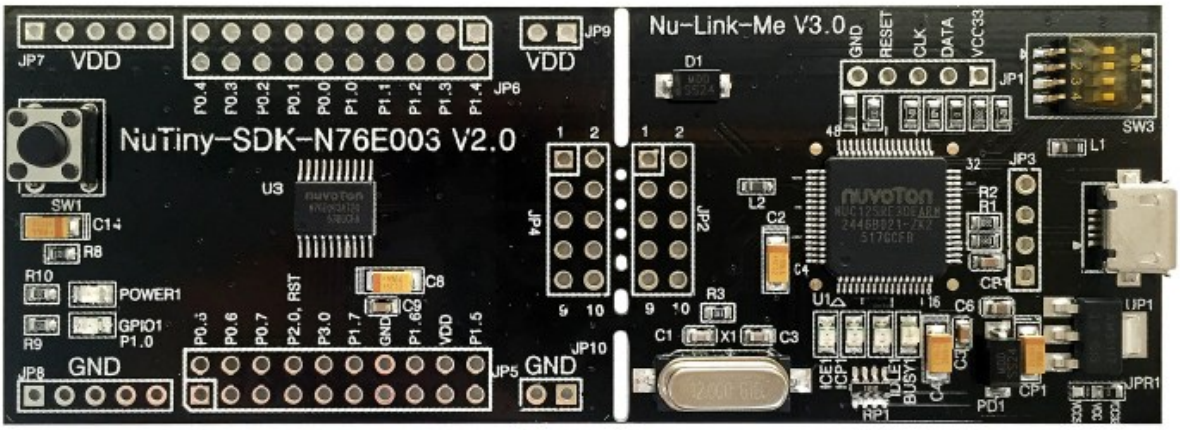
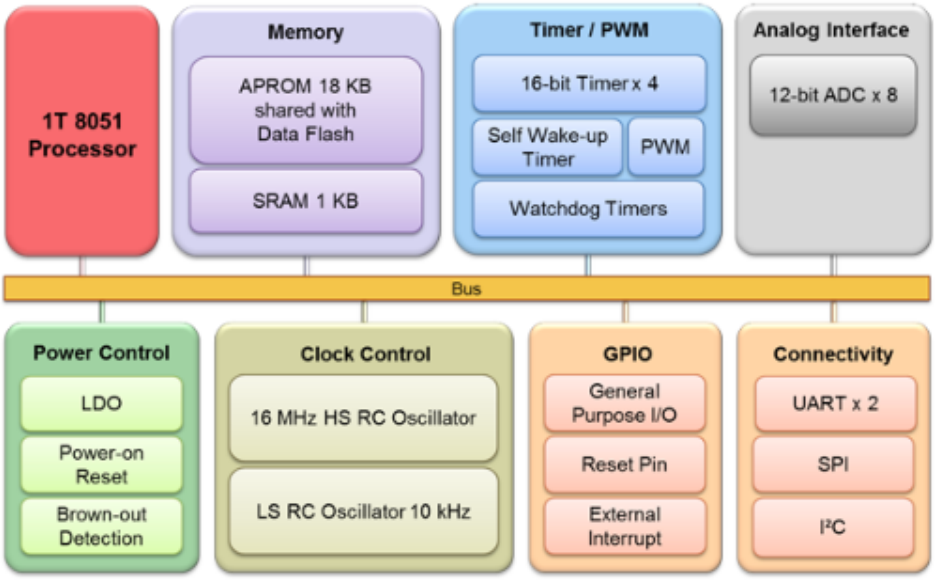
Getting Started with Nuvoton 8-bit Microcontrollers – N76E003

