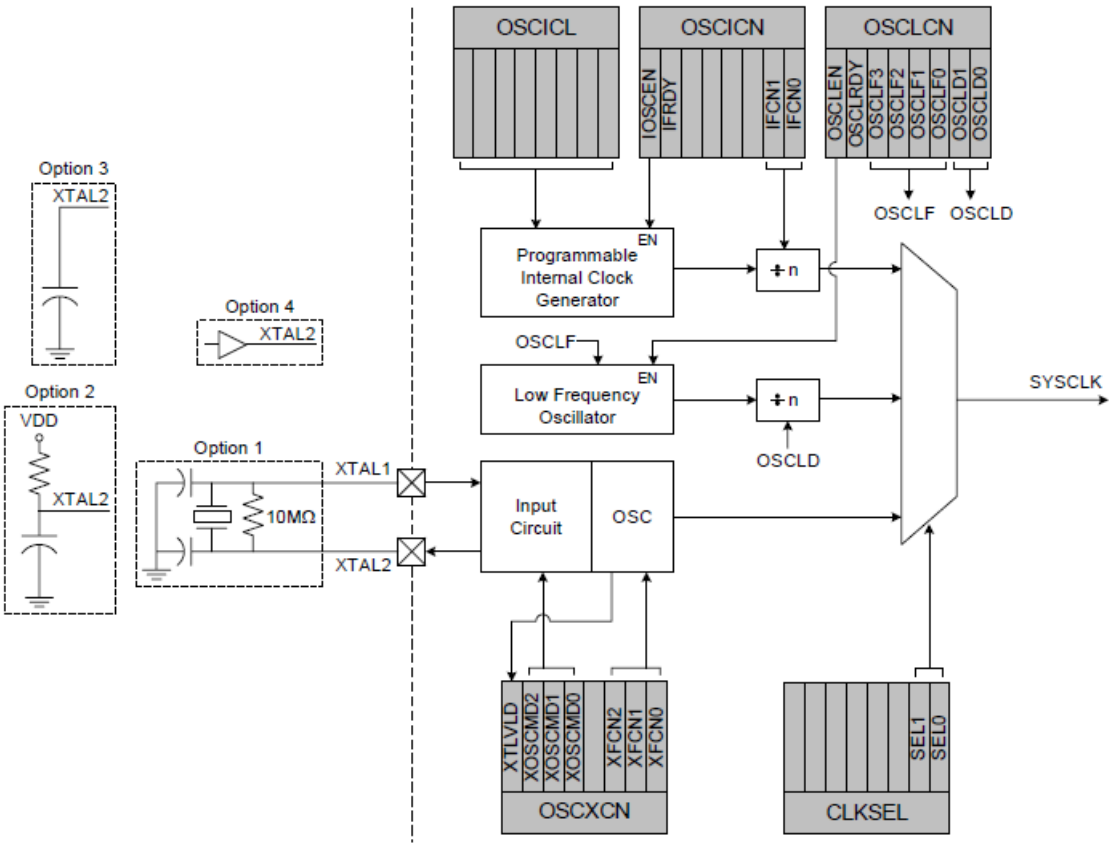

Open-Drain Mode

The general block diagram for a C8051 GPIO pin is shown below:




This arrangement is typical for all 8051-based microcontrollers. The subtle differences are the Schmitt-trigger and analogue input channels.

The clock system is also pretty straightforward. There is only one system clock and it can be fed with several clock sources. These include the internal low-frequency low accuracy oscillator, the programmable high-frequency internal clock generator or an external clock source which can be anything from a crystal oscillator to simple RC oscillators. Faster clocks are needed for fast data processing while slower clocks are intended for low power consumption. To vary clock speed, there are also clock dividers to scale clock sources according to requirement. All of these can be set in code. Refer to the block diagram below:

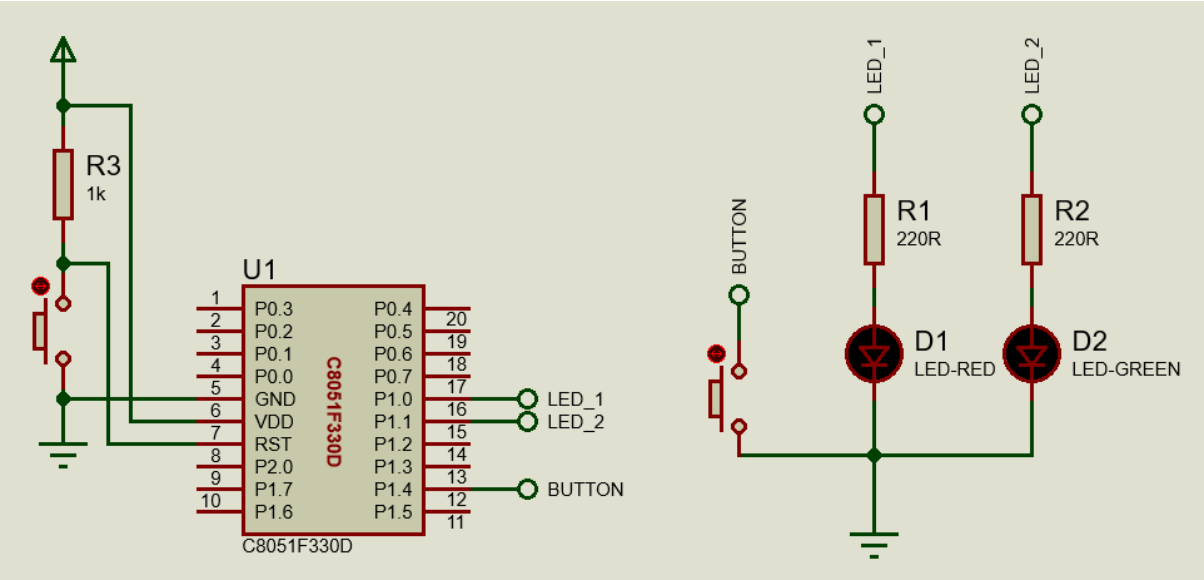


## Code

```
void PCA_Init(void);
void Port_IO_Init(void);
void Init_Device(void);


void main(void)
{
    unsigned char i = 0;

    Init_Device();

    P1_0_bit = 1;
    P1_1_bit = 0;

    while(1)
    {
       if(P1_4_bit == 0)
       {
           delay_ms(100);
           i++;
       }

       switch(i)
       {
           case 0:
           {
               OSCICN = 0x83;
               break;
           }
           case 1:
           {
               OSCICN = 0x82;
               break;
           }
           case 2:
           {
               OSCICN = 0x81;
               break;
           }
           case 3:
           {
               OSCICN = 0x80;
               break;
           }
           default:
           {
               i = 0;
               break;
           }
       }

       P1_0_bit ^= 1;
       P1_1_bit ^= 1;
       delay_ms(100);
    };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
```

```
}


void Port_IO_Init(void)
{
    // P0.0  -  Unassigned,  Open-Drain, Digital
    // P0.1  -  Unassigned,  Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Skipped,     Push-Pull,  Digital
    // P1.1  -  Skipped,     Push-Pull,  Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Skipped,     Open-Drain, Digital
    // P1.5  -  Unassigned,  Open-Drain, Digital
    // P1.6  -  Unassigned,  Open-Drain, Digital
    // P1.7  -  Unassigned,  Open-Drain, Digital

    P1MDOUT = 0x03;
    P1SKIP = 0x13;
    XBR1 = 0x40;
}


void Init_Device(void)
{
    PCA_Init();
    Port_IO_Init();
}
```

## Schematic



## Explanation

To demonstrate both the clock system and the GPIOs, I used a simple LED blinking example.

Two LEDs connected with P1.0 and P1.1 blink alternatively like a police light. These pins are set as push-pull outputs.

A push button connected with P1.4 is also set as an open-drain input. An open-drain input is not purely open from VDD as there is a weak internal pullup. The purpose of this button is to change the clock frequency by altering the clock divider.

```
void Port_IO_Init(void)
{
    // P0.0  -  Unassigned,  Open-Drain, Digital
    // P0.1  -  Unassigned,  Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Skipped,     Push-Pull,  Digital
    // P1.1  -  Skipped,     Push-Pull,  Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Skipped,     Open-Drain, Digital
    // P1.5  -  Unassigned,  Open-Drain, Digital
    // P1.6  -  Unassigned,  Open-Drain, Digital
    // P1.7  -  Unassigned,  Open-Drain, Digital

    P1MDOUT = 0x03;
    P1SKIP = 0x13;
    XBR1 = 0x40;
}
```

By default, the internal high-frequency oscillator is selected as the system clock source. This source is reasonably accurate for typical applications. Inside the code, the oscillator divider is altered to change the system clock frequency.

```
void main(void)
{
    unsigned char i = 0;

    Init_Device();

    P1_0_bit = 1;
    P1_1_bit = 0;

    while(1)
    {
       if(P1_4_bit == 0)
       {
           delay_ms(100);
           i++;
       }

       switch(i)
       {
           case 0:
```

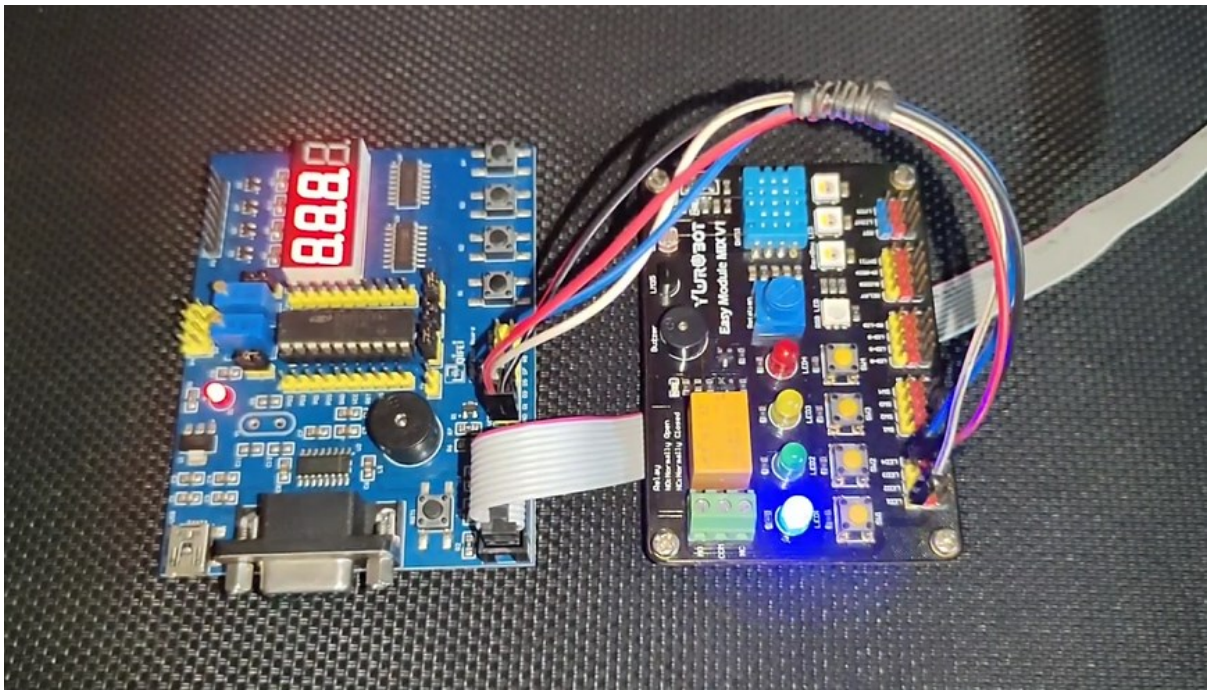
```
            {
                OSCICN = 0x83;
                break;
            }
            case 1:
            {
                OSCICN = 0x82;
                break;
            }
            case 2:
            {
                OSCICN = 0x81;
                break;
            }
            case 3:
            {
                OSCICN = 0x80;
                break;
            }
            default:
            {
                i = 0;
                break;
            }
        }

        P1_0_bit ^= 1;
        P1_1_bit ^= 1;
        delay_ms(100);
    };
}
```

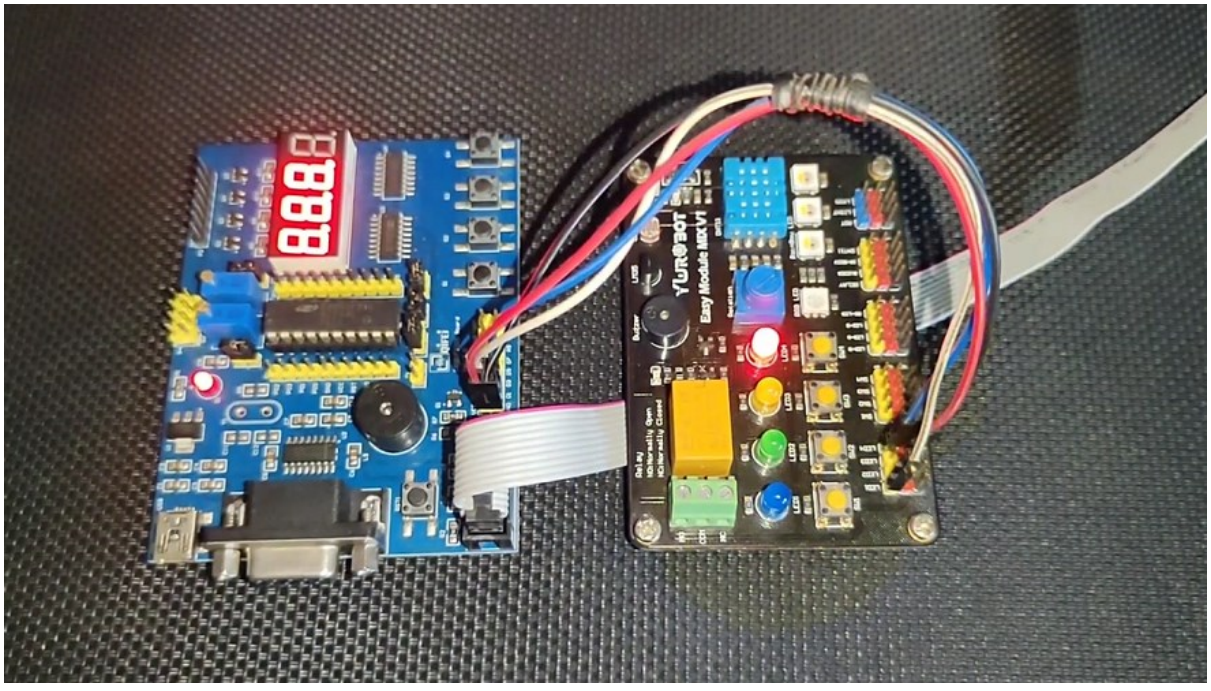
Since the software delay amount is fixed, changing the system clock frequency alters the blink rate.

A word of caution that I would like to share at this point that is, the internal watchdog timer (WDT) is started but default and so we need to stop it before doing anything else. If this is not done then our code would seemingly not work at all as the WDT would repeatedly overflow and reset the microcontroller. In C8051s, the WDT is a part of the PCA module and so the PCA module needs to be configured as follows.

```
void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}
```

## Demo





Demo video link: https://youtu.be/c6nSj4rbI-s

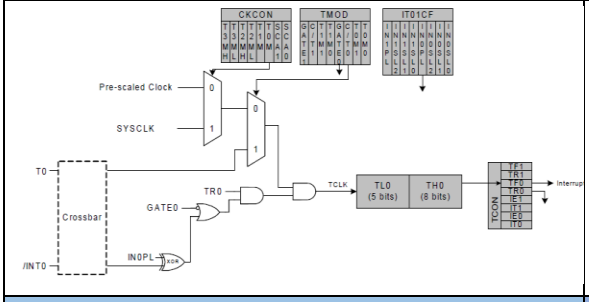
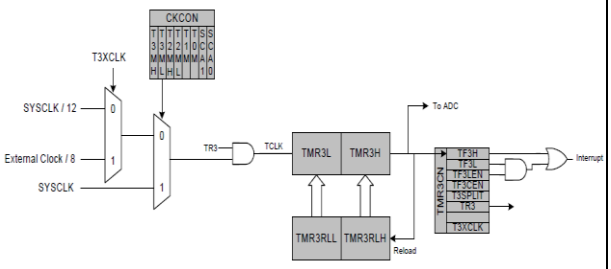
# Timer (TMR) – Countdown Timer

In digital electronics and embedded systems, timing is literally everything and so this makes timers of any microcontroller very important. The C8051F330D microcontroller has 4 timers – Timer 0 to Timer 3.

Timers 0 and 1 are similar to the ones we would expect with any standard 8051 microcontrollers. They can be used as timers or counters. In counter mode, these timers can count external pulses/inputs.

Timers 2 and 3 are additional timers and they have the same features and similar operating procedures. These timers cannot be used as counters.

Timers are the clock source for some internal hardware peripherals such as UARTs and SMBUS. All of these timers can be clocked with various clock sources. The table shown below summarizes the mode of operation of all of these timers.

<table>
<tr><th>Timer 0 and Timer 1</th><th>Timer 2 and Timer 3</th></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>13-bit Timer/Counter</td><td rowspan="2">16-bit Timer with Auto-reload</td></tr>
<tr><td>16-bit Timer/Counter</td></tr>
<tr><td>1x 8-bit Timer/Counter with Auto-reload</td><td rowspan="2">2x 8-bit Timer/Counter with Auto-reload</td></tr>
<tr><td>2x 8-bit Timers/Counters (Timer 0 Only)</td></tr>
</table>

All of these timers can be configured in various ways and at that moment things get complex. Fortunately for us, SiLabs' *Configuration Wizard 2* makes coding a lot easy as it takes care of setting these timers just like other hardware peripherals.

## Code

```
#define LED_DOUT           P1_6_bit
#define LED_CLK            P1_5_bit
#define LED_LATCH          P1_7_bit

#define SET                P1_4_bit
#define INC                P1_3_bit
#define DEC                P1_2_bit
#define ESC                P1_1_bit


unsigned char i = 0;
register unsigned char val = 0;
signed int s = 0;
unsigned int ms = 0;

const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x7F, // .
  0xBF  // -
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


void PCA_Init(void);
void Timer_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void data_converter(unsigned int val);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);


void Timer_2_ISR(void)
iv IVT_ADDR_ET2
ilevel 1
ics ICS_AUTO
{
    ms++;
    if(ms > 999)
    {
        ms = 0;
        s--;
```

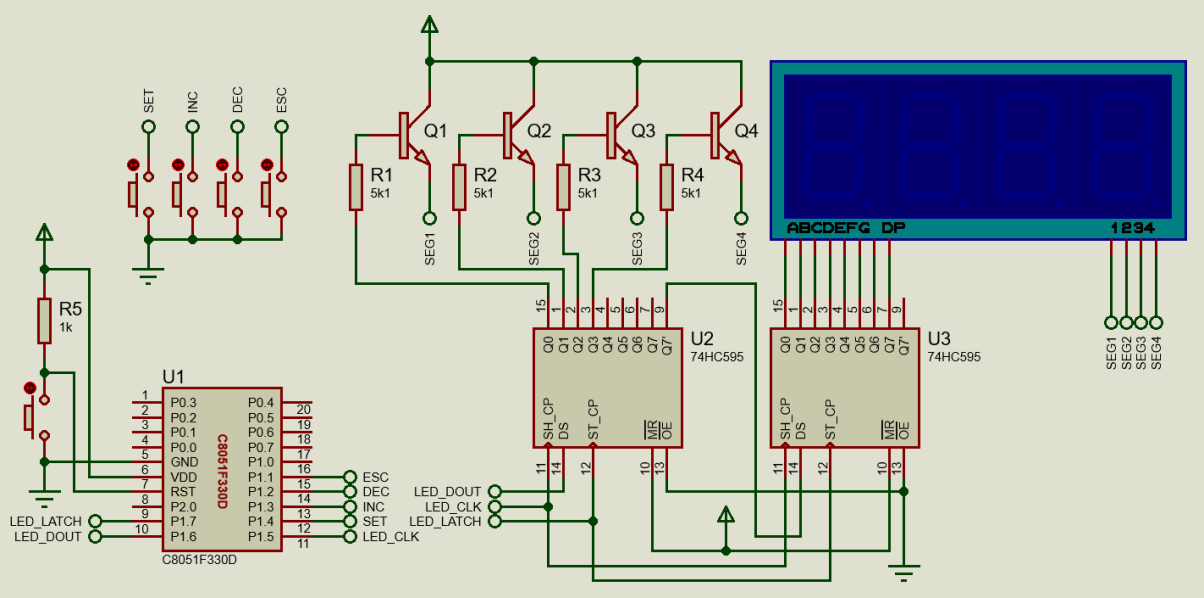
```
        if(s < 0)
        {
           s = 0;
           TR2_bit = 0;
        }
    }

    TMR2CN &= 0x7F;
}


void Timer_3_ISR(void)
iv IVT_ADDR_ET3
ilevel 0
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (s / 1000);
            break;
        }
        case 1:
        {
            val = ((s % 1000) / 100);
            break;
        }
        case 2:
        {
            val = ((s % 100) / 10);
            break;
        }
        case 3:
        {
            val = (s % 10);
            break;
        }
    }

    segment_write(val, i);

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}


void main(void)
{
  unsigned char set_time = 0;
  Init_Device();

  while(1)
  {
      if(SET == 0)
      {
         delay_ms(40);
         while(SET == 0);
```

```
         set_time = 1;
      }

      if(set_time == 1)
      {
          if(INC == 0)
          {
             delay_ms(40);
             s++;

             if(s >= 9999)
             {
                 s = 9999;
             }
          }

          if(DEC == 0)
          {
             delay_ms(40);
             s--;

             if(s <= 0)
             {
                 s = 0;
             }
          }

          if(ESC == 0)
          {
             TR2_bit = 1;
             set_time = 0;
          }
      }
  };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}


void Timer_Init(void)
{
    TMR2RLL = 0x02;
    TMR2RLH = 0xFC;
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}


void Port_IO_Init(void)
{
    // P0.0  -  Unassigned,  Open-Drain, Digital
    // P0.1  -  Unassigned,  Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital
```

```
    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Skipped,     Open-Drain, Digital
    // P1.2  -  Skipped,     Open-Drain, Digital
    // P1.3  -  Skipped,     Open-Drain, Digital
    // P1.4  -  Skipped,     Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P1MDOUT = 0xE0;
    P1SKIP = 0xFE;
    XBR1 = 0x40;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
{
    IE = 0xA0;
    EIE1 = 0x80;
}


void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
    LED_LATCH = 1;
    LED_CLK = 1;
    LED_DOUT = 0;
}


void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}
```

```
void segment_write(unsigned char disp, unsigned char pos)
{
    LED_LATCH = 0;
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}
```

## Schematic




## Explanation

In this demo, two timers – Timer 2 and 3 have been used to make a simple countdown timer. Timer 3 is used for multiplexing the onboard seven-segment displays while Timer 2 is used as the actual countdown timer.

First, let's see how the multiplexing is achieved. With reference to the schematic, the seven-segment displays are driven with a pair of 74HC595 serial-in-parallel-out (SIPO) shift registers. The shift registers are cascaded in series. This means that the final output of the first shift register is connected to the data input of the second shift register while clock and latch pins are shorted. With this configuration, we get a 16-bit wide shift register because the first shift register would shift out data to the second register. Using shift registers in this way allows us to expand three outputs from C8051F330D to literally unlimited number of outputs. Here we just need 16 outputs.

```
void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
```

```
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}
```

74HC595 SIPO registers take data serially in and the input data is shifted when there is a low to high clock transition. One shift register can hold up to 8 bits or one byte of data and so eight clock transitions are needed. The above function does this SIPO part.

Now that we know how to transfer data through the shift registers, we can now focus on how the seven segments are lit. The first eight bits are used to select which seven-segment out of the four to be lit and the second eight bits following it write the code for the LEDs of the selected segment. After sending these two bytes, the shift registers are latched by raising their latch pins high from low, meaning that all outputs of the shift registers are updated.

```
#define LED_DOUT            P1_6_bit
#define LED_CLK             P1_5_bit
#define LED_LATCH           P1_7_bit

....

const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x7F, // .
  0xBF  // -
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};

....

void segment_write(unsigned char disp, unsigned char pos)
{
    LED_LATCH = 0;
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}
```

The above code just writes one seven-segment display at a time but we have four seven-segment displays. All of them need to be updated in such a way that all of them appear to be displayed simultaneously at the same time. Therefore, they need to be updated at a fast refresh rate and in such a way that while updating them no other tasks are blocked. To achieve this, we need a timer with periodic interrupts. Here Timer 3 is used with about a 1ms periodic interrupt interval. The following math shows how to find the ideal timer auto-reload value to achieve the desired timer interrupt interval. Remember that Timer 3 has an auto-reload feature and so we just have to put the calculated value in our code once.

$$\boldsymbol{Timer\ Frequency} = \frac{\boldsymbol{System\ Clock}}{\boldsymbol{Prescalar}} = \frac{\boldsymbol{12.25\ MHz}}{\boldsymbol{12}} = \boldsymbol{1.02083\ MHz}$$

$$\boldsymbol{Timer\ Tick} = \frac{\boldsymbol{1}}{\boldsymbol{Timer\ Frequency}} = \frac{\boldsymbol{1}}{\boldsymbol{1.02083\ MHz}} = \boldsymbol{0.98\mu s} \approx \boldsymbol{1\mu s}$$

$$\boldsymbol{TimerLoadValue} = \boldsymbol{TimerTopValue} - \left(\frac{\boldsymbol{Required\ Interval}}{\boldsymbol{Timer\ Tick}}\right)$$

$$\boldsymbol{TimerLoadValue} = \boldsymbol{65535} - \left(\frac{\boldsymbol{1000\mu s}}{\boldsymbol{0.98\mu s}}\right) = \boldsymbol{64514} = \boldsymbol{0xFC02}$$

The code below shows the timer, system clock and interrupt settings generated by the *Configuration Wizard 2* code generator.

```
void Timer_Init(void)
{
    TMR2RLL = 0x02;
    TMR2RLH = 0xFC;
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
{
    IE = 0xA0;
    EIE1 = 0x80;
}
```

Inside Timer 3 interrupt, the value to be displayed by the seven segments are decoded bytewise and position-wise and the respective seven segment displays are updated. This process is repeated periodically every 1ms.

```
void Timer_3_ISR(void)
iv IVT_ADDR_ET3
ilevel 0
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (s / 1000);
            break;
        }
        case 1:
        {
            val = ((s % 1000) / 100);
            break;
        }
        case 2:
        {
            val = ((s % 100) / 10);
            break;
        }
        case 3:
        {
            val = (s % 10);
            break;
        }
    }

    segment_write(val, i);

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}
```

Timer 2 is set just like Timer 3 and for 1ms interrupt interval but unlike Timer 3, it is not left to run by default. This is so because we want the timer to run after we have set the countdown time in seconds. In the main loop, four onboard push buttons are used to set count down time and start the timer.

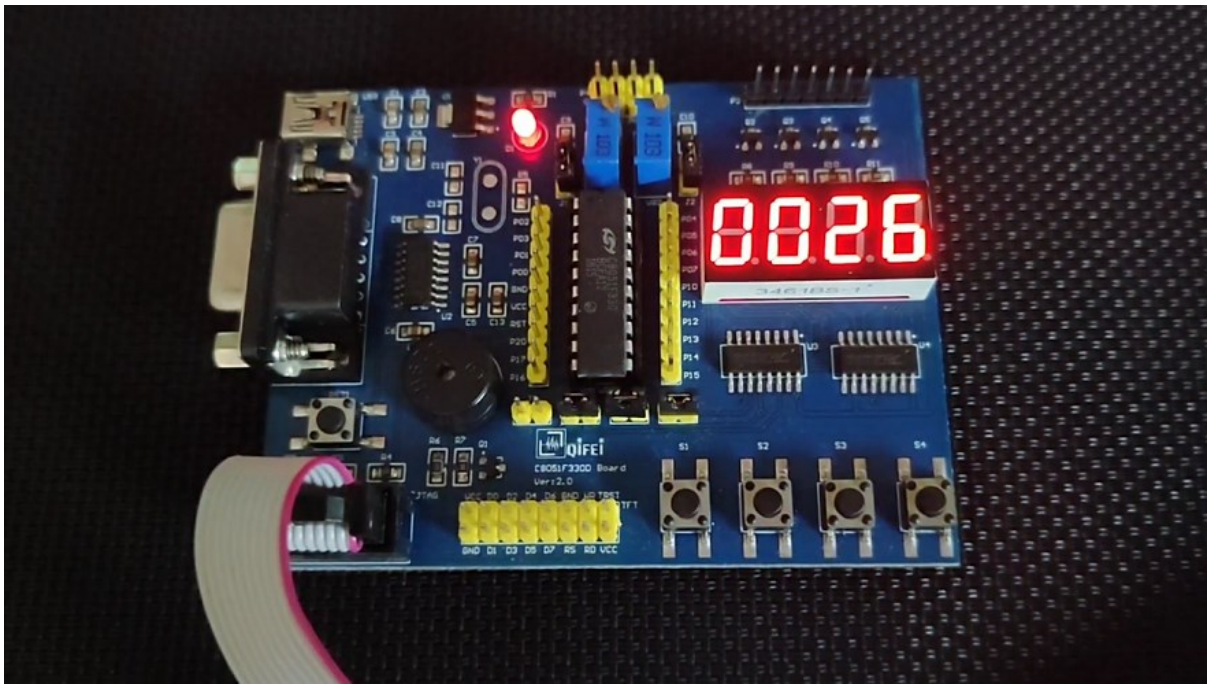
```
while(1)
{
    if(SET == 0)
    {
       delay_ms(40);
       while(SET == 0);

       set_time = 1;
    }

    if(set_time == 1)
    {
        if(INC == 0)
        {
           delay_ms(40);
           s++;

           if(s >= 9999)
           {
```

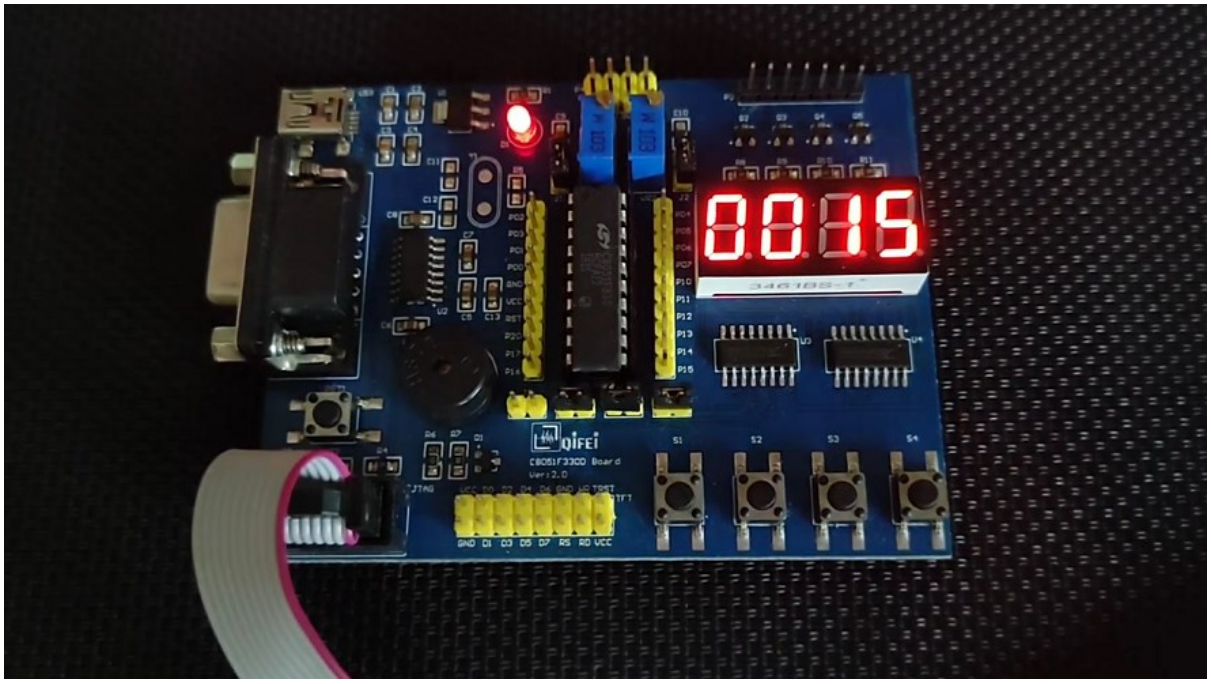
```
                s = 9999;
            }
        }

        if(DEC == 0)
        {
            delay_ms(40);
            s--;

            if(s <= 0)
            {
                s = 0;
            }
        }

        if(ESC == 0)
        {
            TR2_bit = 1;
            set_time = 0;
        }
    }
};
```

Inside the interrupt subroutine of Timer 2, a pair of variables keep track of milliseconds and seconds. Upon every 1000ms (1s) count, the countdown second is decremented. The timer keeps running till the second count has reached zero. When the second count has reached zero the timer is stopped by clearing its timer run flag bit.

```
void Timer_2_ISR(void)
iv IVT_ADDR_ET2
ilevel 1
ics ICS_AUTO
{
    ms++;
    if(ms > 999)
    {
        ms = 0;
        s--;

        if(s < 0)
        {
            s = 0;
            TR2_bit = 0;
        }
    }

    TMR2CN &= 0x7F;
}
```

## Demo

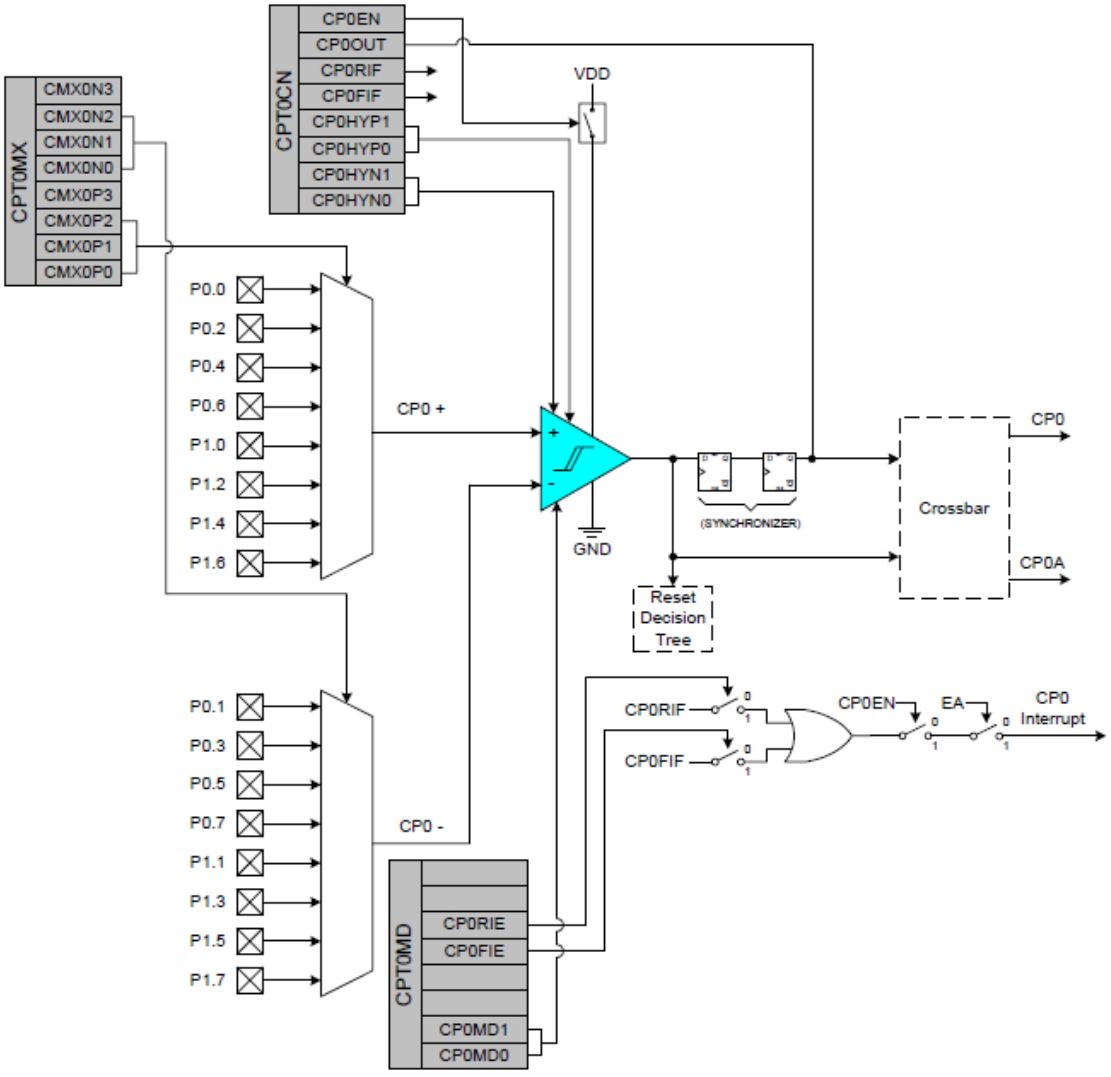



Demo video link: https://youtu.be/rU22K8fVgS0

# Analogue Comparator (CP0) – Capacitance Meter

C8051s come with analogue comparator hardware embedded with them. This comparator can be used to measure voltage levels and take decisions based on voltage levels. Analogue comparators can be used as low battery detectors, voltage threshold detectors, oscillators, etc. The block diagram of C8051F330D's comparator 0 (CP0) is shown below. From the block diagram above, we can see that several inputs can be used. Additionally, there are options for interrupt, hysteresis, synchronizer, output and others.



## Code

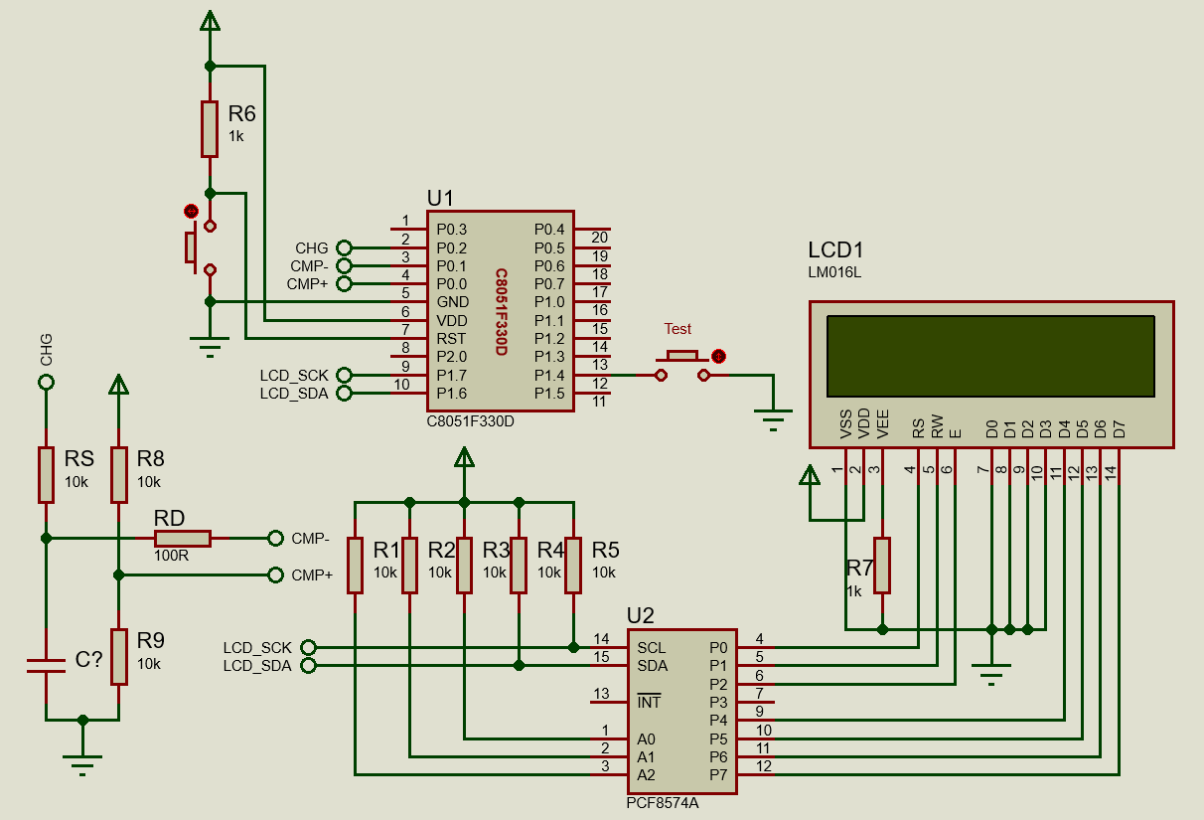
```
#include "LCD_2_Wire.c"
#include "lcd_print.c"

#define measure_button     P1_4_bit

#define sampling_R         10000.0
#define div_factor         (sampling_R * 0.693147)

#define us_per_tick        0.9795918
#define scale_factor       50


unsigned char ovf = 0;
unsigned char measurement_done = 0;


void PCA_Init(void);
void Timer_Init(void);
void Comparator_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void reset_timer_1(void);
unsigned int get_timer_1(void);


void Timer_1_ISR(void)
iv IVT_ADDR_ET1
ilevel 0
ics ICS_AUTO
{
    ovf++;
    TF1_bit = 0;
}


void Analog_Comparator_ISR(void)
iv IVT_ADDR_ECP0
ilevel 0
ics ICS_AUTO
{
    if(CP0FIF_bit)
    {
        measurement_done = 1;
        TCON = 0x00;
        P0_2_bit = 0;
        CP0FIF_bit = 0;
    }
}


void main(void)
{
    float c = 0.0;
    unsigned long cnt = 0;

    Init_Device();
    LCD_init();
    LCD_clear_home();

    LCD_goto(0, 0);
```

```
    LCD_putstr("Capacitance/ F:");

    while(1)
    {
        if((measure_button == 0) && (measurement_done == 0))
        {
            reset_timer_1();
            LCD_goto(0, 1);
            LCD_putstr("Discharging!  ");
            P0MDIN = 0xFE;
            P0MDOUT = 0x06;
            P0_1_bit = 0;
            P0_2_bit = 0;
            delay_ms(4000);
            P0MDIN = 0xFC;
            P0MDOUT = 0x04;
            TCON = 0x40;
            P0_2_bit = 1;
            LCD_goto(0, 1);
            LCD_putstr("Measuring!    ");
            delay_ms(4000);
        }

        if(measurement_done)
        {
           cnt = ovf;
           cnt <<= 16;
           cnt += get_timer_1();
           c = ((cnt * us_per_tick) / div_factor);
           c *= scale_factor;

           LCD_goto(0, 1);
           LCD_putstr("             ");

           if((c > 0.0) && (c < 10000000.0))
           {
               if((c > 0) && (c < 1000))
               {
                   LCD_goto(12, 0);
                   LCD_putstr("n");
               }
               else if((c >= 1000.0) && (c < 10000000.0))
               {
                   c /= 1000.0;
                   LCD_goto(12, 0);
                   LCD_putstr("u");
               }

               print_F(0, 1, c, 1);
           }

           else
           {
               LCD_goto(0, 1);
               LCD_putstr("O.L");
           }

           measurement_done = 0;
        }
    };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
```

```
    PCA0MD = 0x00;
}


void Timer_Init(void)
{
    TMOD = 0x10;
}


void Comparator_Init(void)
{
    CPT0CN = 0x8A;
    delay_us(20);
    CPT0CN &= ~0x30;
    CPT0MX = 0x00;
    CPT0MD = 0x11;
}


void Port_IO_Init(void)
{
    // P0.0  -  Skipped,     Open-Drain, Analog
    // P0.1  -  Skipped,     Open-Drain, Analog
    // P0.2  -  Skipped,     Push-Pull,  Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Skipped,     Open-Drain, Digital
    // P1.5  -  Unassigned,  Open-Drain, Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDIN = 0xFC;
    P0MDOUT = 0x04;
    P1MDOUT = 0xC0;
    P0SKIP = 0x07;
    P1SKIP = 0xD0;
    XBR1 = 0xC0;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
{
    IE = 0x88;
    EIE1 = 0x20;
}


void Init_Device(void)
{
    PCA_Init();
```

```
    Comparator_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}


void reset_timer_1(void)
{
    ovf = 0;
    TH1 = 0x00;
    TL1 = 0x00;
}


unsigned int get_timer_1(void)
{
    unsigned int counts = 0;

    counts = TH1;
    counts <<= 8;
    counts |= TL1;

    return counts;
}
```

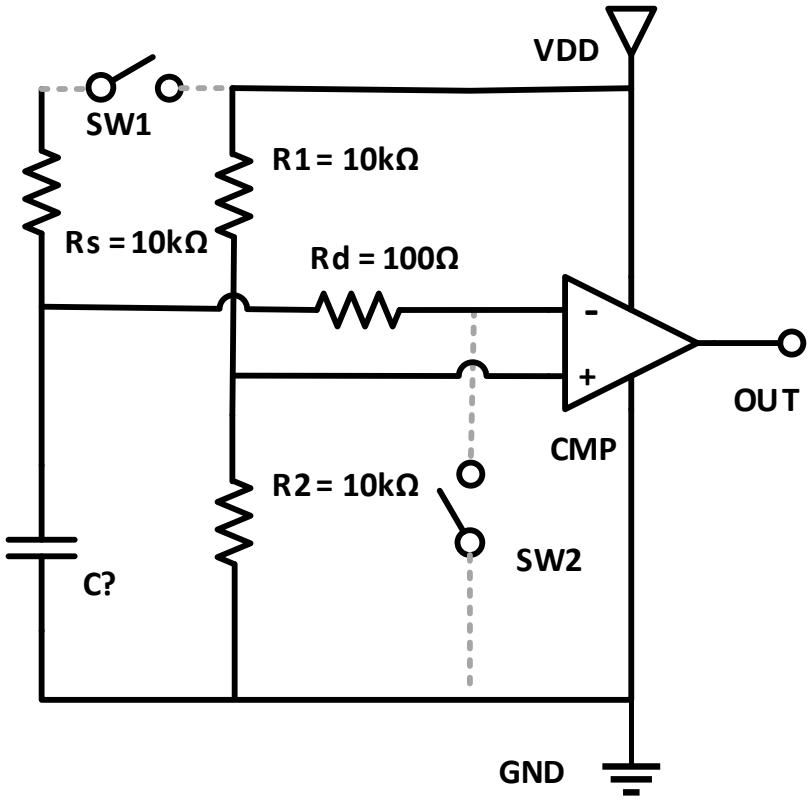
## Schematic



## Explanation

In this section, we will see how we can use the internal analogue comparator of a C8051 to measure unknown capacitance values of capacitors. The principle used is based on determining the time it takes to charge an unknown capacitor to a certain voltage level. Thus, two things are needed to be done in the program – firstly, comparing two voltage levels and secondly, timing the time taken for the comparator to change logic state.

The basic circuit looks like the one shown below:




Before trying to measure the unknown capacitor ***C?***, it is discharged through the current-limiting resistor ***Rd*** by closing switch ***SW2***. SW2 is not a physical switch. The working of SW2 is done inside the microcontroller. The inverting input pin of the internal analogue comparator is temporarily set as an output and then it is set to logic zero for 4 seconds. The capacitor to be measured starts to discharge. Discharging the capacitor ensures that no residual charge is present in it as such charge will definitely affect reading. After discharging SW2 is opened by declaring it as an input.

After discharging, switch ***SW1*** is closed and an internal timer is started. SW1 is just like SW2 and is not an external pin but rather an I/O pin. The non-inverting pin of the comparator is set to half VDD by the voltage divider formed by resistors ***R1*** and ***R2***. Thus, when SW1 is closed with the unknown capacitor fully discharged, logic high state is observed at the output side of the comparator. The unknown capacitor continues to charge via the sampling resistor ***Rs***. When the voltage of the capacitor exceeds a value more than half VDD, the output state of the comparator is flipped low and SW1 is opened. Thus, a high-low transition is observed. When this transition occurs, the timer is stopped, a comparator interrupt is issued and some calculations are performed to back calculate the capacitance of the unknown capacitor.

Now let us see the calculations.

We know:

$$V_c = V_{DD}(1 - e^{-\frac{t}{RC}})$$

Since the capacitor is charged to half VDD, Vc becomes:

$$V_c = 0.5 \times V_{DD}$$

Therefore, the first equation simplifies as follows:

$$0.5 \times V_{DD} = V_{DD}(1 - e^{-\frac{t}{RC}})$$

$$0.5 = 1 - e^{-\frac{t}{RC}}$$

$$e^{-\frac{t}{RC}} = 0.5$$

$$ln(e^{-\frac{t}{RC}}) = ln(0.5)$$

$$\frac{-t}{RC} = -0.693147$$

$$C = \frac{t}{0.693147 \times R}$$

Since Rs = 10kΩ

$$C = \frac{t}{6931.47}$$

P0.0 and P0.1 are set as the open-drain analogue pins since these pins are comparator input pins. P0.2 is the capacitor charging pin or SW1. P1.4 is attached to an onboard push-button that will be used to trigger the measurement process when pressed.

```
void Port_IO_Init(void)
{
    // P0.0  -  Skipped,     Open-Drain, Analog
    // P0.1  -  Skipped,     Open-Drain, Analog
    // P0.2  -  Skipped,     Push-Pull,  Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Skipped,     Open-Drain, Digital
    // P1.5  -  Unassigned,  Open-Drain, Digital
```

```
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDIN = 0xFC;
    P0MDOUT = 0x04;
    P1MDOUT = 0xC0;
    P0SKIP = 0x07;
    P1SKIP = 0xD0;
    XBR1 = 0xC0;
}
```

The comparator is set as follows.

```
void Comparator_Init(void)
{
    CPT0CN = 0x8A;
    delay_us(20);
    CPT0CN &= ~0x30;
    CPT0MX = 0x00;
    CPT0MD = 0x11;
}
```

With these settings, the comparator is set with 10mV hysteresis on both inputs, fast response time and falling-edge comparator interrupt.

The system clock is set to 12.25MHz using the internal RC oscillator.

```
void Oscillator_Init(void)
{
    OSCICN = 0x82;
}
```

The timer used for timing in this example is Timer 1. It is set as a Mode 1 timer with a prescalar of 12 and with overflow interrupt enabled. However, it is not started in the beginning. It will be started during measurement. In this way, one tick of the timer is about 979ns.

```
void Timer_Init(void)
{
    TMOD = 0x10;
}
```

$$\boldsymbol{Timer\ Frequency = \frac{System\ Clock}{Prescalar} = \frac{12.25\ MHz}{12} = 1.02083\ MHz}$$

$$\boldsymbol{Timer\ Tick = \frac{1}{Timer\ Frequency} = \frac{1}{1.02083\ MHz} = 979ns}$$

The main loop has two parts. The first part is about discharging and charging the unknown capacitor. When the push-button is pressed, the timer is reset and the discharge procedure is started by setting the proper logic states of the pin as mentioned earlier. After completing the discharge procedure, the I/O pins are reconfigured and the timer is started as soon as the charging process is initiated.

```
if((measure_button == 0) && (measurement_done == 0))
{
    reset_timer_1();
    LCD_goto(0, 1);
    LCD_putstr("Discharging!  ");
    P0MDIN = 0xFE;
    P0MDOUT = 0x06;
    P0_1_bit = 0;
    P0_2_bit = 0;
    delay_ms(4000);
    P0MDIN = 0xFC;
    P0MDOUT = 0x04;
    TCON = 0x40;
    P0_2_bit = 1;
    LCD_goto(0, 1);
    LCD_putstr("Measuring!    ");
    delay_ms(4000);
}
```

After completing the aforementioned steps, the code waits for the comparator interrupt.

```
void Analog_Comparator_ISR(void)
iv IVT_ADDR_ECP0
ilevel 0
ics ICS_AUTO
{
    if(CP0FIF_bit)
    {
        measurement_done = 1;
        TCON = 0x00;
        P0_2_bit = 0;
        CP0FIF_bit = 0;
    }
}
```

When the comparator interrupt is triggered, the timer and the charging process are both stopped.

We have already seen what math is needed to convert timer count to capacitance value and this second part of the main code is all about so.

```
if(measurement_done)
{
    cnt = ovf;
    cnt <<= 16;
    cnt += get_timer_1();
    c = ((cnt * us_per_tick) / div_factor);
    c *= scale_factor;

    LCD_goto(0, 1);
    LCD_putstr("             ");

    if((c > 0.0) && (c < 10000000.0))
    {
         if((c > 0) && (c < 1000))
         {
              LCD_goto(12, 0);
              LCD_putstr("n");
         }

         else if((c >= 1000.0) && (c < 10000000.0))
         {
             c /= 1000.0;
```

```
            LCD_goto(12, 0);
            LCD_putstr("u");
        }

        print_F(0, 1, c, 1);
    }

    else
    {
        LCD_goto(0, 1);
        LCD_putstr("O.L");
    }

    measurement_done = 0;
}
```

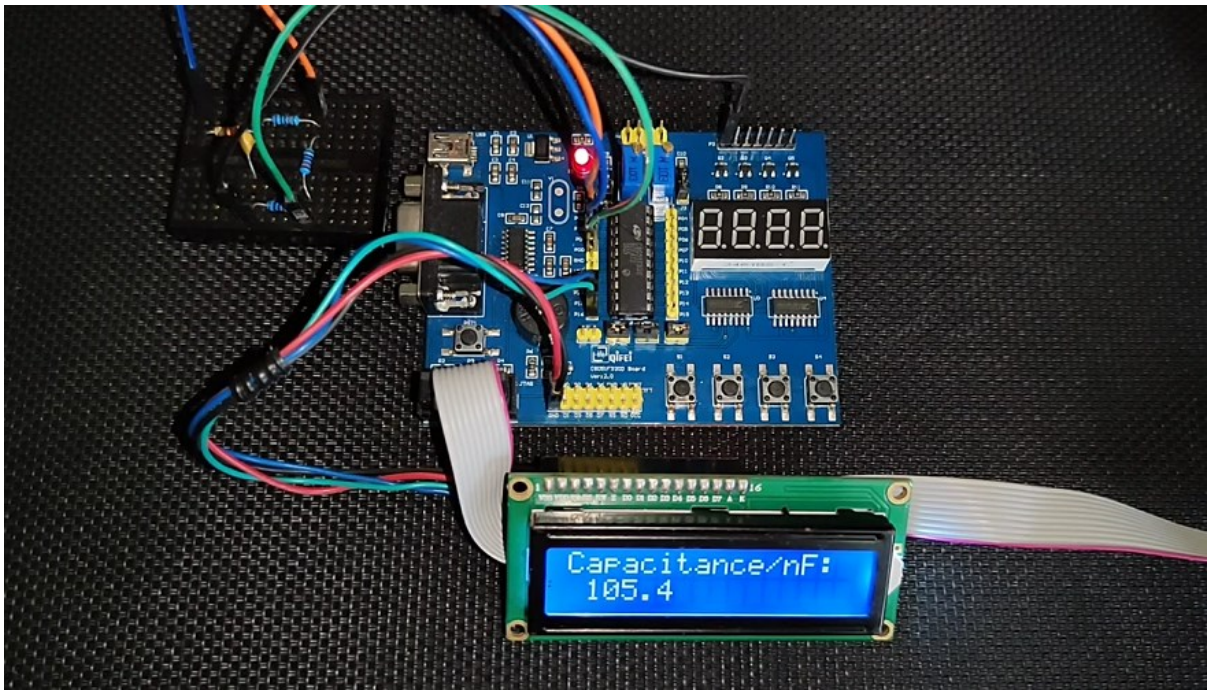
Timer 1 interrupt is used to check if the timer overflowed during measurement. This should only be taken into account when measuring large-value capacitors.

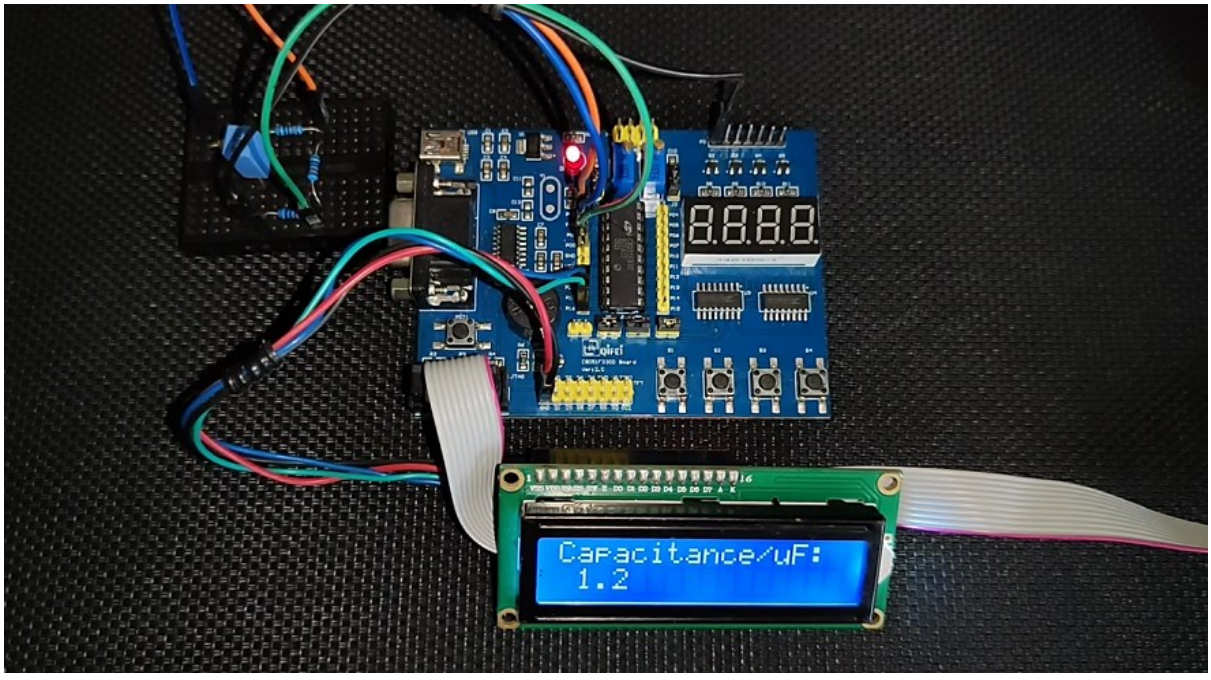
```
void Timer_1_ISR(void)
iv IVT_ADDR_ET1
ilevel 0
ics ICS_AUTO
{
    ovf++;
    TF1_bit = 0;
}
```

Unlike most other examples covered in this document, an I2C LCD is used to show data. The code for the I2C LCD has been shared and it is the same as the ones I used with my past documents on various platforms.

The overall performance of such a capacitance meter is satisfactory and the results are accurate enough for most cases.

## Demo

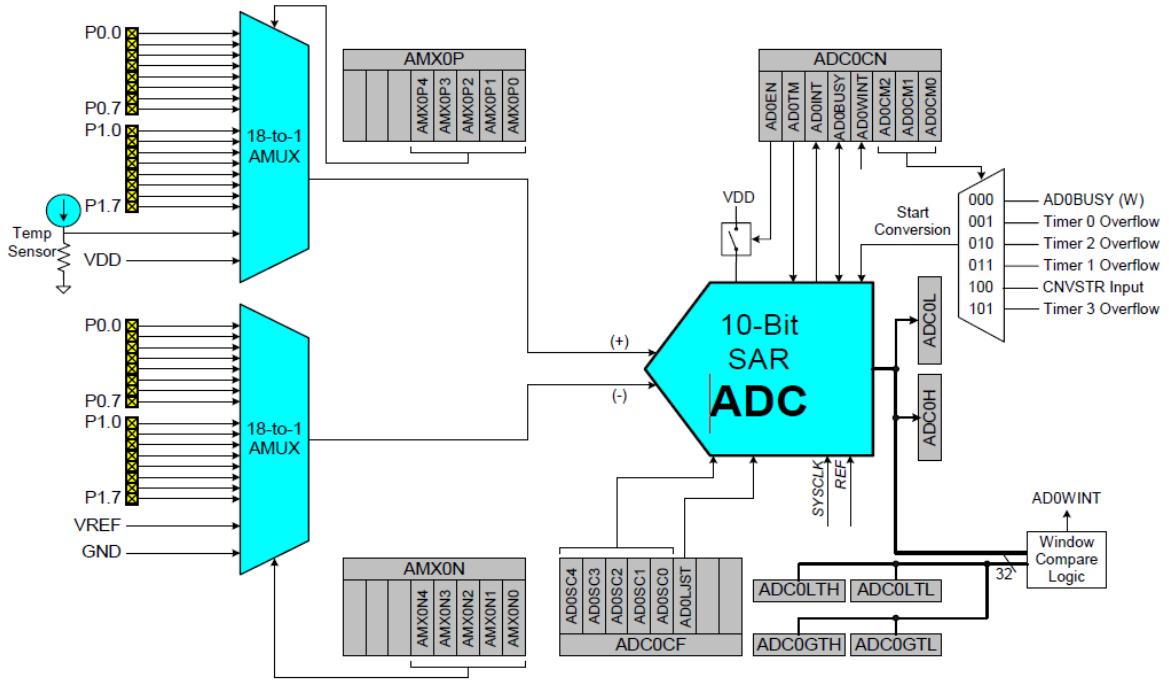



Demo video link: https://youtu.be/C-Gg1HH_d3s

# Analogue-to-Digital Converter (ADC) – Analogue Thermometer

An Analogue-to-Digital Converter (ADC) is a must in any modern-era microcontroller and fortunately, C8051s come with 10-bit 200kbps differential successive-approximation register (SAR) ADCs, unlike traditional ones which are fully digital devices. It is quite advanced in terms of features. In the case of C8051F330D, all pins can be used with it. Two internal analogue multiplexers take care of the pin selection task. Additionally, there is an internal temperature sensor, an internal reference source, an analogue window comparator and tons of other kinds of stuff. The ADC can be triggered by several sources and it also supports interrupts. However, the best features of this ADC are the support for differential inputs and its SAR-type build-up. SAR ADCs are fast compared to other ADC types. The block diagram of the internal ADC of C8051F330D is shown below:




## Code

```
#define LED_DOUT  P1_6_bit
#define LED_CLK   P1_5_bit
#define LED_LATCH P1_7_bit

#define ADC_res    1023.0
#define VDD_mv     3300.0


unsigned char i = 0;
unsigned char pt = 0;
register unsigned char val = 0;
unsigned int value = 0;


const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
```

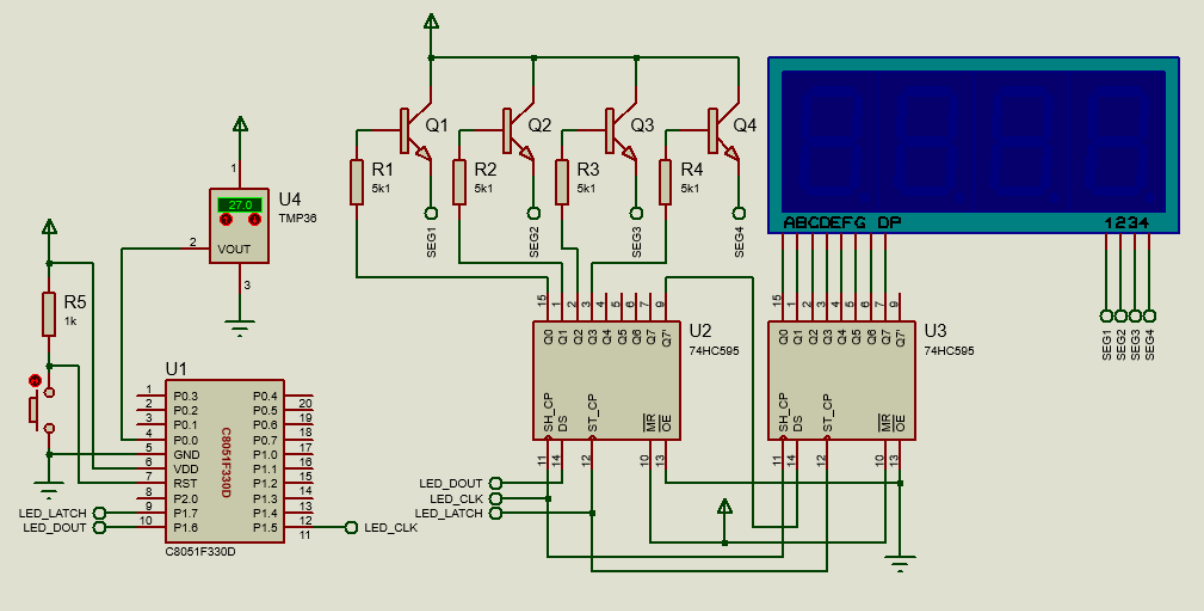
```
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x7F, // .
  0xBF  // -
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


void PCA_Init(void);
void Timer_Init();
void Port_IO_Init();
void Oscillator_Init();
void Interrupts_Init();
void ADC_Init(void);
void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos, unsigned char point);
unsigned int adc_read(void);
void Voltage_Reference_Init(void);
unsigned int adc_avg(unsigned char channel);


void Timer_ISR(void)
iv IVT_ADDR_ET3
ilevel 0
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (value / 1000);
            pt = 0;
            break;
        }
        case 1:
        {
            val = ((value % 1000) / 100);
            pt = 1;
            break;
        }
        case 2:
        {
            val = ((value % 100) / 10);
            pt = 0;
            break;
        }
        case 3:
        {
            val = (value % 10);
            pt = 0;
            break;
        }
```

```
    }

    segment_write(val, i, pt);

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}


void main(void)
{
  float t = 0;

  Init_Device();

  while(1)
  {
     t = (adc_avg(0) * VDD_mv);
     t /= ADC_res;
     value = ((t * 10) - 5000);
     delay_ms(100);
  };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}


void Timer_Init()
{
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}


void Port_IO_Init()
{
    // P0.0  -  Skipped,     Open-Drain, Analog
    // P0.1  -  Unassigned,  Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital
```

```
    P0MDIN = 0xFE;
    P1MDOUT = 0xE0;
    P0SKIP = 0x01;
    P1SKIP = 0xE0;
    XBR1 = 0x40;
}


void Oscillator_Init()
{
    OSCLCN = 0x82;
}


void Interrupts_Init()
{
    IE = 0x80;
    EIE1 = 0x80;
}


void ADC_Init(void)
{
    AMX0P = 0x00;
    AMX0N = 0x11;
    ADC0CF = 0x58;
    ADC0CN = 0x80;
}


void Voltage_Reference_Init(void)
{
    REF0CN = 0x0A;
}


void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
    ADC_Init();
    Voltage_Reference_Init();
}


void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
```

```
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}


void segment_write(unsigned char disp, unsigned char pos, unsigned char point)
{
    unsigned char write_value = segment_code[disp];

    if(point)
    {
        write_value &= segment_code[10];
    }

    LED_LATCH = 0;
    write_74HC595(write_value);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}


unsigned int adc_read(void)
{
   unsigned int ad_value = 0;

   ad_value = ADC0H;
   ad_value <<= 8;
   ad_value |= ADC0L;

   return ad_value;
}


unsigned int adc_avg(unsigned char channel)
{
    unsigned int avg_value = 0;
    signed char samples = 16;

    AMX0P = (channel & 0x1F);
    delay_ms(1);

    while(samples > 0)
    {
        AD0INT_bit = 0;
        AD0BUSY_bit = 1;

        while(AD0INT_bit == 0);
        avg_value += adc_read();

        samples--;
    };

    avg_value >>= 4;

    return avg_value;
}
```

## Schematic




## Explanation

TMP36 is a very popular analogue temperature sensor apart from LM35. It gives voltage output that is proportional to temperature and has an accuracy of 10mV/°C. With an ADC we can read this sensor and determine the temperature it is sensing.

Firstly, we have to declare the pin that we will be using with the ADC. In this example, P0.0 pin is used and so it is set as open-drain analogue input.

```
void Port_IO_Init()
{
    // P0.0  -  Skipped,     Open-Drain, Analog
    // P0.1  -  Unassigned,  Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDIN = 0xFE;
    P1MDOUT = 0xE0;
    P0SKIP = 0x01;
    P1SKIP = 0xE0;
    XBR1 = 0x40;
}
```

Like most of the examples, the system clock is derived from the internal RC oscillator and is set to 12.25MHz.

```
void Oscillator_Init()
{
    OSCLCN = 0x82;
}
```

Note that the ADC is differential in terms of input. Since we are using only one channel, i.e., the positive input of the ADC, the negative input has to be tied to the ground. The ADC clock is set to 1MHz which is good enough for most applications considering Nyquist theory. Software-based trigger-polling method is used. ADC readings are set to be right-justified. These all are set as per the ADC initialization code shown below:

```
void ADC_Init(void)
{
    AMX0P = 0x00;
    AMX0N = 0x11;
    ADC0CF = 0x58;
    ADC0CN = 0x80;
}
```

All ADC measurements have to be done with respect to a reference voltage source and in this example, VDD is used as the reference source. Internal analogue peripherals need to be biased by the internal bias generator. All these are part of the internal reference source peripheral and so these settings are set by tweaking the internal reference source configuration register.

```
void Voltage_Reference_Init(void)
{
    REF0CN = 0x0A;
}
```

The ADC can be read using the following function. It lets the user select a single channel and take an average of 16 samples. The ADC is first triggered by a software trigger and then the ADC conversion completion flag is polled. After polling, the ADC is read. The ADC readings are summed and stored. After acquiring 16 samples, the summed reading is right-shifted by a factor of 4. This shifting is just like dividing by 16 because $2^4$ = 16.

```
unsigned int adc_avg(unsigned char channel)
{
    unsigned int avg_value = 0;
    signed char samples = 16;

    AMX0P = (channel & 0x1F);
    delay_ms(1);

    while(samples > 0)
    {
        AD0INT_bit = 0;
        AD0BUSY_bit = 1;

        while(AD0INT_bit == 0);
        avg_value += adc_read();

        samples--;
    };
```

```
    avg_value >>= 4;

    return avg_value;
}
```

Since the ADC reading is right-justified, the 10-bit ADC reading is done as follows:

```
unsigned int adc_read(void)
{
   unsigned int ad_value = 0;

   ad_value = ADC0H;
   ad_value <<= 8;
   ad_value |= ADC0L;

   return ad_value;
}
```

The high byte is read first and then the low byte is read.

Inside the main loop, the average ADC reading is obtained and the ADC count is converted to millivolts. The millivolt value is then converted to temperature.

```
void main(void)
{
  float t = 0;

  Init_Device();

  while(1)
  {
     t = (adc_avg(0) * VDD_mv);
     t /= ADC_res;
     value = ((t * 10) - 5000);
     delay_ms(100);
  };
}
```

The math behind these steps in as follows.

First, we need to determine the voltage output from the sensor. This is done using the formula below:

$$\boldsymbol{ADC\ Reading\ in\ mV} = \frac{\boldsymbol{Average\ ADC\ Count} \times \boldsymbol{Reference\ Voltage\ in\ mV}}{\boldsymbol{Max\ ADC\ Count}}$$

If we check the datasheet of the TMP36 sensor, we can see the sensor output voltage vs temperature relation along with the output scaling voltage and offset voltage values.

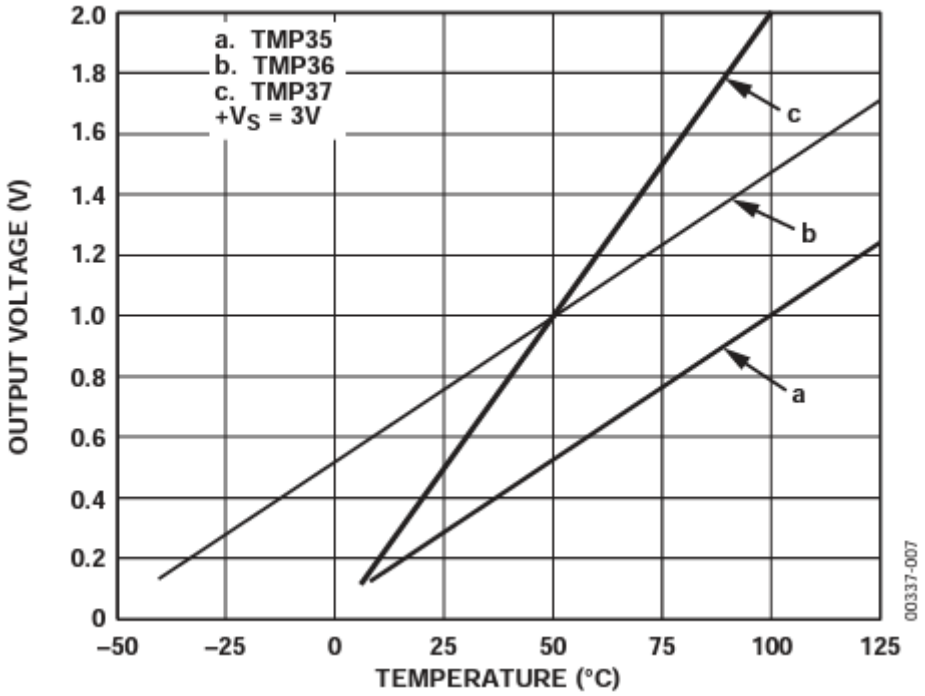



**Table 4. TMP35/TMP36/TMP37 Output Characteristics**

| Sensor | Offset Voltage (V) | Output Voltage Scaling (mV/°C) | Output Voltage at 25°C (mV) |
|---|---|---|---|
| TMP35 | 0 | 10 | 250 |
| TMP36 | 0.5 | 10 | 750 |
| TMP37 | 0 | 20 | 500 |

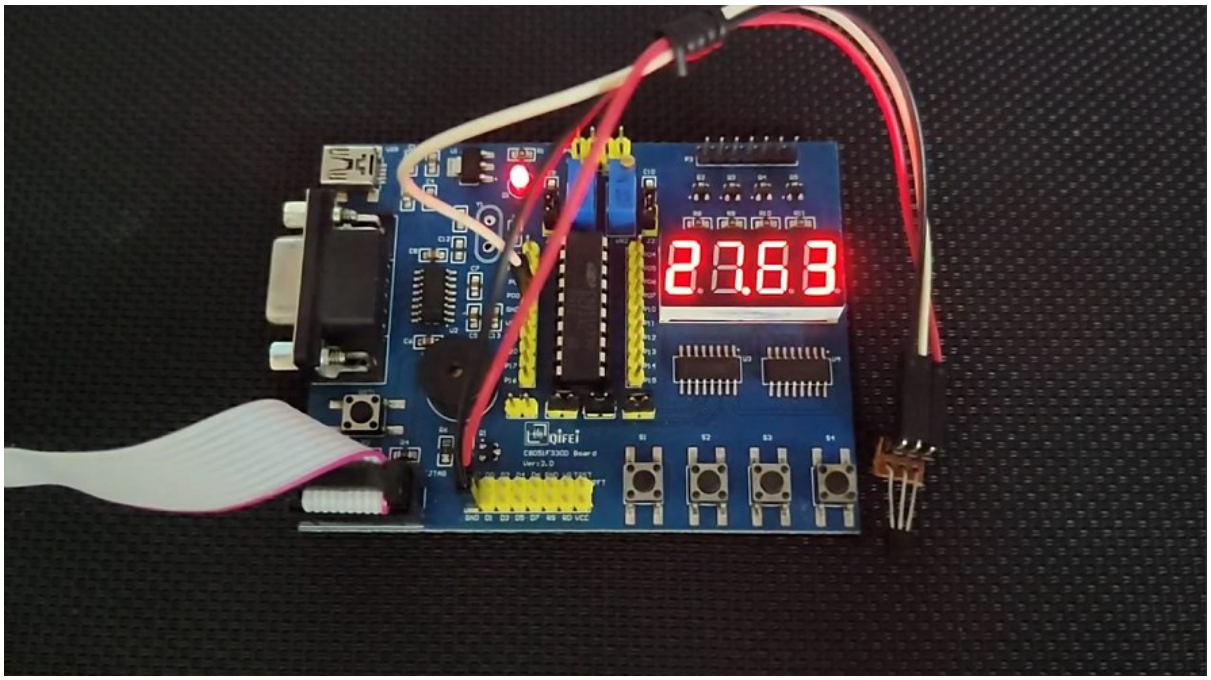
Using the graph and the info above, we can determine the following temperature versus voltage relation:

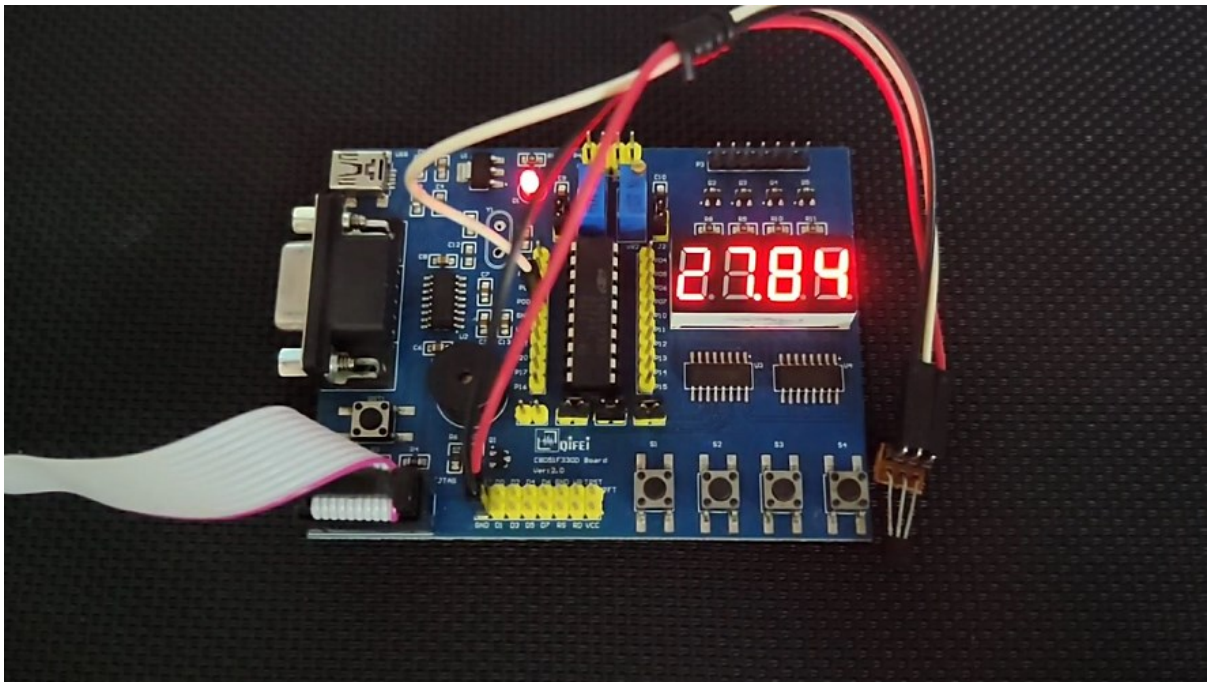
$$\boldsymbol{Temperature} = (\boldsymbol{Output\ Voltage} \times \mathbf{0.1}) - \mathbf{50}$$

To display the temperature on the onboard seven-segment display, the above equation is scaled by a factor of 100 and so it changes to the following:

$$\boldsymbol{Scaled\ Temperature} = (\boldsymbol{Output\ Voltage} \times \mathbf{10}) - \mathbf{5000}$$

## Demo

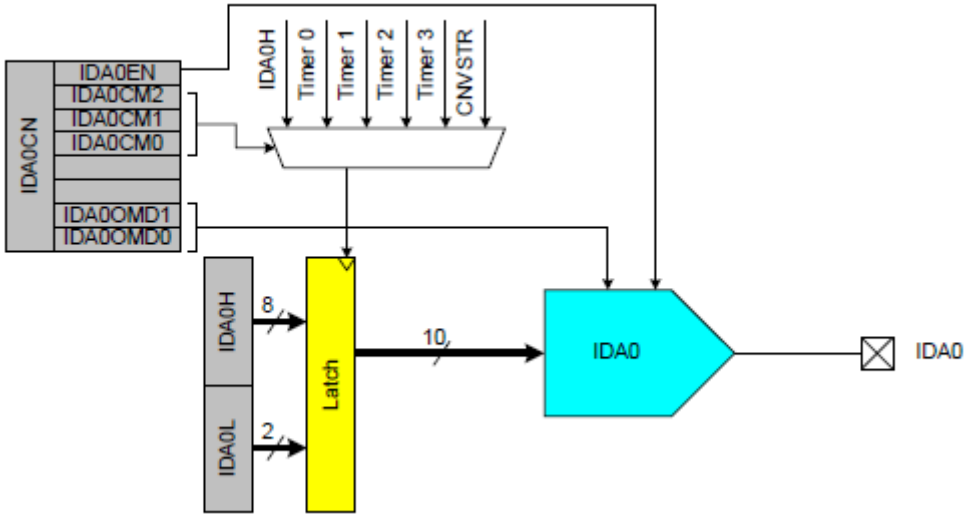




Demo video link: https://youtu.be/IIkpdQ5EISQ

# Digital-to-Analogue Converter (IDA0) – Digital Signal Synthesizer

Digital-to-Analogue Converter (DAC) is just the converse hardware with respect to the ADC. With the exception of ARM microcontrollers and some modern microcontrollers such as the ATXMega series, DAC is not present in common devices and we have to use external DAC chips like the MCP4921 if we need one in a project. Fortunately, C8051F330D has one 10-bit DAC which can be used for a number of applications including audio generation, waveform generation, etc. Unlike most DACs which provide voltage outputs, the DAC present in C8051F330D provides current output instead of voltage output. With a suitable load resistor, we can convert the output current from the DAC to voltage. The block diagram below shows the contents of the DAC. The DAC can be triggered by a number of sources including all timers. These trigger sources are used to update DAC output. Two registers hold the 10-bit digital count that would be converted to current. The digital count is latched upon trigger. Through register coding, it is also possible to set the maximum full-scale current which can be 0.5mA, 1.0mA or 2.0mA.




## Code

```
#define LED_DOUT          P1_6_bit
#define LED_CLK           P1_5_bit
#define LED_LATCH         P1_7_bit

#define MODE_SW           P1_3_bit
#define INC_SW            P1_2_bit
#define DEC_SW            P1_1_bit


unsigned char i = 0;
register unsigned char val = 0;
unsigned int value = 0;


const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
```

```
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x7F, // .
  0xBF  // -
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


const unsigned int code LUT_sine[32] =
{
    512,
    615,
    714,
    804,
    882,
    946,
    991,
    1017,
    1023,
    1007,
    971,
    916,
    845,
    760,
    665,
    564,
    460,
    359,
    264,
    179,
    108,
    53,
    17,
    2,
    7,
    33,
    78,
    142,
    220,
    310,
    409,
    512
};


const unsigned int code LUT_triangle[32] =
{
    512,
    576,
    640,
    704,
    768,
    832,
```

```
    896,
    960,
    1023,
    960,
    896,
    832,
    768,
    704,
    640,
    576,
    512,
    448,
    384,
    320,
    256,
    192,
    128,
    64,
    0,
    64,
    128,
    192,
    256,
    320,
    384,
    448
};


const unsigned int code LUT_square[32] =
{
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023
};
```

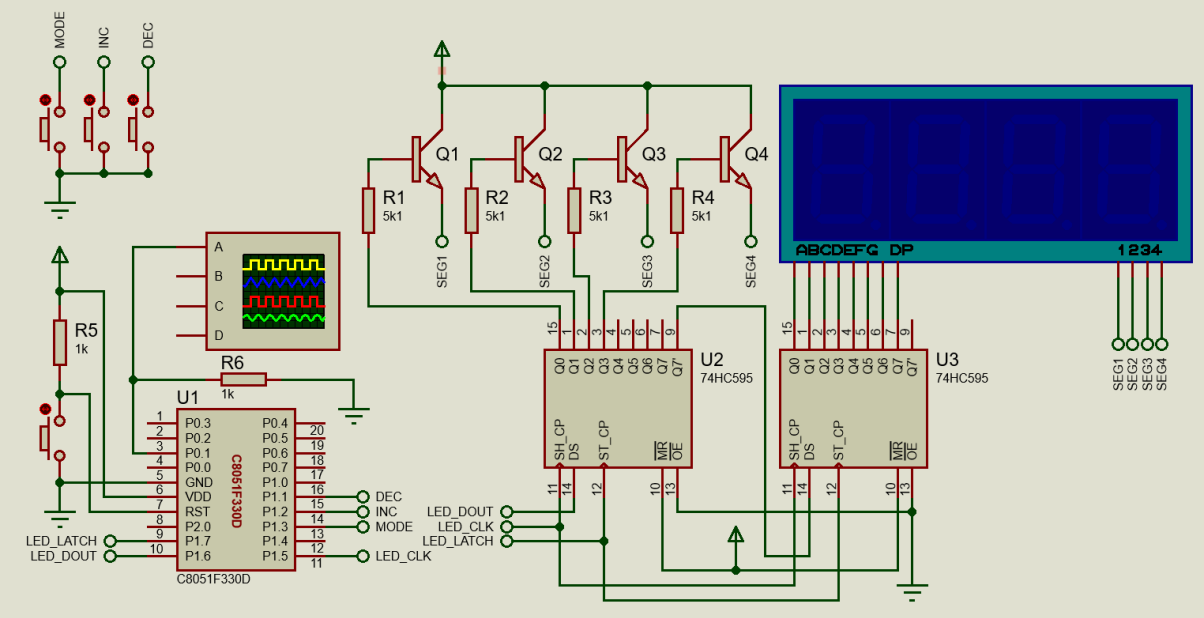
```
void PCA_Init(void);
void Timer_Init(void);
void DAC_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);
void DAC_write(unsigned int dac_value);


void Timer_ISR(void)
iv IVT_ADDR_ET3
ilevel 0
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (value / 1000);
            break;
        }
        case 1:
        {
            val = ((value % 1000) / 100);
            break;
        }
        case 2:
        {
            val = ((value % 100) / 10);
            break;
        }
        case 3:
        {
            val = (value % 10);
            break;
        }
    }

    segment_write(val, i);

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}


void main(void)
{
    signed char l = 0;
    unsigned int j = 0;
    signed int dly = 0;
    unsigned int dac_data = 0;
    unsigned char mode = 0;

    Init_Device();

    while(1)
```

```
{
    if(MODE_SW == 0)
    {
        delay_ms(30);
        l = 0;
        dac_data = 0;
        mode++;
    }

    if(mode >= 3)
    {
        mode = 0;
    }

    if(INC_SW == 0)
    {
        delay_ms(40);
        dly++;
    }

    if(dly > 9999)
    {
        dly = 0;
    }

    if(DEC_SW == 0)
    {
        delay_ms(40);
        dly--;
    }

    if(dly < 0)
    {
        dly = 9999;
    }

    value = dly;

    switch(mode)
    {
        case 1:
        {
            dac_data = LUT_triangle[l];
            break;
        }

        case 2:
        {
            dac_data = LUT_square[l];
            break;
        }

        default:
        {
            dac_data = LUT_sine[l];
            break;
        }
    }

    l++;
    if(l >= 32)
    {
       l = 0;
    }

    DAC_write(dac_data);
```

```
        for(j = 0; j < dly; j++)
        {
            delay_us(1);
        }
    };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}


void Timer_Init(void)
{
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}


void DAC_Init(void)
{
    IDA0CN = 0xF2;
}


void Port_IO_Init(void)
{
    // P0.0  -  Unassigned,  Open-Drain, Digital
    // P0.1  -  Skipped,     Open-Drain, Analog
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Skipped,     Open-Drain, Digital
    // P1.2  -  Skipped,     Open-Drain, Digital
    // P1.3  -  Skipped,     Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDIN = 0xFD;
    P1MDOUT = 0xE0;
    P0SKIP = 0x02;
    P1SKIP = 0xEE;
    XBR1 = 0x40;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
```

```
{
    IE = 0x80;
    EIE1 = 0x80;
}


void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    DAC_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}


void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}


void segment_write(unsigned char disp, unsigned char pos)
{
    LED_LATCH = 0;
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}


void DAC_write(unsigned int dac_value)
{
    unsigned char lb = 0x00;
    unsigned char hb = 0x00;

    dac_value <<= 0x06;

    lb = (dac_value & 0xC0);
    hb = ((dac_value & 0xFF00) >> 0x08);

    IDA0L = lb;
    IDA0H = hb;
}
```

## Schematic




## Explanation

In this example, the internal DAC is used to generate waveforms digitally – sinusoidal, triangular and square waves. The result is a Digital Signal Synthesizer (DSS).

Firstly, the system clock is set to 12.25MHz and is derived from internal oscillator.

```
void Oscillator_Init(void)
{
    OSCICN = 0x82;
}
```

C8051's DAC is set up just by setting the *IDA0CN* register. We need to enable the DAC, define its full-scale current value, and the source that would issue a new write sequence and the output pin.

```
void DAC_Init(void)
{
    IDA0CN = 0xF2;
}
```

The DAC takes 10-bit integer inputs that are separated by two registers and so we need a function to load these registers. This is done by the following function. Notice how the two registers are arranged and loaded.

| IDA0H | | | | | | | | IDA0L | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | | | | | | |

```
void DAC_write(unsigned int dac_value)
{
    unsigned char lb = 0x00;
    unsigned char hb = 0x00;
```

```
    dac_value <<= 0x06;

    lb = (dac_value & 0xC0);
    hb = ((dac_value & 0xFF00) >> 0x08);

    IDA0L = lb;
    IDA0H = hb;
}
```

Three lookup table arrays for three types of waveforms hold the 10-bit values that would be put into the DAC input register. Alternatively, equations could have been used but that would require mathematical computations and would slow down the wave-generation process.

```
const unsigned int code LUT_sine[32] =
{
    512,
    615,
    714,
    804,
    882,
    946,
    991,
    1017,
    1023,
    1007,
    971,
    916,
    845,
    760,
    665,
    564,
    460,
    359,
    264,
    179,
    108,
    53,
    17,
    2,
    7,
    33,
    78,
    142,
    220,
    310,
    409,
    512
};


const unsigned int code LUT_triangle[32] =
{
    512,
    576,
    640,
    704,
    768,
    832,
    896,
    960,
    1023,
    960,
    896,
```

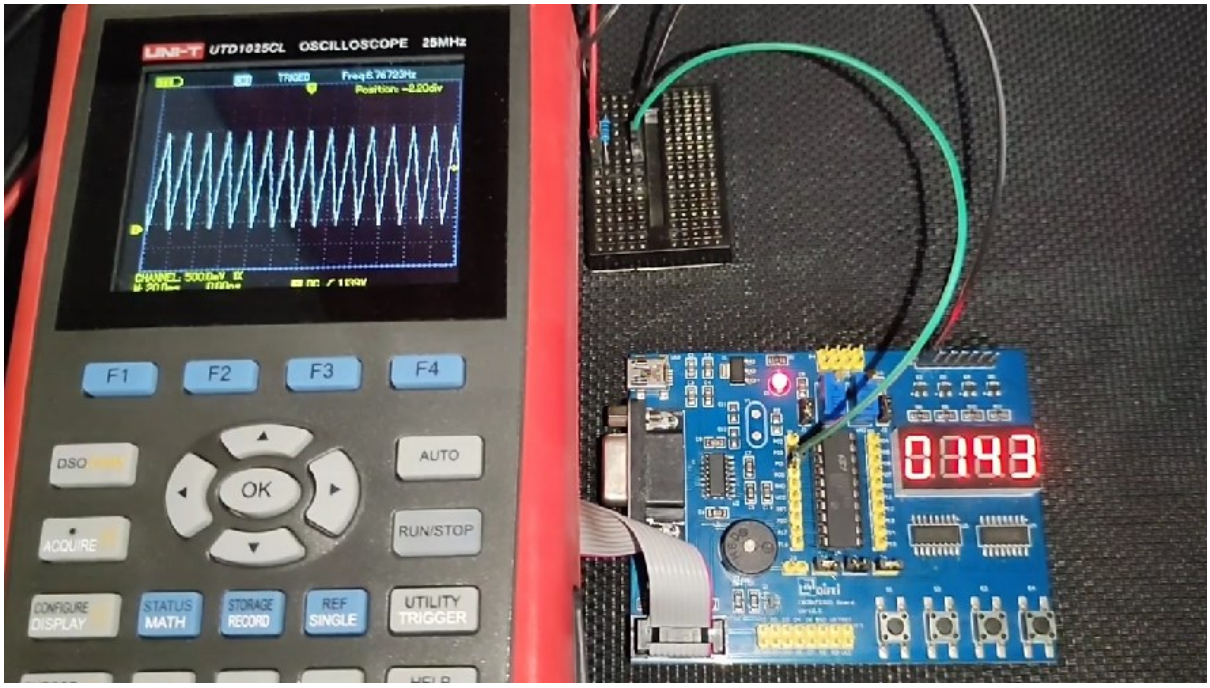
```
    832,
    768,
    704,
    640,
    576,
    512,
    448,
    384,
    320,
    256,
    192,
    128,
    64,
    0,
    64,
    128,
    192,
    256,
    320,
    384,
    448
};


const unsigned int code LUT_square[32] =
{
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023,
    1023
};
```

Three buttons are used to control the DSS. Pressing the button attached to P1.3 switches waveform types while pressing the buttons connected to P1.1 and P1.2 changes the frequency of the output waveform. This is the task inside the main loop. The onboard display gives an indication of period.

```
void main(void)
{
    signed char l = 0;
    unsigned int j = 0;
    signed int dly = 0;
    unsigned int dac_data = 0;
    unsigned char mode = 0;

    Init_Device();

    while(1)
    {
        if(MODE_SW == 0)
        {
            delay_ms(30);
            l = 0;
            dac_data = 0;
            mode++;
        }

        if(mode >= 3)
        {
            mode = 0;
        }

        if(INC_SW == 0)
        {
            delay_ms(40);
            dly++;
        }

        if(dly > 9999)
        {
            dly = 0;
        }

        if(DEC_SW == 0)
        {
            delay_ms(40);
            dly--;
        }

        if(dly < 0)
        {
            dly = 9999;
        }

        value = dly;

        switch(mode)
        {
            case 1:
            {
                dac_data = LUT_triangle[l];
                break;
            }

            case 2:
            {
                dac_data = LUT_square[l];
                break;
            }

            default:
            {
                dac_data = LUT_sine[l];
```

```
                break;
            }
        }

        l++;
        if(l >= 32)
        {
           l = 0;
        }

        DAC_write(dac_data);

        for(j = 0; j < dly; j++)
        {
            delay_us(1);
        }
    };
}
```

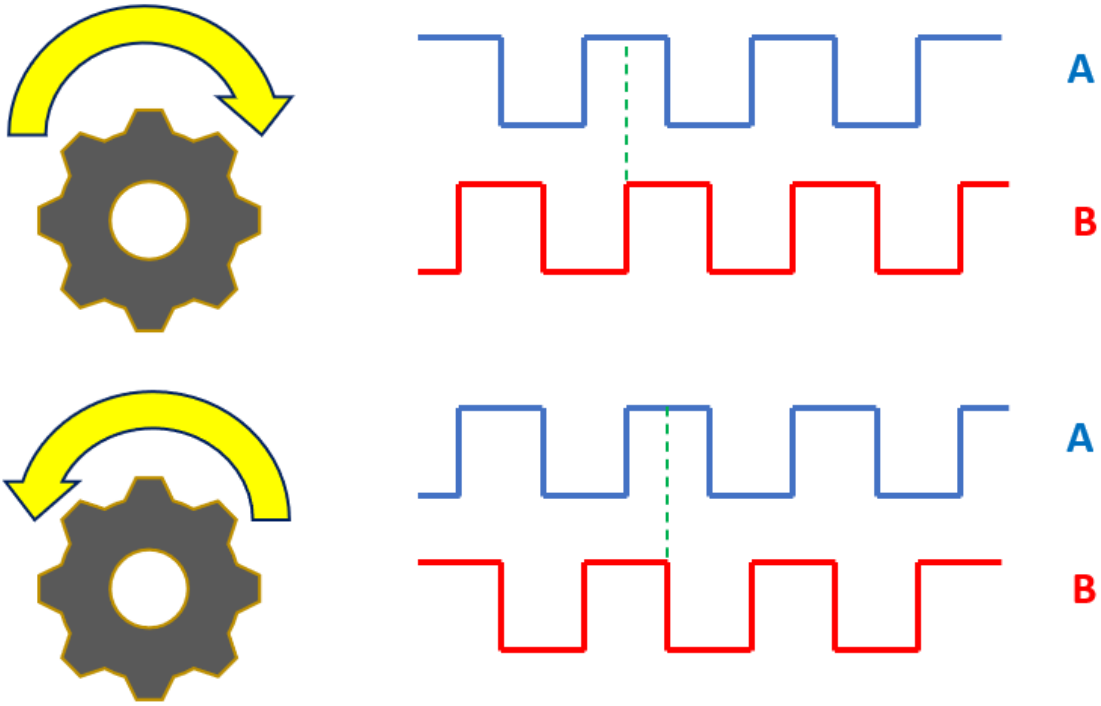
## Demo




Demo video link: https://youtu.be/uiA3NKKdNno

## External Interrupt (EXTI) – Rotary Encoder

External interrupts (EXTI) are usually not needed unless there are tasks related to GPIO pins that require very high attention. One such task is reading a rotary encoder. Usually, a rotary encoder is not always rotated unless a change is needed. Thus, we can avoid polling its inputs in code and use an external interrupt to monitor for changes.




Most rotary encoders have two outputs. They are labelled as *A* and *B* or as *Data* and *Clock*. *A* and *B* have different phase shifts as shown in the diagram above. Let us consider a rotary encode as above. In this situation, we can attach one pin (let's say *A*) to an external interrupt of a C8051 and poll the other pin (let's say *B*) inside the interrupt for its previous and present state. In this way, we can determine if the rotary encoder is rotating in a clockwise or anticlockwise direction.