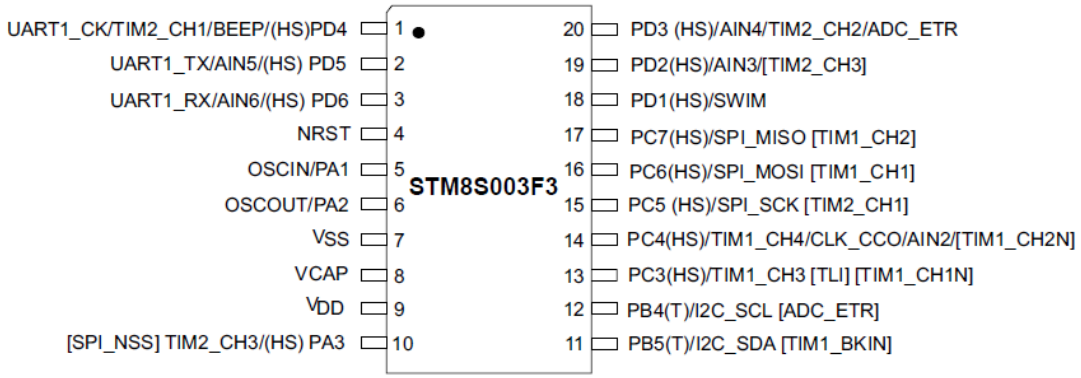
Many of us who are involved in the embedded system industry like 8-bit microcontrollers. They are cheap, easy to use and solve most of the common automation problems. 32-bit micros are in that contrast expensive and are mainly intended for advance-level works that cannot be solved with 8-bit micros. Thus, 32-bit micros are not much demanding as the 8-bit ones. In most places, 8051s, AVRs and PICs are the most used 8-bit micros. The popular Arduino platform is mainly based on 8-bit AVR micros. However, these are not the only 8-bit micros of the whole embedded world. ***Nuvoton*** - a Taiwan-based semiconductor manufacturer, is one such company that has its own flavour of 8-bit and 32-bit micros. The 8-bit micros from Nuvoton are based on the popular 8051 architectures. In this series of articles, we will be discovering Nuvoton ***N76E003*** 1T-8051-based microcontroller.

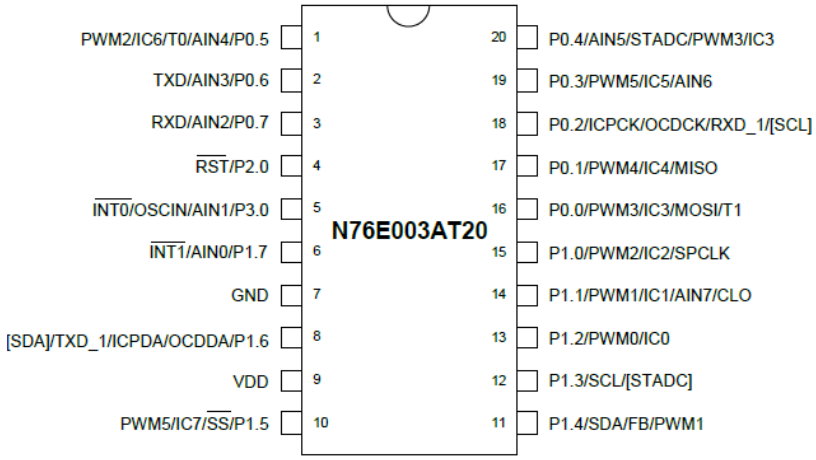


N76E003 vs STM8S003



In terms of outlook, part numbering, pin layout and other basic features, N76E003 looks like a cheap Chinese copy of STMicroelectronics' STM8S003. However, as we all know, looks can be deceptive. Though several similarities, N76E003 is not a replica of STM8S003 in any way. In fact, in terms of architecture and some hardware features, N76E003 varies a lot from STM8S003. For instance, the processor core of these chips is not same. Even after having same pin layouts, N76E003 has several additional hardware than STM8S003, for instance 8 channel - 12-bit ADC.





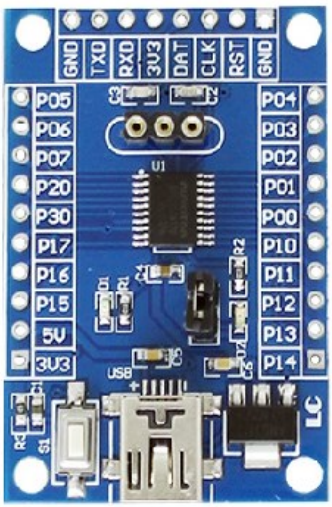
The table below summarizes an actual comparison of these chips.

Features	*N76E003*	*STM8S003*
Operating Voltage	2.40 - 5.50V	2.95 - 5.50V
Flash	18kB	8kB
SRAM	1kB	1kB
EEPROM	Shared with Flash up to 4kB	128B
GPIO	17 + 1 Input Only	16
ADC	8 Channel 12-bit, 400kHz	5 Channel 10-bit, 428kHz
Timers	3 x 16-bit + 1 x 16-bit (for PWM)	1 x 8-bit, 2 x 16-bit
PWM	6 Channel 16-bit	3 x Complementary + 4 Independent
SPI	1 Channel 8Mbit/s	1 Channel 8Mbit/s
I2C	1 Channel 400kHz	1 Channel 400kHz
UART	2 Channel	1 Channel
HIRC	16MHz (2% -40 - 105°C)	16MHz (1.5% -25 - 85°C)
LIRC	10kHz	128kHz
Idle Current Consumption	< 5µA	6µA
Programming Interface	2 Wire	1 Wire
Package	TSSOP20/QFN20	TSSOP20/UFQFPN20/LQFP32

It is a good idea to consider N76E003 as an advanced 8051 micro in STM8S003 form-factor and having several similarities with the STM8S003.