### Code

```
#define LED_DOUT            P1_6_bit
#define LED_CLK             P1_5_bit
#define LED_LATCH           P1_7_bit

#define ROT_ENC_DATA        P0_0_bit
#define ROT_ENC_CLK         P0_1_bit
#define ROT_ENC_SW          P0_2_bit


unsigned char i = 0;
unsigned char val = 0;
unsigned char past_clock_state = 0;
unsigned char present_clock_state = 0;
unsigned char data_state = 0;
signed int value = 0;
unsigned int step_size = 1;
```

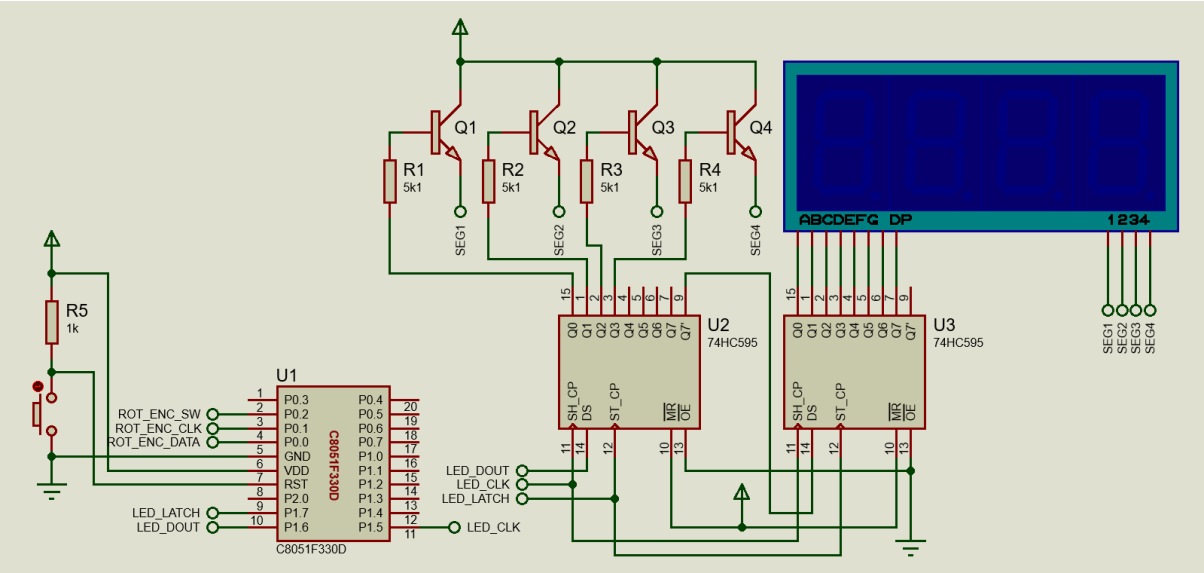
```
const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x7F, // .
  0xBF  // -
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


void PCA_Init(void);
void Timer_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);


void EXTI_0_ISR(void)
iv IVT_ADDR_EX0
ilevel 0
ics ICS_AUTO
{
    present_clock_state = ROT_ENC_CLK;
    data_state = ROT_ENC_DATA;

    if(present_clock_state != past_clock_state)
    {
        if(data_state != present_clock_state)
        {
            value -= step_size;
        }
        else
        {
            value += step_size;
        }

        past_clock_state = present_clock_state;
    }

    if(value > 9999)
    {
        value = 0;
    }
```

```
    if(value < 0)
    {
        value = 9999;
    }
}


void Timer_3_ISR(void)
iv IVT_ADDR_ET3
ilevel 1
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (value / 1000);
            break;
        }
        case 1:
        {
            val = ((value % 1000) / 100);
            break;
        }
        case 2:
        {
            val = ((value % 100) / 10);
            break;
        }
        case 3:
        {
            val = (value % 10);
            break;
        }
    }

    segment_write(val, i);

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}



void main(void)
{
    Init_Device();

    while(1)
    {
        if(ROT_ENC_SW == 0)
        {
            delay_ms(60);
            while(ROT_ENC_SW == 0);
            step_size *= 10;

            if(step_size > 1000)
            {
                step_size = 1;
```

```
            }
        }
    };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}

void Timer_Init(void)
{
    TCON = 0x01;
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}

void Port_IO_Init(void)
{
    // P0.0  -  Skipped,     Open-Drain, Digital
    // P0.1  -  Skipped,     Open-Drain, Digital
    // P0.2  -  Skipped,     Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P1MDOUT = 0xE0;
    P0SKIP = 0x07;
    P1SKIP = 0xE0;
    XBR1 = 0x40;
}

void Oscillator_Init(void)
{
    OSCICN = 0x82;
}

void Interrupts_Init(void)
{
    IE = 0x81;
    IP = 0x01;
    EIE1 = 0x80;
}
void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}
```

```
void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}

void segment_write(unsigned char disp, unsigned char pos)
{
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 0;
    LED_LATCH = 1;
}
```

## Schematic

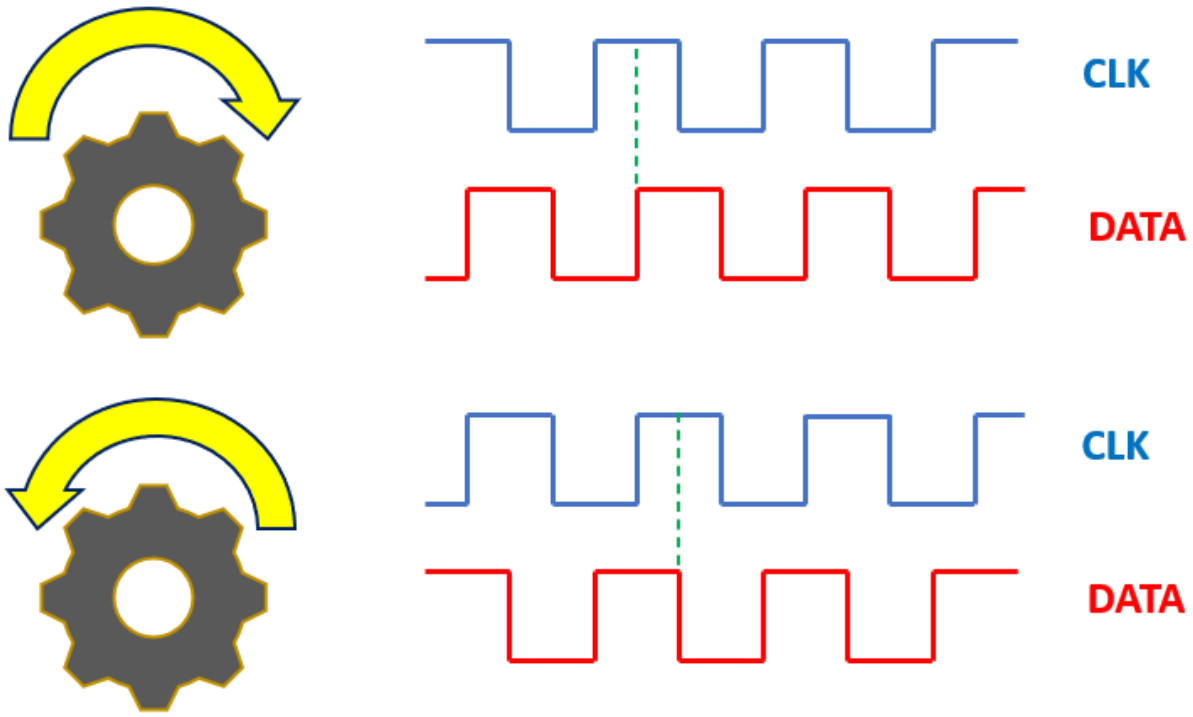



## Explanation

Unlike most other examples, this example is not clock dependent. All we need are an external interrupt and a couple of GPIO pins as inputs.

```
#define ROT_ENC_DATA        P0_0_bit
#define ROT_ENC_CLK         P0_1_bit
#define ROT_ENC_SW          P0_2_bit
```

Now let's see the theory. Suppose the clock pin is attached to the interrupt pin.



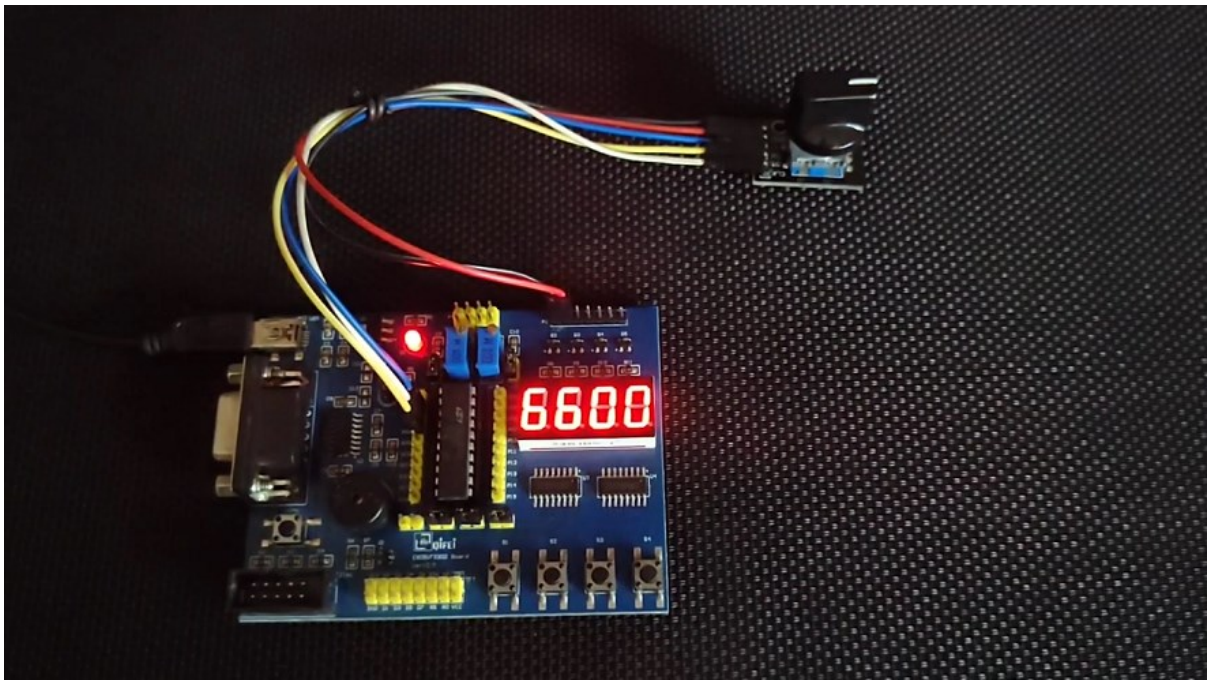

As soon as an interrupt is triggered, both the data and the clock pins are polled. If there is a change in the logic state of the clock input then the state of the data pin is compared against the current state of the clock pin. This comparison tells us the direction of the rotary encoder. Based on direction a variable named *value* is either increment or decremented.

```
void EXTI_0_ISR(void)
iv IVT_ADDR_EX0
ilevel 0
ics ICS_AUTO
{
    present_clock_state = ROT_ENC_CLK;
    data_state = ROT_ENC_DATA;

    if(present_clock_state != past_clock_state)
    {
        if(data_state != present_clock_state)
        {
            value -= step_size;
        }
        else
        {
            value += step_size;
        }
        past_clock_state = present_clock_state;
    }
    if(value > 9999)
    {
        value = 0;
    }
    if(value < 0)
    {
        value = 9999;
    }
}
```

The step size of the change is determined in the main.

```
if(ROT_ENC_SW == 0)
{
    delay_ms(60);
    while(ROT_ENC_SW == 0);
    step_size *= 10;

    if(step_size > 1000)
    {
         step_size = 1;
    }
}
```

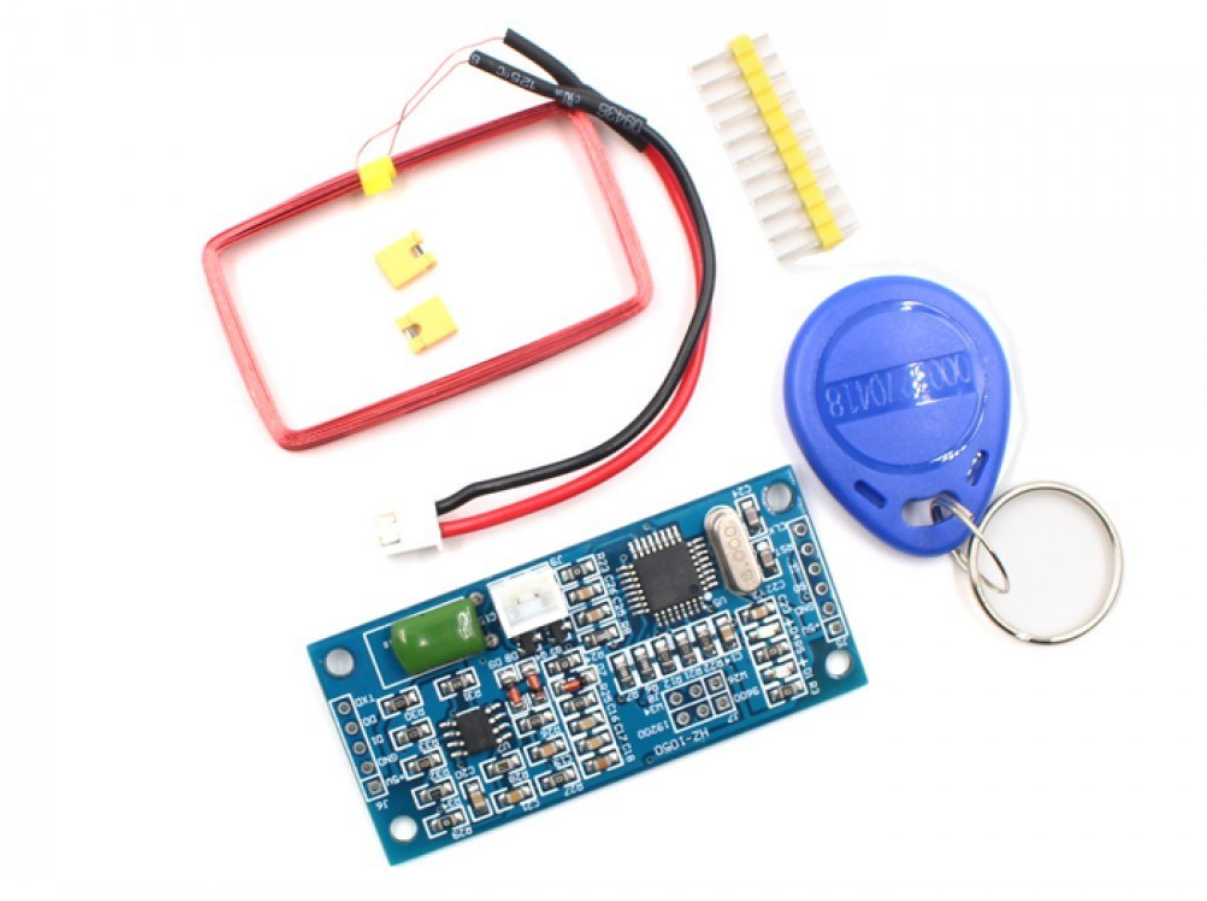
## Demo




Demo video link: https://youtu.be/B-plZxG3h2s

## External Interrupt (EXTI) – RFID Card Reader

C8051F330D microcontroller does not have pin-change interrupt capability but like many other 8051 microcontrollers, it sports two external interrupts – EXTI0 and EXTI1. In combination, these interrupts can be used for a number of applications. Here I used these interrupts for decoding a Wiegand 26 RFID card reader.



To know more about Weigand 26, RFIDs and their working principle be sure to check the following links:

- https://en.wikipedia.org/wiki/Wiegand_interface
- https://getsafeandsound.com/blog/26-bit-wiegand-format
- https://www.securityinfowatch.com/access-identity/access-control/article/10537302/understanding-26bit-wiegand
- https://www.techtarget.com/iotagenda/definition/RFID-radio-frequency-identification
- https://en.wikipedia.org/wiki/Access_control
- https://en.wikipedia.org/wiki/Radio-frequency_identification

## Code

### Waveshare_RGB_LCD.h

```
sbit Soft_I2C_Scl at P1_7_bit;
sbit Soft_I2C_Sda at P1_6_bit;


#define LCD_I2C_address             0x7C
#define RGB_I2C_address             0xC0


/*    Colour Definitions    */
#define REG_RED                     0x04
#define REG_GREEN                   0x03
#define REG_BLUE                    0x02
#define REG_MODE_1                  0x00
#define REG_MODE_2                  0x01
#define REG_OUTPUT                  0x08

/*    LCD Definitions    */
#define LCD_CLEAR_DISPLAY           0x01
#define LCD_RETURN_HOME             0x02
#define LCD_ENTRY_MODE_SET          0x04
#define LCD_DISPLAY_CONTROL         0x08
#define LCD_CURSOR_SHIFT            0x10
#define LCD_FUNCTION_SET            0x20
#define LCD_SET_CGRAM_ADDR          0x40
#define LCD_SET_DDRAM_ADDR          0x80

/*    flags for display entry mode    */
#define LCD_ENTRY_RIGHT             0x00
#define LCD_ENTRY_LEFT              0x02
#define LCD_ENTRY_SHIFT_INCREMENT   0x01
#define LCD_ENTRY_SHIFT_DECREMENT   0x00

/*    flags for display on/off control    */
#define LCD_DISPLAY_ON              0x04
#define LCD_DISPLAY_OFF             0x00
#define LCD_CURSOR_ON               0x02
#define LCD_CURSOR_OFF              0x00
#define LCD_BLINK_ON                0x01
#define LCD_BLINK_OFF               0x00

/*    flags for display/cursor shift    */
#define LCD_DISPLAY_MOVE            0x08
#define LCD_CURSOR_MOVE             0x00
#define LCD_MOVE_RIGHT              0x04
#define LCD_MOVE_LEFT               0x00

/*    flags for function set    */
#define LCD_8_BIT_MODE              0x10
#define LCD_4_BIT_MODE              0x00
#define LCD_2_LINE                  0x08
#define LCD_1_LINE                  0x00
#define LCD_5x8_DOTS                0x00

/*   Data / Command selection   */
#define DAT                         0x40
#define CMD                         0x80


void RGB_LCD_init(void);
```

```
void I2C_bus_write(unsigned char address, unsigned char value_1, unsigned char value_2);
void LCD_I2C_write(unsigned char value, unsigned char mode);
void RGB_I2C_write(unsigned char reg, unsigned char value);
void set_RGB(unsigned char R, unsigned char G, unsigned char B);
void LCD_goto(unsigned char x_pos, unsigned char y_pos);
void LCD_clear_home(void);
void LCD_putchar(char chr);
void LCD_putstr(char *lcd_string);
```

## Waveshare_RGB_LCD.c

```
#include "Waveshare_RGB_LCD.h"


void RGB_LCD_init(void)
{
    Soft_I2C_Init();
    delay_ms(20);

    LCD_I2C_write((LCD_FUNCTION_SET | LCD_2_LINE), CMD);
    delay_ms(5);

    LCD_I2C_write((LCD_FUNCTION_SET | LCD_2_LINE), CMD);
    delay_ms(5);

    LCD_I2C_write((LCD_FUNCTION_SET | LCD_2_LINE), CMD);
    delay_ms(5);

    LCD_I2C_write((LCD_FUNCTION_SET | LCD_2_LINE), CMD);
    LCD_I2C_write((LCD_DISPLAY_CONTROL | LCD_DISPLAY_ON | LCD_CURSOR_OFF | LCD_BLINK_OFF),
CMD);

    LCD_clear_home();

    LCD_I2C_write((LCD_ENTRY_MODE_SET | LCD_ENTRY_LEFT | LCD_ENTRY_SHIFT_DECREMENT), CMD);

    RGB_I2C_write(REG_MODE_1, 0x00);
    RGB_I2C_write(REG_OUTPUT, 0xFF);
    RGB_I2C_write(REG_MODE_2, 0x20);

    set_RGB(127, 127, 127);
}


void I2C_bus_write(unsigned char address, unsigned char value_1, unsigned char value_2)
{
    Soft_I2C_Start();
    Soft_I2C_Write(address);
    Soft_I2C_Write(value_1);
    Soft_I2C_Write(value_2);
    Soft_I2C_Stop();
}


void LCD_I2C_write(unsigned char value, unsigned char mode)
{
    I2C_bus_write(LCD_I2C_address, mode, value);
}


void RGB_I2C_write(unsigned char reg, unsigned char value)
{
    I2C_bus_write(RGB_I2C_address, reg, value);
}
```

```
void set_RGB(unsigned char R, unsigned char G, unsigned char B)
{
    RGB_I2C_write(REG_RED, R);
    RGB_I2C_write(REG_GREEN, G);
    RGB_I2C_write(REG_BLUE, B);
}


void LCD_goto(unsigned char x_pos, unsigned char y_pos)
{
    if(y_pos == 0)
    {
        x_pos |= 0x80;
    }
    else
    {
        x_pos |= 0xC0;
    }

    I2C_bus_write(LCD_I2C_address, 0x80, x_pos);
}


void LCD_clear_home(void)
{
    LCD_I2C_write(LCD_CLEAR_DISPLAY, CMD);
    LCD_I2C_write(LCD_RETURN_HOME, CMD);
    delay_ms(2);
}


void LCD_putchar(char chr)
{
    if((chr >= 0x20) && (chr <= 0x7F))
    {
        LCD_I2C_write(chr, DAT);
    }
}


void LCD_putstr(char *lcd_string)
{
    do
    {
        LCD_putchar(*lcd_string++);
    }while(*lcd_string != '\0') ;
}
```

## lcd_print_rgb.h

```
#define no_of_custom_symbol     1
#define array_size_per_symbol   8
#define array_size              (array_size_per_symbol * no_of_custom_symbol)

void load_custom_symbol(void);
void print_symbol(unsigned char x_pos, unsigned char y_pos, unsigned char symbol_index);
void print_C(unsigned char x_pos, unsigned char y_pos, signed int value);
void print_I(unsigned char x_pos, unsigned char y_pos, signed long value);
void print_D(unsigned char x_pos, unsigned char y_pos, signed int value, unsigned char
points);
void print_F(unsigned char x_pos, unsigned char y_pos, float value, unsigned char points);
```

lcd_print_rgb.c

```
#include "lcd_print_rgb.h"


void load_custom_symbol(void)
{
    unsigned char s = 0;

    const unsigned char custom_symbol[array_size] =
    {
        0x00, 0x06, 0x09, 0x09, 0x06, 0x00, 0x00, 0x00
    };

    LCD_I2C_write(0x40, CMD);

    for(s = 0; s < array_size; s++)
    {
         LCD_I2C_write(custom_symbol[s], DAT);
    }

    LCD_I2C_write(0x80, CMD);
}


void print_symbol(unsigned char x_pos, unsigned char y_pos, unsigned char symbol_index)
{
    LCD_goto(x_pos, y_pos);
    LCD_I2C_write(symbol_index, DAT);
}


void print_C(unsigned char x_pos, unsigned char y_pos, signed int value)
{
     char ch[5] = {0x20, 0x20, 0x20, 0x20, '\0'};

     if(value < 0x00)
     {
        ch[0] = 0x2D;
        value = -value;
     }
     else
     {
        ch[0] = 0x20;
     }

     if((value > 99) && (value <= 999))
     {
         ch[1] = ((value / 100) + 0x30);
         ch[2] = (((value % 100) / 10) + 0x30);
         ch[3] = ((value % 10) + 0x30);
     }
     else if((value > 9) && (value <= 99))
     {
         ch[1] = (((value % 100) / 10) + 0x30);
         ch[2] = ((value % 10) + 0x30);
         ch[3] = 0x20;
     }
     else if((value >= 0) && (value <= 9))
     {
         ch[1] = ((value % 10) + 0x30);
         ch[2] = 0x20;
         ch[3] = 0x20;
     }
```

```
     LCD_goto(x_pos, y_pos);
     LCD_putstr(ch);
}


void print_I(unsigned char x_pos, unsigned char y_pos, signed long value)
{
    char ch[7] = {0x20, 0x20, 0x20, 0x20, 0x20, 0x20, '\0'};

    if(value < 0)
    {
        ch[0] = 0x2D;
        value = -value;
    }
    else
    {
        ch[0] = 0x20;
    }

    if(value > 9999)
    {
        ch[1] = ((value / 10000) + 0x30);
        ch[2] = (((value % 10000)/ 1000) + 0x30);
        ch[3] = (((value % 1000) / 100) + 0x30);
        ch[4] = (((value % 100) / 10) + 0x30);
        ch[5] = ((value % 10) + 0x30);
    }

    else if((value > 999) && (value <= 9999))
    {
        ch[1] = (((value % 10000)/ 1000) + 0x30);
        ch[2] = (((value % 1000) / 100) + 0x30);
        ch[3] = (((value % 100) / 10) + 0x30);
        ch[4] = ((value % 10) + 0x30);
        ch[5] = 0x20;
    }
    else if((value > 99) && (value <= 999))
    {
        ch[1] = (((value % 1000) / 100) + 0x30);
        ch[2] = (((value % 100) / 10) + 0x30);
        ch[3] = ((value % 10) + 0x30);
        ch[4] = 0x20;
        ch[5] = 0x20;
    }
    else if((value > 9) && (value <= 99))
    {
        ch[1] = (((value % 100) / 10) + 0x30);
        ch[2] = ((value % 10) + 0x30);
        ch[3] = 0x20;
        ch[4] = 0x20;
        ch[5] = 0x20;
    }
    else
    {
        ch[1] = ((value % 10) + 0x30);
        ch[2] = 0x20;
        ch[3] = 0x20;
        ch[4] = 0x20;
        ch[5] = 0x20;
    }

    LCD_goto(x_pos, y_pos);
    LCD_putstr(ch);
}
```

```
void print_D(unsigned char x_pos, unsigned char y_pos, signed int value, unsigned char
points)
{
    char ch[5] = {0x2E, 0x20, 0x20, 0x20, 0x20};

    ch[1] = ((value / 100) + 0x30);

    if(points > 1)
    {
        ch[2] = (((value / 10) % 10) + 0x30);

        if(points > 1)
        {
            ch[3] = ((value % 10) + 0x30);
        }
    }

    LCD_goto(x_pos, y_pos);
    LCD_putstr(ch);
}


void print_F(unsigned char x_pos, unsigned char y_pos, float value, unsigned char points)
{
    signed long tmp = 0x00000000;

    tmp = value;
    print_I(x_pos, y_pos, tmp);
    tmp = ((value - tmp) * 1000);

    if(tmp < 0)
    {
       tmp = -tmp;
    }

    if(value < 0)
    {
        value = -value;
        LCD_goto(x_pos, y_pos);
        LCD_putchar(0x2D);
    }
    else
    {
        LCD_goto(x_pos, y_pos);
        LCD_putchar(0x20);
    }

    if((value >= 10000) && (value < 100000))
    {
        print_D((x_pos + 6), y_pos, tmp, points);
    }
    else if((value >= 1000) && (value < 10000))
    {
        print_D((x_pos + 5), y_pos, tmp, points);
    }
    else if((value >= 100) && (value < 1000))
    {
        print_D((x_pos + 4), y_pos, tmp, points);
    }
    else if((value >= 10) && (value < 100))
    {
        print_D((x_pos + 3), y_pos, tmp, points);
    }
    else if(value < 10)
    {
        print_D((x_pos + 2), y_pos, tmp, points);
```

```
    }
}
```

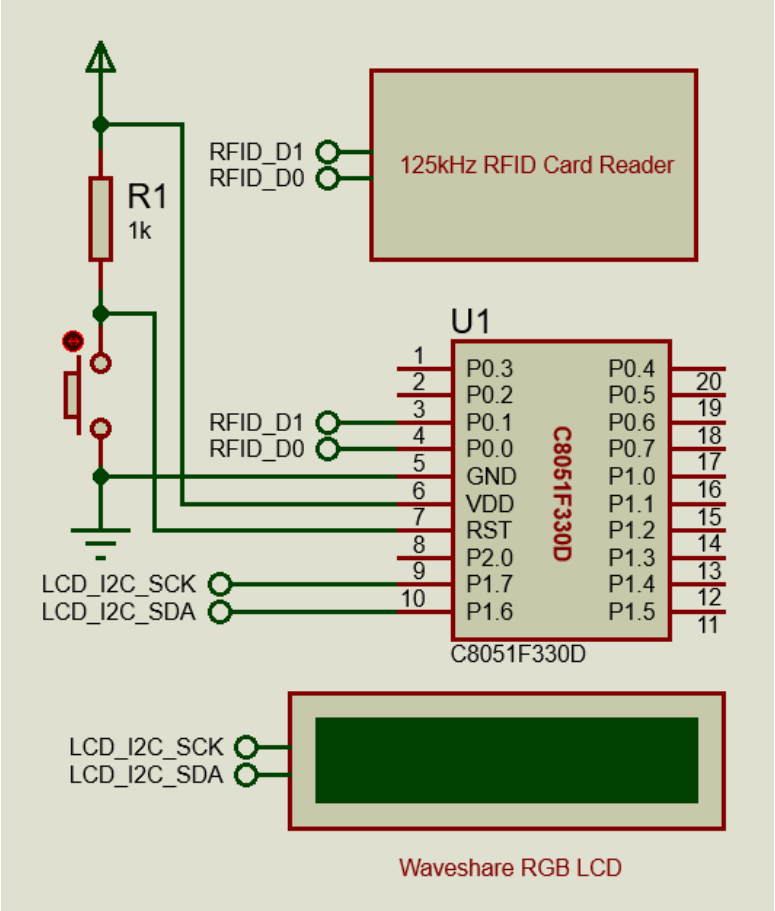
main.c

```
#include "Waveshare_RGB_LCD.c"
#include "lcd_print_rgb.c"


unsigned char count = 0;
unsigned long raw_card_data = 0;


void PCA_Init(void);
void Timer_Init(void);
void Port_IO_Init();
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);


void EXTI_0_ISR(void)
iv IVT_ADDR_EX0
ilevel 0
ics ICS_AUTO
{
    raw_card_data <<= 1;
    count++;
}


void EXTI_1_ISR(void)
iv IVT_ADDR_EX1
ilevel 0
ics ICS_AUTO
{
    raw_card_data <<= 1;
    raw_card_data |= 1;
    count++;
}


void main(void)
{
    unsigned char facility_code = 0;
    unsigned int card_number = 0;

    Init_Device();
    RGB_LCD_init();
    LCD_clear_home();

    LCD_goto(0, 0);
    LCD_putstr("Facility:");

    LCD_goto(0, 1);
    LCD_putstr("Card I.D:");

    while(1)
    {
        if(count >= 25)
        {
            card_number = (raw_card_data & 0xFFFF);
            facility_code = (0xFF & (raw_card_data >> 0x10));
            print_C(12, 0, facility_code);
```

```
            print_I(10, 1, card_number);
            raw_card_data = 0;
            count = 0;
        }
    };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}


void Timer_Init(void)
{
    TCON = 0x05;
}


void Port_IO_Init()
{
    // P0.0  -  Skipped,     Open-Drain, Digital
    // P0.1  -  Skipped,     Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Unassigned,  Open-Drain, Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P1MDOUT = 0xC0;
    P0SKIP = 0x03;
    P1SKIP = 0xC0;
    XBR1 = 0x40;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
{
    IE = 0x85;
    IP = 0x05;
    IT01CF = 0x10;
}


void Init_Device(void)
{
    PCA_Init();
```

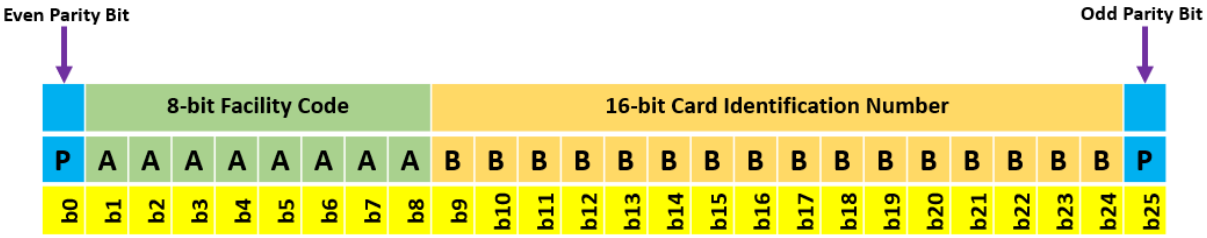
```
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}
```

## Schematic

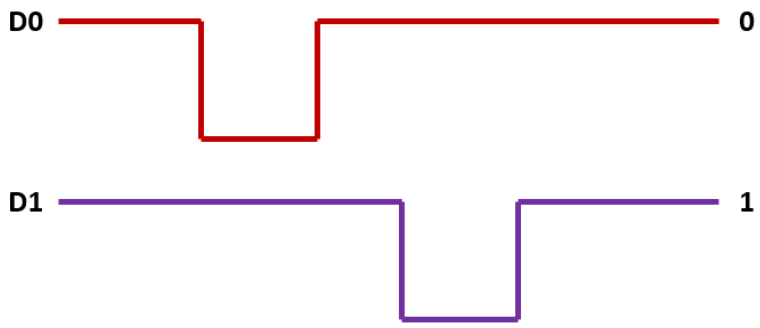



## Explanation

In Wiegand 26 protocol, there are 26 bits which define unique card identification (B) and facility code (A) as shown below:




Bits 0 to 25 are either ones or zeroes and are defined by the D0 and D1 outputs of the Wiegand 26 RFID reader. Normally D0 and D1 remain in the high logic state. When there is a high to low transition in D0 output, a zero is read and similarly when there is such a transition in D1, a one is read. Detecting 26 such transitions results in decoding the card number.




External interrupts EXTI0 (P0.0) and EXTI1 (P0.1) are configured to detect high-to-low or falling edge transitions. These pins are configured as open-drain inputs.

```
void Interrupts_Init(void)
{
    IE = 0x85;
    IP = 0x05;
    IT01CF = 0x10;
}


void Timer_Init(void)
{
    TCON = 0x05;
}
```

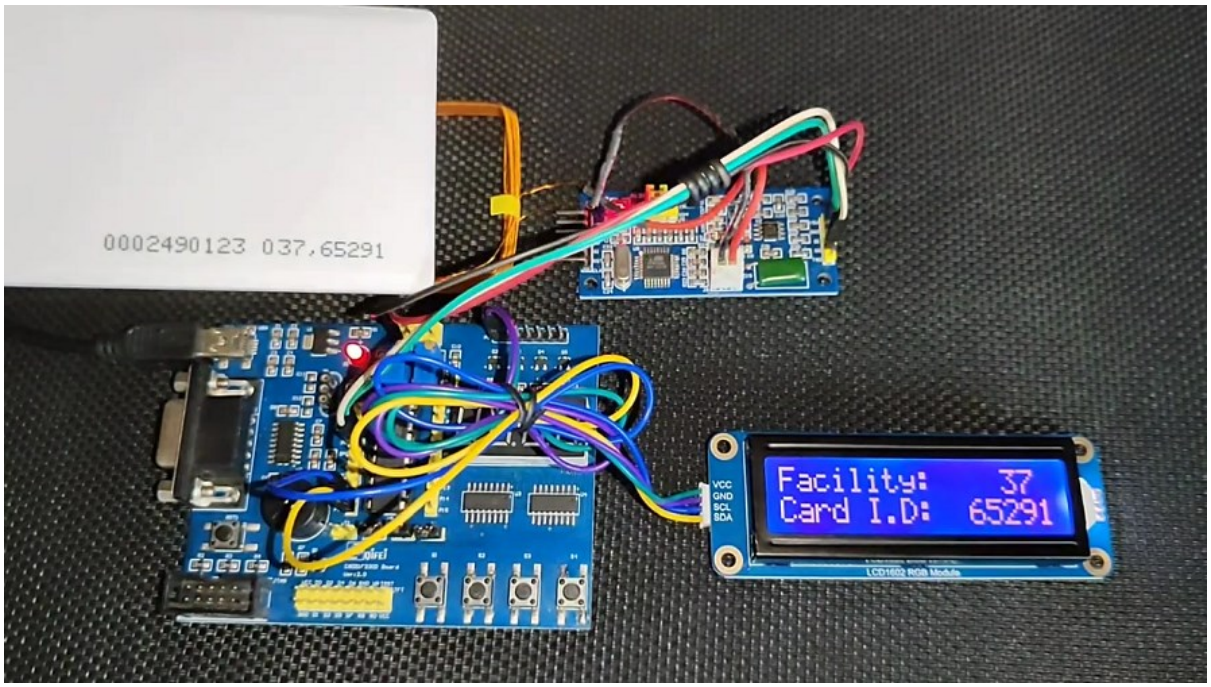
EXTI0 is connected to D0 and EXTI1 is connected to D1 of the RFID reader respectively and so when interrupts are triggered bit info is processed. When EXTI0 is triggered, it means that the bit in the card code is zero. Similarly, when EXTI1 is triggered, it means that the bit in the card code is one. The variable *raw_card_data* contains the card info and upon every interrupt, this variable is shifted while the bit count is stored in the variable *count*. A *"one"* is added to the variable *raw_card_data* when EXTI1 is triggered. For *"zero"*, nothing is needed to be done.

```
void EXTI_0_ISR(void)
iv IVT_ADDR_EX0
ilevel 0
ics ICS_AUTO
{
    raw_card_data <<= 1;
    count++;
}


void EXTI_1_ISR(void)
iv IVT_ADDR_EX1
ilevel 0
ics ICS_AUTO
{
    raw_card_data <<= 1;
    raw_card_data |= 1;
    count++;
}
```

Inside the main code, the peripherals are initialized first. The bit count is checked to see if all 26 bits (0 to 25) have been read. If read so then the raw card info is processed and displayed. After displaying, bit count and raw card data variables are reset to zeroes.

```
void main(void)
{
    unsigned char facility_code = 0;
    unsigned int card_number = 0;

    Init_Device();
    RGB_LCD_init();
    LCD_clear_home();

    LCD_goto(0, 0);
    LCD_putstr("Facility:");

    LCD_goto(0, 1);
    LCD_putstr("Card I.D:");

    while(1)
    {
        if(count >= 25)
        {
            card_number = (raw_card_data & 0xFFFF);
            facility_code = (0xFF & (raw_card_data >> 0x10));
            print_C(12, 0, facility_code);
            print_I(10, 1, card_number);
            raw_card_data = 0;
            count = 0;
        }
    };
}
```

## Demo

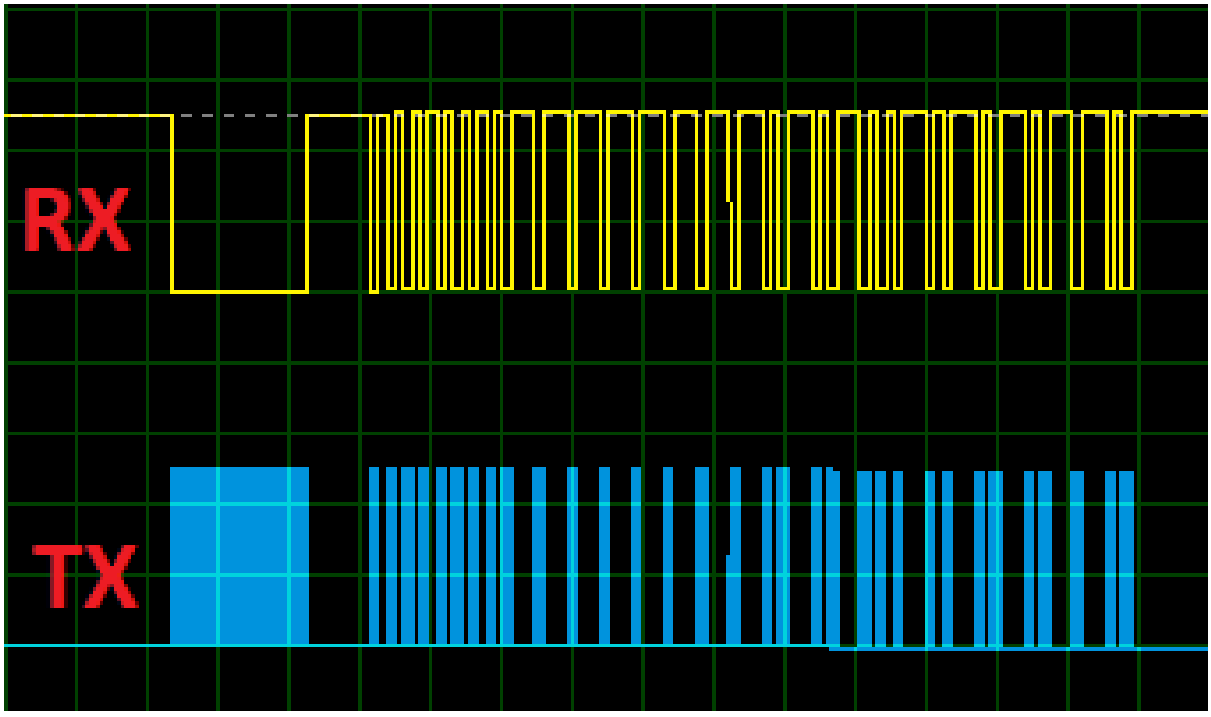



Demo video link: https://youtu.be/uXVXlZ71dZk

# External Interrupt (EXTI) and Timer (TMR) – Decoding IR Remote

Infrared (IR) remote controllers are widely used and cheap methods of remotely controlling appliances such as televisions, stereos, home theatres, etc. An IR remote transmitter sends data to an IR receiver by modulating/mixing data signals with a carrier wave of frequency between 30kHz to 40kHz. This modulation technique ensures the safety of data during long-distance transmissions. At the receiving end, an IR receiver picks up the modulated data and demodulates it, resulting in a stream of pulses. These pulses possess variable properties such as pulse widths, timing, position, and phase, and they carry data. The pulse properties are governed by a set of rules known as the protocol. By decoding the pulses according to the protocol, the sent information can be retrieved.

In most microcontrollers, there is no dedicated hardware for decoding IR remote protocols. Additionally, the microcontroller receiving IR remote data does not have prior knowledge of when it will receive an IR burst. Therefore, a combination of an external interrupt and a timer coupled with some software tricks is required for decoding IR data.




In this segment, we will explore how an external interrupt and a timer can be utilized to decode a NEC IR remote effortlessly. It is worth mentioning that this approach can be extended to decode any IR remote controller. To begin, I suggest studying the NEC IR protocol, which can be found here. For comprehensive information regarding IR remotes, SB-Project's website is highly recommended as an excellent resource.

## Code

### SW_I2C.h

```
#define bit_shift_right(val, shift_size)     (val >> shift_size)
#define bit_shift_left(val, shift_size)      (val << shift_size)
#define bit_mask(bit_pos)                    bit_shift_left(1, bit_pos)

#define bit_set(val, bit_val)               (val |= bit_mask(bit_val))
#define bit_clr(val, bit_val)               (val &= (~bit_mask(bit_val)))
#define bit_tgl(val, bit_val)               (val ^= bit_mask(bit_val))
#define get_bit(val, bit_val)               (val & bit_mask(bit_val))
#define get_reg(val, msk)                   (val & msk)

#define test_if_bit_set(val, bit_val)       (get_bit(val, bit_val) != FALSE)
#define test_if_bit_cleared(val, bit_val)   (get_bit(val, bit_val) == FALSE)

#define test_if_all_bits_set(val, msk)     (get_reg(val, msk) == msk)
#define test_if_any_bit_set(val, msk)      (get_reg(val, msk) != FALSE)


#define SDA_DIR_OUT()                      do{bit_set(P1MDOUT, 6); bit_set(P0SKIP,
6);}while(0)
#define SDA_DIR_IN()                       do{bit_clr(P1MDOUT, 6); bit_set(P0SKIP,
6);}while(0)

#define SCL_DIR_OUT()                      do{bit_set(P1MDOUT, 7); bit_set(P0SKIP,
7);}while(0)
#define SCL_DIR_IN()                       do{bit_clr(P1MDOUT, 7); bit_set(P0SKIP,
7);}while(0)

#define SDA_HIGH()                         bit_set(P1, 6)
#define SDA_LOW()                          bit_clr(P1, 6)

#define SCL_HIGH()                         bit_set(P1, 7)
#define SCL_LOW()                          bit_clr(P1, 7)

#define SDA_IN()                           get_bit(P1, 6)

#define I2C_ACK                            0xFF
#define I2C_NACK                           0x00

#define I2C_timeout                        1000


void SW_I2C_init(void);
void SW_I2C_start(void);
void SW_I2C_stop(void);
unsigned char SW_I2C_read(unsigned char ack);
void SW_I2C_write(unsigned char value);
void SW_I2C_ACK_NACK(unsigned char mode);
unsigned char SW_I2C_wait_ACK(void);
```

### SW_I2C.c

```
#include "SW_I2C.h"


void SW_I2C_init(void)
{
    XBR1 = 0x40;
    SDA_DIR_OUT();
```

```
    SCL_DIR_OUT();
    delay_us(100);
    SDA_HIGH();
    SCL_HIGH();
}


void SW_I2C_start(void)
{
    SDA_DIR_OUT();
    SDA_HIGH();
    SCL_HIGH();
    delay_us(40);
    SDA_LOW();
    delay_us(40);
    SCL_LOW();
}


void SW_I2C_stop(void)
{
    SDA_DIR_OUT();
    SDA_LOW();
    SCL_LOW();
    delay_us(40);
    SDA_HIGH();
    SCL_HIGH();
    delay_us(40);
}


unsigned char SW_I2C_read(unsigned char ack)
{
    unsigned char i = 8;
    unsigned char j = 0;

    SDA_DIR_IN();

    while(i > 0)
    {
        SCL_LOW();
        delay_us(20);
        SCL_HIGH();
        delay_us(20);
        j <<= 1;

        if(SDA_IN() != 0x00)
        {
            j++;
        }

        delay_us(10);
        i--;
    };

    switch(ack)
    {
        case I2C_ACK:
        {
            SW_I2C_ACK_NACK(I2C_ACK);;
            break;
        }
        default:
        {
            SW_I2C_ACK_NACK(I2C_NACK);;
```

```
            break;
        }
    }

    return j;
}


void SW_I2C_write(unsigned char value)
{
    unsigned char i = 8;

    SDA_DIR_OUT();
    SCL_LOW();

    while(i > 0)
    {

        if(((value & 0x80) >> 7) != 0x00)
        {
            SDA_HIGH();
        }
        else
        {
            SDA_LOW();
        }


        value <<= 1;
        delay_us(20);
        SCL_HIGH();
        delay_us(20);
        SCL_LOW();
        delay_us(20);
        i--;
    };
}


void SW_I2C_ACK_NACK(unsigned char mode)
{
    SCL_LOW();
    SDA_DIR_OUT();

    switch(mode)
    {
        case I2C_ACK:
        {
            SDA_LOW();
            break;
        }
        default:
        {
            SDA_HIGH();
            break;
        }
    }

    delay_us(20);
    SCL_HIGH();
    delay_us(20);
    SCL_LOW();
}
```

```
unsigned char SW_I2C_wait_ACK(void)
{
    signed int timeout = 0;

    SDA_DIR_IN();

    SDA_HIGH();
    delay_us(10);
    SCL_HIGH();
    delay_us(10);

    while(SDA_IN() != 0x00)
    {
        timeout++;

        if(timeout > I2C_timeout)
        {
            SW_I2C_stop();
            return 1;
        }
    };

    SCL_LOW();
    return 0;
}
```

PCF8574.h

```
#include "SW_I2C.c"


#define PCF8574_address                 0x4E

#define PCF8574_write_cmd               PCF8574_address
#define PCF8574_read_cmd                (PCF8574_address | 1)


void PCF8574_init(void);
unsigned char PCF8574_read(void);
void PCF8574_write(unsigned char data_byte);
```

PCF8574.c

```
#include "PCF8574.h"


void PCF8574_init(void)
{
    SW_I2C_init();
    delay_ms(20);
}


unsigned char PCF8574_read(void)
{
    unsigned char port_byte = 0;

    SW_I2C_start();
    SW_I2C_write(PCF8574_read_cmd);
    port_byte = SW_I2C_read(I2C_NACK);
    SW_I2C_stop();
```

```
    return port_byte;
}


void PCF8574_write(unsigned char data_byte)
{
    SW_I2C_start();
    SW_I2C_write(PCF8574_write_cmd);
    SW_I2C_ACK_NACK(I2C_ACK);
    SW_I2C_write(data_byte);
    SW_I2C_ACK_NACK(I2C_ACK);
    SW_I2C_stop();
}
```

LCD_2_Wire.h

```
#include "PCF8574.c"


#define clear_display        0x01
#define goto_home            0x02

#define cursor_direction_inc 0x06
#define cursor_direction_dec 0x04
#define display_shift        0x05
#define display_no_shift     0x04

#define display_on           0x0C
#define display_off          0x0A
#define cursor_on            0x0A
#define cursor_off           0x08
#define blink_on             0x09
#define blink_off            0x08

#define _8_pin_interface     0x30
#define _4_pin_interface     0x20
#define _2_row_display       0x28
#define _1_row_display       0x20
#define _5x10_dots           0x60
#define _5x7_dots            0x20

#define BL_ON                1
#define BL_OFF               0

#define dly                  2

#define DAT                  1
#define CMD                  0


void LCD_init(void);
void LCD_toggle_EN(void);
void LCD_send(unsigned char value, unsigned char mode);
void LCD_4bit_send(unsigned char lcd_data);
void LCD_putstr(char *lcd_string);
void LCD_putchar(char char_data);
void LCD_clear_home(void);
void LCD_goto(unsigned char x_pos, unsigned char y_pos);
```

LCD_2_Wire.c

```
#include "LCD_2_Wire.h"


static unsigned char bl_state;
static unsigned char data_value;


void LCD_init(void)
{
    PCF8574_init();
    delay_ms(10);

    bl_state = BL_ON;
    data_value = 0x04;
    PCF8574_write(data_value);

    delay_ms(10);

    LCD_send(0x33, CMD);
    LCD_send(0x32, CMD);

    LCD_send((_4_pin_interface | _2_row_display | _5x7_dots), CMD);
    LCD_send((display_on | cursor_off | blink_off), CMD);
    LCD_send((clear_display), CMD);
    LCD_send((cursor_direction_inc | display_no_shift), CMD);
}


void LCD_toggle_EN(void)
{
    data_value |= 0x04;
    PCF8574_write(data_value);
    delay_ms(1);
    data_value &= 0xF9;
    PCF8574_write(data_value);
    delay_ms(1);
}


void LCD_send(unsigned char value, unsigned char mode)
{
    switch(mode)
    {
       case CMD:
       {
          data_value &= 0xF4;
          break;
       }
       case DAT:
       {
          data_value |= 0x01;
          break;
       }
    }

    switch(bl_state)
    {
       case BL_ON:
       {
          data_value |= 0x08;
          break;
       }
```

```
        case BL_OFF:
        {
           data_value &= 0xF7;
           break;
        }
    }

    PCF8574_write(data_value);
    LCD_4bit_send(value);
    delay_ms(1);
}


void LCD_4bit_send(unsigned char lcd_data)
{
    unsigned char temp = 0x00;

    temp = (lcd_data & 0xF0);
    data_value &= 0x0F;
    data_value |= temp;
    PCF8574_write(data_value);
    LCD_toggle_EN();

    temp = (lcd_data & 0x0F);
    temp <<= 0x04;
    data_value &= 0x0F;
    data_value |= temp;
    PCF8574_write(data_value);
    LCD_toggle_EN();
}


void LCD_putstr(char *lcd_string)
{
    do
    {
        LCD_putchar(*lcd_string++);
    }while(*lcd_string != '\0') ;
}


void LCD_putchar(char char_data)
{
    if((char_data >= 0x20) && (char_data <= 0x7F))
    {
        LCD_send(char_data, DAT);
    }
}


void LCD_clear_home(void)
{
    LCD_send(clear_display, CMD);
    LCD_send(goto_home, CMD);
}


void LCD_goto(unsigned char x_pos,unsigned char y_pos)
{
    if(y_pos == 0)
    {
        LCD_send((0x80 | x_pos), CMD);
    }
    else
    {
```

```
        LCD_send((0x80 | 0x40 | x_pos), CMD);
    }
}
```

lcd_print.h

```
#define no_of_custom_symbol     1
#define array_size_per_symbol   8
#define array_size              (array_size_per_symbol * no_of_custom_symbol)


void load_custom_symbol(void);
void print_symbol(unsigned char x_pos, unsigned char y_pos, unsigned char symbol_index);
void print_C(unsigned char x_pos, unsigned char y_pos, signed int value);
void print_I(unsigned char x_pos, unsigned char y_pos, signed long value);
void print_D(unsigned char x_pos, unsigned char y_pos, signed int value, unsigned char
points);
void print_F(unsigned char x_pos, unsigned char y_pos, float value, unsigned char points);
```

lcd_print.c

```
#include "lcd_print.h"


void load_custom_symbol(void)
{
    unsigned char s = 0;

    const unsigned char custom_symbol[array_size] =
    {
        0x00, 0x06, 0x09, 0x09, 0x06, 0x00, 0x00, 0x00
    };

    LCD_send(0x40, CMD);

    for(s = 0; s < array_size; s++)
    {
         LCD_send(custom_symbol[s], DAT);
    }

    LCD_send(0x80, CMD);
}


void print_symbol(unsigned char x_pos, unsigned char y_pos, unsigned char symbol_index)
{
    LCD_goto(x_pos, y_pos);
    LCD_send(symbol_index, DAT);
}


void print_C(unsigned char x_pos, unsigned char y_pos, signed int value)
{
     char ch[5] = {0x20, 0x20, 0x20, 0x20, '\0'};

     if(value < 0x00)
     {
        ch[0] = 0x2D;
        value = -value;
     }
     else
     {
```

```
        ch[0] = 0x20;
    }

    if((value > 99) && (value <= 999))
    {
        ch[1] = ((value / 100) + 0x30);
        ch[2] = (((value % 100) / 10) + 0x30);
        ch[3] = ((value % 10) + 0x30);
    }
    else if((value > 9) && (value <= 99))
    {
        ch[1] = (((value % 100) / 10) + 0x30);
        ch[2] = ((value % 10) + 0x30);
        ch[3] = 0x20;
    }
    else if((value >= 0) && (value <= 9))
    {
        ch[1] = ((value % 10) + 0x30);
        ch[2] = 0x20;
        ch[3] = 0x20;
    }

    LCD_goto(x_pos, y_pos);
    LCD_putstr(ch);
}


void print_I(unsigned char x_pos, unsigned char y_pos, signed long value)
{
    char ch[7] = {0x20, 0x20, 0x20, 0x20, 0x20, 0x20, '\0'};

    if(value < 0)
    {
        ch[0] = 0x2D;
        value = -value;
    }
    else
    {
        ch[0] = 0x20;
    }

    if(value > 9999)
    {
        ch[1] = ((value / 10000) + 0x30);
        ch[2] = (((value % 10000)/ 1000) + 0x30);
        ch[3] = (((value % 1000) / 100) + 0x30);
        ch[4] = (((value % 100) / 10) + 0x30);
        ch[5] = ((value % 10) + 0x30);
    }

    else if((value > 999) && (value <= 9999))
    {
        ch[1] = (((value % 10000)/ 1000) + 0x30);
        ch[2] = (((value % 1000) / 100) + 0x30);
        ch[3] = (((value % 100) / 10) + 0x30);
        ch[4] = ((value % 10) + 0x30);
        ch[5] = 0x20;
    }
    else if((value > 99) && (value <= 999))
    {
        ch[1] = (((value % 1000) / 100) + 0x30);
        ch[2] = (((value % 100) / 10) + 0x30);
        ch[3] = ((value % 10) + 0x30);
        ch[4] = 0x20;
        ch[5] = 0x20;
    }
```

```
    else if((value > 9) && (value <= 99))
    {
        ch[1] = (((value % 100) / 10) + 0x30);
        ch[2] = ((value % 10) + 0x30);
        ch[3] = 0x20;
        ch[4] = 0x20;
        ch[5] = 0x20;
    }
    else
    {
        ch[1] = ((value % 10) + 0x30);
        ch[2] = 0x20;
        ch[3] = 0x20;
        ch[4] = 0x20;
        ch[5] = 0x20;
    }

    LCD_goto(x_pos, y_pos);
    LCD_putstr(ch);
}


void print_D(unsigned char x_pos, unsigned char y_pos, signed int value, unsigned char
points)
{
    char ch[5] = {0x2E, 0x20, 0x20, 0x20, 0x20};

    ch[1] = ((value / 100) + 0x30);

    if(points > 1)
    {
        ch[2] = (((value / 10) % 10) + 0x30);

        if(points > 1)
        {
            ch[3] = ((value % 10) + 0x30);
        }
    }

    LCD_goto(x_pos, y_pos);
    LCD_putstr(ch);
}


void print_F(unsigned char x_pos, unsigned char y_pos, float value, unsigned char points)
{
    signed long tmp = 0x00000000;

    tmp = value;
    print_I(x_pos, y_pos, tmp);
    tmp = ((value - tmp) * 1000);

    if(tmp < 0)
    {
       tmp = -tmp;
    }

    if(value < 0)
    {
        value = -value;
        LCD_goto(x_pos, y_pos);
        LCD_putchar(0x2D);
    }
    else
    {
```

```
        LCD_goto(x_pos, y_pos);
        LCD_putchar(0x20);
    }

    if((value >= 10000) && (value < 100000))
    {
        print_D((x_pos + 6), y_pos, tmp, points);
    }
    else if((value >= 1000) && (value < 10000))
    {
        print_D((x_pos + 5), y_pos, tmp, points);
    }
    else if((value >= 100) && (value < 1000))
    {
        print_D((x_pos + 4), y_pos, tmp, points);
    }
    else if((value >= 10) && (value < 100))
    {
        print_D((x_pos + 3), y_pos, tmp, points);
    }
    else if(value < 10)
    {
        print_D((x_pos + 2), y_pos, tmp, points);
    }
}
```

main.c

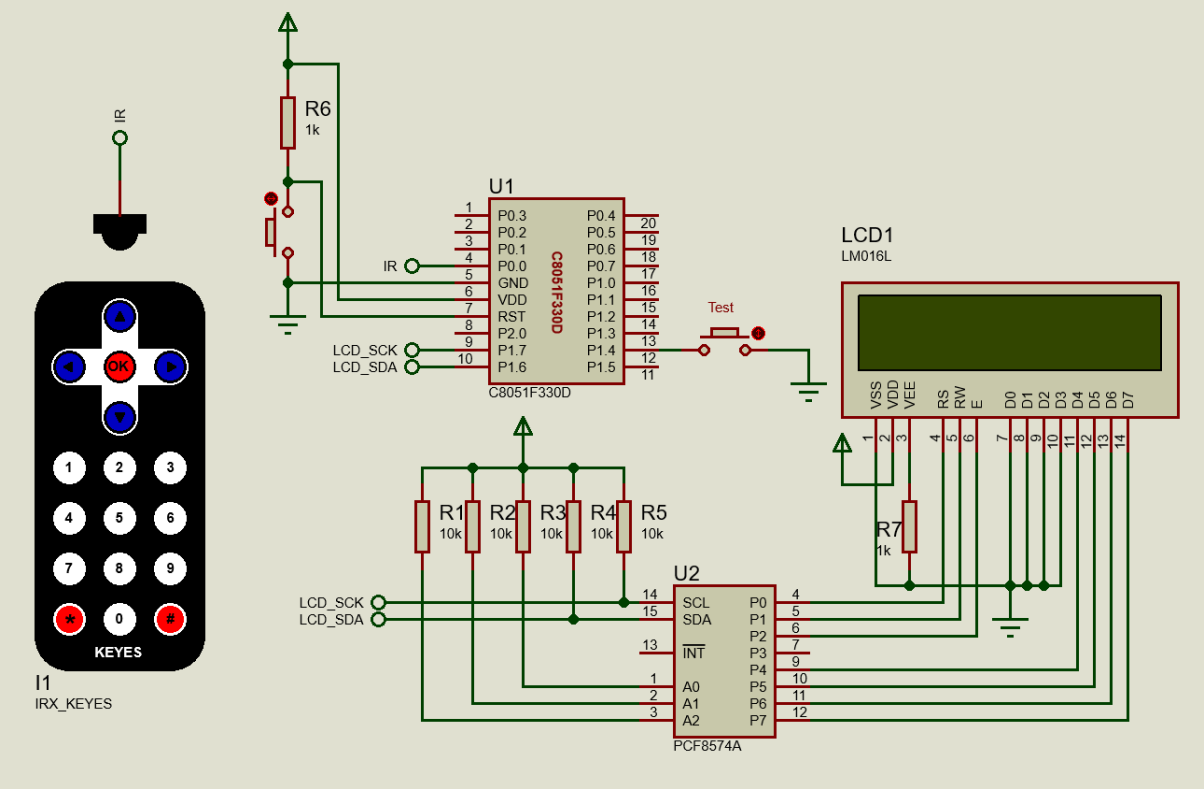
```
#include "LCD_2_Wire.c"
#include "lcd_print.c"


#define sync_high          16000
#define sync_low           10800
#define one_high            2700
#define one_low             1800
#define zero_high           1400
#define zero_low             900


bit received;
unsigned char bits = 0;
unsigned int frames[33];


void PCA_Init(void);
void Timer_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void erase_frames(void);
unsigned int get_timer(void);
void set_timer(void);
unsigned char decode(unsigned char start_pos, unsigned char end_pos);
void decode_NEC(unsigned char *addr, unsigned char *cmd);


void IR_receive(void)
iv IVT_ADDR_EX0
ilevel 0
ics ICS_AUTO
{
    frames[bits] = get_timer();
```

```
    bits++;
    TR0_bit = 1;

    if(bits >= 33)
    {
       received = 1;
       TR0_bit = 0;
    }
    set_timer();
}


void main(void)
{
    unsigned char i = 0;

    unsigned char address = 0;
    unsigned char command = 0;

    Init_Device();
    LCD_init();
    LCD_clear_home();

    LCD_goto(0, 0);
    LCD_putstr("ADR:");
    LCD_goto(0, 1);
    LCD_putstr("CMD:");

    while(1)
    {
       if(received)
       {
           decode_NEC(&address, &command);
           print_I(12, 0, address);
           print_I(12, 1, command);
           erase_frames();
       }
    };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}


void Timer_Init(void)
{
    TCON = 0x01;
    TMOD = 0x01;
}


void Port_IO_Init(void)
{
    // P0.0  -  Skipped,     Open-Drain, Digital
    // P0.1  -  Unassigned,  Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital
```

```
    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Unassigned,  Open-Drain, Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P1MDOUT = 0xC0;
    P0SKIP = 0x01;
    P1SKIP = 0xC0;
    XBR1 = 0x40;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
{
    IE = 0x81;
    IT01CF = 0x00;
}


void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}


void erase_frames(void)
{
     delay_ms(90);

     for(bits = 0; bits < 33; bits++)
     {
         frames[bits] = 0x0000;
     }

     set_timer();
     received = 0;
     bits = 0;
}


unsigned int get_timer(void)
{
    unsigned int time = 0;

    time = TH0;
    time <<= 8;
    time |= TL0;

    return time;
}
```

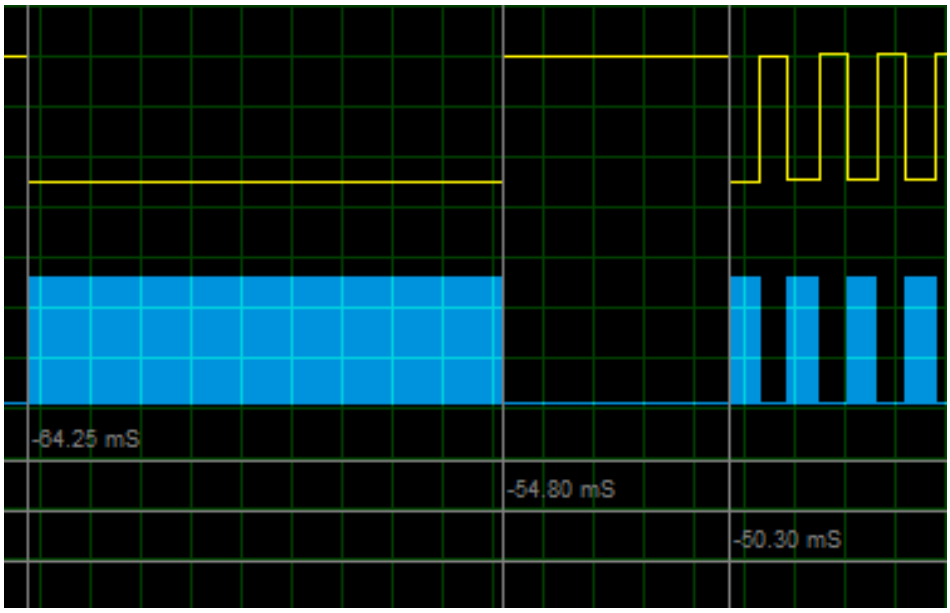
```
void set_timer(void)
{
     TH0 = 0;
     TL0 = 0;
}


unsigned char decode(unsigned char start_pos, unsigned char end_pos)
{
     unsigned char value = 0;

     for(bits = start_pos; bits <= end_pos; bits++)
     {
           value <<= 1;
           if((frames[bits] >= one_low) && (frames[bits] <= one_high))
           {
               value |= 1;
           }
           else if((frames[bits] >= zero_low) && (frames[bits] <= zero_high))
           {
               value |= 0;
           }
           else if((frames[bits] >= sync_low) && (frames[bits] <= sync_high))
           {
               return 0xFF;
           }
     }

     return value;
}


void decode_NEC(unsigned char *addr, unsigned char *cmd)
{
     *addr = decode(2, 9);
     *cmd = decode(18, 25);
}
```

## Schematic



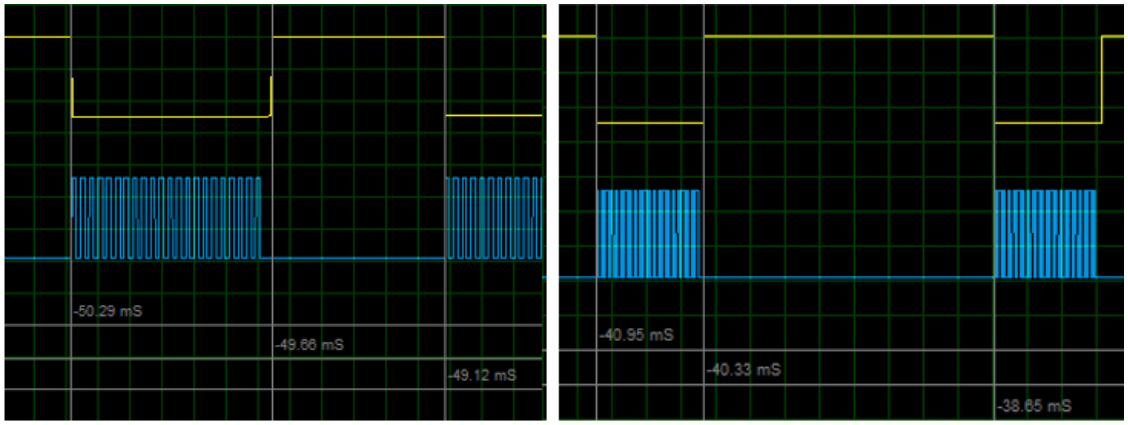

## Explanation

An NEC transmission consists of a total of 33 pulses. The initial pulse serves as the sync pulse, while the subsequent 32 pulses carry the address and command information. These 32 bits can be divided into two groups. The first group comprises 16 bits, which include the address and its inverted counterpart. The second group consists of another 16 bits, containing the command and its inverted form. Comparing the non-inverted signals with the inverted ones allows for data integrity verification.



In the case of the NEC IR protocol, the received IR data is represented as a sequence of pulses. These pulses correspond to sync bits, ones, and zeros. Notably, the pulse widths vary within this protocol. The sync bit is characterized by a high pulse lasting 9ms, followed by a low pulse of 4.5ms, resulting in a total pulse time of 13.5ms. Refer to the timing diagram below, where the blue pulses represent those from the transmitter, and the yellow pulses represent those received by the receiver. It is evident that the pulse streams are inverted, with this inversion occurring at the IR receiver.

In the NEC IR protocol, a logic one is indicated by a pulse with a high time of 560µs and a low time of 2.25ms, resulting in a total pulse time of 2.81ms. Conversely, a logic zero is represented by a pulse with a high time of 560µs and a low time of 1.12ms, giving a total pulse time of 1.68ms.




These timings can vary up to ±30% of their theoretical values due to several factors such as medium, temperature, reflections, etc. Thus, we have to define these time tolerances in our code.

```
#define sync_high          16000
#define sync_low           10800
#define one_high            2700
#define one_low             1800
#define zero_high           1400
#define zero_low             900
```

In this example, the system clock is set to 12.25MHz. We have to be as precise as possible with timings because we are dealing with a time-sensitive application.

```
void Oscillator_Init(void)
{
    OSCICN = 0x82;
}
```

Setting up an external interrupt to detect falling edges and a timer are crucial parts in detecting IR pulse streams. This leaves the MCU free for other non-critical tasks. P0.0 is tied to external interrupt 0 (INT0) as per the code below. Timer 0 is also used.

```
void Timer_Init(void)
{
    TCON = 0x01;
    TMOD = 0x01;
}
```

```
....

void Interrupts_Init(void)
{
    IE = 0x81;
    IT01CF = 0x00;
}
```

Timer 0 is operating at a speed of approximately 1MHz, where 1 timer tick equals 1µs. This timer is clocked with the 12.25MHz system clock, which has been divided by a prescaler of 12. In this example, the code utilizes the *"gate"* feature of this timer, which automatically starts or stops the timer. There are two possible cases with the *gate* feature. Either the timer runs like other timers when instructed to run via software, or it runs when it is coded to run and the INT0 pin is in a logic high state. The latter feature is used in this example. Note that the timer overflow interrupt is not needed here.

Upon an external interrupt, the timer immediately stops due to the falling edge that triggered the interrupt and the subsequent low level at the INT0 pin. The count of the timer is captured and then stored in an array. TR0 bit is also set high so that the timer can run again.

```
void IR_receive(void)
iv IVT_ADDR_EX0
ilevel 0
ics ICS_AUTO
{
    frames[bits] = get_timer();
    bits++;
    TR0_bit = 1;

    if(bits >= 33)
    {
       received = 1;
       TR0_bit = 0;
    }
    set_timer();
}
```

After capturing 33 pulses, the timer is fully stopped in order to process captured pulses.

The captured pulse times are compared against the timing extremes as mentioned earlier and sorted accordingly. This results in decoding IR data.

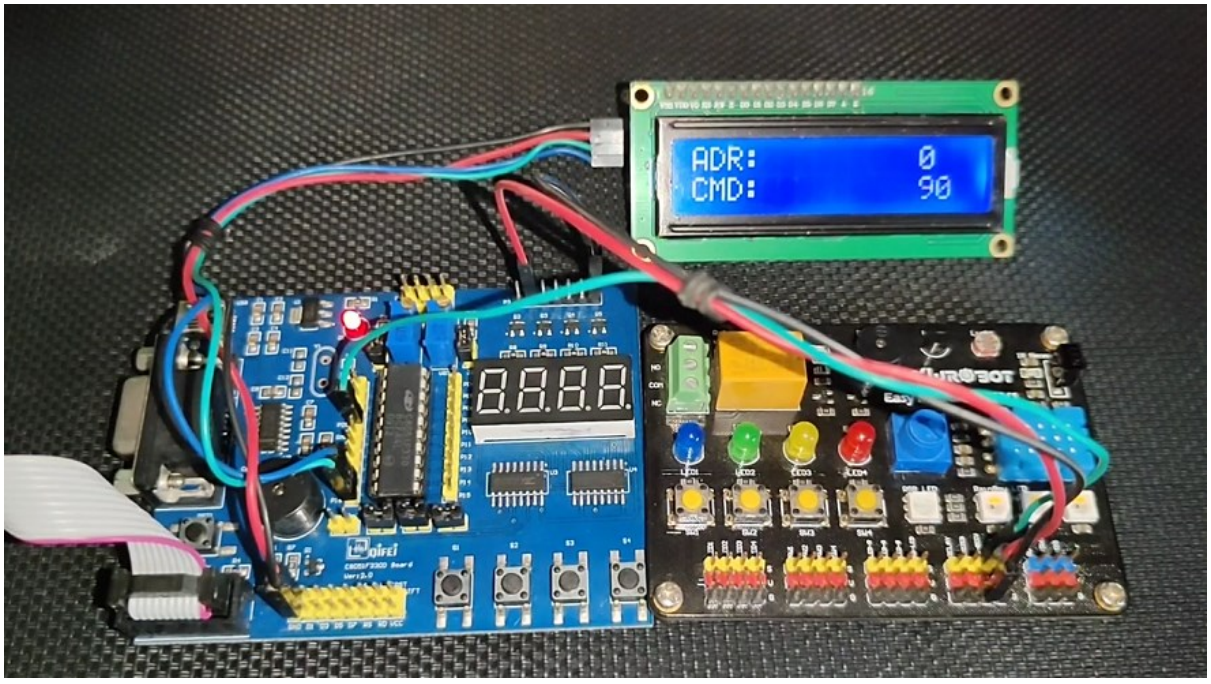
```
unsigned char decode(unsigned char start_pos, unsigned char end_pos)
{
     unsigned char value = 0;

     for(bits = start_pos; bits <= end_pos; bits++)
     {
           value <<= 1;
           if((frames[bits] >= one_low) && (frames[bits] <= one_high))
           {
               value |= 1;
           }
           else if((frames[bits] >= zero_low) && (frames[bits] <= zero_high))
           {
               value |= 0;
           }
           else if((frames[bits] >= sync_low) && (frames[bits] <= sync_high))
           {
               return 0xFF;
```

```
            }
        }

        return value;
}


void decode_NEC(unsigned char *addr, unsigned char *cmd)
{
        *addr = decode(2, 9);
        *cmd = decode(18, 25);
}
```

In the main loop, after receiving and decoding an NEC frame, the address and command data are displayed on an LCD. Following this, the microcontroller is readied to receive a new NEC frame.

```
if(received)
{
    decode_NEC(&address, &command);
    print_I(12, 0, address);
    print_I(12, 1, command);
    erase_frames();
}
```

## Demo

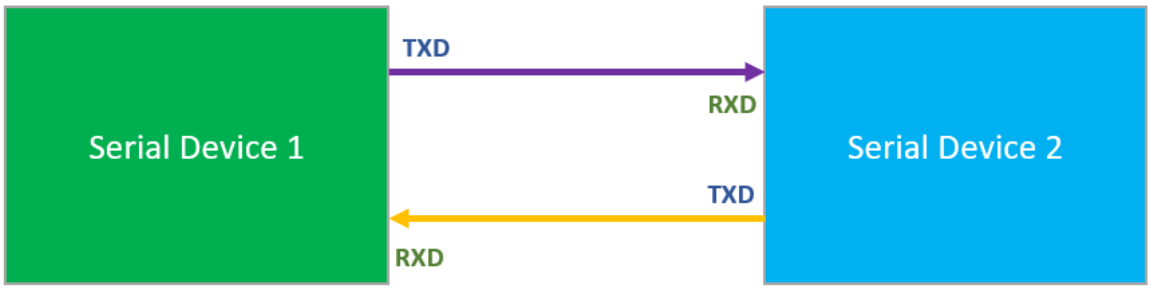



Demo video link: https://youtu.be/lv1Pp7kGRrE

# Serial Communication (UART) – HC-SR04 SONAR Reader

Serial communication via UART peripheral is the most widely used communication method between computers and microcontrollers, portable devices and so on. Many sensors and communication devices such as GSM modems, GPS receivers, RF modules, etc employ UART-based communication to transact data with host devices. UART peripheral is also the backbone of several other communications such as MODBUS, LIN, IrDA and so on.

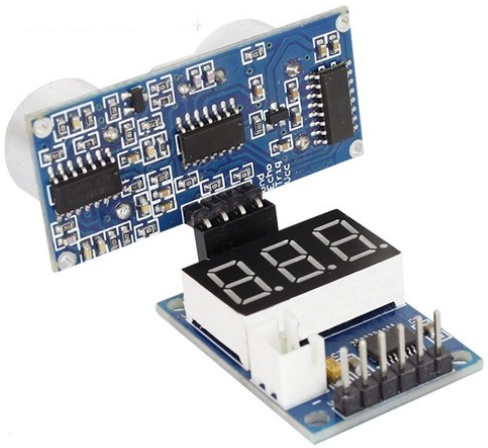
Only a pair of wires along with a ground wire are needed to establish communication between devices. Unless there is a requirement for longer range or voltage-level shifting, no additional hardware is needed to be able to use this peripheral. The following diagram below shows a simplified UART bus.




To learn more about UART visit the following link:

https://learn.mikroe.com/uart-serial-communication

To demonstrate the use of UART peripheral, I used a HC-SR04-to-serial converter. It is also known as a digital tape. This device measures and shows the distance between it and an object facing it using a HC-SR04 ultrasonic SONAR sensor. The extra feature of this device is the measurement output via serial peripherals.



## Code

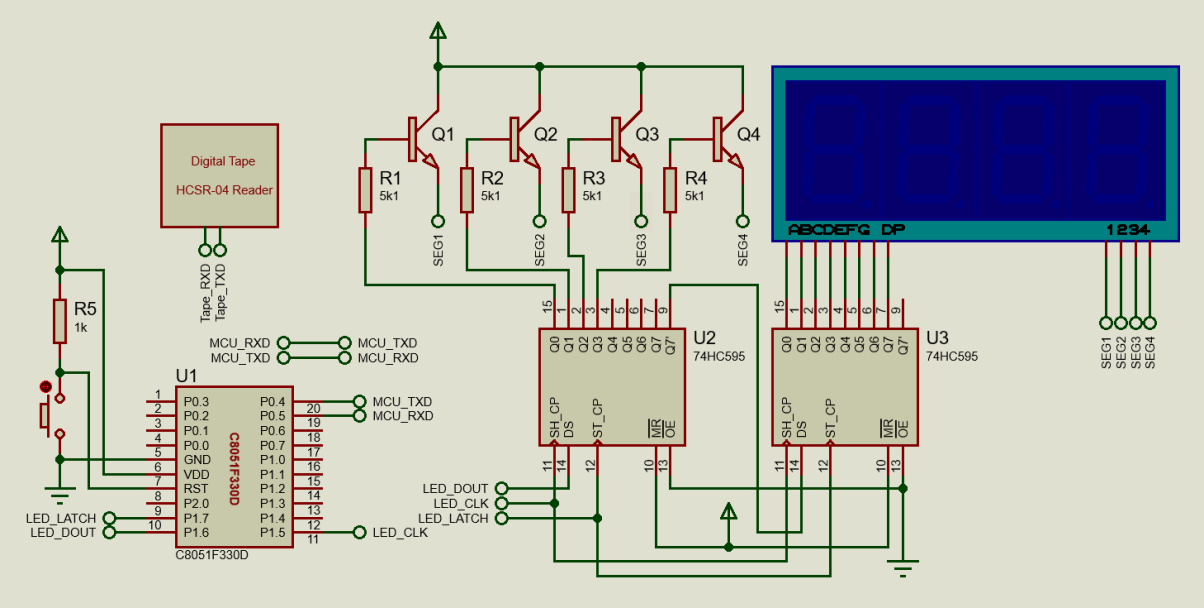
```
#define LED_DOUT P1_6_bit
#define  LED_CLK P1_5_bit
#define  LED_LATCH P1_7_bit

unsigned int d = 0;
unsigned char cnt = 0;
unsigned char value = 0;
char rx_buffer[18];

const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x7F, // .
  0xBF  // -
};

const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};

void PCA_Init(void);
void Timer_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);

void UART0_ISR(void)
iv IVT_ADDR_ES0
ilevel 0
ics ICS_AUTO
{
    rx_buffer[cnt++] = UART_Read();
    RI0_bit = 0;
}

void Timer_ISR(void)
iv IVT_ADDR_ET3
ilevel 1
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            value = (d / 1000);
            break;
        }
```

```
        case 1:
        {
            value = ((d % 1000) / 100);
            break;
        }
        case 2:
        {
            value = ((d % 100) / 10);
            break;
        }
        case 3:
        {
            value = (d % 10);
            break;
        }
    }

    if(d >= 40000)
    {
        segment_write(11, i);
    }
    else
    {
        segment_write(value, i);
    }

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}

void main(void)
{
     unsigned char i = 0x00;
     unsigned char j = 0x00;
     unsigned char k = 0x00;
     unsigned char l = 0x00;
     unsigned int multiplier = 1;
     unsigned int range = 0x0000;

     Init_Device();
     UART1_Init(9600);

     while(1)
     {
         if(cnt >= 18)
         {
             for(i = 0; i < 18; i++)
             {
                 if(rx_buffer[i] == 'D')
                 {
                     j = i;
                     j += 2;
                     break;
                 }
             }

             for(i = j; i < 18; i++)
             {
                 if(rx_buffer[i] == ' ')
                 {
```

```
                    k = i;
                    break;
                }
            }

            range = 0;
            multiplier = 1;
            l = ((k - j) - 1);
            for(i = 0; i < l; i++)
            {
                multiplier *= 10;
            }

            for(i = j; i < k; i++)
            {
               range += ((rx_buffer[i] - 0x30) * multiplier);
               multiplier /= 10;
            }

            d = range;
            cnt = 0x00;
        }

        delay_ms(40);
     };
}

void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}

void Timer_Init(void)
{
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}

void Port_IO_Init(void)
{
    // P0.0  -  Unassigned,  Open-Drain, Digital
    // P0.1  -  Unassigned,  Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  TX0 (UART0), Push-Pull,  Digital
    // P0.5  -  RX0 (UART0), Push-Pull,  Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDOUT = 0x30;
    P1MDOUT = 0xE0;
    P1SKIP = 0xE0;
    XBR0 = 0x01;
    XBR1 = 0x40;
}
```

```
void Oscillator_Init(void)
{
    OSCICN = 0x82;
}

void Interrupts_Init(void)
{
    IE = 0x90;
    EIE1 = 0x80;
}

void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}

void write_74HC595(unsigned char send_data)
{
    signed char clks = 0x08;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}

void segment_write(unsigned char disp, unsigned char pos)
{
    LED_LATCH = 0;
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}
```

## Schematic




## Explanation

To simplify things, the MikroC compiler's UART library is used in this example. The rest of the setup is just like all other examples. Thus, we can focus on data processing and working of the UART peripheral.

```
Init_Device();
UART1_Init(9600);
```

The digital tape provides distance in the following string format at a baud rate of 9600 bps:

**D:XXX\r\n**

Thus, the UART of C8051F330 has to read 8 bytes (7 actually and a null character) of data. Interrupt-based data acquisition is used because this method not only frees up the main loop but also addresses the fact that no one knows when and from which part of data would be transmitted and received.

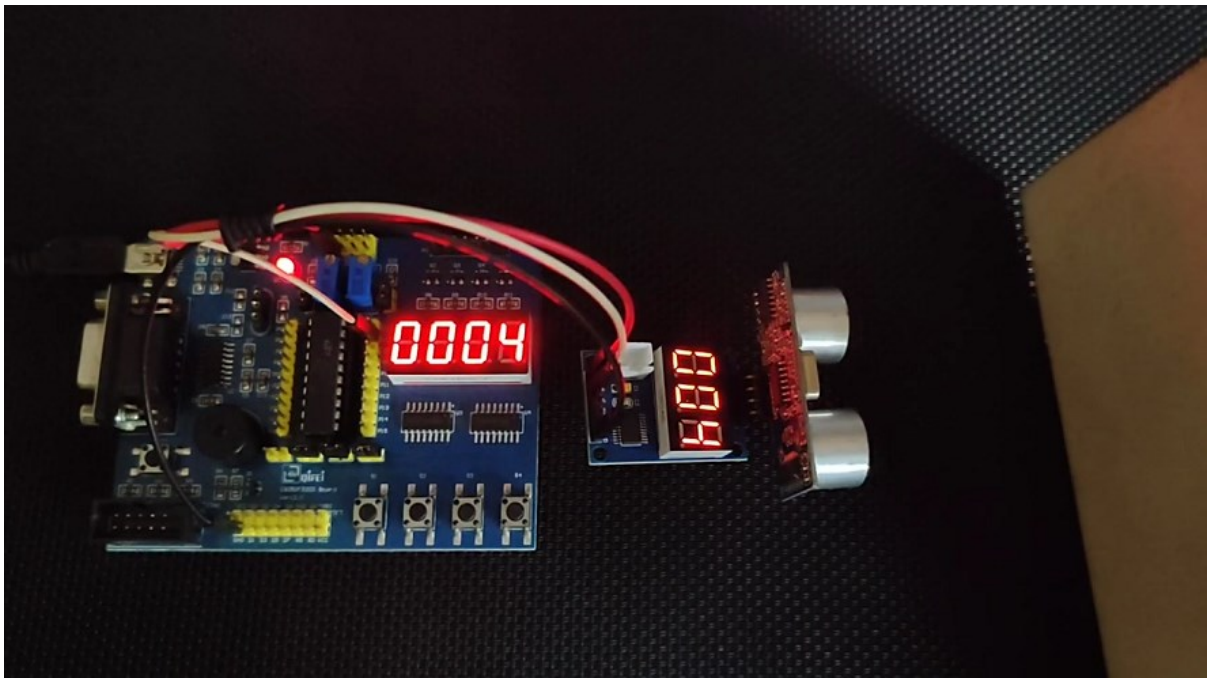
Firstly, a global data buffer is needed that would store any data received by the UART reception interrupt. This buffer should be larger than the actual number of data bytes that would be received and this is so to accommodate at least two successive transmissions from the tape. At every interrupt, the UART is read and the buffer is filled sequentially.

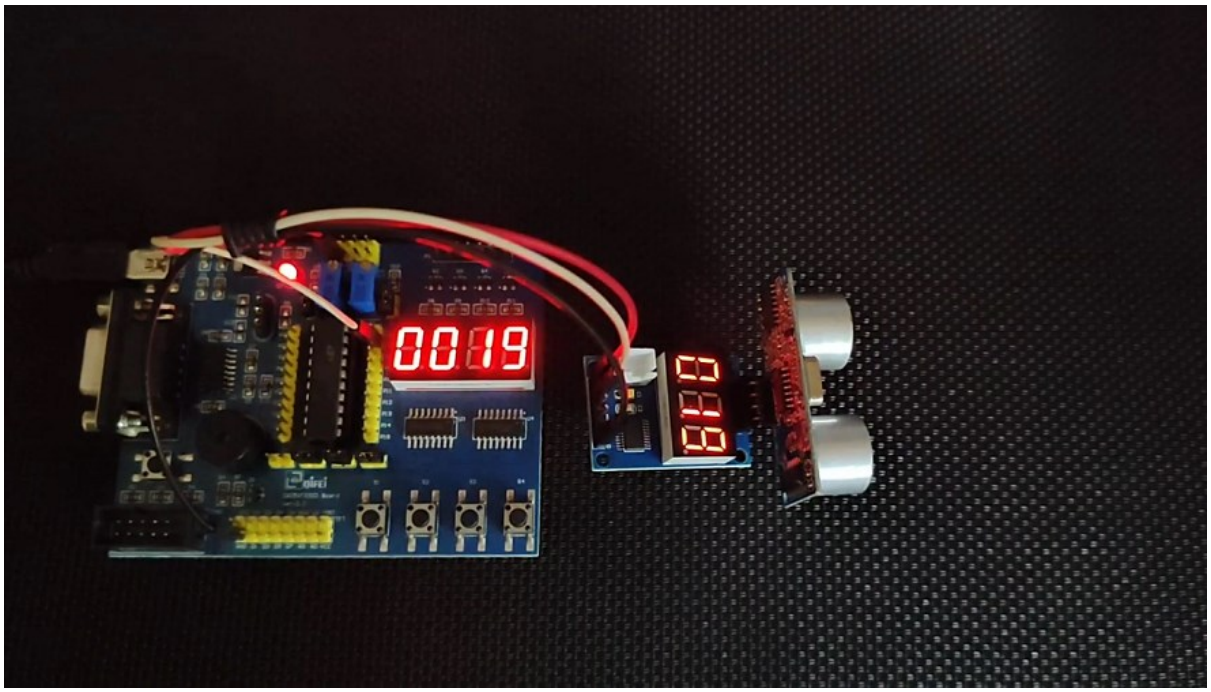
```
char rx_buffer[18];

....

void UART0_ISR(void)
iv IVT_ADDR_ES0
ilevel 0
ics ICS_AUTO
{
    rx_buffer[cnt++] = UART_Read();
    RI0_bit = 0;
}
```

When the buffer inside the interrupt is full, the code starts searching for the string of characters as per the mentioned format the tape is supposed to send. Once the character "*D*" is found, the code then searches for numeric characters before null character. The numeric characters are then converted to numbers and the distance is computed and shown on the display.

```
void main(void)
{
     unsigned char i = 0x00;
     unsigned char j = 0x00;
     unsigned char k = 0x00;
     unsigned char l = 0x00;
     unsigned int multiplier = 1;
     unsigned int range = 0x0000;

     Init_Device();
     UART1_Init(9600);

     while(1)
     {
         if(cnt >= 18)
         {
             for(i = 0; i < 18; i++)
             {
                 if(rx_buffer[i] == 'D')
                 {
                     j = i;
                     j += 2;
                     break;
                 }
             }
             for(i = j; i < 18; i++)
             {
                 if(rx_buffer[i] == ' ')
                 {
                     k = i;
                     break;
                 }
             }

             range = 0;
             multiplier = 1;
             l = ((k - j) - 1);
             for(i = 0; i < l; i++)
             {
                 multiplier *= 10;
             }
             for(i = j; i < k; i++)
             {
                range += ((rx_buffer[i] - 0x30) * multiplier);
                multiplier /= 10;
             }

             d = range;
             cnt = 0x00;
         }

         delay_ms(40);
     };
}
```

## Demo

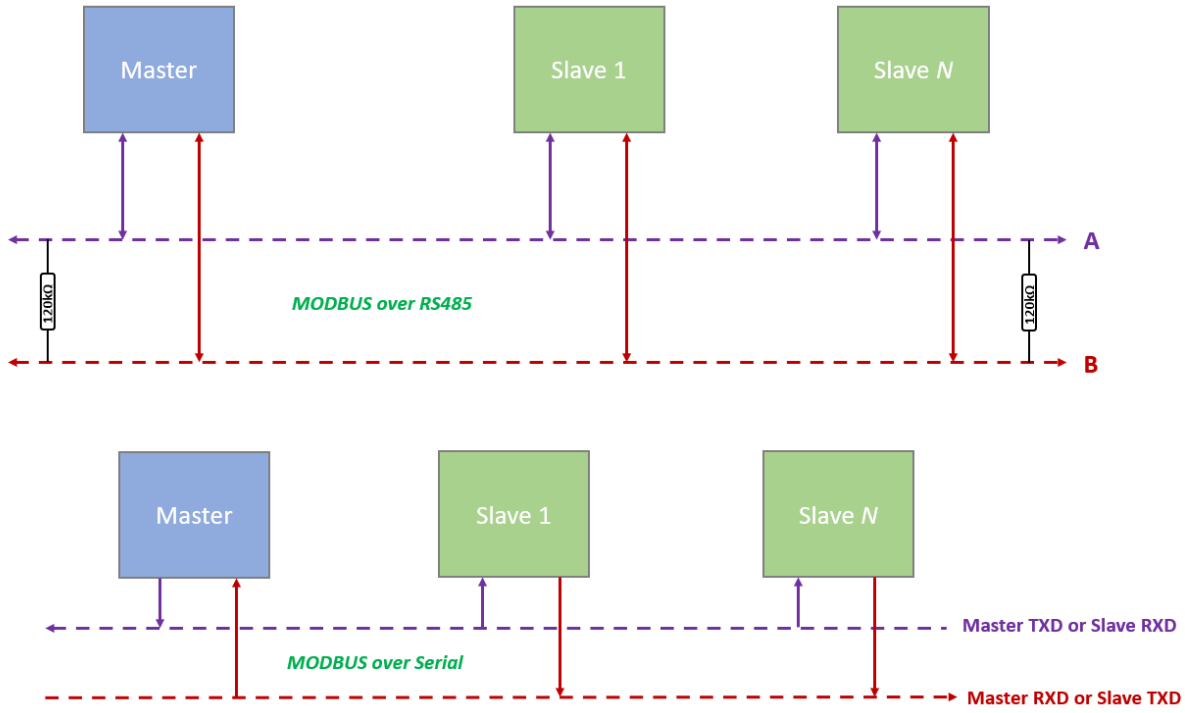



Demo video link: https://youtu.be/eZXi5qdfD3g

# MODBUS – ToF200 Distance Sensor

MODBUS RTU protocol is the industry standard for reliable serial communication among devices via RS232, RS485, etc. In MODBUS RTU, data from and to a device is sent and received in a specific packet format that encapsulates ID, function code, location address, number of bytes, data and a CRC field. If one or some fields of this encapsulation is wrong then the entire frame is discarded as garbage.




Shown above are two common types of MODBUS hardware implementation. The first one utilizes RS485 as a medium for longer range while the latter is mostly seen in short-range and onboard components. The basic MODBUS protocol is the same in both cases except for the medium.

MODBUS is itself a large topic to discuss here and so to know more about MODBUS, visit these links:

- https://en.wikipedia.org/wiki/Modbus
- https://modbus.org/docs/PI_MBUS_300.pdf
- https://modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf

‘

## Code

### ToF200.h

```
#define ToF200_TX_data_packet_size                    8
#define ToF200_RX_data_packet_size                    16

#define ToF200_slave_default_ID                       0x01

//registers
#define ToF200_special_register                       0x0001
#define ToF200_slave_ID_register                      0x0002
#define ToF200_baud_rate_register                     0x0003
#define ToF200_range_precision_register               0x0004
#define ToF200_output_control_register                0x0005
#define ToF200_load_calibration_register              0x0006
#define ToF200_offset_correction_register             0x0007
#define ToF200_xtalk_correction_register              0x0008
#define ToF200_i2c_enable_register                    0x0009
#define ToF200_measurement_register                   0x0010
#define ToF200_offset_calibration_register            0x0020
#define ToF200_xtalk_calibration_register             0x0021

//parameters
#define ToF200_restore_default                        0xAA55
#define ToF200_reboot                                 0x1000
#define ToF200_comm_test                              0x0000

#define ToF200_baud_rate_115200                       0x0000
#define ToF200_baud_rate_38400                        0x0001
#define ToF200_baud_rate_9600                         0x0002

#define ToF200_high_precision_1200mm                  0x0001
#define ToF200_medium_precision_2000mm                0x0002
#define ToF200_low_precision_1200mm                   0x0003

#define ToF200_do_not_load_calibration                0x0000
#define ToF200_load_calibration                       0x0001

#define ToF200_i2c_not_prohibited                     0x0000
#define ToF200_i2c_prohibited                         0x0001

//other constant parameters
#define ToF200_default_max_distance                   2000

#define MODBUS_read_holding_registers_function_code   0x03
#define MODBUS_write_single_register_function_code    0x06


unsigned char cnt;
unsigned char rx_buffer[ToF200_RX_data_packet_size];


unsigned int make_word(unsigned char HB, unsigned char LB);
void get_HB_LB(unsigned int value, unsigned char *HB, unsigned char *LB);
unsigned int MODBUS_RTU_CRC16(unsigned char *data_input, unsigned char data_length);
void flush_RX_buffer(void);
void MODBUS_TX(unsigned char slave_ID, unsigned char function_code, unsigned char reg,
unsigned char value);
unsigned int ToF_get_range(void);
```

ToF200.c

```
#include "ToF200.h"


unsigned int make_word(unsigned char HB, unsigned char LB)
{
    unsigned int tmp = 0;

    tmp = HB;
    tmp <<= 8;
    tmp |= LB;

    return tmp;
}


void get_HB_LB(unsigned int value, unsigned char *HB, unsigned char *LB)
{
    *LB = (unsigned char)(value & 0x00FF);
    *HB = (unsigned char)((value & 0xFF00) >> 8);
}


unsigned int MODBUS_RTU_CRC16(unsigned char *data_input, unsigned char data_length)
{
    unsigned char n = 8;
    unsigned char s = 0;
    unsigned int CRC_word = 0xFFFF;

    for(s = 0; s < data_length; s++)
    {
      CRC_word ^= ((unsigned int)data_input[s]);

      n = 8;
      while(n > 0)
      {
        if((CRC_word & 0x0001) == 0)
        {
          CRC_word >>= 1;
        }

        else
        {
          CRC_word >>= 1;
          CRC_word ^= 0xA001;
        }

        n--;
      }
    }

    return CRC_word;
}


void flush_RX_buffer(void)
{
    signed char s = (ToF200_TX_data_packet_size - 1);

    while(s > -1)
    {
        rx_buffer[s] = 0x00;
        s--;
```

```
    };
}


void MODBUS_TX(unsigned char slave_ID, unsigned char function_code, unsigned char reg,
unsigned char value)
{
    unsigned char i  = 0x00;
    unsigned char lb = 0x00;
    unsigned char hb = 0x00;
    unsigned int temp = 0x0000;

    unsigned char tx_buffer[ToF200_TX_data_packet_size];

    tx_buffer[0x00] = slave_ID;
    tx_buffer[0x01] = function_code;

    get_HB_LB(reg, &hb, &lb);

    tx_buffer[0x02] = hb;
    tx_buffer[0x03] = lb;

    get_HB_LB(value, &hb, &lb);

    tx_buffer[0x04] = hb;
    tx_buffer[0x05] = lb;

    temp = MODBUS_RTU_CRC16(tx_buffer, 6);
    get_HB_LB(temp, &hb, &lb);

    tx_buffer[0x06] = lb;
    tx_buffer[0x07] = hb;

    flush_RX_buffer();

    for(i = 0; i < ToF200_TX_data_packet_size; i++)
    {
        UART_Write(tx_buffer[i]);
    }

    cnt = 0x00;
    delay_ms(40);
}


unsigned int ToF_get_range(void)
{
    unsigned int CRC_1 = 0x0000;
    unsigned int CRC_2 = 0x0000;

    unsigned int distance = 5000;

    MODBUS_TX(ToF200_slave_default_ID,
              MODBUS_read_holding_registers_function_code,
              ToF200_measurement_register,
              0x0001);

    if(rx_buffer[0x00] == ToF200_slave_default_ID)
    {
        if(rx_buffer[0x01] == MODBUS_read_holding_registers_function_code)
        {
            if(rx_buffer[0x02] == 0x02)
            {
                CRC_1 = MODBUS_RTU_CRC16(rx_buffer, 5);
                CRC_2 = make_word(rx_buffer[0x06], rx_buffer[0x05]);
```

```
                if(CRC_1 == CRC_2)
                {
                    distance = make_word(rx_buffer[0x03], rx_buffer[0x04]);
                    if(distance > ToF200_default_max_distance)
                    {
                        distance = 40000;
                    }
                }

                else
                {
                    distance = 40000;
                }
            }
        }
    }

    return distance;
}
```

main.c

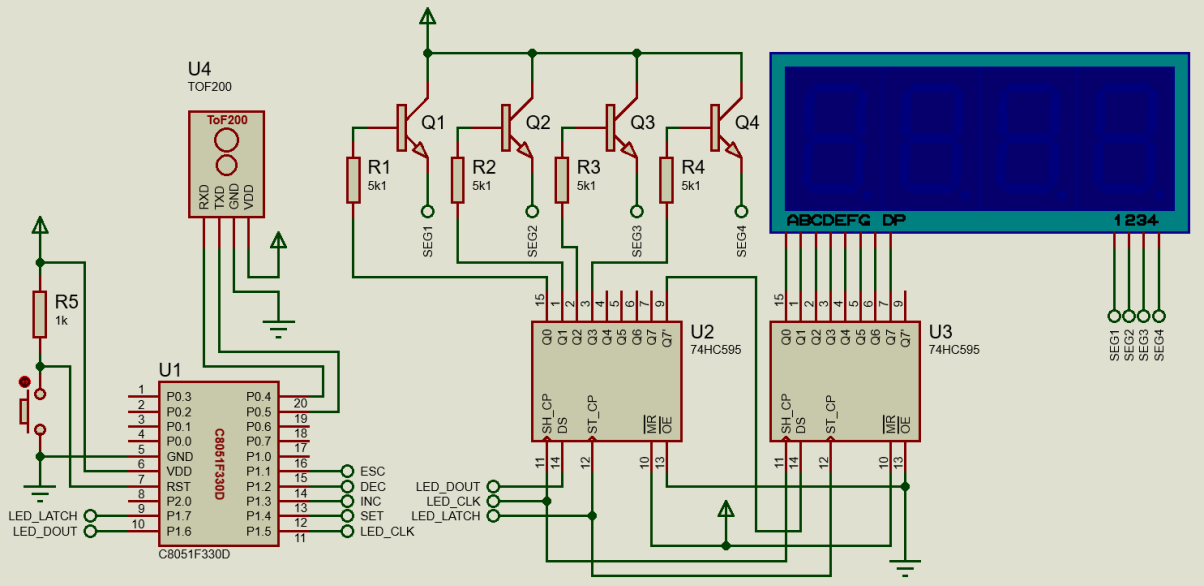
```
#include "ToF200.c"


#define LED_DOUT P1_6_bit
#define  LED_CLK P1_5_bit
#define  LED_LATCH P1_7_bit


unsigned int d = 0;
unsigned char i = 0;
unsigned char value = 0;


const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x7F, // .
  0xBF  // -
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


void PCA_Init(void);
void Timer_Init(void);
```

```
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);


void UART0_ISR(void)
iv IVT_ADDR_ES0
ilevel 0
ics ICS_AUTO
{
    rx_buffer[cnt++] = UART_Read();
    RI0_bit = 0;
}


void Timer_ISR(void)
iv IVT_ADDR_ET3
ilevel 1
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            value = (d / 1000);
            break;
        }
        case 1:
        {
            value = ((d % 1000) / 100);
            break;
        }
        case 2:
        {
            value = ((d % 100) / 10);
            break;
        }
        case 3:
        {
            value = (d % 10);
            break;
        }
    }

    if(d >= 40000)
    {
        segment_write(11, i);
    }
    else
    {
        segment_write(value, i);
    }

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}
```

```
void main(void)
{
     Init_Device();

     while(1)
     {
         d = ((float)ToF_get_range());
         delay_ms(400);
     };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}


void Timer_Init(void)
{
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}


void Port_IO_Init(void)
{
    // P0.0  -  Unassigned,  Open-Drain, Digital
    // P0.1  -  Unassigned,  Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  TX0 (UART0), Push-Pull,  Digital
    // P0.5  -  RX0 (UART0), Push-Pull,  Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDOUT = 0x30;
    P1MDOUT = 0xE0;
    P1SKIP = 0xE0;
    XBR0 = 0x01;
    XBR1 = 0x40;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
{
```

```
    IE = 0x90;
    EIE1 = 0x80;
}


void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
    UART1_Init(115200);
}


void write_74HC595(unsigned char send_data)
{
    signed char clks = 0x08;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}


void segment_write(unsigned char disp, unsigned char pos)
{
    LED_LATCH = 0;
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}
```

## Schematic




## Explanation

In this example, a ToF200 IR rangefinder sensor is used. This sensor communicates via MODBUS RTU frames over UART medium. Therefore, we would need to set up the UART peripheral and UART reception interrupt.

```
void Port_IO_Init(void)
{
    // P0.4  -  TX0 (UART0), Push-Pull,  Digital
    // P0.5  -  RX0 (UART0), Push-Pull,  Digital

    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDOUT = 0x30;
    P1MDOUT = 0xE0;
    P1SKIP = 0xE0;
    XBR0 = 0x01;
    XBR1 = 0x40;
}

void Interrupts_Init(void)
{
    IE = 0x90;
    EIE1 = 0x80;
}

void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
    UART1_Init(115200);
}
```

The *ToF200.h* header file describes the functions, registers and parameter values. The codes in *ToF200.c* source file is what we need to understand. The following functions are typical data manipulation functions. Of these, the *MODBUS_RTU_CRC16* function is a bit critical because it generates CRC value when MODBUS frame data is passed to it. This function has been described by MODBUS documentation and so we can simply avoid trying to recreate it on our own.

```
unsigned int make_word(unsigned char HB, unsigned char LB);
void get_HB_LB(unsigned int value, unsigned char *HB, unsigned char *LB);
unsigned int MODBUS_RTU_CRC16(unsigned char *data_input, unsigned char data_length);
void flush_RX_buffer(void);
```

The sensor will send out data when it is asked to return distance. To do so, we have to use the following function.

```
void MODBUS_TX(unsigned char slave_ID, unsigned char function_code, unsigned char reg,
unsigned char value)
{
    unsigned char i  = 0x00;
    unsigned char lb = 0x00;
    unsigned char hb = 0x00;
    unsigned int temp = 0x0000;

    unsigned char tx_buffer[ToF200_TX_data_packet_size];

    tx_buffer[0x00] = slave_ID;
    tx_buffer[0x01] = function_code;

    get_HB_LB(reg, &hb, &lb);
    tx_buffer[0x02] = hb;
    tx_buffer[0x03] = lb;

    get_HB_LB(value, &hb, &lb);
    tx_buffer[0x04] = hb;
    tx_buffer[0x05] = lb;

    temp = MODBUS_RTU_CRC16(tx_buffer, 6);
    get_HB_LB(temp, &hb, &lb);
    tx_buffer[0x06] = lb;
    tx_buffer[0x07] = hb;

    flush_RX_buffer();

    for(i = 0; i < ToF200_TX_data_packet_size; i++)
    {
        UART_Write(tx_buffer[i]);
    }

    cnt = 0x00;
    delay_ms(40);
}
```

What this function does is simply transmit data from our C8051F330D microcontroller to the ToF200 sensor in the following MODBUS RTU frame format. The first byte is the slave ID followed by the holding register read function, the register to be accessed, data length and CRC high and low bytes computed from these values. Before sending these bytes, the reception buffer array, *rx_buffer* which would hold the data from the sensor is flushed or cleared of any previous value and the byte counter variable *cnt* is set to zero.