Hardware Tools

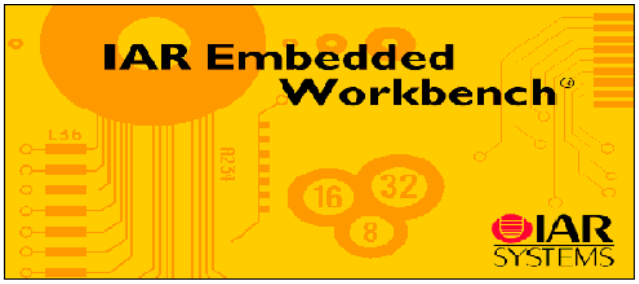
To get started only two things are needed – a prototyping/development board and a programmer called NuLink. There are two options either you can buy the expensive official NuTiny SDK boards with built-in NULink programmer/debugger or you can buy the cheap unofficial ones as shown below.



My personal choice is the unofficial one.

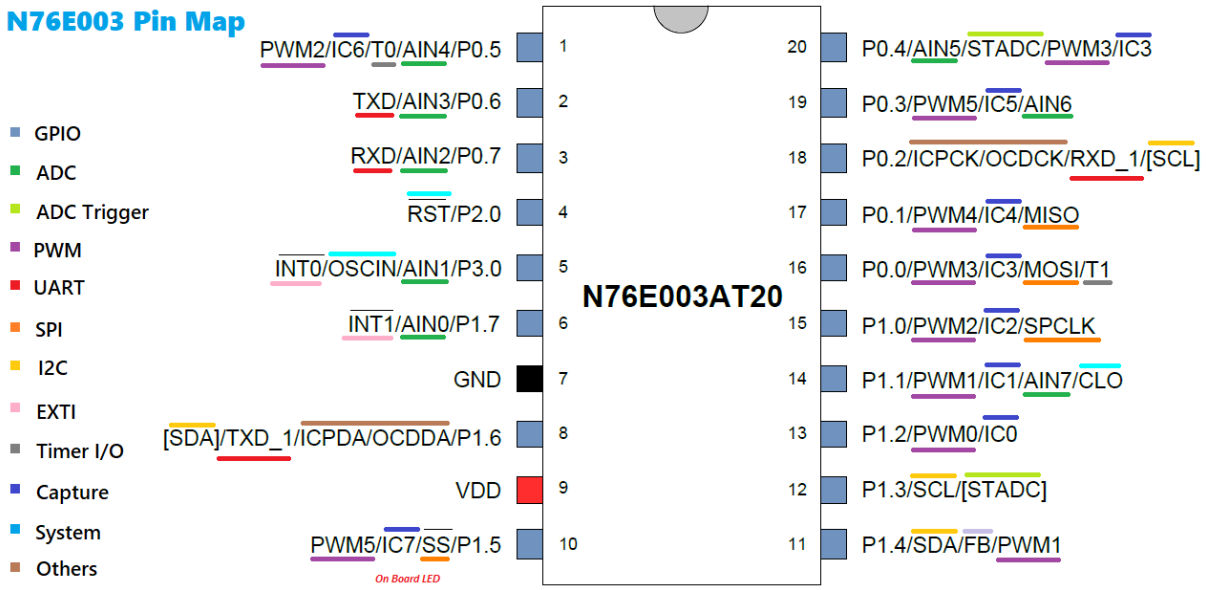
Software Tools

Despite Nuvoton being a giant in manufacturing some of the industry's coolest chips, it greatly lags in terms of software tools. Unlike other manufacturers like STMicroelectronics and Microchip, Nuvoton doesn't have a free C compiler of its own. To use Nuvoton 8-bit micros, we have to use either ***Keil Micro Vision*** or ***IAR Embedded Workbench***. Both industry-standard tools but are not free. However, there are trial and evaluation versions of these tools.



The next stuffs that we will be needing are device datasheet, drivers, GUI for NuLink and sample codes with Nuvoton's official header and source files. These are available here.

We will also be needing a pin map because there are few pins in N76E003 and they have multiple functions.



How to get started?