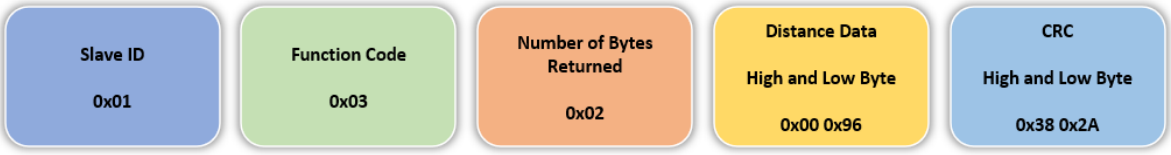

The data received from the sensor is read using the UART RX interrupt. Whenever a new byte is received by the microcontroller's UART RX pin, an interrupt is triggered. The received bytes are stored in a buffer array.

```
void UART0_ISR(void)
iv IVT_ADDR_ES0
ilevel 0
ics ICS_AUTO
{
    rx_buffer[cnt++] = UART_Read();
    RI0_bit = 0;
}
```

The received data buffer array is then processed to compute distance. The following function returns the range of objects detected by the sensor.

```
unsigned int ToF_get_range(void)
{
    unsigned int CRC_1 = 0x0000;
    unsigned int CRC_2 = 0x0000;
    unsigned int distance = 5000;

    MODBUS_TX(ToF200_slave_default_ID,
              MODBUS_read_holding_registers_function_code,
              ToF200_measurement_register,
              0x0001);

    if(rx_buffer[0x00] == ToF200_slave_default_ID)
    {
        if(rx_buffer[0x01] == MODBUS_read_holding_registers_function_code)
        {
            if(rx_buffer[0x02] == 0x02)
            {
                CRC_1 = MODBUS_RTU_CRC16(rx_buffer, 5);
                CRC_2 = make_word(rx_buffer[0x06], rx_buffer[0x05]);
                if(CRC_1 == CRC_2)
                {
                    distance = make_word(rx_buffer[0x03], rx_buffer[0x04]);
                    if(distance > ToF200_default_max_distance)
                    {
                        distance = 40000;
                    }
                }

                else
                {
                    distance = 40000;
                }
            }
        }
    }

    return distance;
}
```

Firstly, the function transmits commands to the sensor to return range data. After that, the sensor begins transmitting range data and thus, UART RX interrupt triggers. Data is likely to be received in the following frame format.

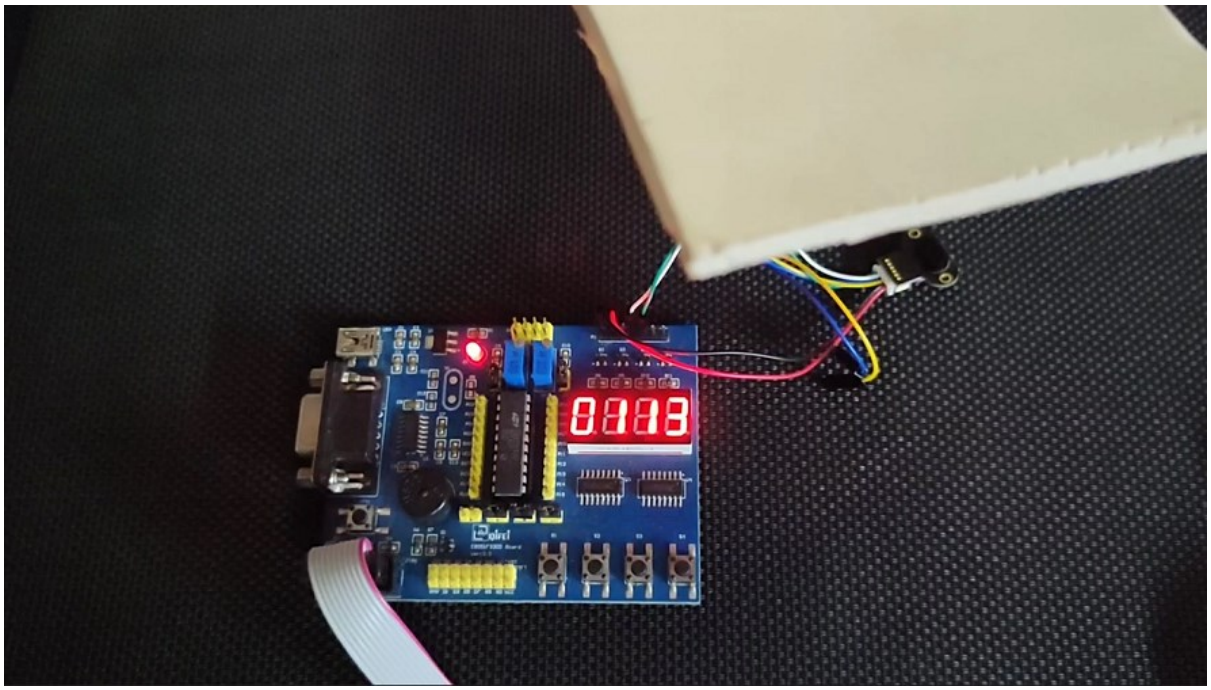



Inside the *ToF_get_range* function, slave ID, function code and the number of returned bytes are checked against what they should be. If these are okay, the received CRC value is checked against the computed CRC value. If both of these values match then the distance value is read and returned.

Inside the main loop, the distance is read every 400ms and the read data is shown on the onboard seven-segment LED display.

```
void main(void)
{
     Init_Device();

     while(1)
     {
         d = ((float)ToF_get_range());
         delay_ms(400);
     };
}
```

## Demo



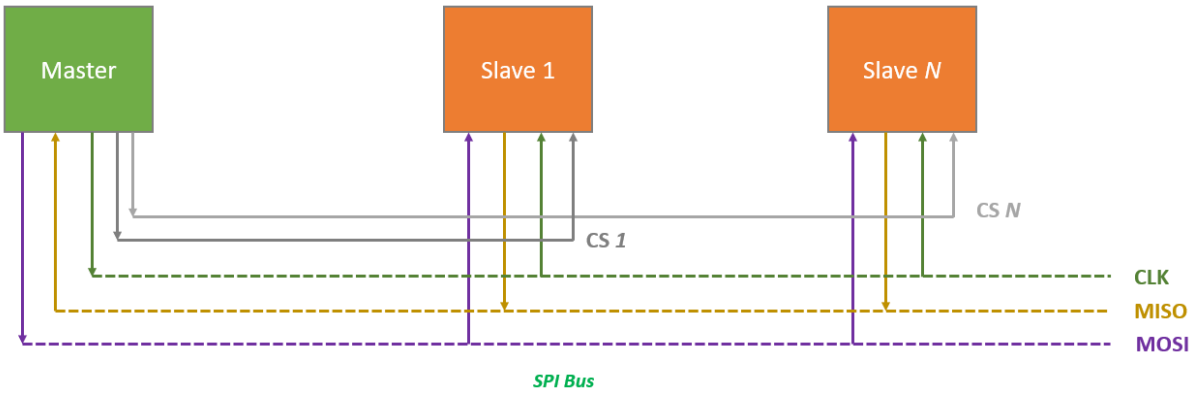
Demo video link: https://youtu.be/fUFDKg7su7A

# SPI – MAX31865 PT100 RTD Amplifier

Serial Peripheral Interface (SPI) communication is a multi-wire high-speed short-distance synchronous communication method. Three (half-duplex) or four (full-duplex) pins are all that are needed to establish communication between a host (master) and a slave. These pins are as follows:

- ***Master-In-Slave-Out (MISO)*** connected to ***Slave-Data-Out (SDO)***.
- ***Master-Out-Slave-In (MOSI)*** connected to ***Slave-Data-In (SDI)***.
- ***Serial Clock (SCLK)*** connected to ***Slave Clock (SCK)***.
- ***Slave Select (SS)*** connected to ***Chip Select (CS)***.


Multiple slaves can be present in a common SPI bus but their chip-select pins have to be different. This is where SPI communication falls behind UART and I2C communications. However, the raw speed advantage makes SPI the number one choice for displays, flash memories, sensors, etc.




To know more about SPI bus, the following links are very informative:

- https://learn.mikroe.com/spi-bus
- https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi
- http://ww1.microchip.com/downloads/en/devicedoc/spi.pdf
- http://tronixstuff.com/2011/05/13/tutorial-arduino-and-the-spi-bus
- https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html
- http://www.circuitbasics.com/basics-of-the-spi-communication-protocol

## Code

### MAX31865.h

```
#define MAX31865_CONFIG_REG                 0x00
#define MAX31865_RTD_MSB_REG                0x01
#define MAX31865_RTD_LSB_REG                0x02
#define MAX31865_HFAULT_MSB_REG             0x03
#define MAX31865_HFAULT_LSB_REG             0x04
#define MAX31865_LFAULT_MSB_REG             0x05
#define MAX31865_LFAULT_LSB_REG             0x06
#define MAX31865_FAULT_STATUS_REG           0x07


/* Configuration Definitions */
#define MAX31865_CONFIG_BIAS                0x80
#define MAX31865_CONFIG_MODE_AUTO           0x40
#define MAX31865_CONFIG_MODE_OFF            0x00
#define MAX31865_CONFIG_1SHOT               0x20
#define MAX31865_CONFIG_3_WIRE              0x10
#define MAX31865_CONFIG_24_WIRE             0x00
#define MAX31865_CONFIG_FAULT_STATUS        0x02
#define MAX31865_CONFIG_FILTER_50Hz         0x01
#define MAX31865_CONFIG_FILTER_60Hz         0x00

/* Fault Definitions */
#define MAX31865_FAULT_HIGH_THRESHOLD       0x80
#define MAX31865_FAULT_LOW_THRESHOLD        0x40
#define MAX31865_FAULT_REF_IN_LOW           0x20
#define MAX31865_FAULT_REF_IN_HIGH          0x10
#define MAX31865_FAULT_RTD_IN_LOW           0x08
#define MAX31865_FAULT_OV_UV                0x04

#define MAX31865_RTD_A                      0.00390803
#define MAX31865_RTD_B                      -0.000000577
#define MAX31865_Reference_Resistance       430.0
#define MAX31865_RTD_Nominal_Value          100.0 // for PT100

#define MAX31865_CS                         P0_3_bit


void MAX31865_init(void);
unsigned char MAX31865_read_byte(unsigned char address);
unsigned int MAX31865_read_word(unsigned char address);
void MAX31865_write_byte(unsigned char address, unsigned char value);
void MAX31865_write_word(unsigned char address, unsigned char lb, unsigned char hb);
unsigned int MAX31865_get_RTD(void);
signed int MAX31865_get_temperature(void);
```

### MAX31865.c

```
#include "MAX31865.h"


void MAX31865_init(void)
{
    SPI1_Init_Advanced(1000000, _SPI_CLK_IDLE_LO | _SPI_CLK_ACTIVE_2_IDLE | _SPI_MASTER);
    delay_ms(10);

    MAX31865_CS = 1;

    MAX31865_write_byte(MAX31865_CONFIG_REG, \
```

```
                        MAX31865_CONFIG_BIAS | \
                        MAX31865_CONFIG_MODE_AUTO | \
                        MAX31865_CONFIG_3_WIRE | \
                        MAX31865_CONFIG_FAULT_STATUS | \
                        MAX31865_CONFIG_FILTER_50Hz);
}


unsigned char MAX31865_read_byte(unsigned char address)
{
    unsigned char retval = 0x00;

    MAX31865_CS = 0;
    SPI_Write(address & 0x7F);
    retval = SPI_Read(0x00);
    MAX31865_CS = 1;

    return retval;
}


unsigned int MAX31865_read_word(unsigned char address)
{
    unsigned char lb = 0x00;
    unsigned char hb = 0x00;
    unsigned int retval = 0x0000;

    hb = MAX31865_read_byte(address);
    lb = MAX31865_read_byte(address + 1);

    retval = hb;
    retval <<= 0x08;
    retval |= lb;

    return retval;
}


void MAX31865_write_byte(unsigned char address, unsigned char value)
{
    MAX31865_CS = 0;
    SPI_Write(address | 0x80);
    SPI_Write(value);
    MAX31865_CS = 1;
}


void MAX31865_write_word(unsigned char address, unsigned char lb, unsigned char hb)
{
    MAX31865_write_byte(address, hb);
    MAX31865_write_byte((address + 1), lb);
}


unsigned int MAX31865_get_RTD(void)
{
  unsigned int rtd_value = 0x00;

  rtd_value = MAX31865_read_word(MAX31865_RTD_MSB_REG);
  rtd_value >>= 1;

  return rtd_value;
}
```

```
signed int MAX31865_get_temperature(void)
{
    float rt = 0.0;
    signed int t_value = 0;

    t_value = MAX31865_get_RTD();
    rt = (MAX31865_Reference_Resistance * t_value);
    rt /= 32768.0;

    rt /= MAX31865_RTD_Nominal_Value;
    rt = (rt - 1.0);
    t_value = (rt / MAX31865_RTD_A);

    return t_value;
}
```

main.c

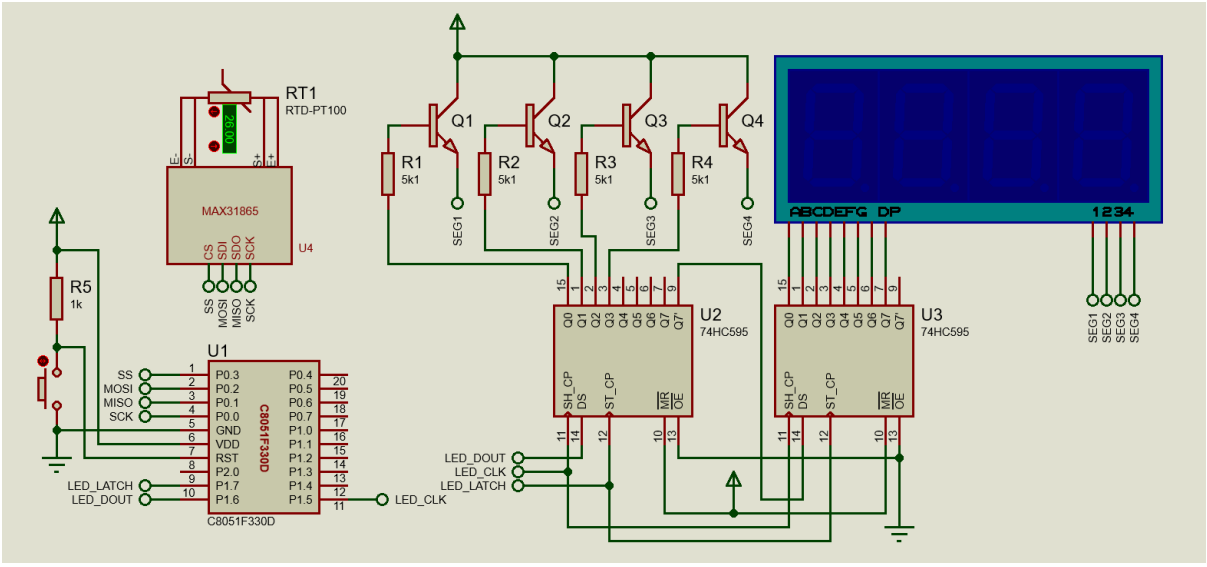
```
#include "MAX31865.c"


#define LED_DOUT          P1_6_bit
#define LED_CLK           P1_5_bit
#define LED_LATCH         P1_7_bit


unsigned char i = 0;
register unsigned char val = 0;
unsigned int value = 0;

const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x9C, // degree
  0xC6  // C
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


void PCA_Init(void);
void Timer_Init(void);
void SPI_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);
```

```
void Timer_ISR()
iv IVT_ADDR_ET3
ilevel 1
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (value / 10);
            break;
        }
        case 1:
        {
            val = (value % 10);
            break;
        }
        case 2:
        {
            val = 10;
            break;
        }
        case 3:
        {
            val = 11;
            break;
        }
    }

    segment_write(val, i);

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}


void main(void)
{
    Init_Device();
    MAX31865_init();

    while(1)
    {
        value = MAX31865_get_temperature();
        delay_ms(600);
    };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}


void Timer_Init(void)
```

```
{
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}


void SPI_Init(void)
{
    SPI0CFG = 0x40;
    SPI0CN = 0x01;
    SPI0CKR = 0x05;
}


void Port_IO_Init(void)
{
    // P0.0  -  SCK  (SPI0), Push-Pull,  Digital
    // P0.1  -  MISO (SPI0), Open-Drain, Digital
    // P0.2  -  MOSI (SPI0), Push-Pull,  Digital
    // P0.3  -  Skipped,     Push-Pull,  Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDOUT = 0x0D;
    P1MDOUT = 0xE0;
    P0SKIP = 0x08;
    P1SKIP = 0xE0;
    XBR0 = 0x02;
    XBR1 = 0x40;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
{
    IE = 0x80;
    EIE1 = 0x80;
}


void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    SPI_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
```

```
}


void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}


void segment_write(unsigned char disp, unsigned char pos)
{
    LED_LATCH = 0;
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}
```

## Schematic



## Explanation

MAX31865 is a PT100-to-digital converter. It converts resistance data from a PT100 RTD sensor to digital temperature data that is sent to a host microcontroller via an SPI bus.

Firstly, the system clock is set to 12.25MHz.

```
void Oscillator_Init(void)
{
    OSCICN = 0x82;
}
```

Pins P0.0 to P0.3 are set as SPI pins in the crossbar.

```
void Port_IO_Init(void)
{
    // P0.0  -  SCK  (SPI0), Push-Pull,  Digital
    // P0.1  -  MISO (SPI0), Open-Drain, Digital
    // P0.2  -  MOSI (SPI0), Push-Pull,  Digital
    // P0.3  -  Skipped,     Push-Pull,  Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDOUT = 0x0D;
    P1MDOUT = 0xE0;
    P0SKIP = 0x08;
    P1SKIP = 0xE0;
    XBR0 = 0x02;
    XBR1 = 0x40;
}
```

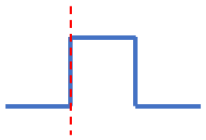
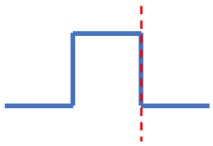
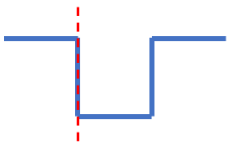
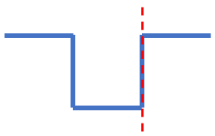
MAX31865 header file contains all the constants and function prototypes that would be eventually needed in the MAX31865 source file.

Now let's look at what's inside the source file. The first thing in the MAX31865 source file is the initialization of the MAX31865 along with the initialization of the SPI peripheral of the host C8051F330. The built-in SPI library of MikroC is used to do the peripheral hardware initialization.

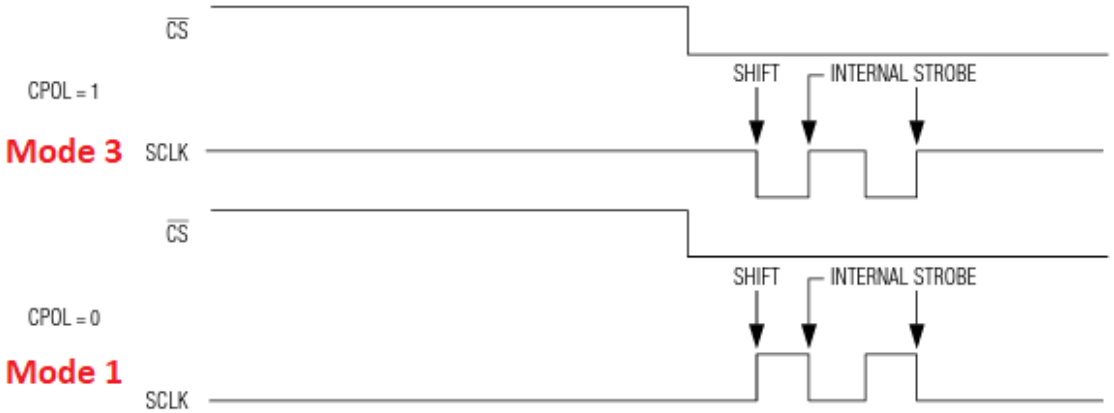
```
void MAX31865_init(void)
{
    SPI1_Init_Advanced(1000000, _SPI_CLK_IDLE_LO | _SPI_CLK_ACTIVE_2_IDLE | _SPI_MASTER);
    delay_ms(10);

    MAX31865_CS = 1;

    MAX31865_write_byte(MAX31865_CONFIG_REG, \
                        MAX31865_CONFIG_BIAS | \
                        MAX31865_CONFIG_MODE_AUTO | \
```

```
                    MAX31865_CONFIG_3_WIRE | \
                    MAX31865_CONFIG_FAULT_STATUS | \
                    MAX31865_CONFIG_FILTER_50Hz);
}
```

The SPI protocol defines four different modes of operation, also known as SPI bus modes. The modes differ in terms of the clock polarity and clock phase and the timing of the data signals. The four SPI bus modes are as follows:

| Mode | CPOL | CPHA | Description | Clock Signal Representation |
|---|---|---|---|---|
| 0 | 0 | 0 | The idle state of clock signal is low and data is sampled on the leading or first edge of the clock. This is the most commonly used mode. |  |
| 1 | 0 | 1 | The idle state of clock signal is low and data is sampled on the trailing or second edge of the clock. |  |
| 2 | 1 | 0 | The idle state of clock signal is high and data is sampled on the first edge of the clock. |  |
| 3 | 1 | 1 | The idle state of clock signal is high and data is sampled on the second edge of the clock. |  |

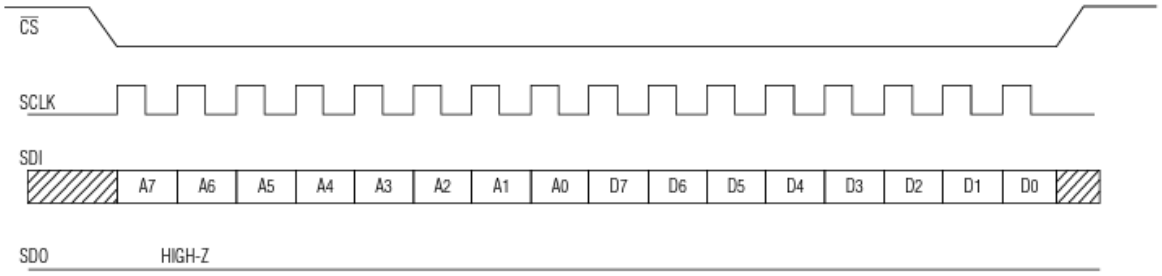
As per the datasheet of MAX31865, the chip supports both Mode 1 and Mode 3 SPI bus modes and so, here, Mode 1 is used.



The next important functions are the SPI byte read and write functions.

```
void MAX31865_write_byte(unsigned char address, unsigned char value)
{
    MAX31865_CS = 0;
    SPI_Write(address | 0x80);
    SPI_Write(value);
    MAX31865_CS = 1;
}
```

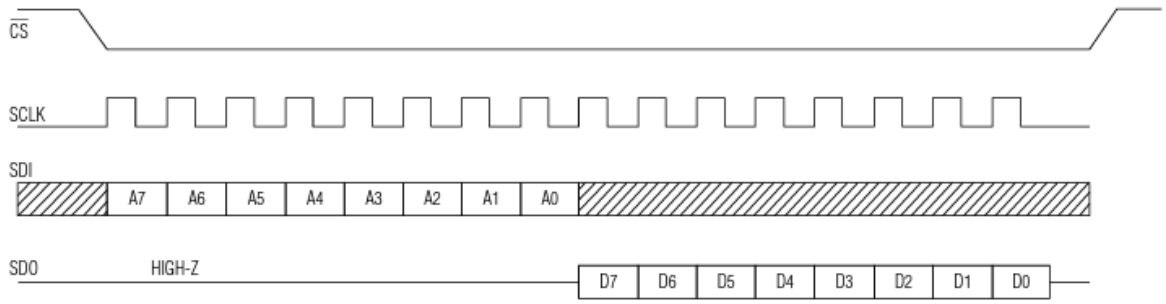
The write function is very simple. Writing to the chip is initiated by holding the chip/slave select pin low and sending the address and data bytes, i.e., two bytes are sent. Once the transaction is completed, the chip select pin is held high. During this process, no data reading is done. The polarity and phase of SPI signals are maintained automatically. The timing diagram for the SPI write is shown below and it clearly shows what has been described.




Reading the chip is similar to write process. The only difference that exists is the reading process itself. During an SPI bus read, a dummy byte is sent to the chip. The following function is for SPI byte read:

```
unsigned char MAX31865_read_byte(unsigned char address)
{
    unsigned char retval = 0x00;

    MAX31865_CS = 0;
    SPI_Write(address & 0x7F);
    retval = SPI_Read(0x00);
    MAX31865_CS = 1;

    return retval;
}
```

The timing diagram for the SPI bus read matches the code given above.




The rest of the code is all about collecting and processing RTD data as per the datasheet.

Just like byte read and write, there are functions for word-sized read and write. These functions utilize the byte read and write functions.

```
unsigned int MAX31865_read_word(unsigned char address)
{
    unsigned char lb = 0x00;
    unsigned char hb = 0x00;
    unsigned int retval = 0x0000;

    hb = MAX31865_read_byte(address);
    lb = MAX31865_read_byte(address + 1);

    retval = hb;
    retval <<= 0x08;
    retval |= lb;

    return retval;
}

....

void MAX31865_write_word(unsigned char address, unsigned char lb, unsigned char hb)
{
    MAX31865_write_byte(address, hb);
    MAX31865_write_byte((address + 1), lb);
}
```

Going further, there is a function to read raw data from the RTD sensor.

```
unsigned int MAX31865_get_RTD(void)
{
  unsigned int rtd_value = 0x00;

  rtd_value = MAX31865_read_word(MAX31865_RTD_MSB_REG);
  rtd_value >>= 1;

  return rtd_value;
}
```

As per the datasheet, the raw RTD data is 15-bits wide and so a word read operation is needed.

| REGISTER | RTD MSBS (01h) REGISTER | | | | | | | | RTD LSBS (02h) REGISTER | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| RTD Resistance Data | MSB | — | — | — | — | — | — | — | — | — | — | — | — | — | LSB | Fault |
| Bit Weighting | $2^{14}$ | $2^{13}$ | $2^{12}$ | $2^{11}$ | $2^{10}$ | $2^{9}$ | $2^{8}$ | $2^{7}$ | $2^{6}$ | $2^{5}$ | $2^{4}$ | $2^{3}$ | $2^{2}$ | $2^{1}$ | $2^{0}$ | — |
| Decimal Value | 16384 | 8192 | 4096 | 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | — |

With the raw RTD data extracted, we can calculate RTD resistance and temperature after further data processing. This is done in the following function:

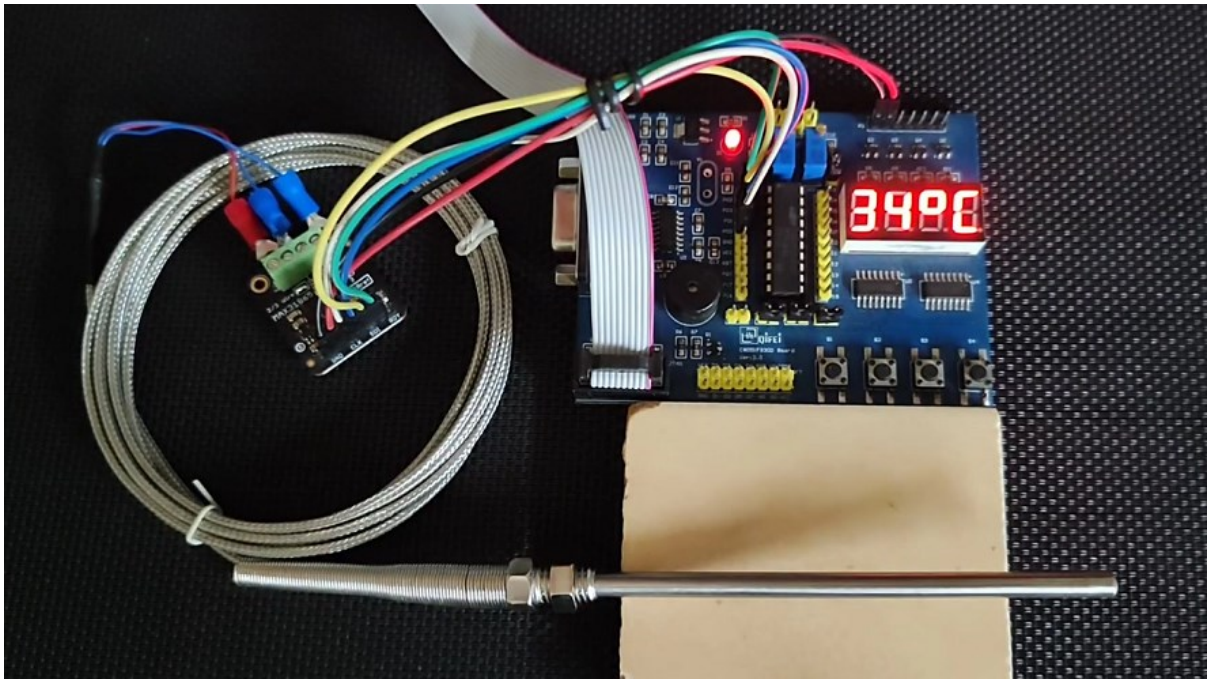
```
signed int MAX31865_get_temperature(void)
{
    float rt = 0.0;
    signed int t_value = 0;
```

```
    t_value = MAX31865_get_RTD();
    rt = (MAX31865_Reference_Resistance * t_value);
    rt /= 32768.0;

    rt /= MAX31865_RTD_Nominal_Value;
    rt = (rt - 1.0);
    t_value = (rt / MAX31865_RTD_A);

    return t_value;
}
```

Inside the main, the temperature read by the chip is displayed on the on board seven-segment display.

```
void main(void)
{
    Init_Device();
    MAX31865_init();

    while(1)
    {
        value = MAX31865_get_temperature();
        delay_ms(600);
    };
}
```

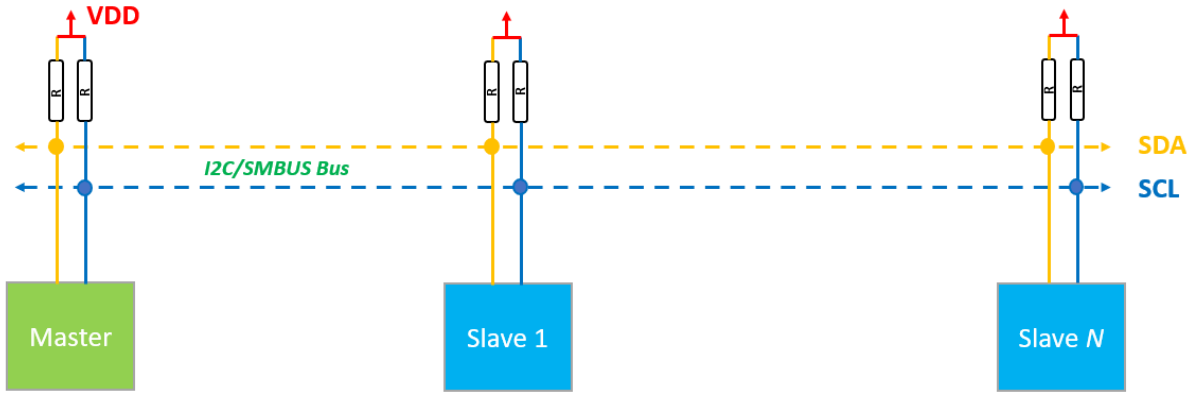
## Demo



Demo video link: https://youtu.be/xW0eEuduWUE

# I2C/SMBUS – HMC5883L Digital Compass

Going through the datasheet of C805F330D, we will not find any I2C hardware present in it. The same applies to the MikroC compiler. However, there is an SMBus hardware present. SMBus and I2C are compatible with each other, the only differences being the maximum bus speed and the timeout feature. Both I2C and SMBus are two-wire interfaces. Thus, whenever we need to hook up I2C devices with our C8051 microcontrollers, we can use SMBus without any issues.




There are tons of articles dedicated to I2C and SMBus protocols and so to know more about them the following links are very informative:

- https://learn.mikroe.com/i2c-everything-need-know
- https://learn.sparkfun.com/tutorials/i2c
- http://www.ti.com/lsds/ti/interface/i2c-overview.page
- http://www.robot-electronics.co.uk/i2c-tutorial
- https://www.i2c-bus.org/i2c-bus
- http://i2c.info
- https://www.totalphase.com/support/articles/200349186-Differences-between-I2C-and-SMBus
- https://www.nxp.com/docs/en/application-note/AN4471.pdf
- https://www.i2c-bus.org/smbus/

## Code

### HMC5883L.h

```
#define HMC5883L_READ_ADDR       0x3D
#define HMC5883L_WRITE_ADDR      0x3C

#define Config_Reg_A             0x00
#define Config_Reg_B             0x01
#define Mode_Reg                 0x02
#define X_MSB_Reg                0x03
#define X_LSB_Reg                0x04
#define Z_MSB_Reg                0x05
#define Z_LSB_Reg                0x06
#define Y_MSB_Reg                0x07
#define Y_LSB_Reg                0x08
#define Status_Reg               0x09
#define ID_Reg_A                 0x0A
#define ID_Reg_B                 0x0B
#define ID_Reg_C                 0x0C

#define PI                      3.142
#define declination_angle     -0.5167        // For Uttara, Dhaka, Bangladesh


signed int X_axis = 0;
signed int Y_axis = 0;
signed int Z_axis = 0;
float m_scale = 1.0;


unsigned int make_word(unsigned char HB, unsigned char LB);
void HMC5883L_init(void);
unsigned char HMC5883L_read(unsigned char reg);
void HMC5883L_write(unsigned char reg_address, unsigned char value);
void HMC5883L_read_data(void);
void HMC5883L_scale_axes(void);
void HMC5883L_set_scale(float gauss);
float HMC5883L_heading(void);
```

### HMC5883L.c

```
#include "HMC5883L.h"


unsigned int make_word(unsigned char HB, unsigned char LB)
{
    unsigned int val = 0;

    val = HB;
    val <<= 8;
    val |= LB;

    return val;
}


void HMC5883L_init(void)
{
    SMBus1_Init(20000);
    HMC5883L_write(Config_Reg_A, 0x70);
    HMC5883L_write(Config_Reg_B, 0xA0);
```

```
    HMC5883L_write(Mode_Reg, 0x00);
    HMC5883L_set_scale(1.3);
}


unsigned char HMC5883L_read(unsigned char reg)
{
    unsigned char val = 0;

    SMBus1_Start();
    SMBus1_Write(HMC5883L_WRITE_ADDR);
    SMBus1_Write(reg);
    SMBus1_Repeated_Start();
    SMBus1_Write(HMC5883L_READ_ADDR);
    val = SMBus1_Read(0);
    SMBus1_Stop();

    return(val);
}


void HMC5883L_write(unsigned char reg_address, unsigned char value)
{
    SMBus1_Start();
    SMBus1_Write(HMC5883L_WRITE_ADDR);
    SMBus1_Write(reg_address);
    SMBus1_Write(value);
    SMBus1_Stop();
}


void HMC5883L_read_data(void)
{
    unsigned char lsb = 0;
    unsigned char msb = 0;

    SMBus1_Start();
    SMBus1_Write(HMC5883L_WRITE_ADDR);
    SMBus1_Write(X_MSB_Reg);
    SMBus1_Repeated_Start();
    SMBus1_Write(HMC5883L_READ_ADDR);

    msb = SMBus1_Read(1);
    lsb = SMBus1_Read(1);
    X_axis = make_word(msb, lsb);

    msb = SMBus1_Read(1);
    lsb = SMBus1_Read(1);
    Z_axis = make_word(msb, lsb);

    msb = SMBus1_Read(1);
    lsb = SMBus1_Read(0);
    Y_axis = make_word(msb, lsb);

    SMBus1_Stop();
}


void HMC5883L_scale_axes(void)
{
    X_axis *= m_scale;
    Z_axis *= m_scale;
    Y_Axis *= m_scale;
}
```

```
void HMC5883L_set_scale(float gauss)
{
    unsigned char value = 0;

    if(gauss == 0.88)
    {
                value = 0x00;
                m_scale = 0.73;
        }

        else if(gauss == 1.3)
        {
                value = 0x01;
                m_scale = 0.92;
        }

        else if(gauss == 1.9)
        {
                value = 0x02;
                m_scale = 1.22;
        }

        else if(gauss == 2.5)
        {
                value = 0x03;
                m_scale = 1.52;
        }

        else if(gauss == 4.0)
        {
                value = 0x04;
                m_scale = 2.27;
        }

        else if(gauss == 4.7)
        {
                value = 0x05;
                m_scale = 2.56;
        }

        else if(gauss == 5.6)
        {
                value = 0x06;
                m_scale = 3.03;
        }

        else if(gauss == 8.1)
        {
                value = 0x07;
                m_scale = 4.35;
        }

        value <<= 5;
        HMC5883L_write(Config_Reg_B, value);
}


float HMC5883L_heading(void)
{
    float heading = 0.0;

    HMC5883L_read_data();
    HMC5883L_scale_axes();
    heading = atan2(Y_axis, X_axis);
    heading += declination_angle;
```

```
    if(heading < 0.0)
    {
            heading += (2.0 * PI);
    }

    if(heading > (2.0 * PI))
    {
            heading -= (2.0 * PI);
    }

    heading *= (180.0 / PI);

    return heading;
}
```

main.c

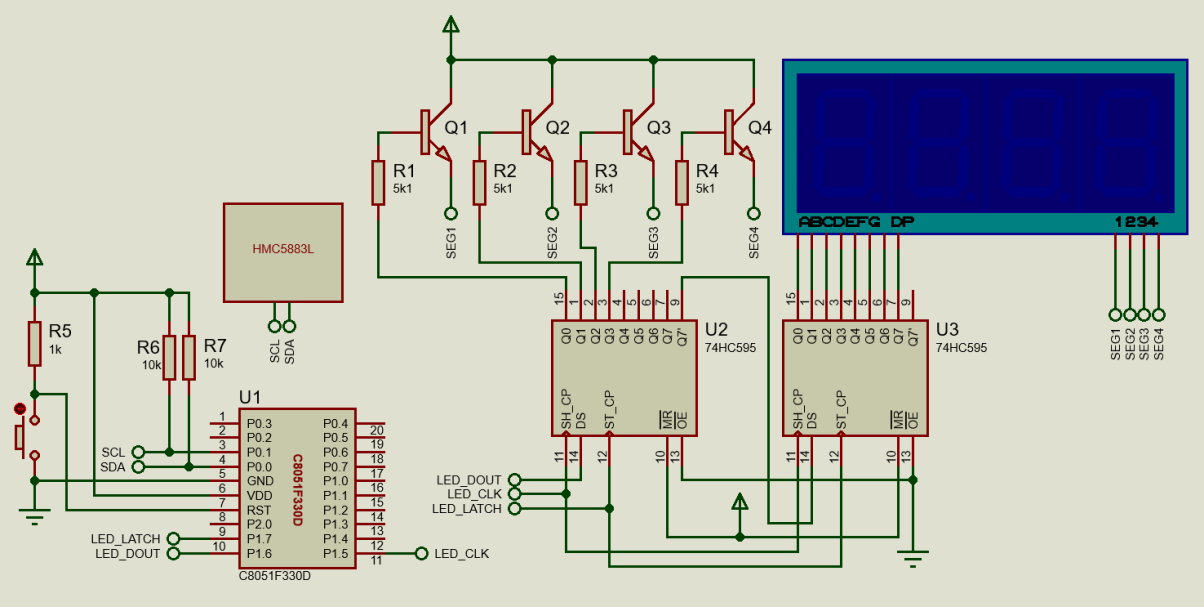
```
#include "HMC5883L.c"


#define LED_DOUT P1_6_bit
#define  LED_CLK P1_5_bit
#define  LED_LATCH P1_7_bit


unsigned char i = 0;
signed int h = 0;
unsigned char val = 0;


const unsigned char code segment_code[13] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x7F, // .
  0xBF, // -
  0x9C  // deg
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


void PCA_Init(void);
void SMBus_Init(void);
void Timer_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
```

```
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);


void Timer_ISR()
iv IVT_ADDR_ET3
ilevel 0
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (h / 100);
            break;
        }
        case 1:
        {
            val = ((h % 100) / 10);
            break;
        }
        case 2:
        {
            val = (h % 10);
            break;
        }
        case 3:
        {
            val = 12;
            break;
        }
    }

    segment_write(val, i);

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}


void main(void)
{
     Init_Device();

     while(1)
     {
         h = HMC5883L_heading();
         delay_ms(200);
     }
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}
```

```
void SMBus_Init(void)
{
    SMB0CF = 0x80;
}


void Timer_Init(void)
{
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}


void Port_IO_Init(void)
{
    // P0.0  -  SDA (SMBus), Open-Drain, Digital
    // P0.1  -  SCL (SMBus), Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P1MDOUT = 0xE0;
    P1SKIP = 0xE0;
    XBR0 = 0x04;
    XBR1 = 0x40;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
{
    IE = 0x80;
    EIE1 = 0x80;
}


void Init_Device(void)
{
    PCA_Init();
    SMBus_Init();
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
    HMC5883L_init();
}
```

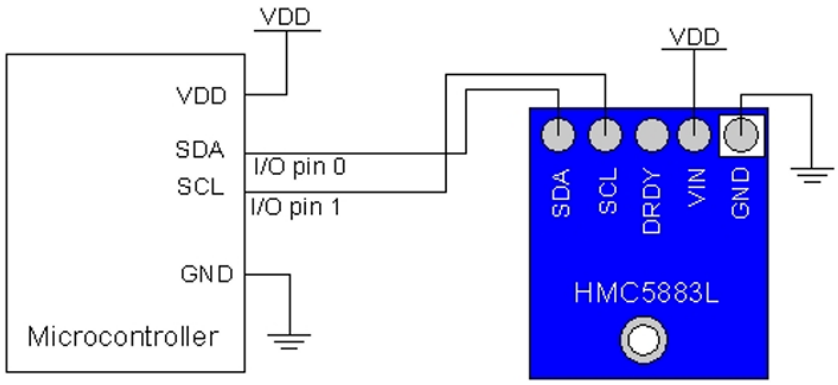
```
void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}


void segment_write(unsigned char disp, unsigned char pos)
{
    LED_LATCH = 0;
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}
```

## Schematic



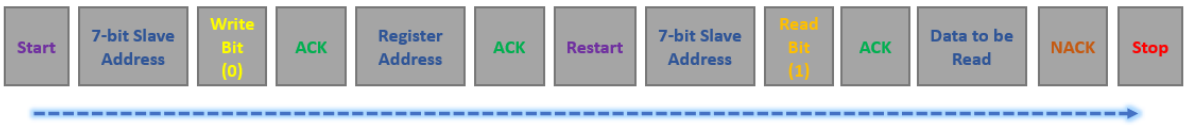
## Explanation

HMC5883L is a popular digital geomagnetic sensor or simply compass sensor that uses the I2C bus interface for communication. Since I2C and SMBUS are similar in terms of working principle, we can use the SMBUS interface to hook up this sensor with C8051F330. Shown below is the basic interfacing diagram:




To be able to successfully use SMBUS/I2C interface, we have to code according to the instructions given in the device datasheet. We have to be specifically attentive to I2C bus read and write operations because these are the operations that are used for data transactions. Generally, these operations do not vary from device to device.

I2C/SMBUS read process can be simply defined by the following functions and flow diagram. In our case with the HMC5883L digital compass, we have to begin communication by initiating a "Start" condition in the bus. This is followed by providing the 7-bit I2C slave address of the compass along with the write bit. If things are sent without any issue, the slave would issue an acknowledgement. It is not mandatory to check acknowledgement at this time. Acknowledgement is only needed to be checked when data is read but we have to specify which register is required to be read. Before actually reading the register, we want to read the I2C bus is restarted along with the 7-bit I2C slave address of the compass with the read bit attached to it.



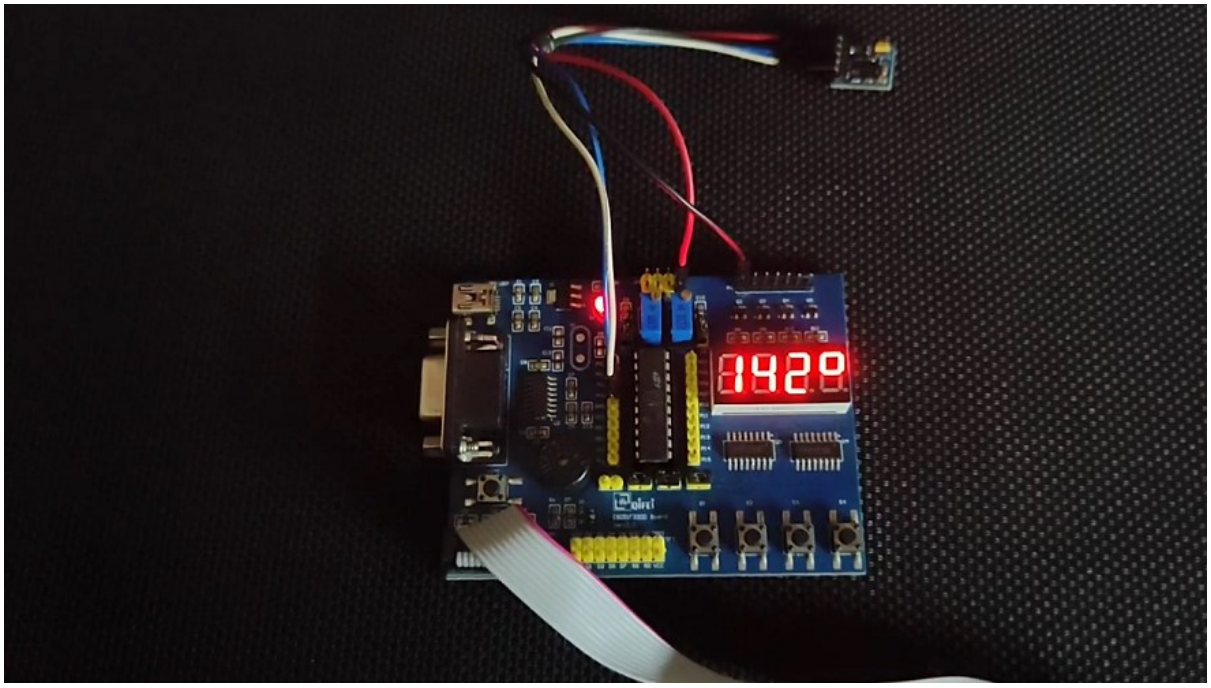

```
unsigned char HMC5883L_read(unsigned char reg)
{
    unsigned char val = 0;

    SMBus1_Start();
    SMBus1_Write(HMC5883L_WRITE_ADDR);
    SMBus1_Write(reg);
    SMBus1_Repeated_Start();
    SMBus1_Write(HMC5883L_READ_ADDR);
    val = SMBus1_Read(0);
    SMBus1_Stop();
    return(val);
}
```

The writing process is yet simpler all we have to do is to specify the register we wish to write followed by the value we want to write.


```
void HMC5883L_write(unsigned char reg_address, unsigned char value)
{
    SMBus1_Start();
    SMBus1_Write(HMC5883L_WRITE_ADDR);
    SMBus1_Write(reg_address);
    SMBus1_Write(value);
    SMBus1_Stop();
}
```

Once we establish these two read-and-write processes, we are left with coding the compass as per the datasheet. This enables us to use the compass for real world applications.

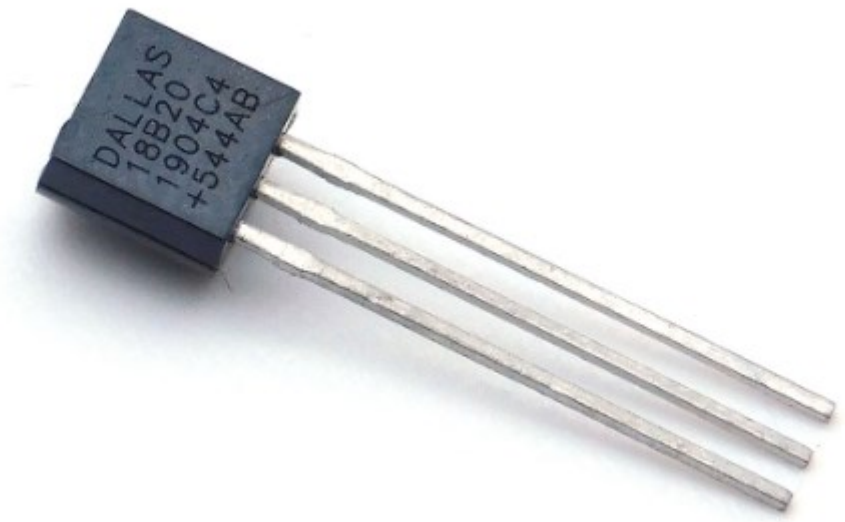
## Demo



Demo video link: https://youtu.be/50upZAW4W_A

## One Wire Bus (OW) – DS18B20 Digital Temperature Sensor

MikroC compiler packs a lot of common regular-use libraries apart from SPI, UART and I2C libraries. One wire library is one such library that is not usually seen in other compilers. This library allows us to communicate with one-wire devices from Dallas like DS18B20 digital temperature sensors and other one-wire devices from the same manufacturer.




### Code

```
#define LED_DOUT                        P1_6_bit
#define LED_CLK                         P1_5_bit
#define LED_LATCH                       P1_7_bit

#define DS18B20_CONVERT_T               0x44
#define DS18B20_READ_SCRATCHPAD         0xBE
#define DS18B20_SKIP_ROM                0xCC


sbit OW_Bit at P1_4_bit;


unsigned char i = 0;
register unsigned char val = 0;
unsigned int value = 0;

const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
```

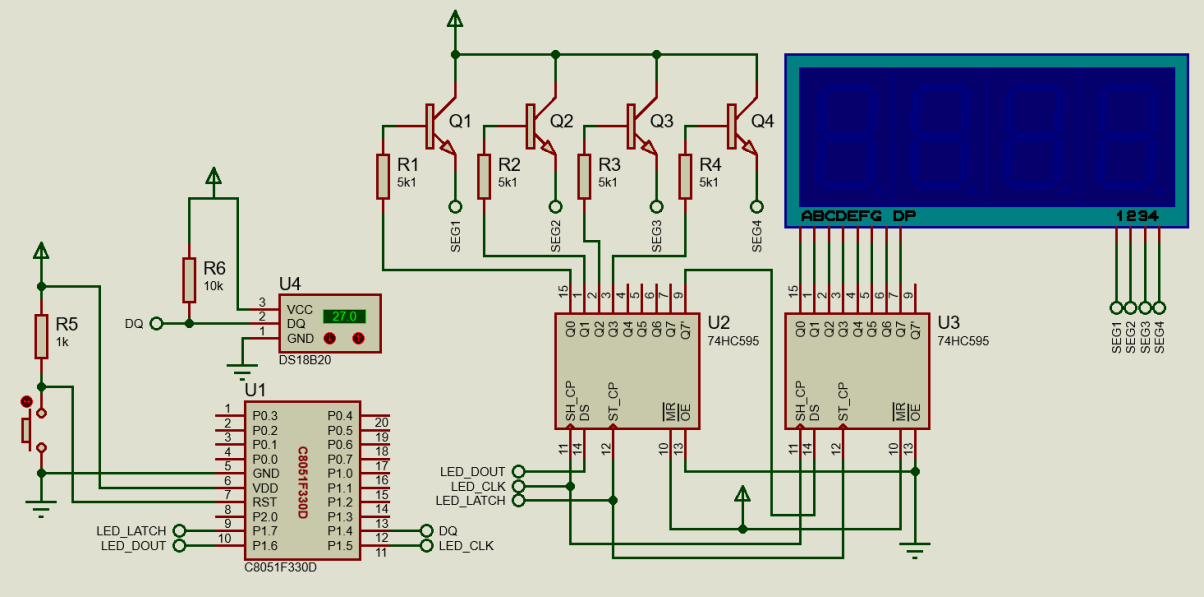
```
  0x9C, // degree
  0xC6  // C
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


void Reset_Sources_Init(void);
void PCA_Init(void);
void Timer_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);


void Timer_ISR(void)
iv IVT_ADDR_ET3
ilevel 1
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (value / 10);
            break;
        }
        case 1:
        {
            val = (value % 10);
            break;
        }
        case 2:
        {
            val = 10;
            break;
        }
        case 3:
        {
            val = 11;
            break;
        }
    }

    segment_write(val, i);
    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}
```

```
void main(void)
{
    unsigned temp = 0x00;
    Init_Device();

    while(1)
    {
        Ow_Reset();
        Ow_Write(DS18B20_SKIP_ROM);
        Ow_Write(DS18B20_CONVERT_T);
        Delay_us(120);

        Ow_Reset();
        Ow_Write(DS18B20_SKIP_ROM);
        Ow_Write(DS18B20_READ_SCRATCHPAD);

        temp =  Ow_Read();
        temp = ((Ow_Read() << 0x08) + temp);
        value = temp >> 0x04;

        P0_1_bit = 1;
        delay_ms(300);
        P0_1_bit = 0;
        delay_ms(300);

    }
}


void Reset_Sources_Init(void)
{
    RSTSRC = 0x04;
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}

void Timer_Init(void)
{
    TMR3CN = 0x04;
    TMR3RLL = 0xCA;
    TMR3RLH = 0xFA;
}


void Port_IO_Init(void)
{
    // P0.0  -  Unassigned,  Open-Drain, Digital
    // P0.1  -  Skipped,     Push-Pull,  Digital
    // P0.2  -  Skipped,     Open-Drain, Analog
    // P0.3  -  Skipped,     Open-Drain, Analog
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Skipped,     Open-Drain, Digital
```

```
    // P1.5  -  Unassigned,  Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDIN = 0xF3;
    P0MDOUT = 0x02;
    P1MDOUT = 0xE0;
    P0SKIP = 0x0E;
    P1SKIP = 0xD0;
    XBR1 = 0x40;
}


void Oscillator_Init(void)
{
    int i = 0;
    OSCICN = 0x81;

    if(MCDRSF_bit == 1)
    {
        CLKSEL = 0x00;

        for(i = 0; i < 9; i++)
        {
            P0_1_bit = 1;
            delay_ms(60);
            P0_1_bit = 0;
            delay_ms(60);
        }
    }
    else
    {
        OSCXCN = 0x67;
        for (i = 0; i < 3000; i++);  // Wait 1ms for initialization
        while ((OSCXCN & 0x80) == 0);
        CLKSEL = 0x01;

        for(i = 0; i < 9; i++)
        {
            P0_1_bit = 1;
            delay_ms(45);
            P0_1_bit = 0;
            delay_ms(45);
        }
    }

    delay_ms(2000);
}


void Interrupts_Init(void)
{
    IE = 0x80;
    EIE1 = 0x80;
}


void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Reset_Sources_Init();
    Interrupts_Init();
```

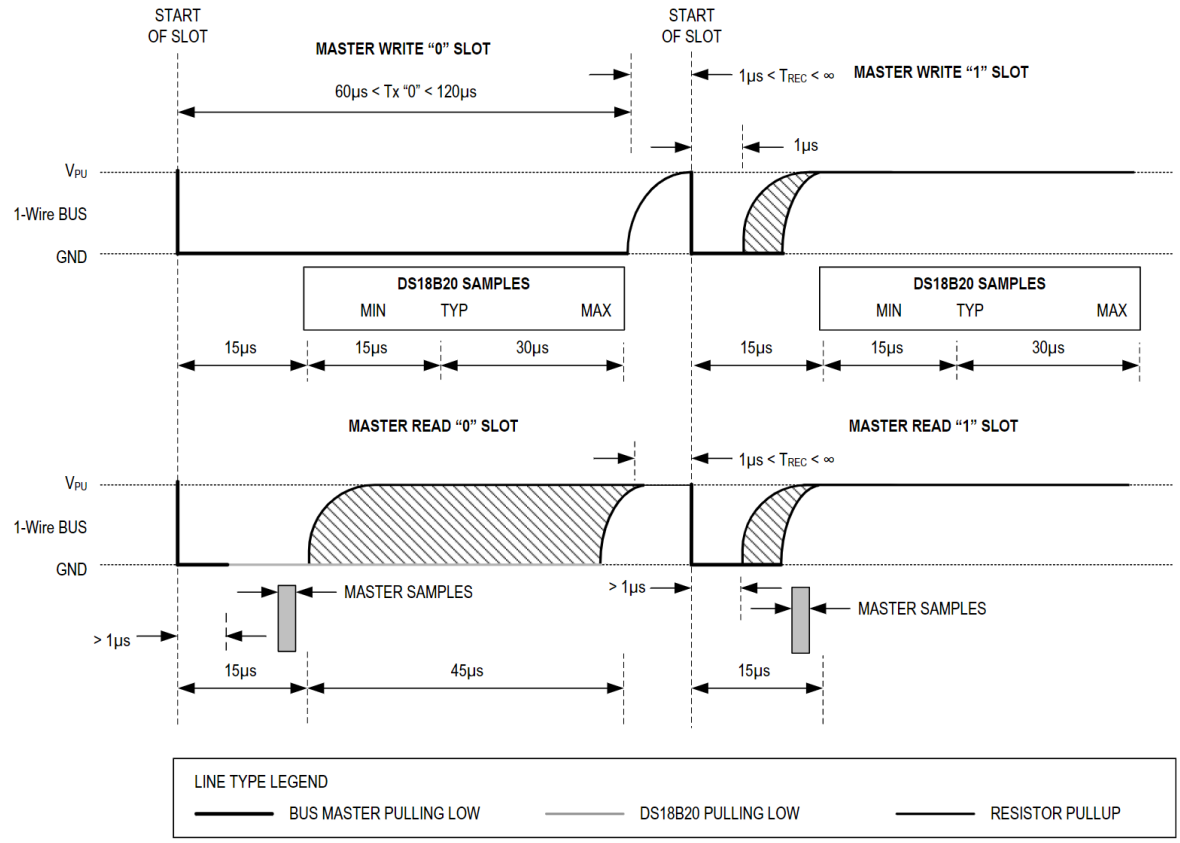
```
}


void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}


void segment_write(unsigned char disp, unsigned char pos)
{
    LED_LATCH = 0;
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}
```

## Schematic



## Explanation

DS18B20 digital temperature sensor employs a time-slotting mechanism for data transactions, the basics of which are shown below.




MikroC compiler has a dedicated one-wire library for this sensor and many other devices that employ one-wire protocol as per specification from Dallas. In this example, I used their library to communicate with a DS18B20 temperature sensor and the entire process is very simple.

The main code begins by issuing one-wire reset command (*Ow_Reset*) followed by ROM skipping and temperature convert commands. The reset command acts like a flush. The ROM skip command is used because there is only one DS18B20 present in the one-wire bus in this example. If there were multiple sensors present in the same bus then we could identify them by reading their unique identification code stored inside their ROMs. The *"temperature convert"* command, on the other hand, reads the internal ADC of the sensor and converts the analogue temperature value to digital binary data. After this, we have to wait for a small duration of 120 microseconds.

The above process is repeated but the temperature conversion command is replaced with the scratchpad read command because the converted temperature binary data is stored in an internal scratchpad or RAM location. The sensor is read and necessary conversions are done to extract temperature in human-readable decimal format. The temperature is then displayed on the onboard seven-segment display.

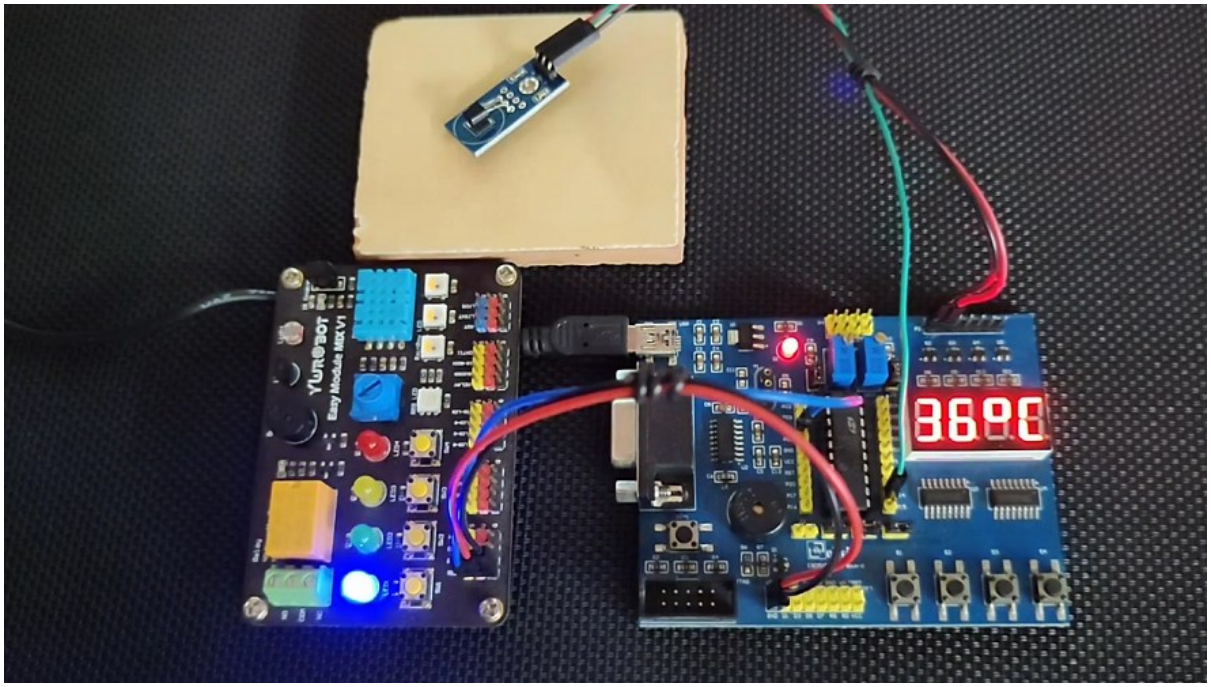
```
void main(void)
{
    unsigned temp = 0x00;
    Init_Device();

    while(1)
    {
        Ow_Reset();
        Ow_Write(DS18B20_SKIP_ROM);
        Ow_Write(DS18B20_CONVERT_T);
        Delay_us(120);

        Ow_Reset();
        Ow_Write(DS18B20_SKIP_ROM);
        Ow_Write(DS18B20_READ_SCRATCHPAD);

        temp =  Ow_Read();
        temp = ((Ow_Read() << 0x08) + temp);
        value = temp >> 0x04;

        P0_1_bit = 1;
        delay_ms(300);
        P0_1_bit = 0;
        delay_ms(300);

    }
}
```

There is one thing in the code that needs to be discussed here. In this example code, I used a trick to ensure that the program works even if the 12MHz external oscillator attached to the microcontroller fails for some reason. For this, I enabled the missing clock detector reset. It works simply by resetting the microcontroller in the event of a missing clock signal from an external oscillator. After a reset or during start-up, the code checks whether there is a history of reset due to external oscillator failure or not. This is done by checking the ***MCDESF*** flag bit. If there was a reset due to external oscillator failure then the code switches to the internal oscillator automatically or else the external oscillator is selected. The oscillator selection is shown by flashing a LED connected with P0.1 pin. The flashing rate indicates which clock source has been selected. A fast flash rate indicates external oscillator selection while a slow flash rate indicates internal oscillator selection.

```
void Reset_Sources_Init(void)
{
    RSTSRC = 0x04;
}


....


void Oscillator_Init(void)
{
    int i = 0;
    OSCICN = 0x81;

    if(MCDRSF_bit == 1)
    {
        CLKSEL = 0x00;

        for(i = 0; i < 9; i++)
        {
            P0_1_bit = 1;
            delay_ms(60);
```

```
            P0_1_bit = 0;
            delay_ms(60);
        }
    }
    else
    {
        OSCXCN = 0x67;
        for (i = 0; i < 3000; i++);  // Wait 1ms for initialization
        while ((OSCXCN & 0x80) == 0);
        CLKSEL = 0x01;

        for(i = 0; i < 9; i++)
        {
            P0_1_bit = 1;
            delay_ms(45);
            P0_1_bit = 0;
            delay_ms(45);
        }
    }

    delay_ms(2000);
}
```

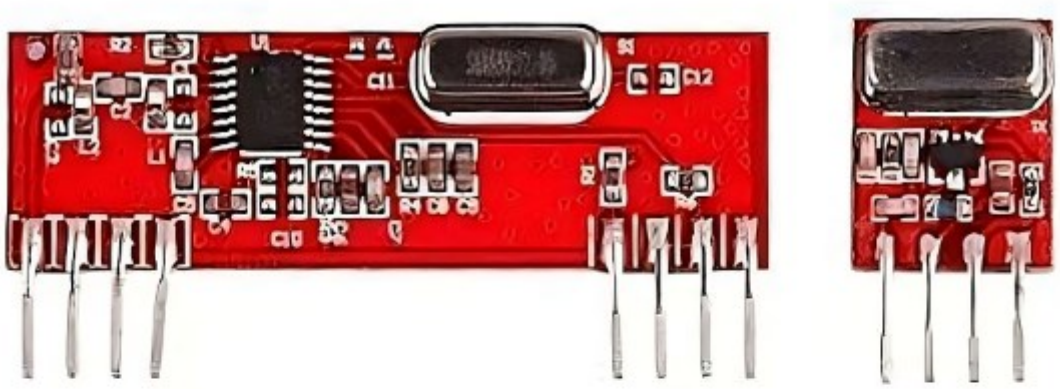
## Demo



Demo video link: https://youtu.be/qE_B88fMvfY

## Bit-Banging – RF Communication with RF Modules

In many situations, we need to create our communication protocol and in those cases bit-banging I/O pins are the only method we have to resort to. On the other hand, there is no other better way other than bit-banging to reliably transmit-receive data between low-cost 315/433 MHz RF modules. Using UART is an alternative but some microcontrollers lack these hardware peripherals while in other cases, it may have other uses. Another important thing to note is the fact that RF modules of these frequency ranges pickup lot of background RF noise and so there is no simple technique to reliably transmit and receive valid data. Thus, we need some sort of data encapsulation.



## Code

### RF TX main.c

```
#define LED_DOUT            P1_6_bit
#define LED_CLK             P1_5_bit
#define LED_LATCH           P1_7_bit

#define INC_SW              P1_4_bit
#define DEC_SW              P1_3_bit

#define RF_RX               P0_0_bit
#define RF_TX               P0_1_bit


void PCA_Init(void);
void Timer_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);
void send_data(unsigned char value);
void RF_send(unsigned char rf_data);


unsigned char i = 0;
unsigned char val = 0;
signed int value = 0;

const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x7F, // .
  0xBF  // -
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


void Timer_3_ISR(void)
iv IVT_ADDR_ET3
ilevel 0
ics ICS_AUTO
{
    switch(i)
```

```
    {
        case 0:
        {
            val = (value / 1000);
            break;
        }
        case 1:
        {
            val = ((value % 1000) / 100);
            break;
        }
        case 2:
        {
            val = ((value % 100) / 10);
            break;
        }
        case 3:
        {
            val = (value % 10);
            break;
        }
    }

    segment_write(val, i);

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}


void main(void)
{
    Init_Device();

    while(1)
    {
        if(INC_SW == 0)
        {
            delay_ms(40);
            value++;
        }

        if(DEC_SW == 0)
        {
            delay_ms(40);
            value--;
        }

        if(value > 200)
        {
            value = 0;
        }

        if(value < 0)
        {
            value = 99;
        }

        send_data(value);
        delay_ms(100);
```

```
    };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}


void Timer_Init(void)
{
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}


void Port_IO_Init(void)
{
    // P0.0  -  Skipped,     Open-Drain, Digital
    // P0.1  -  Skipped,     Push-Pull,  Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDOUT = 0x02;
    P1MDOUT = 0xE0;
    P0SKIP = 0x03;
    P1SKIP = 0xE0;
    XBR1 = 0x40;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
{
    IE = 0x80;
    EIE1 = 0x80;
}


void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    Port_IO_Init();
```

```
    Oscillator_Init();
    Interrupts_Init();
}


void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}


void segment_write(unsigned char disp, unsigned char pos)
{
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 0;
    LED_LATCH = 1;
}


void send_data(unsigned char value)
{
     signed char s = 20;
     unsigned char CRC = 0;

     CRC = (value & 0xAA);

     while(s > 0x00)
     {
           RF_TX = 1;
           delay_us(500);
           RF_TX = 0;
           delay_us(500);
           s--;
     }
     delay_us(100);

     RF_send(value);
     RF_send(CRC);
}


void RF_send(unsigned char rf_data)
{
    signed char s = 0x08;

    while(s > 0x00)
    {
```

```
        RF_TX = 1;

        if(rf_data & 0x80)
        {
            delay_ms(2);
        }
        else
        {
            delay_ms(1);
        }

        RF_TX = 0;
        delay_ms(1);
        rf_data <<= 1;
        s--;
    }
}
```

RF RX main.c

```
#define LED_DOUT            P1_6_bit
#define LED_CLK             P1_5_bit
#define LED_LATCH           P1_7_bit

#define RF_RX               P0_0_bit
#define RF_TX               P0_1_bit

#define sync                0x09
#define error               0x06


void PCA_Init(void);
void Timer_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);
unsigned char receive_data(void);
signed long decode_data(void);


unsigned char i = 0;
unsigned char val = 0;
signed int value = 0;

const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x7F, // .
  0xBF  // -
};
```

```
const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


void Timer_3_ISR(void)
iv IVT_ADDR_ET3
ilevel 0
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (value / 1000);
            break;
        }
        case 1:
        {
            val = ((value % 1000) / 100);
            break;
        }
        case 2:
        {
            val = ((value % 100) / 10);
            break;
        }
        case 3:
        {
            val = (value % 10);
            break;
        }
    }

    segment_write(val, i);

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}


void main(void)
{
    Init_Device();

    while(1)
    {
        value = decode_data();
    };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
```

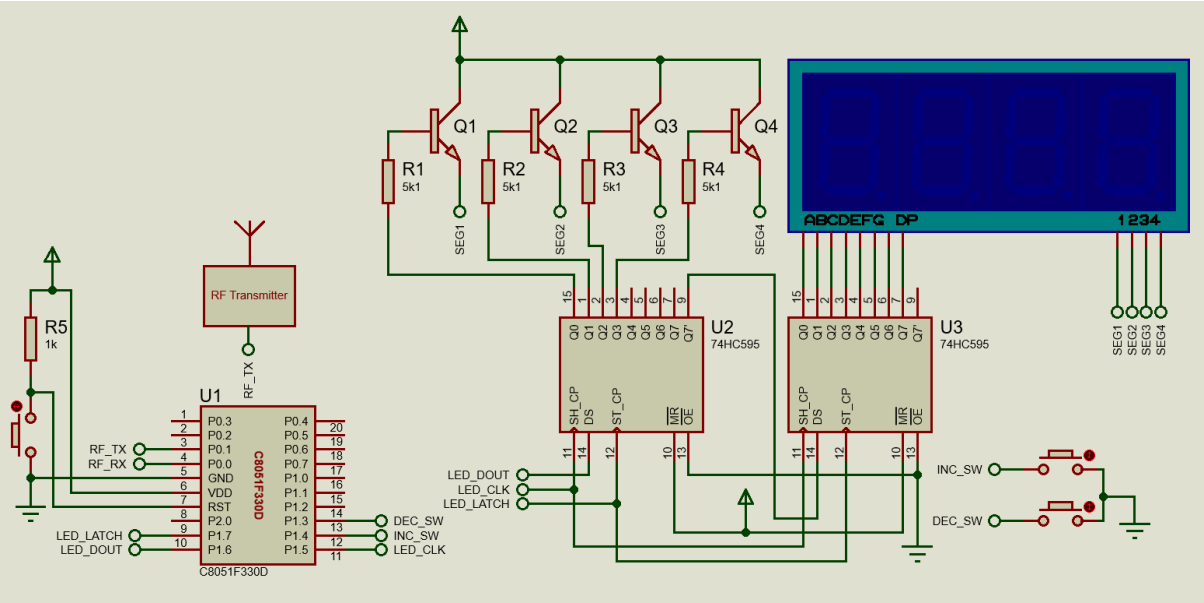
```
    PCA0MD = 0x00;
}


void Timer_Init(void)
{
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}


void Port_IO_Init(void)
{
    // P0.0  -  Skipped,     Open-Drain, Digital
    // P0.1  -  Skipped,     Push-Pull,  Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDOUT = 0x02;
    P1MDOUT = 0xE0;
    P0SKIP = 0x03;
    P1SKIP = 0xE0;
    XBR1 = 0x40;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
{
    IE = 0x80;
    EIE1 = 0x80;
}


void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}


void write_74HC595(unsigned char send_data)
{
```

```
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}


void segment_write(unsigned char disp, unsigned char pos)
{
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 0;
    LED_LATCH = 1;
}


unsigned char receive_data(void)
{
    unsigned char t = 0;

    while(!RF_RX);
    while(RF_RX)
    {
        t++;
        delay_us(10);
    };

    if((t > 25) && (t < 75))
    {
        return sync;
    }
    else if((t > 175) && (t < 225))
    {
        return 1;
    }
    else if((t > 75) && (t < 125))
    {
        return 0;
    }
    else
    {
        return error;
    }
}


signed long decode_data(void)
{
    unsigned char d = 0;
    unsigned char s = 0;
    unsigned long value = 0;
```

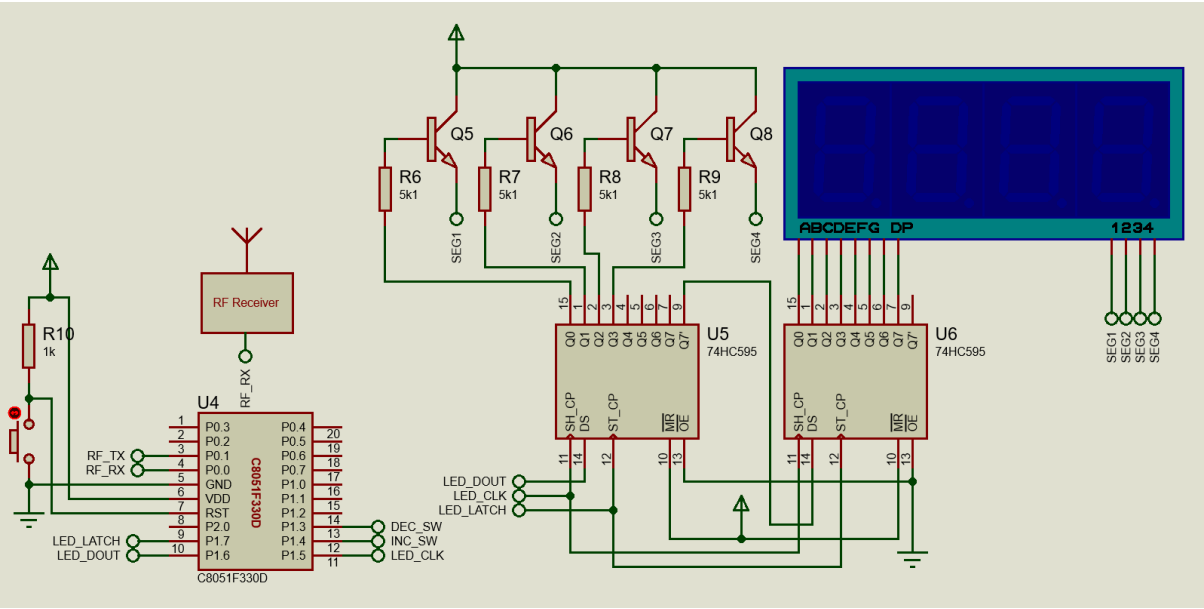
```
    unsigned char v1 = 0;
    unsigned char v2 = 0;

    while(receive_data() != sync);

    d = receive_data();
    while(d == sync)
    {
         d = receive_data();
    };

    while(s < 15)
    {
        switch(d)
        {
            case 1:
            {
                value |= 1;
                break;
            }
            case 0:
            {
                break;
            }
            case sync:
            case error:
            {
                return -1;
            }
        }
        s++;
        value <<= 1;
        d = receive_data();
    }

    v1 = (value >> 8);
    v2 = (value & 0x00FF);
    delay_ms(4);

    if((v1 & 0xAA) == v2)
    {
        return v1;
    }
    else
    {
        return -1;
    }
}
```

## Schematic

### RF TX

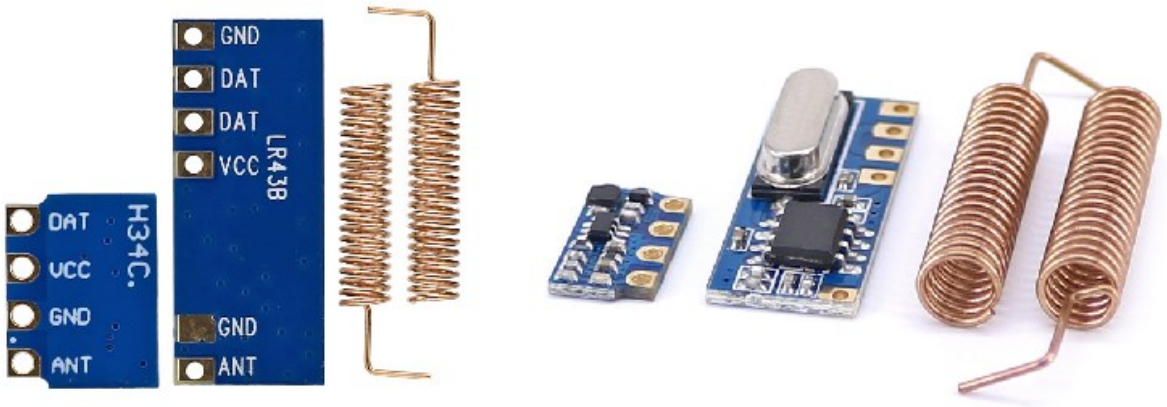



### RF RX

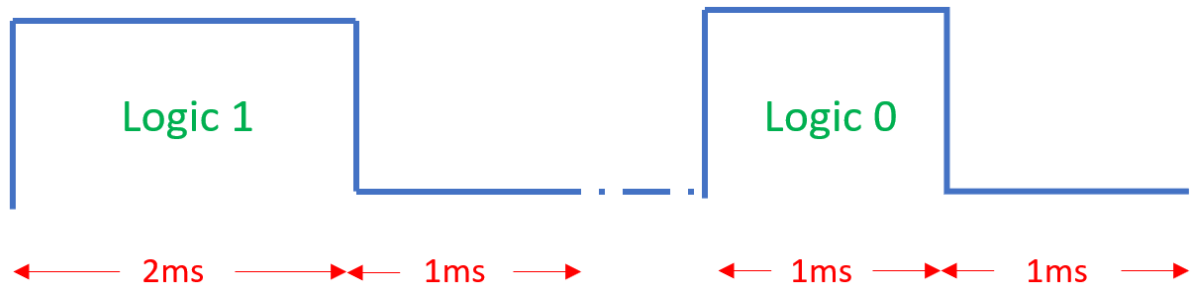


## Explanation

RF modules usually come in pairs - a transmitter and a receiver. *P0.0* and *P0.1* are *RF TX* and *RF RX* pins respectively. These pins are bit-banged. Two C8051F330D microcontroller boards have been used in this example – one is a transmitter and the other is a receiver. The modules used in this example are similar to the ones shown below although any other RF module of similar nature would work fine. I have tested this example with various other RF modules including 315 MHz ones without any issues.




The bit-banging technique employed here is based on the time-slotting technique similar to DS18B20 digital temperature sensor. Ones and zeroes are defined by pulse widths. There is no additional hardware peripheral needed apart from I/O pins to achieve bit-banging.




As shown in the illustration above, a *"One"* is a pulse of 2 ms high followed by a 1 ms low while a *"Zero"* is a pulse of 1 ms high followed by a 1 ms low pin state. The following function will take RF data byte to be sent via RF module and will convert it to time-slotting pulses. Data is sent from MSB to LSB.

```
void RF_send(unsigned char rf_data)
{
    signed char s = 0x08;

    while(s > 0x00)
    {
         RF_TX = 1;

         if(rf_data & 0x80)
         {
             delay_ms(2);
         }
```

```
        else
        {
            delay_ms(1);
        }

        RF_TX = 0;
        delay_ms(1);
        rf_data <<= 1;
        s--;
    }
}
```

The above function is just for time-slotting-based bit-banging. However, data that is to be sent needs to be encapsulated in a proper manner so that whatever we send is reliably received by the receiver without any issue. Therefore, we need a mechanism to add a cyclic redundancy check (CRC) along with the data. The following function will accept a byte of data and send it to an RF receiver via an RF medium. There are two steps in between. Generally, an RF receiver module tends to receive a lot of background RF noise due to its default high RF gain setting. Due to this effect, it continuously outputs random pulses even when nothing valid is being transmitted by any transmitter. Thus, a technique is needed to reduce the RF receiver's gain so that it can pick up the correct transmitted data. This can be achieved by sending a stream of high-frequency pulses. Once the RF gain is reduced, we can reliably send data. However, there could be errors during the reception of data and so a CRC check becomes mandatory. This CRC check would ensure whether the data received is correct or false. The *send_data* function given below does all of these steps. For simplicity, the CRC here is simply the result of an AND operation between data to be sent and 0xAA.

```
void send_data(unsigned char value)
{
     signed char s = 20;
     unsigned char CRC = 0;

     CRC = (value & 0xAA);

     while(s > 0x00)
     {
           RF_TX = 1;
           delay_us(500);
           RF_TX = 0;
           delay_us(500);
           s--;
     }
     delay_us(100);

     RF_send(value);
     RF_send(CRC);
}
```

Now let us look at the receiver side. The receiver side has just to decode and reconstruct what has been sent to it. Since the time-slotting technique has been used to send data, we just have to check pulse widths to determine if received pulses are sync, one or zero pulses. For logic low at the RF RX pin, we have to do nothing. It is only during a logic high that we time the high period. The timing is done in 10µs steps in order to ensure pulse timing accuracy while ensuring that all kinds of pulses are properly polled. A time-keeping variable, t is responsible for keeping time. It gets incremented every 10µs when the RF RX pin is held in the high state. When the RF Rx pin gets low afterward, pulse width time is obtained.

Due to the effects of environment, temperature, medium, oscillator mismatch and other factors, pulse timings may deviate and so all timings are compared in ranges instead of fixed values of 0.5ms, 1ms and 2ms, i.e., there are tolerances in timings. Any pulse that does not fit in these ranges is discarded as a glitch. The *receive_data* function given below does all of the aforementioned tasks.

```
unsigned char receive_data(void)
{
    unsigned char t = 0;

    while(!RF_RX);
    while(RF_RX)
    {
        t++;
        delay_us(10);
    };

    if((t > 25) && (t < 75))
    {
        return sync;
    }
    else if((t > 175) && (t < 225))
    {
        return 1;
    }
    else if((t > 75) && (t < 125))
    {
        return 0;
    }
    else
    {
        return error;
    }
}
```

Now that we understand how the pulses are obtained, we can now focus on how the received pulses are decoded to obtain sent data. Until sync pulses have been successfully obtained, no other received data is processed. If sync pulses have been detected properly then the next 16 bits are polled and the sent data and CRC are reconstructed. At the end of these steps, CRC is calculated and cross-checked against the sent CRC value.

```
#define sync                0x09
#define error               0x06

....

signed long decode_data(void)
{
    unsigned char d = 0;
    unsigned char s = 0;
    unsigned long value = 0;
    unsigned char v1 = 0;
    unsigned char v2 = 0;

    while(receive_data() != sync);

    d = receive_data();
    while(d == sync)
    {
         d = receive_data();
    };
```

```
    while(s < 15)
    {
        switch(d)
        {
            case 1:
            {
                value |= 1;
                break;
            }
            case 0:
            {
                break;
            }
            case sync:
            case error:
            {
                return -1;
            }
        }
        s++;
        value <<= 1;
        d = receive_data();
    }

    v1 = (value >> 8);
    v2 = (value & 0x00FF);
    delay_ms(4);

    if((v1 & 0xAA) == v2)
    {
        return v1;
    }
    else
    {
        return -1;
    }
}
```

In the main code of the transmitter, the value of a variable named *value* is changed using two onboard push buttons connected to P1.4 and P1.4 and sent every 100ms via RF medium using an RF transmitter module.

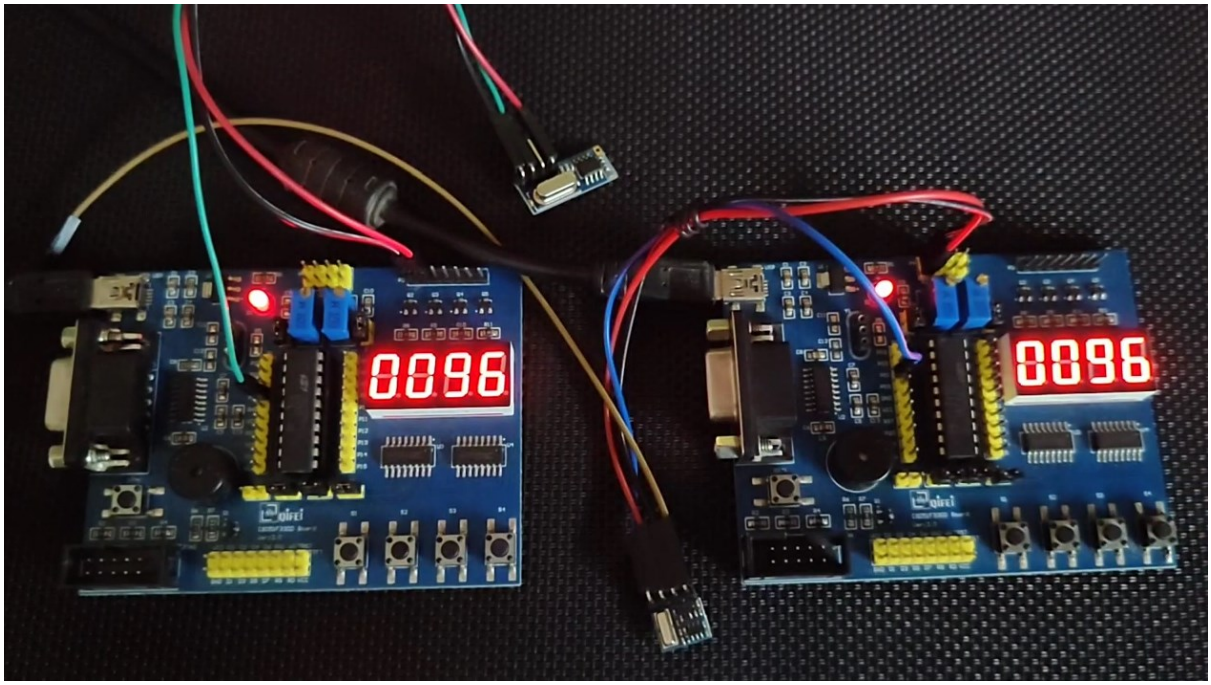
```
void main(void)
{
    Init_Device();

    while(1)
    {
        if(INC_SW == 0)
        {
            delay_ms(40);
            value++;
        }

        if(DEC_SW == 0)
        {
            delay_ms(40);
            value--;
        }

        if(value > 200)
        {
            value = 0;
        }
```

```
        if(value < 0)
        {
            value = 99;
        }

        send_data(value);
        delay_ms(100);
    };
}
```

At the receiver side, the value sent by the transmitter is decoded and shown on the onboard seven-segment display.

```
void main(void)
{
    Init_Device();

    while(1)
    {
        value = decode_data();
    };
}
```

## Demo




Demo video link: https://youtu.be/Hu_T04OuNSM

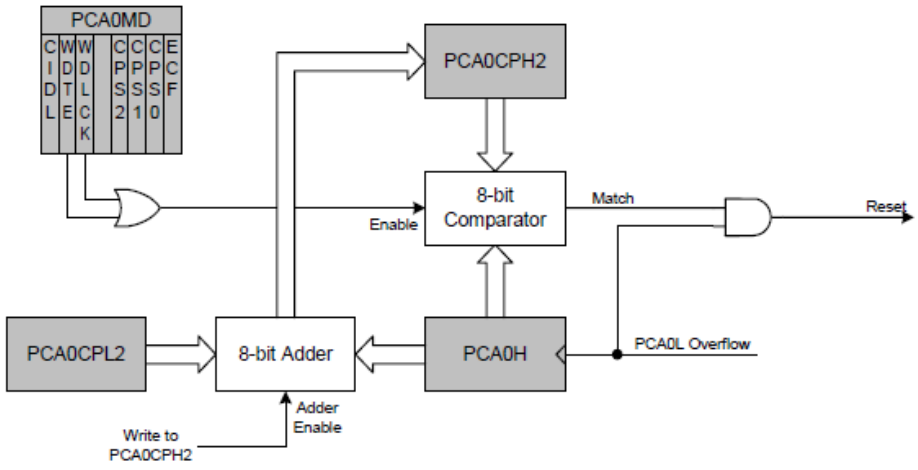
# Watchdog Timer (WDT)

As anyone would expect in a microcontroller, C8051s pack watchdog timers for resetting them in the event of unexpected or unforeseen loop or software failure, leading to unwanted code execution halt.

Unlike most microcontrollers, however, there is no separate and dedicated watchdog timer in C8051F330D. The watchdog timer of C8051F330D is part of the Programmable Array Counter (PCA) hardware's module 2 and so there are some advantages and disadvantages to this arrangement.

In terms of advantages, firstly no additional external hardware is needed and secondly, the implementation is not complex. The disadvantages are, dependency on the system clock for timing rather than a separate independent watchdog timer clock, loss of one PCA module if watchdog timer mode is used and difficulty in locating it in the datasheet.

According to the block diagram shown below, the PCA watchdog timer works by using the compare-match principle. If a match occurs between PCA0CPH2 and PCA0H while the WDT is enabled, a reset will be generated. By default, the watchdog timer is enabled.




## Code

```
#define LED_DOUT          P1_6_bit
#define LED_CLK           P1_5_bit
#define LED_LATCH         P1_7_bit
#define BUTTON            P1_4_bit


unsigned char i = 0;
register unsigned char val = 0;
unsigned int value = 0;

const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
```

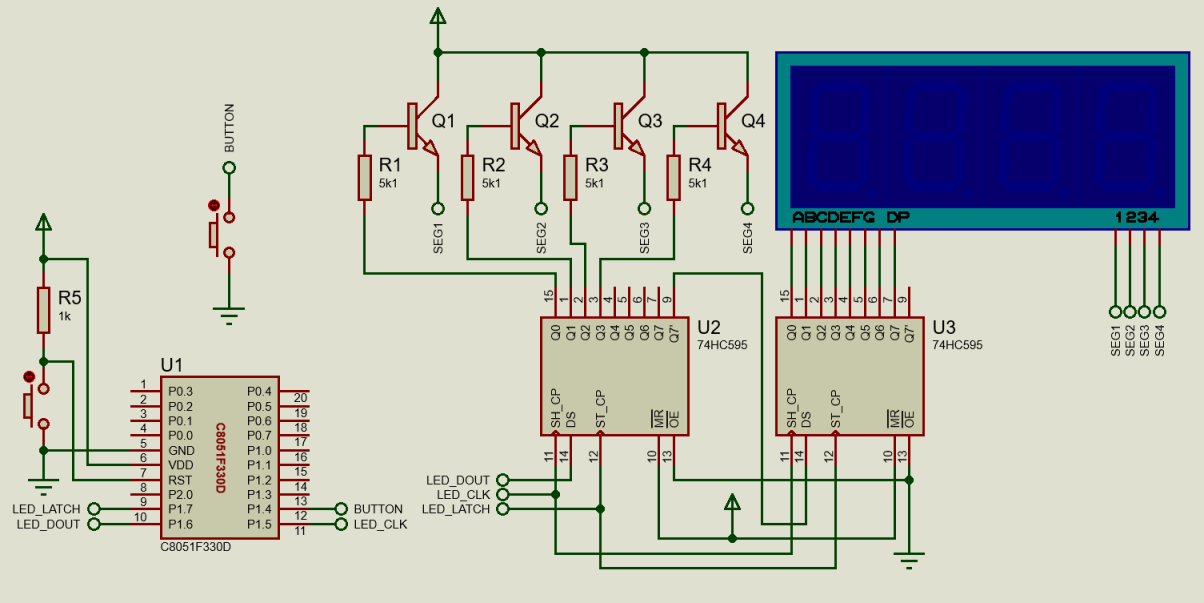
```
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x7F, // .
  0xBF  // -
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);


void Timer_ISR()
iv IVT_ADDR_ET3
ilevel 0
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (value / 1000);
            break;
        }
        case 1:
        {
            val = ((value % 1000) / 100);
            break;
        }
        case 2:
        {
            val = ((value % 100) / 10);
            break;
        }
        case 3:
        {
            val = (value % 10);
            break;
        }
    }

    segment_write(val, i);

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}


void main(void)
```

```
{
  signed int s = 200;

  Init_Device();

  while(1)
  {
      value++;
      delay_ms(20);
      if(BUTTON == 1)
      {
         PCA0CPH2 = 0;
         delay_ms(20);
         PCA0CPH2 = 0;
         delay_ms(20);
         PCA0CPH2 = 0;
      }
      else
      {
         PCA0CPH2 = 0;
         while(s > 0)
         {
             PCA0CPH2 = 0;
             delay_ms(30);
             s--;
         }
         while(1);
      }
  };
}


void PCA_Init()
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
    PCA0CPM2 = 0x01;
    PCA0CPL2 = 0xFF;
    PCA0MD |= 0x40;
    PCA0CPH2 = 0;
}


void Timer_Init(void)
{
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
    PCA0CPH2 = 0;
}


void Port_IO_Init(void)
{
    // P0.0  -  Unassigned,  Open-Drain, Digital
    // P0.1  -  Unassigned,  Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
```

```
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Skipped,     Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P1MDOUT = 0xE0;
    P1SKIP = 0xF0;
    XBR1 = 0x40;
    PCA0CPH2 = 0;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
    PCA0CPH2 = 0;
}


void Interrupts_Init(void)
{
    IE = 0x80;
    EIE1 = 0x80;
    PCA0CPH2 = 0;
}


void Init_Device(void)
{
    PCA0MD = 0x00;

    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
    PCA_Init();

    LED_LATCH = 1;
    LED_CLK = 1;
    LED_DOUT = 0;
    PCA0CPH2 = 0;
}


void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
```

```
    }
}


void segment_write(unsigned char disp, unsigned char pos)
{
    LED_LATCH = 0;
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}
```

## Schematic



## Explanation

The following formula and table show some typical watchdog timeout values.

| System Clock (Hz) | PCA0CPL2 | Timeout Interval (ms) |
|---|---|---|
| 24,500,000 | 255 | 32.1 |
| 24,500,000 | 128 | 16.2 |
| 24,500,000 | 32 | 4.1 |
| 18,432,000 | 255 | 42.7 |
| 18,432,000 | 128 | 21.5 |
| 18,432,000 | 32 | 5.5 |
| 11,059,200 | 255 | 71.1 |
| 11,059,200 | 128 | 35.8 |
| 11,059,200 | 32 | 9.2 |
| 3,062,500[2] | 255 | 257 |
| 3,062,500[2] | 128 | 129.5 |
| 3,062,500[2] | 32 | 33.1 |
| 32,000 | 255 | 24576 |
| 32,000 | 128 | 12384 |
| 32,000 | 32 | 3168 |

**Notes:**
1. Assumes SYSCLK/12 as the PCA clock source, and a PCA0L value of 0x00 at the update time.
2. Internal SYSCLK reset frequency = Internal Oscillator divided by 8.

$$\boldsymbol{Offset} = ((\mathbf{256} \times \boldsymbol{PCA0CPL2}) + (\mathbf{256} - \boldsymbol{PCA0L}))$$

and

$$\boldsymbol{Timeout} = \frac{\boldsymbol{Offset} \times \mathbf{12}}{\boldsymbol{System\ Clock}}$$

Thus with 24.5MHz internal oscillator, *PCA0CPL2* = 255 and *PCA0L* = 0, the timeout is calculated as below:

$$\boldsymbol{Offset} = \big((\mathbf{256} \times \mathbf{255}) + (\mathbf{256} - \mathbf{0})\big) = \mathbf{65536}$$

and

$$\boldsymbol{Timeout} = \frac{\mathbf{65536} \times \mathbf{12}}{\mathbf{24.5}\boldsymbol{MHz}} \approx \mathbf{32.1}\boldsymbol{ms}$$

The ***Config2*** code configuration wizard can be used to generate necessary code automatically and we can avoid all of these calculations but it is better to know how things are done manually.

The setup for this example is just like all other examples covered so far. An onboard push button attached with pin P1.4 and the onboard seven-segment display are both used.

The PCA watchdog setup for the calculation shown above is shown below.

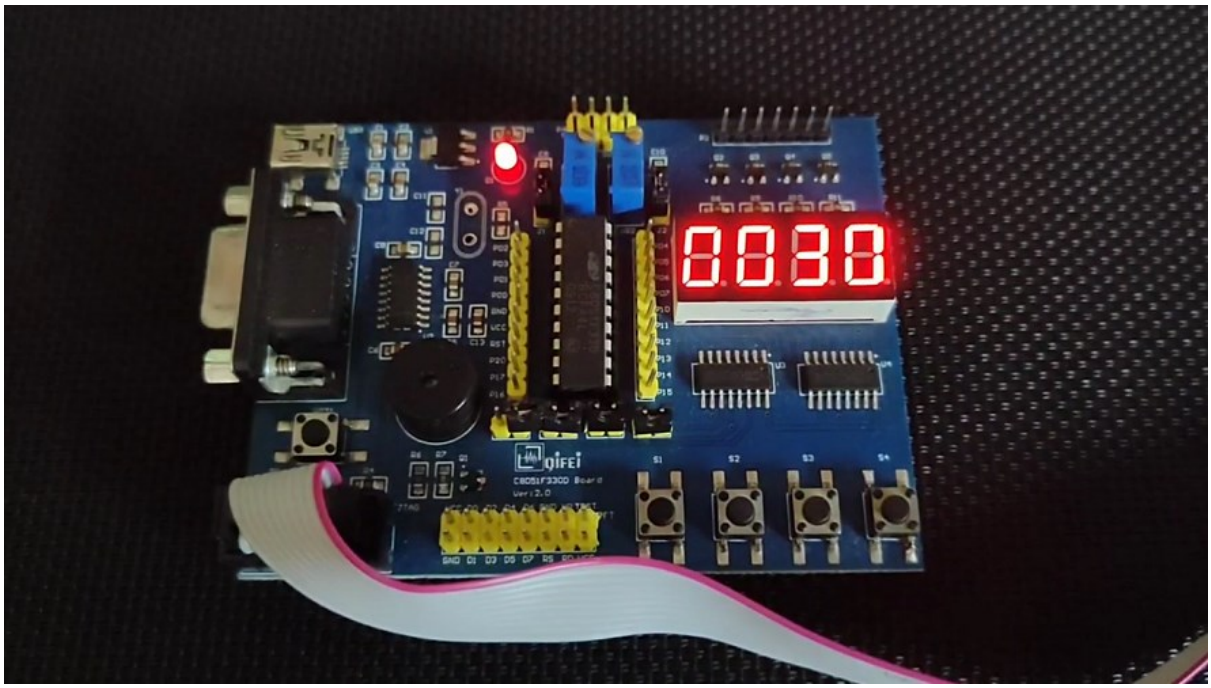
```
void PCA_Init()
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
    PCA0CPM2 = 0x01;
    PCA0CPL2 = 0xFF;
    PCA0MD |= 0x40;
    PCA0CPH2 = 0;
}
```

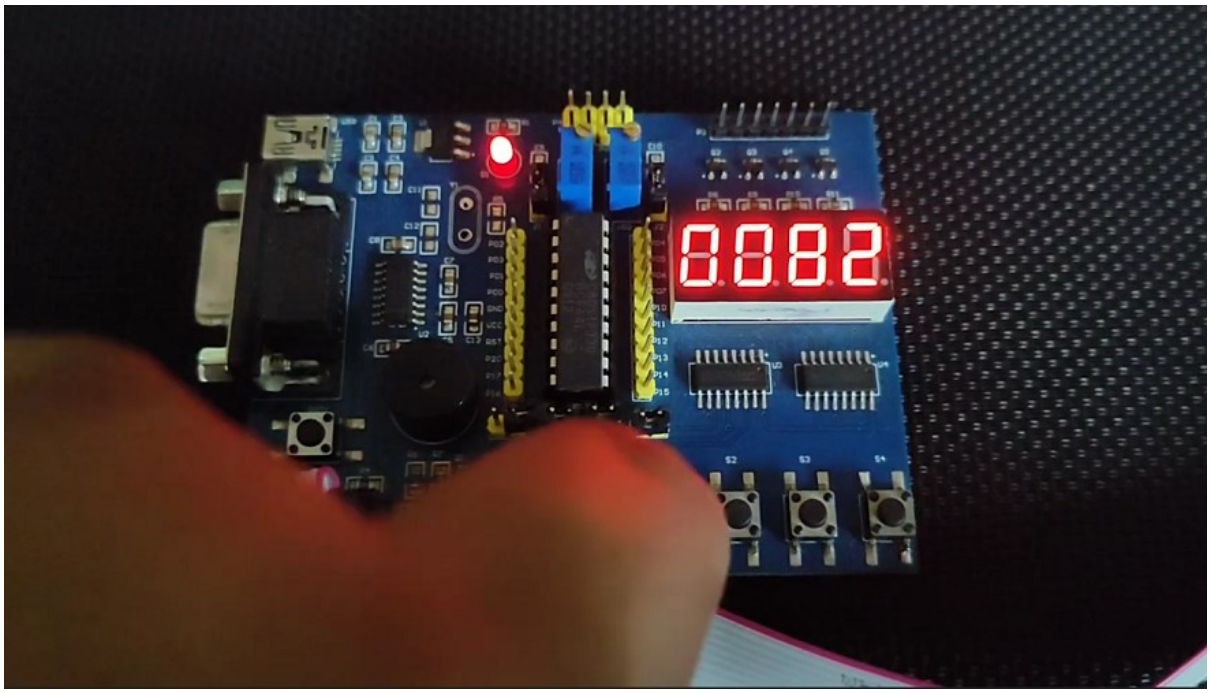
After start-up, the code begins incrementing a variable named *value* and its value is shown on the onboard seven-segment display.

```
void main(void)
{
  signed int s = 200;

  Init_Device();

  while(1)
  {
      value++;
      delay_ms(20);
      if(BUTTON == 1)
      {
         PCA0CPH2 = 0;
         delay_ms(20);
         PCA0CPH2 = 0;
         delay_ms(20);
         PCA0CPH2 = 0;
      }
      else
      {
         PCA0CPH2 = 0;
         while(s > 0)
         {
             PCA0CPH2 = 0;
             delay_ms(30);
             s--;
         }
         while(1);
      }
  };
}
```

Unless the push button which is active-low by working principle is pressed, the variable keeps incrementing. Since the watchdog timer timeout is set to 32ms, the watchdog timer needs to be reset before this time expires and this is done by clearing the *PCA0CPH2* register every 20ms. If the button is pressed, the code first enters a *while* loop that still resets the watchdog timer every 30ms without incrementing the variable *value*. This simulates an unforeseen software bug or *stuck-like* or *hang-like* condition. The code then enters into a jobless indefinite *while* loop. Since the *PCA0CPH2* register is no longer cleared inside the jobless indefinite loop, it is gradually and internally increment until the timeout is reached. After the timeout period expires, the microcontroller is reset and the process repeats.

## Demo



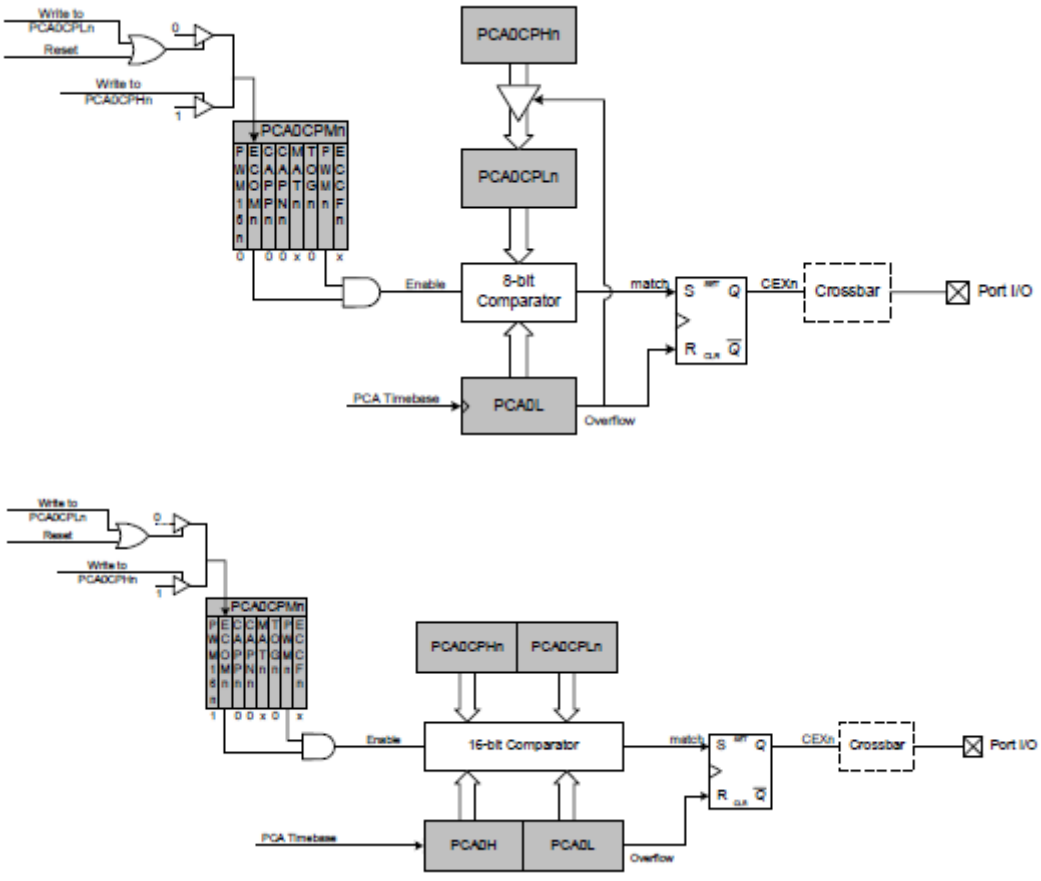

Demo video link: https://youtu.be/Tewf0BUnf8g

# PCA in PWM Mode – Clover RGB LED Lights

C8051s come with internal ***Programmable Counter Array (PCA)*** peripherals with different numbers of PCA I/O channels. PCA hardware can be used for waveform/signal input capture, high-speed frequency outputs, software timer, etc apart from PWM outputs.

The very basic thing that we can do with the internal PCA peripheral is Pulse-Width Modulation (PWM) generation. It needs no new mentioning that PWM is used in numerous applications like switch-mode power supplies, motor control, LED driving, etc.

8 or 16-bit PWMs can be obtained from C8051s and they also offer many other advanced features that are typically needed in motor and power control applications. The following block diagrams show the PCA modules in 8- and 16-bit modes respectively. Both of these modes rely on the compare-match principle to achieve PWM generation.



## Code

```
const code unsigned int LUT_1[32] =
{
	0,
	6423,
	12785,
	19023,
	25079,
	30892,
	36409,
	41574,
	46340,
	50659,
	54490,
	57796,
	60546,
	62713,
	64275,
	65219,
	65532,
	65219,
	64275,
	62713,
	60546,
	57796,
	54490,
	50659,
	46340,
	41574,
	36409,
	30892,
	25079,
	19023,
	12785,
	6423,
};


const code unsigned int LUT_2[32] =
{
	54490,
	57796,
	60546,
	62713,
	64275,
	65219,
	65532,
	65219,
	64275,
	62713,
	60546,
	57796,
	54490,
	50659,
	46340,
	41574,
	36409,
	30892,
	25079,
	19023,
	12785,
	6423,
	0,
	6423,
```

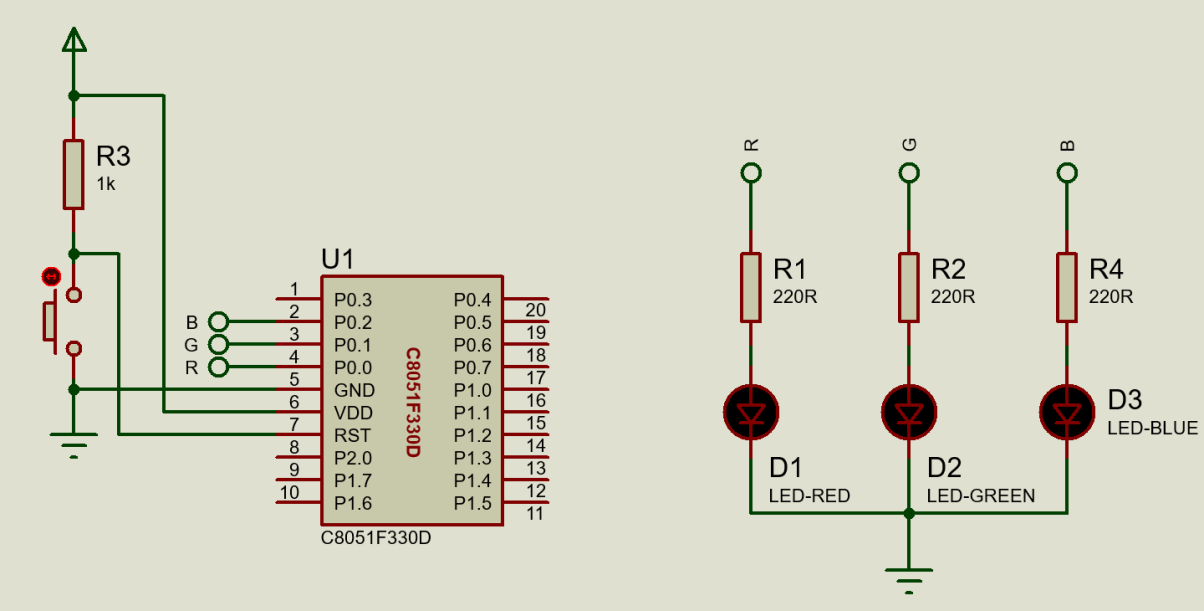
```
	12785,
	19023,
	25079,
	30892,
	36409,
	41574,
	46340,
	50659
};


const code unsigned int LUT_3[32] =
{
	60546,
	57796,
	54490,
	50659,
	46340,
	41574,
	36409,
	30892,
	25079,
	19023,
	12785,
	6423,
	0,
	6423,
	12785,
	19023,
	25079,
	30892,
	36409,
	41574,
	46340,
	50659,
	54490,
	57796,
	60546,
	62713,
	64275,
	65219,
	65532,
	65219,
	64275,
	62713
};


void PCA_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Init_Device(void);
void get_HB_LB(unsigned int value, unsigned char *HB, unsigned char *LB);
void PWM_0_duty_cycle(unsigned int value);
void PWM_1_duty_cycle(unsigned int value);
void PWM_2_duty_cycle(unsigned int value);


void main(void)
{
    unsigned int i = 0;
    unsigned char j = 0;
    unsigned char mode = 1;

    Init_Device();
```

```
while(1)
{
    switch(mode)
    {
        case 1:
        {
            PWM_1_duty_cycle(0);
            PWM_2_duty_cycle(0);

            for(j = 0; j < 6; j++)
            {
                for(i = 0; i < 32; i++)
                {
                    PWM_0_duty_cycle(LUT_1[i]);
                    delay_ms(45);
                }
            }
            break;
        }

        case 2:
        {
            PWM_0_duty_cycle(0);
            PWM_2_duty_cycle(0);

            for(j = 0; j < 6; j++)
            {
                for(i = 0; i < 32; i++)
                {
                    PWM_1_duty_cycle(LUT_1[i]);
                    delay_ms(45);
                }
            }
            break;
        }

        case 3:
        {
            PWM_0_duty_cycle(0);
            PWM_1_duty_cycle(0);

            for(j = 0; j < 6; j++)
            {
                for(i = 0; i < 32; i++)
                {
                    PWM_2_duty_cycle(LUT_1[i]);
                    delay_ms(45);
                }
            }
            break;
        }

        default:
        {
            for(j = 0; j < 10; j++)
            {
                for(i = 0; i < 32; i++)
                {
                    PWM_0_duty_cycle(LUT_1[i]);
                    PWM_1_duty_cycle(LUT_2[i]);
                    PWM_2_duty_cycle(LUT_3[i]);
                    delay_ms(200);
                }
            }
            break;
```

```
            }
        }

        mode++;
        if(mode > 3)
        {
            mode  = 0;
        }
    };
}


void PCA_Init(void)
{
    PCA0CN = 0x40;
    PCA0MD &= ~0x40;
    PCA0MD = 0x02;
    PCA0CPM0 = 0xC2;
    PCA0CPM1 = 0xC2;
    PCA0CPM2 = 0xC2;
    PCA0L = 0xC0;
    PCA0H = 0x4F;
    PCA0CPL0 = 0xFF;
    PCA0CPL1 = 0xFF;
    PCA0CPL2 = 0xFF;
    PCA0CPH0 = 0xFF;
    PCA0CPH1 = 0xFF;
    PCA0CPH2 = 0xFF;
}


void Port_IO_Init(void)
{
    // P0.0  -  CEX0 (PCA),  Push-Pull,  Digital
    // P0.1  -  CEX1 (PCA),  Push-Pull,  Digital
    // P0.2  -  CEX2 (PCA),  Push-Pull,  Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Unassigned,  Open-Drain, Digital
    // P1.6  -  Unassigned,  Open-Drain, Digital
    // P1.7  -  Unassigned,  Open-Drain, Digital

    P0MDOUT = 0x07;
    XBR1 = 0x43;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Init_Device(void)
{
    PCA_Init();
```

```
    Port_IO_Init();
    Oscillator_Init();
}


void get_HB_LB(unsigned int value, unsigned char *HB, unsigned char *LB)
{
    *LB = (unsigned char)(value & 0x00FF);
    *HB = (unsigned char)((value & 0xFF00) >> 0x08);
}



void PWM_0_duty_cycle(unsigned int value)
{
    unsigned char hb = 0x00;
    unsigned char lb = 0x00;

    get_HB_LB(value, &hb, &lb);

    PCA0CPL0 = lb;
    PCA0CPH0 = hb;
    CCF0_bit = 0;
}


void PWM_1_duty_cycle(unsigned int value)
{
    unsigned char hb = 0x00;
    unsigned char lb = 0x00;

    get_HB_LB(value, &hb, &lb);

    PCA0CPL1 = lb;
    PCA0CPH1 = hb;
    CCF1_bit = 0;
}


void PWM_2_duty_cycle(unsigned int value)
{
    unsigned char hb = 0x00;
    unsigned char lb = 0x00;

    get_HB_LB(value, &hb, &lb);

    PCA0CPL2 = lb;
    PCA0CPH2 = hb;
    CCF2_bit = 0;
}
```

## Schematic




## Explanation

This demo uses multi-channel 16-bit PWMs to smoothly change the colours of three RGB LEDs. The LEDs are embedded in a leaf-like board from DFRobot called clover light. 8-bit PWM is functionally the same as 16-bit PWM. 8-bit PWM has a lower resolution but it can be used for high-frequency applications.

Red, Blue and Green colours from the LED are mixed in different ratios to form different colours. This mixing of colours is achieved using PWM and so all three PCA channels are used to drive the LEDs. Since PWM is output and not input, all of the PCA pins are set as digital push-pull outputs in the crossbar. PWM pins are labelled as ***CEXn*** pins.

```
void Port_IO_Init(void)
{
    // P0.0  -  CEX0 (PCA),  Push-Pull,  Digital
    // P0.1  -  CEX1 (PCA),  Push-Pull,  Digital
    // P0.2  -  CEX2 (PCA),  Push-Pull,  Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Unassigned,  Open-Drain, Digital
    // P1.6  -  Unassigned,  Open-Drain, Digital
    // P1.7  -  Unassigned,  Open-Drain, Digital

    P0MDOUT = 0x07;
    XBR1 = 0x43;
}
```

The system clock source is set to 12.5MHz and so one tick is 80ns.

```
void Oscillator_Init(void)
{
    OSCICN = 0x82;
}
```

Timing is very important because PWM signals are timed outputs with variable pulse widths and so variable or unstable time bases would result in odd irregular waveforms.

A clock derived from the system clock or some other source is fed to the PCA peripheral and its counter keeps counting.

PWM is generated by comparing and matching the values of ***PCA0CPn*** and ***PCA0*** registers. A PWM output pin is set when the contents of PCA0 counter match with the contents of PCA0CPn registers and it is reset when the contents of PCA0 counter overflow.

We can optionally load the PCA counter. This has no effect on PWM frequency but it does affect PWM duty cycle.

In this example, the PCA is set to generate three channel PWMs at approximately 46Hz. The PCA clock prescalar is set to 4 and so the 12.25MHz clock is scaled down as per the following calculation:

$$\boldsymbol{PCA\ Clock\ Frequency} = \frac{\boldsymbol{System\ Clock}}{\boldsymbol{Prescalar}}$$

$$\boldsymbol{PCA\ Clock\ Frequency} = \frac{\boldsymbol{12.25\ MHz}}{\boldsymbol{4}} = \boldsymbol{3.0625\ MHz}$$

Since we are using 16-bit resolution PWM in this example, the PWM frequency is calculated as follows:

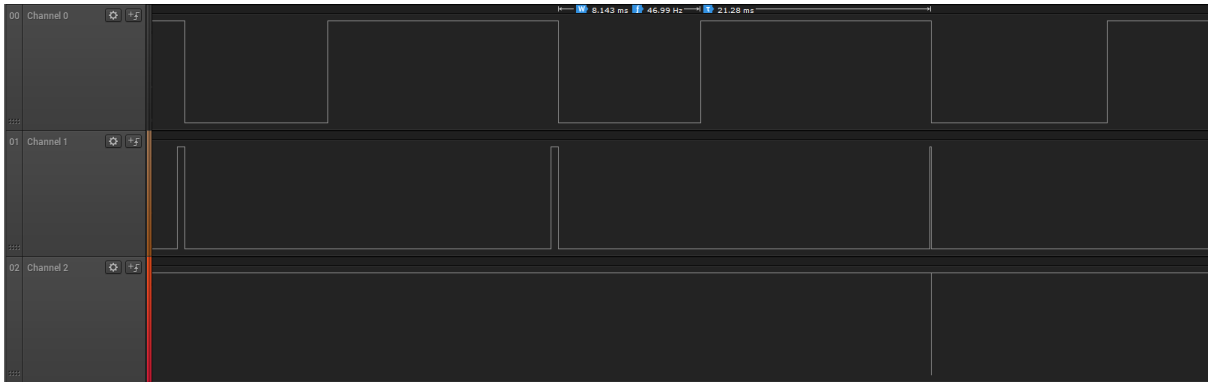
$$\boldsymbol{PWM\ Frequency} = \frac{\boldsymbol{PCA\ Clock\ Frequency}}{\boldsymbol{2^{PWM\ Resolution\ as\ Bit\ Size}}}$$

$$\boldsymbol{PCA\ Clock\ Frequency} = \frac{\boldsymbol{3.0625\ MHz}}{\boldsymbol{65536}} = \boldsymbol{46.73\ Hz}$$

```
void PCA_Init(void)
{
    PCA0CN = 0x40;
    PCA0MD &= ~0x40;
    PCA0MD = 0x02;
    PCA0CPM0 = 0xC2;
    PCA0CPM1 = 0xC2;
    PCA0CPM2 = 0xC2;
    PCA0L = 0xC0;
    PCA0H = 0x4F;
    PCA0CPL0 = 0xFF;
```

```
    PCA0CPL1 = 0xFF;
    PCA0CPL2 = 0xFF;
    PCA0CPH0 = 0xFF;
    PCA0CPH1 = 0xFF;
    PCA0CPH2 = 0xFF;
}
```

Note that the PCA counter is preloaded with a value and this is just to show that it has no effect on PWM frequency. The Saleae Logic analyser capture shown below shows the frequency of the three PWM waveforms.




According to the datasheet PWM duty cycle for 8-bit and 16-bit PWMs are given as follows:

$$\boldsymbol{PWM\ Duty\ Cycle} = \frac{(\mathbf{256} - \boldsymbol{PCA0CPHn})}{\mathbf{256}}\ (\boldsymbol{for}\ \mathbf{8}\ \boldsymbol{bit\ PWM})$$

$$\boldsymbol{PWM\ Duty\ Cycle} = \frac{(\mathbf{65536} - \boldsymbol{PCA0CPn})}{\mathbf{65536}}\ (\boldsymbol{for}\ \mathbf{16}\ \boldsymbol{bit\ PWM})$$

The following functions can be used to alter the duty cycles of respective PWM channels. These functions take 16-bit input and break the input into two bytes. These bytes are then loaded to respective *PCA0CPLn* and *PCA0CPHn* registers. PWM output is updated with the new values in *PCA0CP* registers upon clearing the capture-compare flag (*CCFn*).

```
void PWM_0_duty_cycle(unsigned int value)
{
    unsigned char hb = 0x00;
    unsigned char lb = 0x00;

    get_HB_LB(value, &hb, &lb);

    PCA0CPL0 = lb;
    PCA0CPH0 = hb;
    CCF0_bit = 0;
}

void PWM_1_duty_cycle(unsigned int value)
{
    unsigned char hb = 0x00;
    unsigned char lb = 0x00;
```

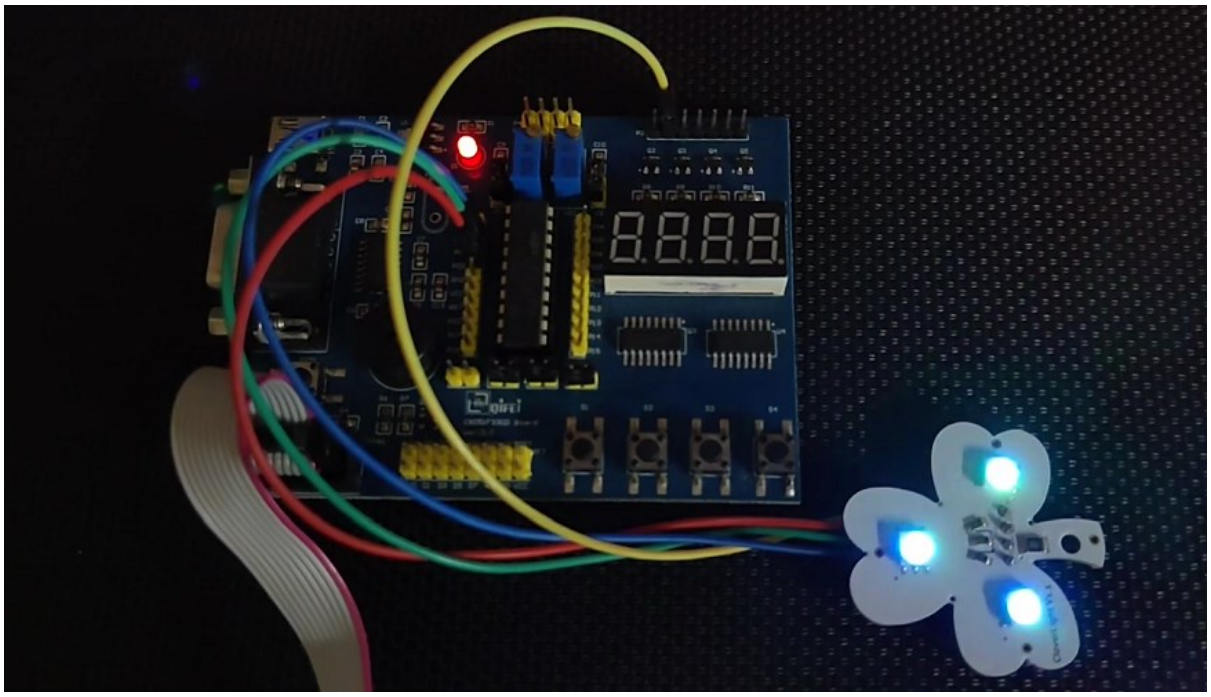
```
    get_HB_LB(value, &hb, &lb);

    PCA0CPL1 = lb;
    PCA0CPH1 = hb;
    CCF1_bit = 0;
}

void PWM_2_duty_cycle(unsigned int value)
{
    unsigned char hb = 0x00;
    unsigned char lb = 0x00;

    get_HB_LB(value, &hb, &lb);

    PCA0CPL2 = lb;
    PCA0CPH2 = hb;
    CCF2_bit = 0;
}
```

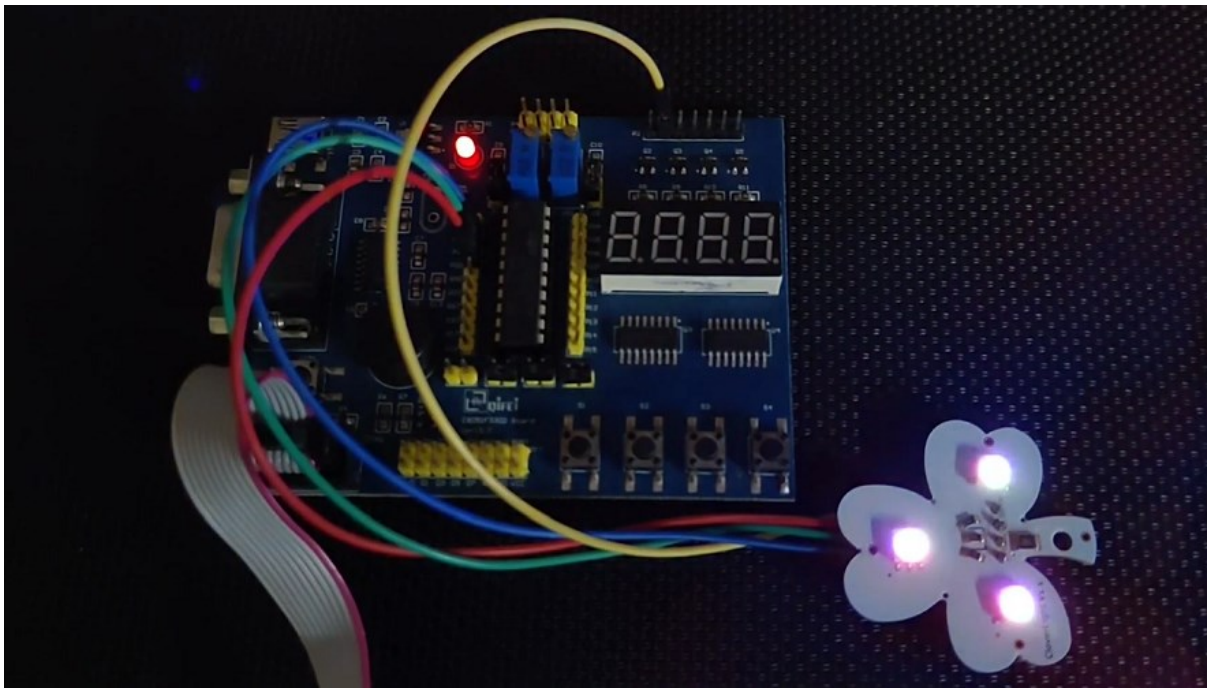
Inside the main loop, all three PWM channels are driven using the data from respective lookup tables. These lookup tables contain sinusoidal PWM duty cycle values.

```
void main(void)
{
    unsigned int i = 0;
    unsigned char j = 0;
    unsigned char mode = 1;

    Init_Device();

    while(1)
    {
        switch(mode)
        {
            case 1:
            {
                PWM_1_duty_cycle(0);
                PWM_2_duty_cycle(0);

                for(j = 0; j < 6; j++)
                {
                    for(i = 0; i < 32; i++)
                    {
                        PWM_0_duty_cycle(LUT_1[i]);
                        delay_ms(45);
                    }
                }
                break;
            }

            case 2:
            {
                PWM_0_duty_cycle(0);
                PWM_2_duty_cycle(0);

                for(j = 0; j < 6; j++)
                {
                    for(i = 0; i < 32; i++)
                    {
                        PWM_1_duty_cycle(LUT_1[i]);
                        delay_ms(45);
                    }
                }
                break;
```

```
            }

            case 3:
            {
                PWM_0_duty_cycle(0);
                PWM_1_duty_cycle(0);

                for(j = 0; j < 6; j++)
                {
                    for(i = 0; i < 32; i++)
                    {
                        PWM_2_duty_cycle(LUT_1[i]);
                        delay_ms(45);
                    }
                }
                break;
            }

            default:
            {
                for(j = 0; j < 10; j++)
                {
                    for(i = 0; i < 32; i++)
                    {
                        PWM_0_duty_cycle(LUT_1[i]);
                        PWM_1_duty_cycle(LUT_2[i]);
                        PWM_2_duty_cycle(LUT_3[i]);
                        delay_ms(200);
                    }
                }
                break;
            }
        }

        mode++;
        if(mode > 3)
        {
            mode  = 0;
        }
    };
}
```

## Demo



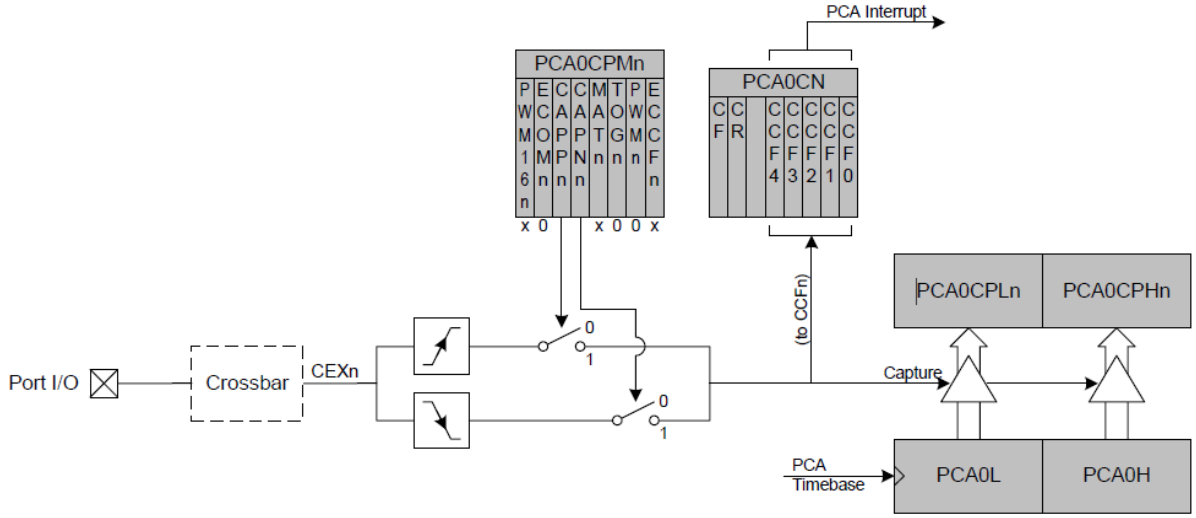


Demo video link: https://youtu.be/CHn6_LVyuLY

## PCA in Capture Mode – TCS3200 Colour Sensor

In many applications, we need to measure pulse widths and frequencies of signals and in such cases, accurate measurement can be done using a capture module. We can also use external interrupts and timers too but with capture hardware things are far more accurate and precise.

The block diagram below shows the PCA hardware in edge-capture mode. With a defined time base, PCA keeps ticking. When there is an edge capture event, PCA counter values are copied to capture-compare registers. The type of edge to be captured and many other things can be set by tweaking internal registers and it is possible to capture falling and rising edges. Capturing consecutive like edges results in deducing period or frequency while capturing consecutive edges that have different polarities result in deducing pulse width.




### Code

```
#include "Waveshare_RGB_LCD.c"
#include "lcd_print_rgb.c"


#define RED_filter      0x00
#define BLUE_filter     0x01
#define CLEAR_filter    0x02
#define GREEN_filter    0x03


#define S2              P1_4_bit
#define S3              P1_5_bit


unsigned long count = 0;
unsigned int overflow = 0;
unsigned int past_capture = 0;
unsigned int current_capture = 0;


void PCA_Init(void);
void Voltage_Reference_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
```

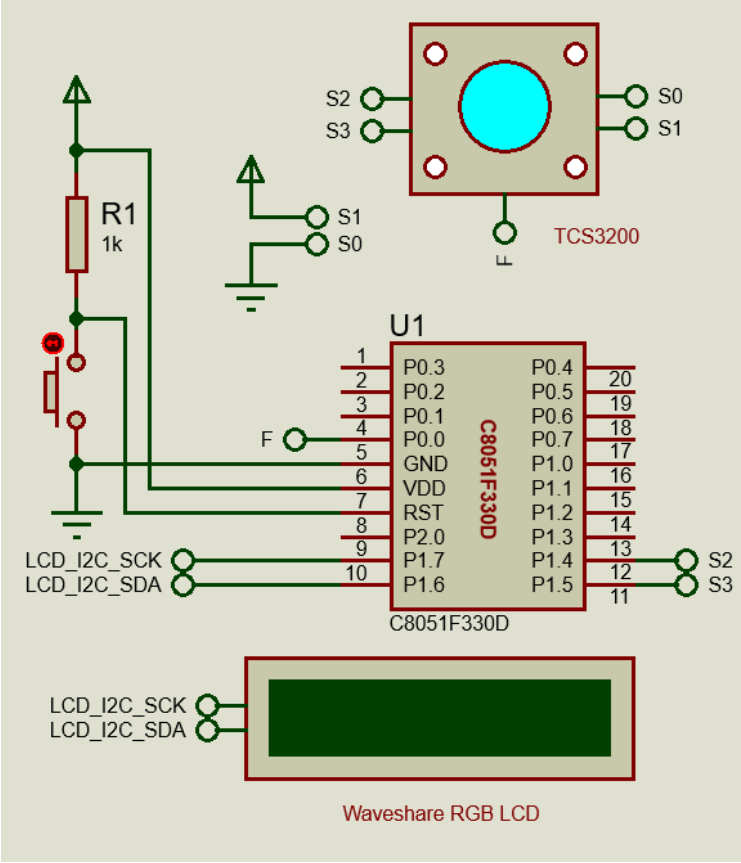
```
void Init_Device(void);
unsigned int get_capture_count(void);


void PCA0_ISR(void)
iv IVT_ADDR_EPCA0
ilevel 0
ics ICS_AUTO
{
    if(CF_bit)
    {
        overflow++;
        CF_bit = 0;
    }

    if(CCF0_bit)
    {
        current_capture = get_capture_count();
        count = ((overflow << 16) + (current_capture - past_capture));
        past_capture = current_capture;
        overflow = 0x00000;
        current_capture = 0;
        CCF0_bit = 0;
    }
}


void main(void)
{
    unsigned long f = 0;
    unsigned long fr = 0;
    unsigned long fg = 0;
    unsigned long fb = 0;
    unsigned long fc = 0;
    unsigned char mode = 0;

    Init_Device();
    RGB_LCD_init();
    LCD_clear_home();

    LCD_goto(0, 0);
    LCD_putstr(" R%  G%  B%  fC");

    while(1)
    {
        switch(mode)
        {
            case RED_filter:
            {
                S2 = 0;
                S3 = 0;
                break;
            }
            case BLUE_filter:
            {
                S2 = 0;
                S3 = 1;
                break;
            }
            case CLEAR_filter:
            {
                S2 = 1;
                S3 = 0;
                break;
            }
```

```
            case GREEN_filter:
            {
                S2 = 1;
                S3 = 1;
                break;
            }
        }
        delay_ms(100);

        f = (1020833 / ((float)count));

        switch(mode)
        {
            case RED_filter:
            {
                fr = f;
                break;
            }
            case BLUE_filter:
            {
                fb = f;
                break;
            }
            case CLEAR_filter:
            {
                fc = f;
                break;
            }
            case GREEN_filter:
            {
                fg = f;
                break;
            }
        }

        mode++;

        if(mode > 3)
        {
            fr = (((float)fr / (float)fc) * 100);
            fg = (((float)fg / (float)fc) * 100);
            fb = (((float)fb / (float)fc) * 100);

            print_C(0, 1, fr);
            print_C(4, 1, fg);
            print_C(8, 1, fb);
            print_I(12, 1, fc);

            fr <<= 1;
            fg <<= 1;
            fb <<= 1;
            set_RGB(fr, fg, fb);

            mode = 0;
            delay_ms(400);
        }
    };
}


void PCA_Init(void)
{
    PCA0CN = 0x40;
    PCA0MD &= ~0x40;
    PCA0MD = 0x01;
    PCA0CPM0 = 0x11;
```

```
}


void Voltage_Reference_Init(void)
{
    REF0CN = 0x08;
}


void Port_IO_Init(void)
{
    // P0.0  -  CEX0 (PCA),  Open-Drain, Digital
    // P0.1  -  Unassigned,  Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Skipped,     Push-Pull,  Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P1MDOUT = 0xF0;
    P1SKIP = 0xF0;
    XBR1 = 0x41;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
{
    IE = 0x80;
    EIE1 = 0x10;
}


void Init_Device(void)
{
    PCA_Init();
    Voltage_Reference_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}


unsigned int get_capture_count(void)
{
    unsigned char lb = 0x00;
    unsigned char hb = 0x00;
    unsigned int cnt = 0x0000;

    hb = PCA0CPH0;
```

```
    lb = PCA0CPL0;

    cnt = hb;
    cnt <<= 8;
    cnt |= lb;

    return cnt;
}
```

## Schematic




## Explanation

Some sensors do not communicate using UART, I2C or SPI methods. Such is the case with the TCS3200 colour sensor from TAOS. This sensor is a light-to-frequency output and so we need to convert frequency output to usable data that represents reflected light colour ratios. TCS3200 has four types of photodiode arrays for sensing colours, three of which are sensitive to red, green and blue colours while a single one consists of a clear or no colour filter. Thus, PCA capture is needed to extract colour info.

Since capture mode is time-sensitive, the system clock setting must be known. In this example, the system clock is set to 12.25MHz.

```
void Oscillator_Init(void)
{
    OSCICN = 0x82;
}
```

The PCA hardware's channel 0 is set to capture failing edges with PCA counter overflow and PCA channel 0 (*CEX0*) interrupts both enabled and a PCA counter time base source of system clock divided by 12 selected.

```
void PCA_Init(void)
{
    PCA0CN = 0x40;
    PCA0MD &= ~0x40;
    PCA0MD = 0x01;
    PCA0CPM0 = 0x11;
}
```

With these settings, the tick resolution of the PCA counter is as follows:

$$\boldsymbol{PCA\ Counter\ Timebase\ Frequency} = \frac{\boldsymbol{System\ Clock}}{\boldsymbol{Prescalar}} = \frac{\boldsymbol{12.25\ MHz}}{\boldsymbol{12}} = \boldsymbol{1.02083\ MHz}$$

$$\boldsymbol{PCA\ Counter\ Tick} = \frac{\boldsymbol{1}}{\boldsymbol{PCA\ \ Timebase\ Frequency}} = \frac{\boldsymbol{1}}{\boldsymbol{1.02083\ MHz}} \approx \boldsymbol{1\mu s}$$

So, in theory, the PCA module can measure the following maximum and minimum frequencies as per Nyquist criterion:

$$\boldsymbol{Max.Measurement\ Frequency} = \frac{\boldsymbol{PCA\ Counter\ Timebase\ Frequency\ Clock}}{\boldsymbol{2}}$$

$$\boldsymbol{Max.Measurement\ Frequency} = \frac{\boldsymbol{1.02083\ MHz}}{\boldsymbol{2}} \approx \boldsymbol{510\ kHz}$$

$$\boldsymbol{Min.Measurement\ Frequency} = \frac{\boldsymbol{PCA\ Counter\ Timebase\ Frequency\ Clock}}{\boldsymbol{Max.PCA\ Counter\ Value}}$$

$$\boldsymbol{Min.Measurement\ Frequency} = \frac{\boldsymbol{1.02083\ MHz}}{\boldsymbol{65535}} \approx \boldsymbol{16\ Hz}$$

Referring to the electrical characteristics of the sensor as shown below, three ranges of frequency outputs can be obtained from it. The first one would not be suitable as the frequency range exceeds the hardware limit of the PCA module as per the settings discussed. The second and third ones are the preferred ones. I have chosen the last one because this range will not overload the PCA module and the readings would be computed quickly. Note with higher frequency settings, the precision of measurement is largely increased but computation and back-calculation times are increased too.

**Electrical Characteristics at $T_A$ = 25°C, $V_{DD}$ = 5 V (unless otherwise noted)**

| PARAMETER | | TEST CONDITIONS | MIN | TYP | MAX | UNIT |
|---|---|---|---|---|---|---|
| $V_{OH}$ | High-level output voltage | $I_{OH}$ = –2 mA | 4 | 4.5 | | V |
| $V_{OL}$ | Low-level output voltage | $I_{OL}$ = 2 mA | | 0.25 | 0.40 | V |
| $I_{IH}$ | High-level input current | | | | 5 | µA |
| $I_{IL}$ | Low-level input current | | | | 5 | µA |
| $I_{DD}$ | Supply current | Power-on mode | | 1.4 | 2 | mA |
| | | Power-down mode | | | 0.1 | µA |
| | Full-scale frequency (See Note 4) | S0 = H, S1 = H | 500 | 600 | | kHz |
| | | S0 = H, S1 = L | 100 | 120 | | kHz |
| | | S0 = L, S1 = H | 10 | 12 | | kHz |
| | Temperature coefficient of responsivity | λ ≤ 700 nm, –25°C ≤ $T_A$ ≤ 70°C | | ±200 | | ppm/°C |
| $k_{SVS}$ | Supply voltage sensitivity | $V_{DD}$ = 5 V ±10% | | ±0.5 | | %/V |

NOTE 4: Full-scale frequency is the maximum operating frequency of the device without saturation.

Now that the theories are established, let us see what happens when falling edges are captured. Whenever a falling edge is detected by the PCA module, the value of the PCA counter is immediately copied to the respective capture-compare register set. Two successive like edges will provide two time captures. The difference between these successive captures results in finding the period. We also need to take into account of any PCA counter overflow that has occurred during these measurements. All these are done inside the PCA interrupt subroutine.

```
void PCA0_ISR(void)
iv IVT_ADDR_EPCA0
ilevel 0
ics ICS_AUTO
{
    if(CF_bit)
    {
        overflow++;
        CF_bit = 0;
    }

    if(CCF0_bit)
    {
        current_capture = get_capture_count();
        count = ((overflow << 16) + (current_capture - past_capture));
        past_capture = current_capture;
        overflow = 0x00000;
        current_capture = 0;
        CCF0_bit = 0;
    }
}
```

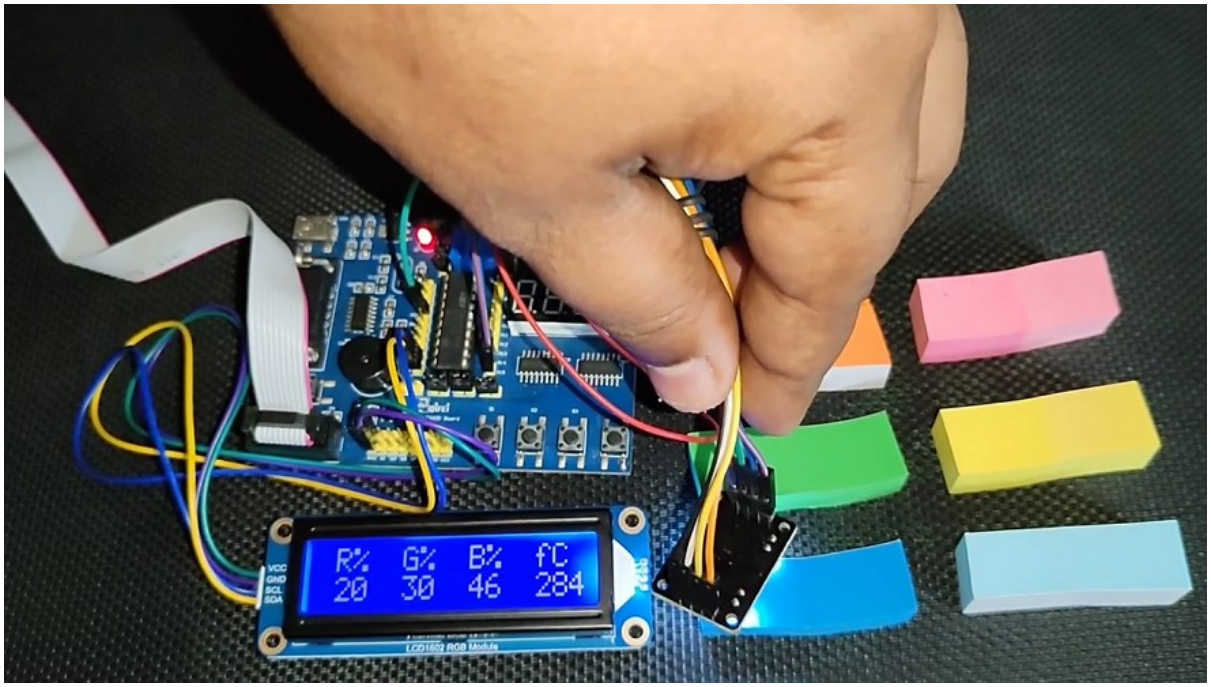
Inside the main loop, the photodiode arrays of the sensor are switched by switching the states of its *S2* and *S3* inputs. After switching each filter, the respective frequency reading is extracted and stored. The frequency readings are converted to percentages with respect to the clear filter. These percentages represent the amount of colour reflected with respect to the full visible light spectrum. The Waveshare RGB LCD's backlight colour tries to mimic the detected colour. This is done by providing RGB data from TCS3200 colour sensor to the PCA9633 IC of the LCD's RGB backlight driver.

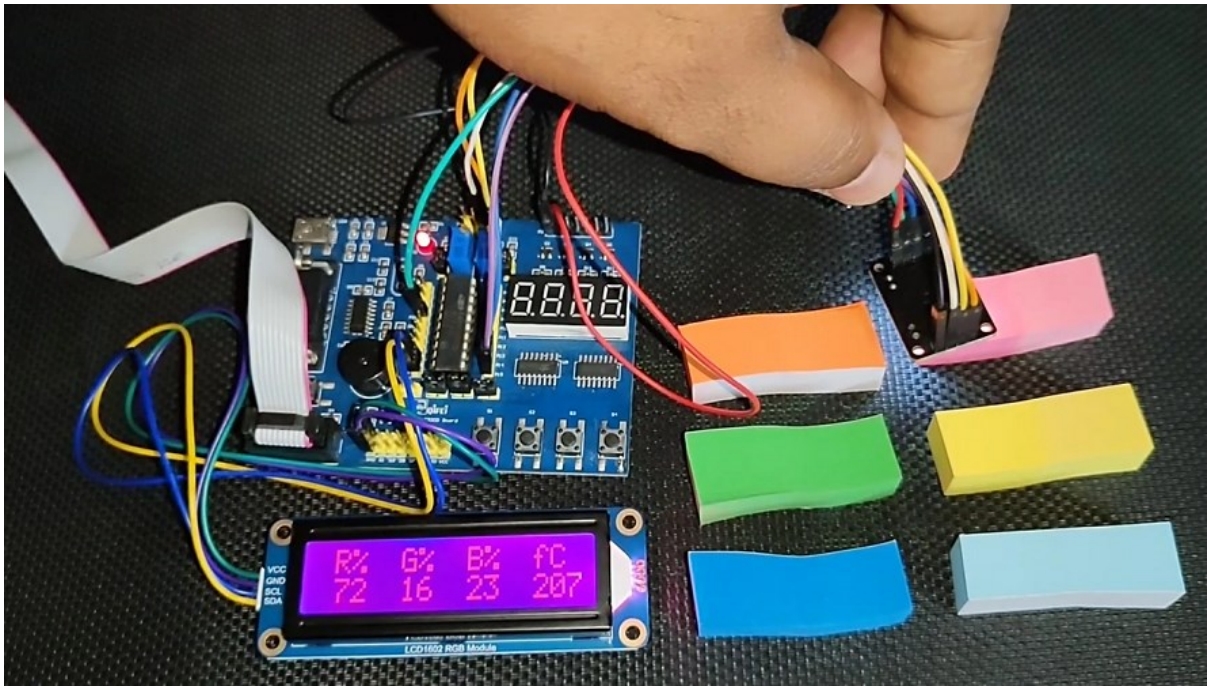
```
void main(void)
{
    unsigned long f = 0;
    unsigned long fr = 0;
    unsigned long fg = 0;
    unsigned long fb = 0;
```

```
    unsigned long fc = 0;
    unsigned char mode = 0;

    Init_Device();
    RGB_LCD_init();
    LCD_clear_home();

    LCD_goto(0, 0);
    LCD_putstr(" R%  G%  B%  fC");

    while(1)
    {
        switch(mode)
        {
            case RED_filter:
            {
                S2 = 0;
                S3 = 0;
                break;
            }
            case BLUE_filter:
            {
                S2 = 0;
                S3 = 1;
                break;
            }
            case CLEAR_filter:
            {
                S2 = 1;
                S3 = 0;
                break;
            }
            case GREEN_filter:
            {
                S2 = 1;
                S3 = 1;
                break;
            }
        }
        delay_ms(100);

        f = (1020833 / ((float)count));

        switch(mode)
        {
            case RED_filter:
            {
                fr = f;
                break;
            }
            case BLUE_filter:
            {
                fb = f;
                break;
            }
            case CLEAR_filter:
            {
                fc = f;
                break;
            }
            case GREEN_filter:
            {
                fg = f;
                break;
            }
        }
```

```
        mode++;

        if(mode > 3)
        {
            fr = (((float)fr / (float)fc) * 100);
            fg = (((float)fg / (float)fc) * 100);
            fb = (((float)fb / (float)fc) * 100);

            print_C(0, 1, fr);
            print_C(4, 1, fg);
            print_C(8, 1, fb);
            print_I(12, 1, fc);

            fr <<= 1;
            fg <<= 1;
            fb <<= 1;
            set_RGB(fr, fg, fb);

            mode = 0;
            delay_ms(400);
        }
    };
}
```

## Demo



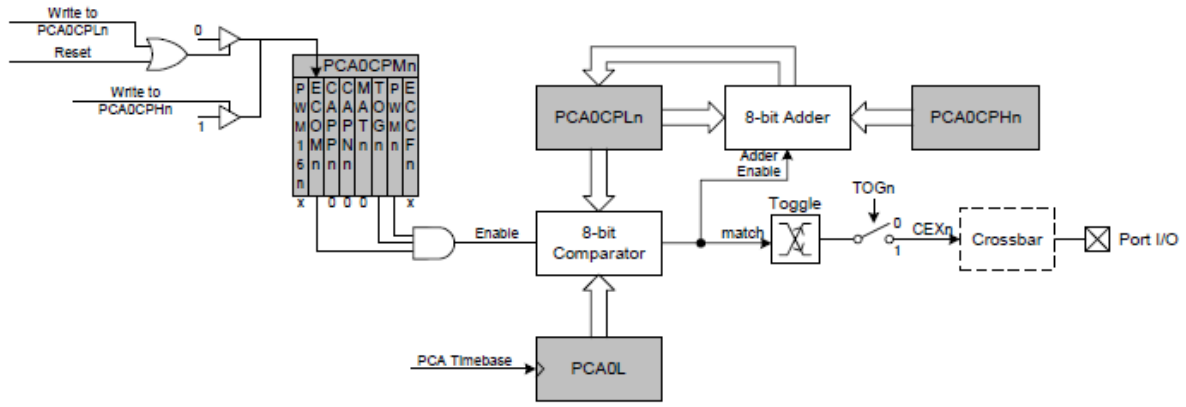




Demo video link: https://youtu.be/U8M_c7FBQAE

## PCA in Frequency Output Mode – IR UART

In the early days of modern cell phones, sharing files between two cell phones was done using IrDA. There was no Wi-Fi or Bluetooth transfer available at that time. In many applications as such, short-range line-of-sight (LOS) wireless communication is needed and for cases, the infrared medium is the best solution and C8051s come with the right hardware for making IR transmitters because of the PCA peripheral's ability to generate frequency outputs. Such outputs can be used to generate modulation or carrier waves which can then be mixed with data signals to make desired bursts of IR data.

The block diagram below shows the PCA peripheral in frequency output mode and from this diagram, we can notice that the compare-match principle is used to achieve frequency output.




### Code

#### IR TX main.c

```
#define LED_DOUT    P1_6_bit
#define LED_CLK     P1_5_bit
#define LED_LATCH   P1_7_bit

#define ADC_res     1023.0
#define VDD_mv      3200.0

#define freq        (6125000 / 2 /38000)


unsigned char i = 0;
register unsigned char val = 0;
unsigned int value = 0;

const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
```

```
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x9C, // degree
  0xC6  // C
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


void PCA_Init(void);
void Timer_Init(void);
void UART_Init(void);
void ADC_Init(void);
void Voltage_Reference_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);


void Timer_ISR(void)
iv IVT_ADDR_ET3
ilevel 0
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (value / 10);
            break;
        }
        case 1:
        {
            val = (value % 10);
            break;
        }
        case 2:
        {
            val = 10;
            break;
        }
        case 3:
        {
            val = 11;
            break;
        }
    }

    segment_write(val, i);

    i++;

    if(i > 3)
    {
```

```
       i = 0;
    }

    TMR3CN &= 0x7F;
}


void main(void)
{
    unsigned int t = 0;
    char tmp = 0;
    Init_Device();
    UART1_Init(1200);
    ADC1_Init_Advanced(_INTERNAL_REF | _RIGHT_ADJUSTMENT);

    while(1)
    {
        t = (ADC1_Get_Sample(16) * VDD_mv);
        t /= 1023.0;
        t = (((float)t - 776.0) / 2.86);
        value = t;
        tmp = t;
        UART1_Write(0xAA);
        UART1_Write((tmp / 10) + 0x30);
        UART1_Write((tmp % 10) + 0x30);
        delay_ms(400);
    };
}


void PCA_Init(void)
{
    PCA0CN = 0x40;
    PCA0MD &= ~0x40;
    PCA0MD = 0x02;
    PCA0CPM1 = 0x46;
    PCA0CPH1 = 0x14;
}


void Timer_Init(void)
{
    TMR3CN = 0x04;
    TMR3RLL = 0x01;
    TMR3RLH = 0xFE;
}


void UART_Init(void)
{
    SCON0 = 0x10;
}


void ADC_Init(void)
{
    AMX0P = 0x10;
    AMX0N = 0x11;
    ADC0CF = 0xF0;
    ADC0CN = 0x80;
}


void Voltage_Reference_Init(void)
{
```

```
    REF0CN = 0x0F;
}


void Port_IO_Init(void)
{
    // P0.0  -  CEX0 (PCA),  Open-Drain, Digital
    // P0.1  -  CEX1 (PCA),  Push-Pull,  Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  TX0 (UART0), Push-Pull,  Digital
    // P0.5  -  RX0 (UART0), Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDOUT = 0x12;
    P1MDOUT = 0xE0;
    P1SKIP = 0xE0;
    XBR0 = 0x01;
    XBR1 = 0x42;
}


void Oscillator_Init(void)
{
    OSCICN = 0x81;
}


void Interrupts_Init(void)
{
    IE = 0x80;
    EIE1 = 0x80;
}


void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    UART_Init();
    ADC_Init();
    Voltage_Reference_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}


void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
```

```
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}


void segment_write(unsigned char disp, unsigned char pos)
{
    LED_LATCH = 0;
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}
```

## IR RX main.c

```
#define LED_DOUT          P1_6_bit
#define LED_CLK           P1_5_bit
#define LED_LATCH         P1_7_bit


unsigned char i = 0;
register unsigned char val = 0;
unsigned int value = 0;
unsigned char cnt = 0;

unsigned char rx_buffer[3] = {0x00, 0x00, 0x00};


const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
  0x90, // 9
  0x9C, // degree
  0xC6  // C
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};
```

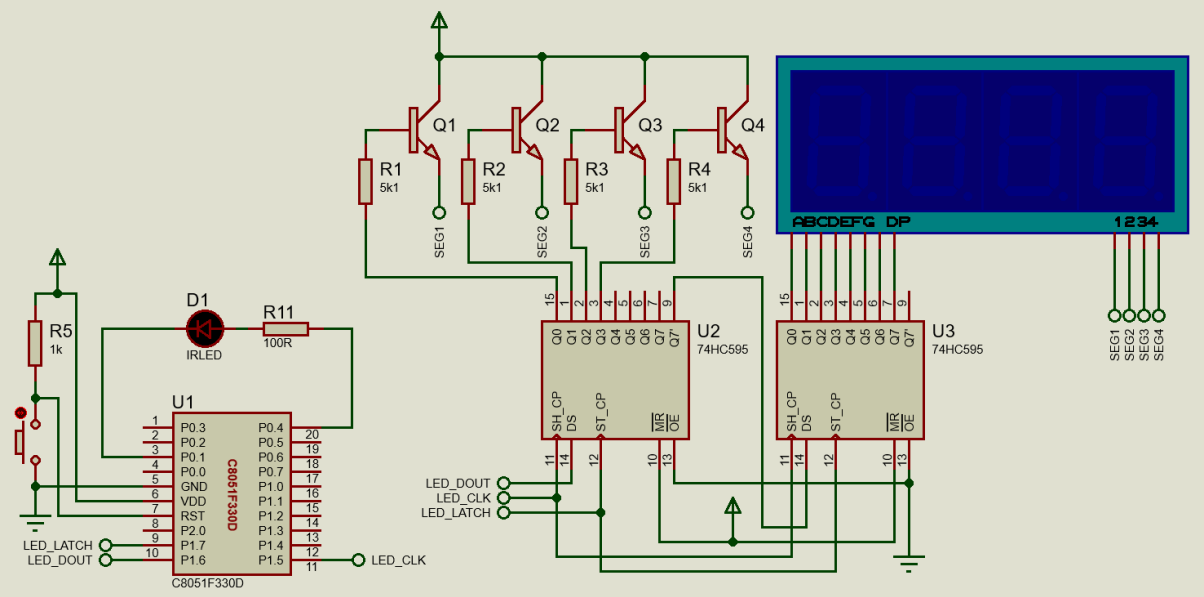
```
void PCA_Init(void);
void Timer_Init(void);
void UART_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);


void UART0_ISR(void)
iv IVT_ADDR_ES0
ilevel 0
ics ICS_AUTO
{
    rx_buffer[cnt++] = UART_Read();
    RI0_bit = 0;
}


void Timer_ISR(void)
iv IVT_ADDR_ET3
ilevel 1
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (value / 10);
            break;
        }
        case 1:
        {
            val = (value % 10);
            break;
        }
        case 2:
        {
            val = 10;
            break;
        }
        case 3:
        {
            val = 11;
            break;
        }
    }

    segment_write(val, i);

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}


void main(void)
{
```

```
    unsigned char tmp = 0x00;
    unsigned char temp = 0x00;

    Init_Device();
    UART1_Init(1200);

    while(1)
    {
        if(cnt >= 3)
        {
            if(rx_buffer[0] == 0xAA)
            {
               temp = (rx_buffer[1] - 0x30);
               tmp = (temp * 10);
               temp = (rx_buffer[2] - 0x30);
               tmp += temp;

               value = tmp;
            }
            cnt = 0;
        }
    };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}


void Timer_Init(void)
{
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}


void UART_Init(void)
{
    SCON0 = 0x10;
}


void Port_IO_Init(void)
{
    // P0.0  -  Unassigned,  Open-Drain, Digital
    // P0.1  -  Unassigned,  Open-Drain, Digital
    // P0.2  -  Unassigned,  Open-Drain, Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  TX0 (UART0), Push-Pull,  Digital
    // P0.5  -  RX0 (UART0), Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Unassigned,  Open-Drain, Digital
    // P1.4  -  Unassigned,  Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital
```

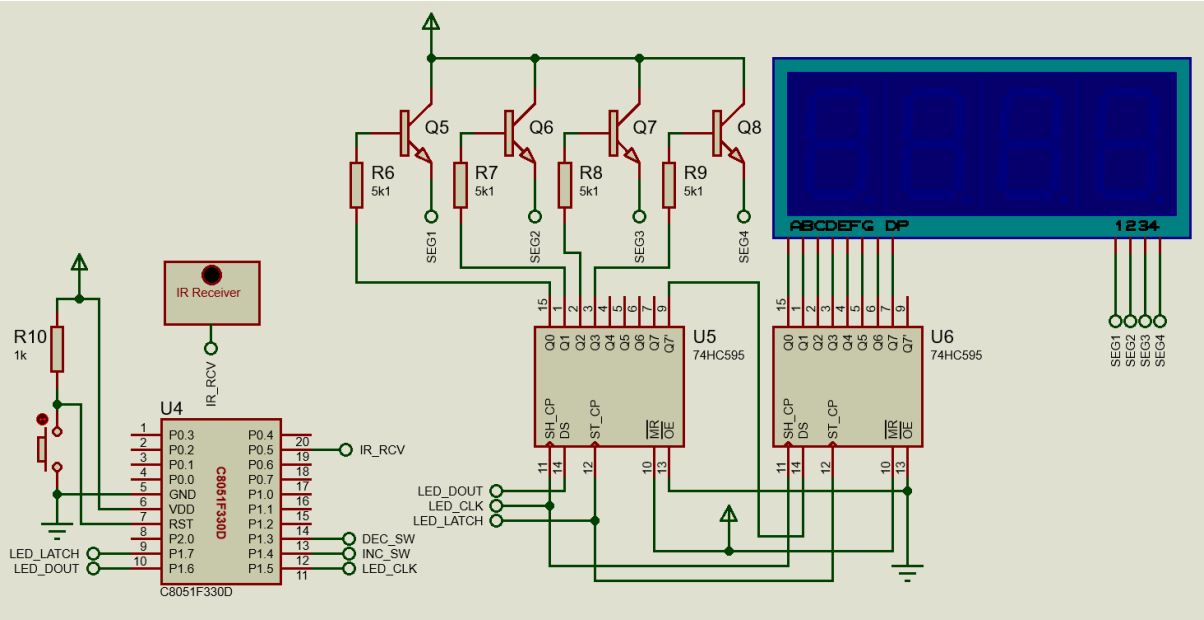
```
    P0MDOUT = 0x10;
    P1MDOUT = 0xE0;
    P1SKIP = 0xE0;
    XBR0 = 0x01;
    XBR1 = 0x40;
}


void Oscillator_Init(void)
{
    OSCICN = 0x82;
}


void Interrupts_Init(void)
{
    IE = 0x90;
    EIE1 = 0x80;
}


void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    UART_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}


void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}


void segment_write(unsigned char disp, unsigned char pos)
{
    LED_LATCH = 0;
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}
```

## Schematic

### IR TX



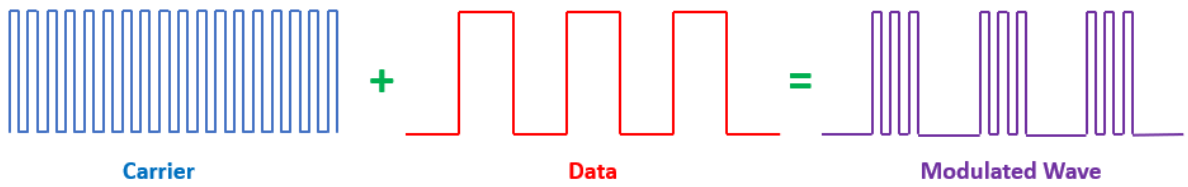

### IR RX



## Explanation

IR UART is simply ordinary slow-speed UART but over an infrared (IR) medium. It is slow in terms of the baud rate or the number of bits transferred per second. Higher baud rates are not suitable for wireless mediums. Only a pair of IR LED and IR receiver modules are needed.

To transfer data over an IR medium we need a carrier wave and have it mixed with data to generate a modulated wave that is receivable by the IR receiver module. The idea is rudimentarily shown in the drawing below:




Like the RF module example, two C8051F330D boards are used in this example. One of these two is an IR transmitter while the other is an IR receiver. The transmitter is responsible for showing and transmitting its internal temperature data while the receiver is responsible for showing this data on its onboard seven-segment displays.

First, let's see the transmitter side coding. The transmitter uses an IR LED to transmit data via an IR medium. The IRLED anode and cathode pins are connected to the TXD (P0.4) and PCA CEX1 (P0.1) pins respectively. Connecting the IR LED this way automatically ensures modulation. The TXD pin is responsible for sending data while the PCA pin ensures transmission of carrier wave.

```
void Port_IO_Init(void)
{
    // P0.1  -  CEX1 (PCA),  Push-Pull,  Digital
    // P0.4  -  TX0 (UART0), Push-Pull,  Digital

    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDOUT = 0x12;
    P1MDOUT = 0xE0;
    P1SKIP = 0xE0;
    XBR0 = 0x01;
    XBR1 = 0x42;
}
```

The system clock is set to 6.125MHz by scaling the internal oscillator.

```
void Oscillator_Init(void)
{
    OSCICN = 0x81;
}
```

Let us now see the settings that have let us output 38kHz carrier wave from the PCA module. According to the device datasheet, the following formula should be used to generate frequency output:

$$F_{CEXn} = \frac{F_{PCA}}{2 \times PCA0CPHn}$$

According to the settings, the PCA clock has a frequency of:

$$F_{PCA} = \frac{System\ Clock}{PCA\ Prescaler}$$

$$F_{PCA} = \frac{6125000\ Hz}{4} = 1531250\ Hz$$

and the frequency output from the PCA channel is calculated to be:

$$F_{CEXn} = \frac{1531250\ Hz}{2 \times 20} = 38281.25\ Hz$$

This ensures the required 38KHz carrier wave.

```
void PCA_Init(void)
{
    PCA0CN = 0x40;
    PCA0MD &= ~0x40;
    PCA0MD = 0x02;
    PCA0CPM1 = 0x46;
    PCA0CPH1 = 0x14;
}
```

The PCA interrupts are not needed and are optional.

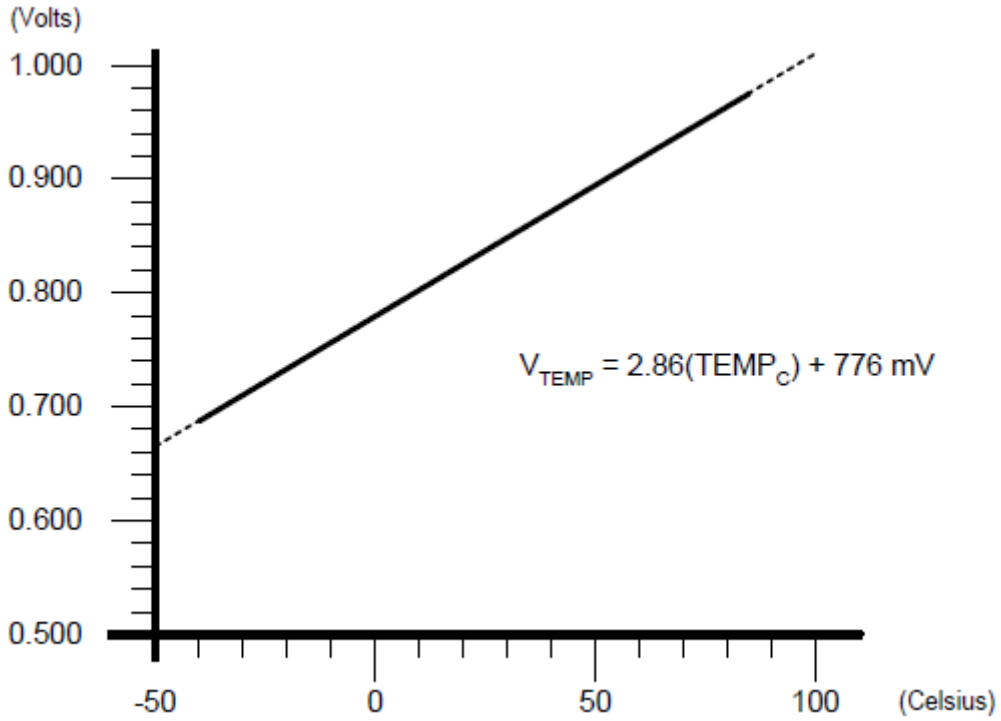
All that is left at this stage on the transmitter side is the main code. Here, we can see that the UART is set with a 1200 baud rate using MikroC's built-in UART library. The communication speed is slow but it is set so because of a trade-off between speed and reliability.

```
void main(void)
{
    unsigned int t = 0;
    char tmp = 0;

    Init_Device();
    UART1_Init(1200);
    ADC1_Init_Advanced(_INTERNAL_REF | _RIGHT_ADJUSTMENT);

    while(1)
    {
        t = (ADC1_Get_Sample(16) * VDD_mv);
        t /= 1023.0;
        t = (((float)t - 776.0) / 2.86);

        value = t;
        tmp = t;
```

```
        UART1_Write(0xAA);
        UART1_Write((tmp / 10) + 0x30);
        UART1_Write((tmp % 10) + 0x30);
        delay_ms(400);
    };
}
```

Note that, here, the internal temperature sensor is used as a source of data to be transmitted. The main loop reads this sensor, displays the temperature and sends the temperature to the C8051F330D board nearby.

To be able to use the internal temperature sensor (ADC channel 16), we need to use the internal reference. The following graph shows the relation between the output voltage from the sensor and temperature.




The equation in this graph gives voltage as a function of temperature. However, we need temperature data from voltage since we can measure voltage from ADC and certainly not temperature directly. Thus, the equation above needs rearrangement.

$$\boldsymbol{V_{TEMP} = 2.86(TEMP_C) + 776}$$

$$\boldsymbol{TEMP_C = \frac{(V_{TEMP} - 776)}{2.86}}$$

where

$$V_{TEMP} = \frac{ADC(Temperature\ Sensor\ Channel) \times VDD\ in\ mV}{Max\ ADC\ Count} = \frac{(ADC \times 3300)}{1023}$$

These have been used inside the main loop.

Lastly, temperature data is sent using UART in the following packet format:


The code for the receiver side is very easy because the IR receiver module internally completes the demodulation task and provide raw UART output.

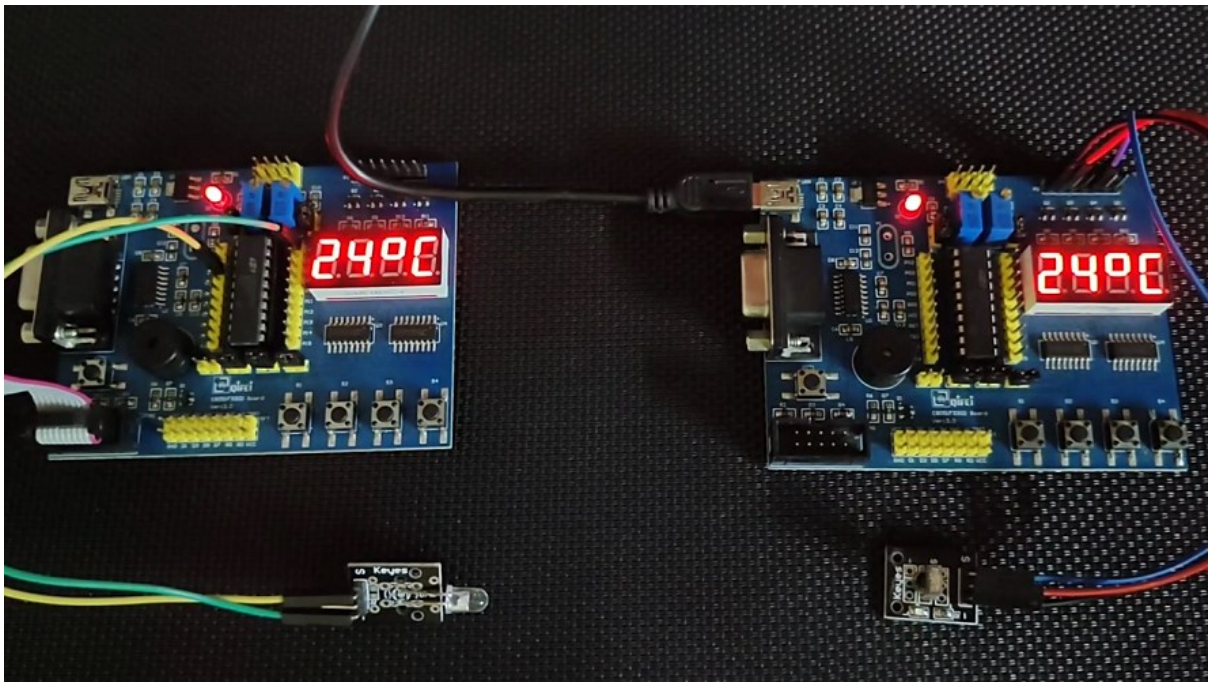
Firstly, data received by the UART is received using the interrupt method. Received serial data are stored in an array buffer called *rx_buffer*.

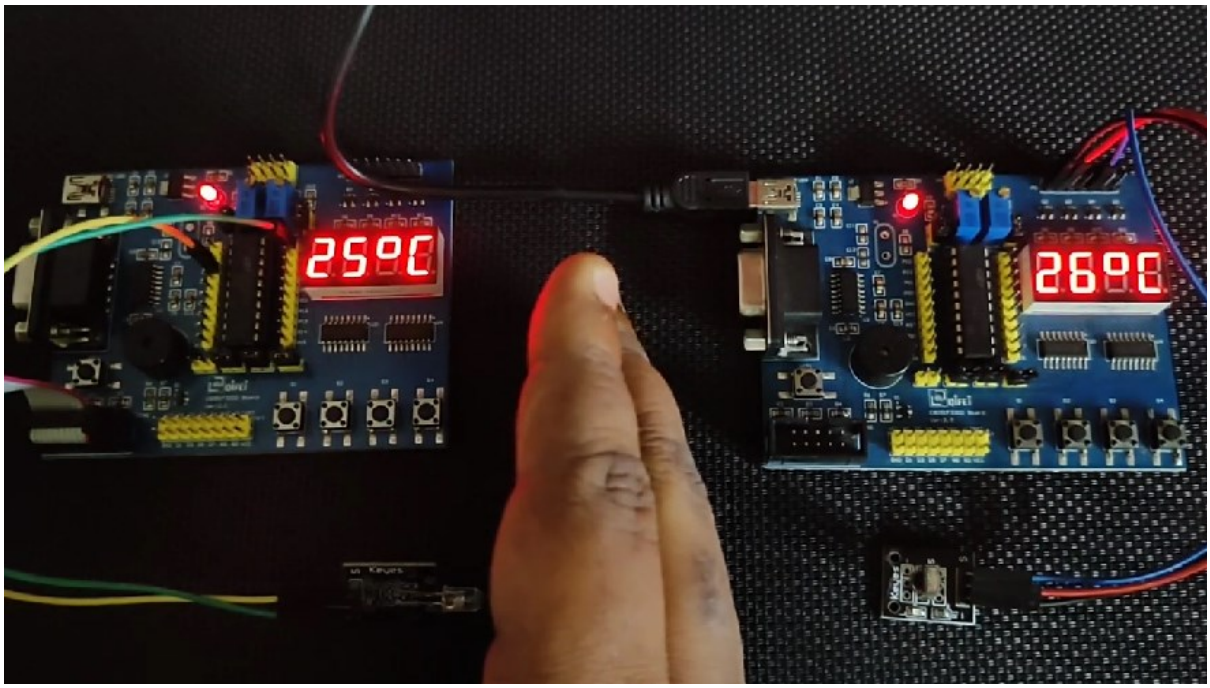
```
void UART0_ISR(void)
iv IVT_ADDR_ES0
ilevel 0
ics ICS_AUTO
{
    rx_buffer[cnt++] = UART_Read();
    RI0_bit = 0;
}
```

Since three bytes of data are to be received, the main code waits for three-byte counts before processing the received data. Firstly, the preamble header is looked for. If the first byte matches with the expected header (0xAA) then the next two bytes are assumed to contain temperature data. After extracting the bytes, the temperature data received is shown on the onboard seven-segment display.

```
void main(void)
{
    unsigned char tmp = 0x00;
    unsigned char temp = 0x00;

    Init_Device();
    UART1_Init(1200);

    while(1)
    {
        if(cnt >= 3)
        {
            if(rx_buffer[0] == 0xAA)
            {
               temp = (rx_buffer[1] - 0x30);
               tmp = (temp * 10);
               temp = (rx_buffer[2] - 0x30);
               tmp += temp;

               value = tmp;
            }
            cnt = 0;
        }
    };
}
```

## Demo




Demo video link: https://youtu.be/YV_2f-8bnIs

## Software PWM – Servo Motor Control

Sometimes we run out of PWM channels and in some cases, the PWM pins may have alternative use in a project. For a tiny microcontroller like the C8051F330D, both cases are very likely. In such scenarios, we can code software PWMs just like what we do with software-based communications like software-based I2C and software-based SPI. Any GPIO pin can be used for software PWM and that is the biggest advantage. There are a few limitations though with this concept. Firstly, some coding is needed and secondly, software PWM techniques are not fast and less accurate not to mention the limited options it offers, unlike hardware PWM.


### Code

```
#define LED_DOUT            P1_6_bit
#define LED_CLK             P1_5_bit
#define LED_LATCH           P1_7_bit

#define INC_SW              P1_4_bit
#define DEC_SW              P1_3_bit

#define duty_min            5
#define duty_max            25
#define counts              240


unsigned char i = 0;
register unsigned char val = 0;
unsigned char count = 0;
unsigned char duty_cycle_1 = duty_min;
unsigned char duty_cycle_2 = duty_max;

const unsigned char code segment_code[12] =
{
  0xC0, // 0
  0xF9, // 1
  0xA4, // 2
  0xB0, // 3
  0x99, // 4
  0x92, // 5
  0x82, // 6
  0xF8, // 7
  0x80, // 8
```

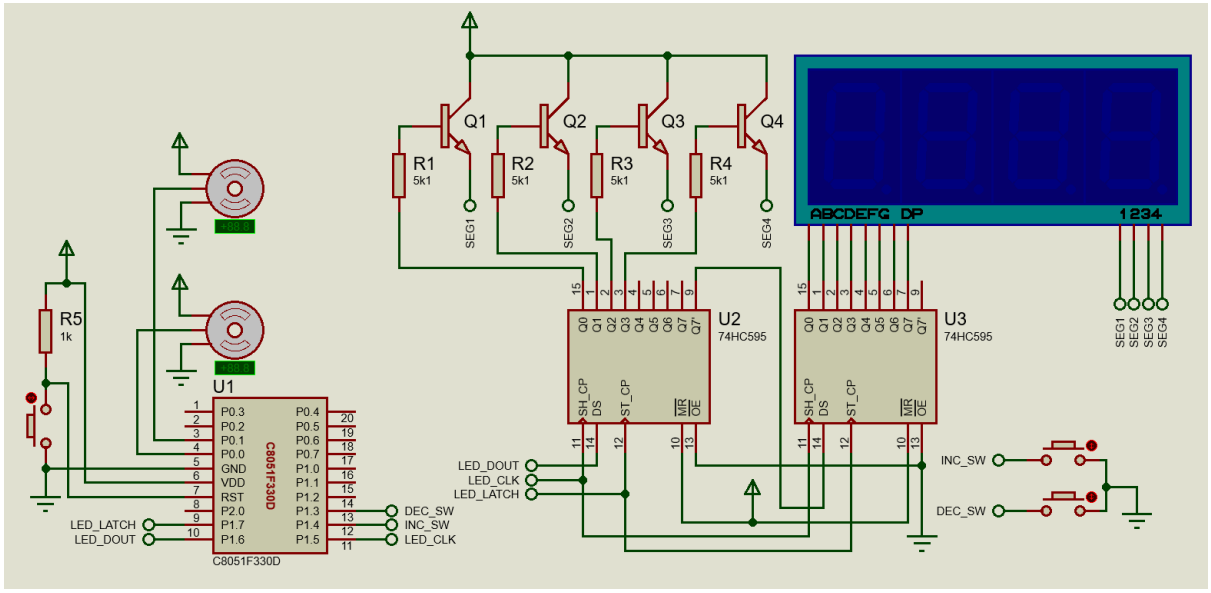
```
  0x90, // 9
  0x7F, // .
  0xBF  // -
};


const unsigned char code display_pos[4] =
{
  0xF7, //1st Display
  0xFB, //2nd Display
  0xFD, //3rd Display
  0xFE  //4th Display
};


void PCA_Init(void);
void Timer_Init(void);
void Port_IO_Init(void);
void Oscillator_Init(void);
void Interrupts_Init(void);
void Init_Device(void);
void write_74HC595(unsigned char send_data);
void segment_write(unsigned char disp, unsigned char pos);


void Timer_2_ISR(void)
iv IVT_ADDR_ET2
ilevel 0
ics ICS_AUTO
{
    count++;

    if(count <= counts)
    {
        if(count <= duty_cycle_1)
        {
            P0_0_bit = 1;
        }
        else
        {
            P0_0_bit = 0;
        }

        if(count <= duty_cycle_2)
        {
            P0_1_bit = 1;
        }
        else
        {
            P0_1_bit = 0;
        }
    }

    else
    {
        count = 0;
    }
    P0_2_bit = ~P0_2_bit;
    TF2H_bit = 0;
}


void Timer_3_ISR(void)
iv IVT_ADDR_ET3
ilevel 1
```

```
ics ICS_AUTO
{
    switch(i)
    {
        case 0:
        {
            val = (duty_cycle_1 / 1000);
            break;
        }
        case 1:
        {
            val = ((duty_cycle_1 % 1000) / 100);
            break;
        }
        case 2:
        {
            val = ((duty_cycle_1 % 100) / 10);
            break;
        }
        case 3:
        {
            val = (duty_cycle_1 % 10);
            break;
        }
    }

    segment_write(val, i);

    i++;

    if(i > 3)
    {
       i = 0;
    }

    TMR3CN &= 0x7F;
}


void main(void)
{
    Init_Device();

    while(1)
    {
        if(INC_SW == 0)
        {
            delay_ms(60);
            duty_cycle_1++;
            duty_cycle_2--;
        }

        if(DEC_SW == 0)
        {
            delay_ms(60);
            duty_cycle_1--;
            duty_cycle_2++;
        }

        if(duty_cycle_1 >= duty_max)
        {
            duty_cycle_1 = duty_max;
            duty_cycle_2 = duty_min;
        }

        if(duty_cycle_1 <= duty_min)
```

```
        {
            duty_cycle_1 = duty_min;
            duty_cycle_2 = duty_max;
        }
    };
}


void PCA_Init(void)
{
    PCA0MD &= ~0x40;
    PCA0MD = 0x00;
}

void Timer_Init(void)
{
    TMR2CN = 0x04;
    TMR2RLL = 0x99;
    TMR2RLH = 0xFF;
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}

void Port_IO_Init(void)
{
    // P0.0  -  Skipped,     Push-Pull,  Digital
    // P0.1  -  Skipped,     Push-Pull,  Digital
    // P0.2  -  Skipped,     Push-Pull,  Digital
    // P0.3  -  Unassigned,  Open-Drain, Digital
    // P0.4  -  Unassigned,  Open-Drain, Digital
    // P0.5  -  Unassigned,  Open-Drain, Digital
    // P0.6  -  Unassigned,  Open-Drain, Digital
    // P0.7  -  Unassigned,  Open-Drain, Digital

    // P1.0  -  Unassigned,  Open-Drain, Digital
    // P1.1  -  Unassigned,  Open-Drain, Digital
    // P1.2  -  Unassigned,  Open-Drain, Digital
    // P1.3  -  Skipped,     Open-Drain, Digital
    // P1.4  -  Skipped,     Open-Drain, Digital
    // P1.5  -  Skipped,     Push-Pull,  Digital
    // P1.6  -  Skipped,     Push-Pull,  Digital
    // P1.7  -  Skipped,     Push-Pull,  Digital

    P0MDOUT = 0x07;
    P1MDOUT = 0xE0;
    P0SKIP = 0x07;
    P1SKIP = 0xF8;
    XBR1 = 0x40;
}

void Oscillator_Init(void)
{
    OSCICN = 0x82;
}

void Interrupts_Init(void)
{
    IE = 0xA0;
    EIE1 = 0x80;
}


void Init_Device(void)
{
```

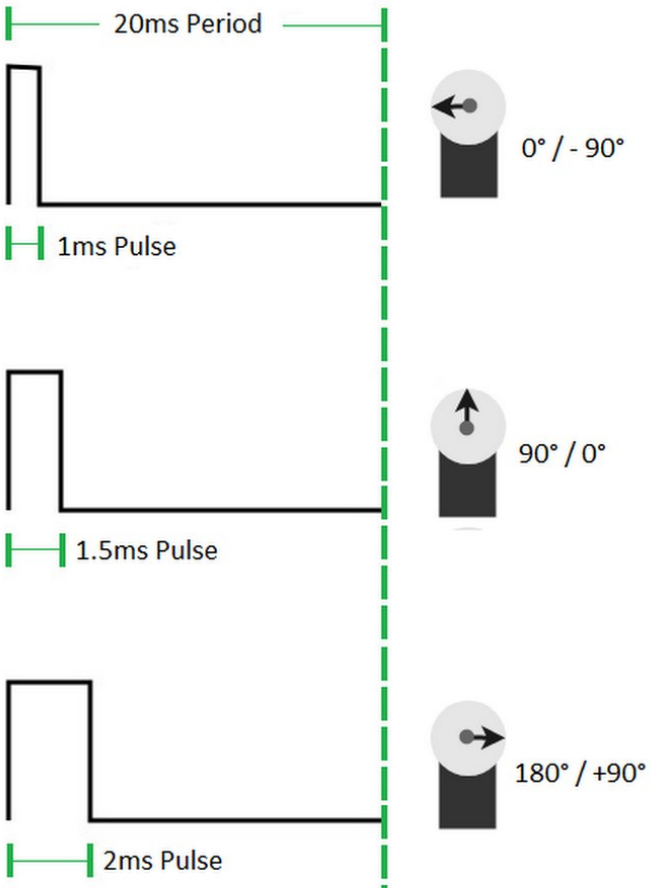
```
    PCA_Init();
    Timer_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}


void write_74HC595(unsigned char send_data)
{
    signed char clks = 8;

    while(clks > 0)
    {
        if((send_data & 0x80) == 0x00)
        {
          LED_DOUT = 0;
        }
        else
        {
          LED_DOUT = 1;
        }

        LED_CLK = 0;
        send_data <<= 1;
        clks--;
        LED_CLK = 1;
    }
}


void segment_write(unsigned char disp, unsigned char pos)
{
    LED_LATCH = 0;
    write_74HC595(segment_code[disp]);
    write_74HC595(display_pos[pos]);
    LED_LATCH = 1;
}
```

## Schematic



## Explanation

In this example, software PWM is used to drive two servo motors. Firstly, let us examine the servo motor signal timing diagram given below. From this diagram, it is evident that to drive a servo motor we need a PWM signal with a 20ms time period (50Hz) and 5 – 10% duty cycle.




To code software PWM, all we need are GPIOs and a timer. Since a timer is needed, we have to be careful about the system clock setting. Here, the system clock is set to 12.25MHz, i.e., one system clock tick is about 80ns.

```
void Oscillator_Init(void)
{
    OSCICN = 0x82;
}
```

Timer 3 is used for the onboard seven-segment displays and it is just like all other examples. Timer 2 is the PWM timer. It is set with a prescaler of 12 and a reload value of 65433. Thus, it is set to tick about every 100µs.

$$\boldsymbol{Timer\ Frequency} = \frac{\boldsymbol{System\ Clock}}{\boldsymbol{Prescalar}} = \frac{\boldsymbol{12.25\ MHz}}{\boldsymbol{12}} = \boldsymbol{1.02083\ MHz}$$

$$\boldsymbol{Timer\ Tick} = \frac{\boldsymbol{1}}{\boldsymbol{Timer\ Frequency}} = \frac{\boldsymbol{1}}{\boldsymbol{1.02083\ MHz}} = \boldsymbol{0.98\mu s}$$

$$\boldsymbol{Timer\ Interrupt\ Invterval} = (\boldsymbol{Top\ Value} - \boldsymbol{Reload\ Value}) \times \boldsymbol{Timer\ Tick}$$

$$\boldsymbol{Timer\ Interrupt\ Invterval} = (\boldsymbol{65535} - \boldsymbol{65433}) \times \boldsymbol{0.98\mu s} \approx \boldsymbol{100\mu s}$$

```
void Timer_Init(void)
{
    TMR2CN = 0x04;
    TMR2RLL = 0x99;
    TMR2RLH = 0xFF;
    TMR3CN = 0x04;
    TMR3RLL = 0x02;
    TMR3RLH = 0xFC;
}


void Interrupts_Init(void)
{
    IE = 0xA0;
    EIE1 = 0x80;
}
```

Inside the Timer 2 interrupt, a variable named *count* is incremented from 0 to 240. This variable defines the timer period. Since the variable resets after every 240 counts, the PWM period is calculated to be:

$$\boldsymbol{PWM\ Period} = \boldsymbol{Timer\ Tick} \times \boldsymbol{count} = \boldsymbol{100\mu s} \times \boldsymbol{240} = \boldsymbol{24ms}$$

As stated, a servo motor is typically run using pulses having a 20ms time period and here we achieved just that, although not exactly 20ms. This difference is kept intentionally to balance off any tolerance in timing.

```
#define duty_min          5
#define duty_max          25
#define counts            240

....

void Timer_2_ISR(void)
iv IVT_ADDR_ET2
ilevel 0
ics ICS_AUTO
{
    count++;
```

```
    if(count <= counts)
    {
        if(count <= duty_cycle_1)
        {
            P0_0_bit = 1;
        }
        else
        {
            P0_0_bit = 0;
        }

        if(count <= duty_cycle_2)
        {
            P0_1_bit = 1;
        }
        else
        {
            P0_1_bit = 0;
        }
    }

    else
    {
        count = 0;
    }
    P0_2_bit = ~P0_2_bit;
    TF2H_bit = 0;
}
```

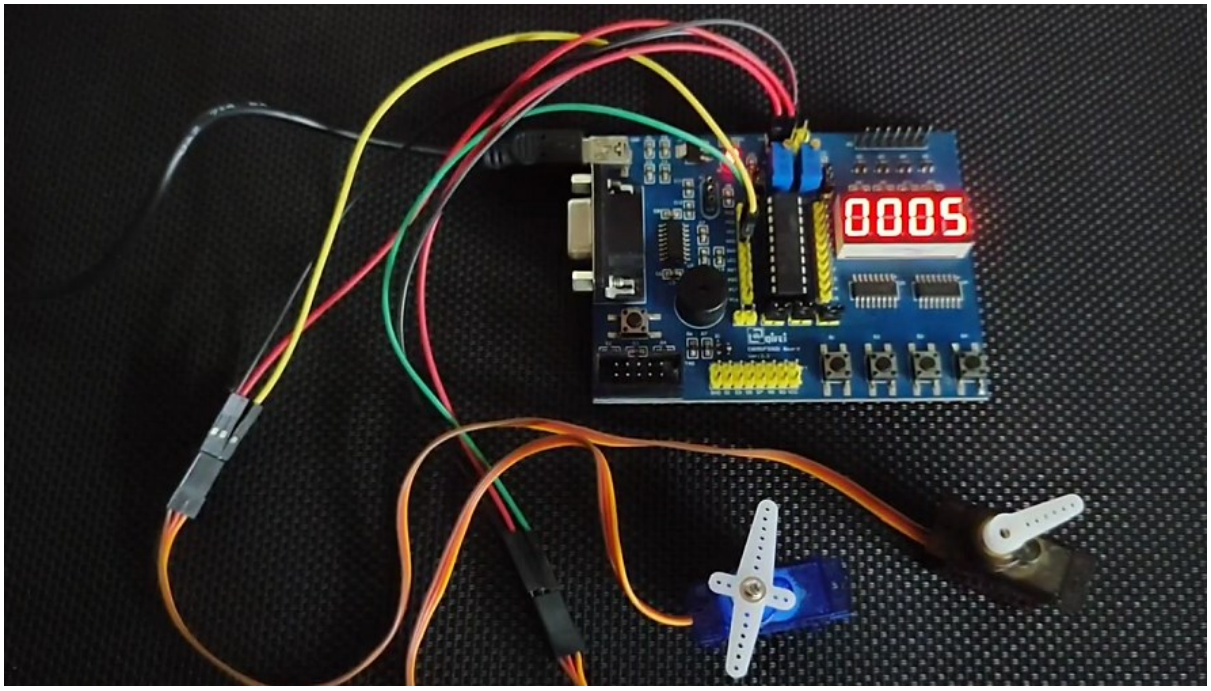
Variables named *duty_cycle_1* and *duty_cycle_2* are used to control the duty cycles of two PWM channels. From 0 count to the value stored in these variables, respective GPIOs are set high and when the count variable exceeds these values, respective GPIOs are set low.

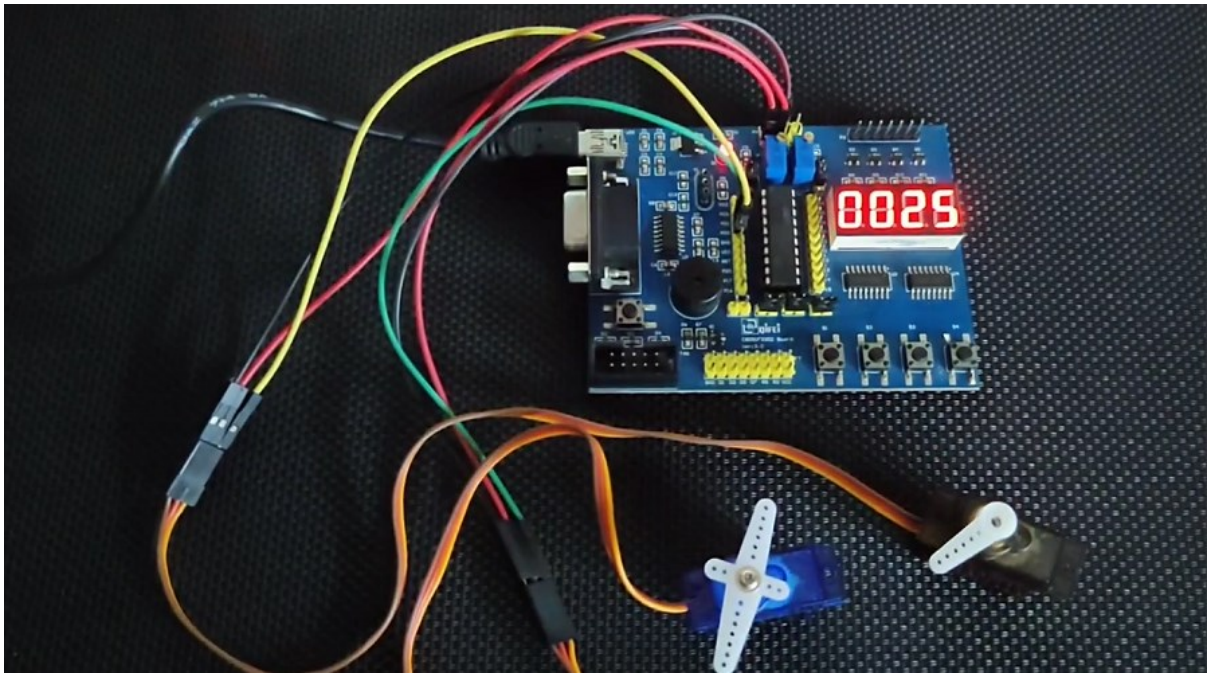
In the main, two onboard push buttons connected with pins P1.3 and P1.4 are pressed to change the duty cycle and the corresponding servo rotations are altered.

```
void main(void)
{
    Init_Device();

    while(1)
    {
        if(INC_SW == 0)
        {
            delay_ms(60);
            duty_cycle_1++;
            duty_cycle_2--;
        }
        if(DEC_SW == 0)
        {
            delay_ms(60);
            duty_cycle_1--;
            duty_cycle_2++;
        }
        if(duty_cycle_1 >= duty_max)
        {
            duty_cycle_1 = duty_max;
            duty_cycle_2 = duty_min;
        }
        if(duty_cycle_1 <= duty_min)
        {
            duty_cycle_1 = duty_min;
            duty_cycle_2 = duty_max;
```

```
        }
    };
}
```

## Demo




Demo video link: https://youtu.be/ynxU_R_mW4A

# Epilogue

In conclusion, SiLabs C8051 microcontrollers have proven to be a versatile and powerful choice for embedded systems development although at present times ARM microcontrollers with advanced features like Wi-Fi, BLE and other features are dominant players. The rich feature set of C8051s, combined with robust performance and low power consumption, make them a preferred option for a wide range of small and medium-scale applications.

We explored some key aspects of SiLabs C8051 microcontrollers, including their internals, features, and other things. I am impressed by the analogue and digital capabilities that enabled seamless interfacing with various sensors and devices. Apart from the hardware side, the software tools are also tweaked for rapid development.

As technology continues to advance, SiLabs remains committed to enhancing its microcontroller offerings, providing continuous support and innovation to developers worldwide. With a vast ecosystem and a dedicated community, the possibilities for leveraging the power of SiLabs C8051 microcontrollers are limitless.

In conclusion, SiLabs C8051 microcontrollers are a dependable and flexible solution for embedded systems development. Their outstanding features, comprehensive development tools, and energy-efficient design make them a compelling choice for any project.

Although it is irrelevant here, this document has been fine-tuned with the help of ChatGPT and Grammarly. These two platforms have saved a lot of time while checking and editing this document. I, humbly, give credit to developers who are also programmers like me and worked on such platforms.

Code Examples and Libraries used in this tutorial can be downloaded from here.

All demo videos of this tutorial can be found in this Youtube playlist.

Happy coding.

*Author: Shawon M. Shahryiar*
*https://www.youtube.com/c/sshahryiar*
*https://github.com/sshahryiar*
*https://www.facebook.com/groups/microarena*
*https://www.facebook.com/microarena*

*26.05.2023*