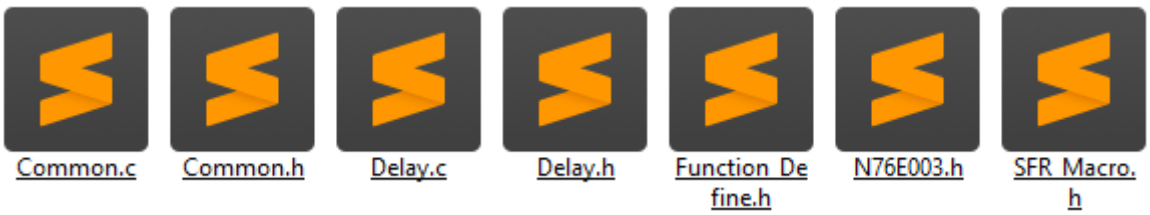
I have made two videos – one for Keil and the other for IAR. These videos show in details how to start building projects with N76E003 using these compilers.

https://www.youtube.com/watch?v=hJ-SyFDZ8go

https://www.youtube.com/watch?v=xZ5gkk6WQpw

Nuvoton Files

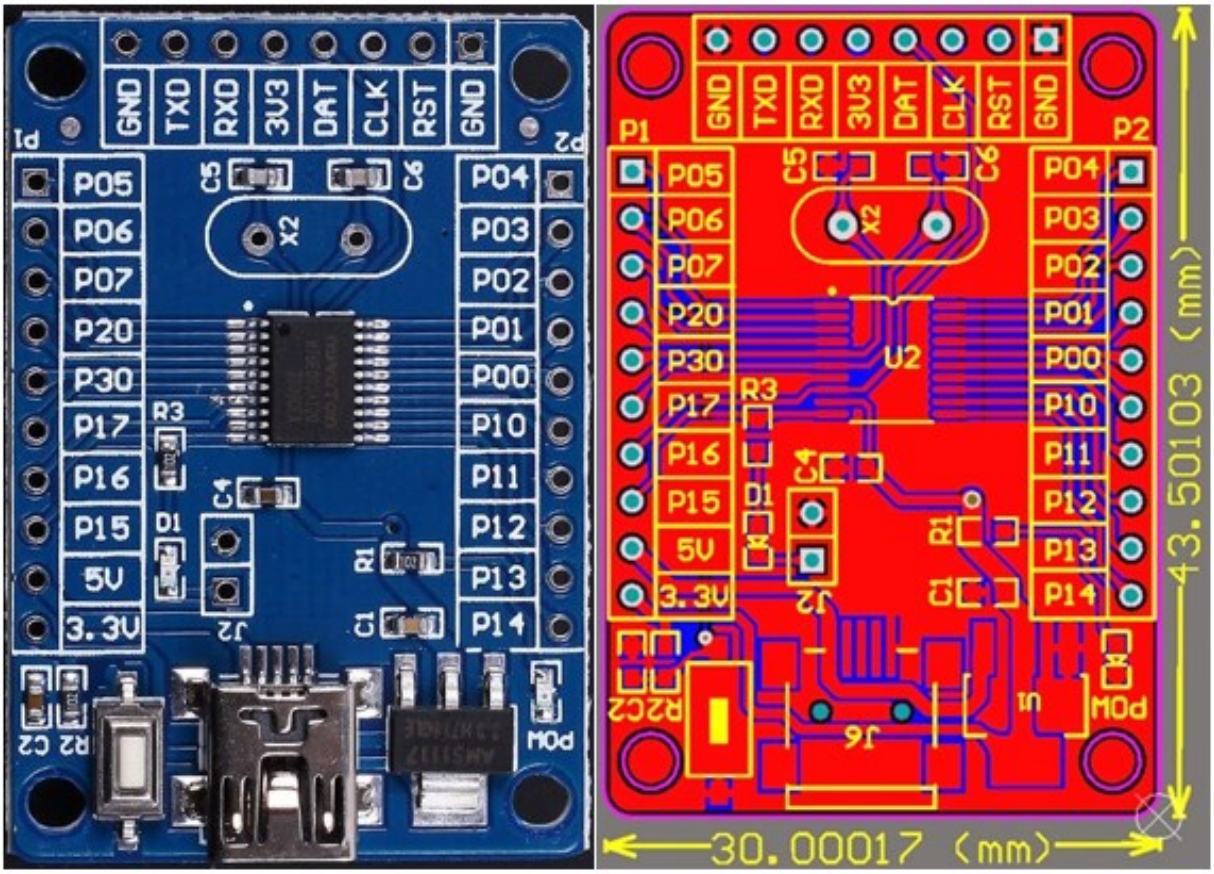
No matter which compiler you use ultimately, you'll need the following header (*.h*) and source files (*.c*) to get things done properly.



Now what these files do? These files make up something like a standard peripheral library. They define registers and functions that are needed to simplify coding. For instance, the ***Delay*** files dictate the software delay functions by using hardware timers. Likewise, ***SFR_Macro*** and ***Function_Define*** files define hardware-based functions and SFR uses. I highly recommend going through the files.

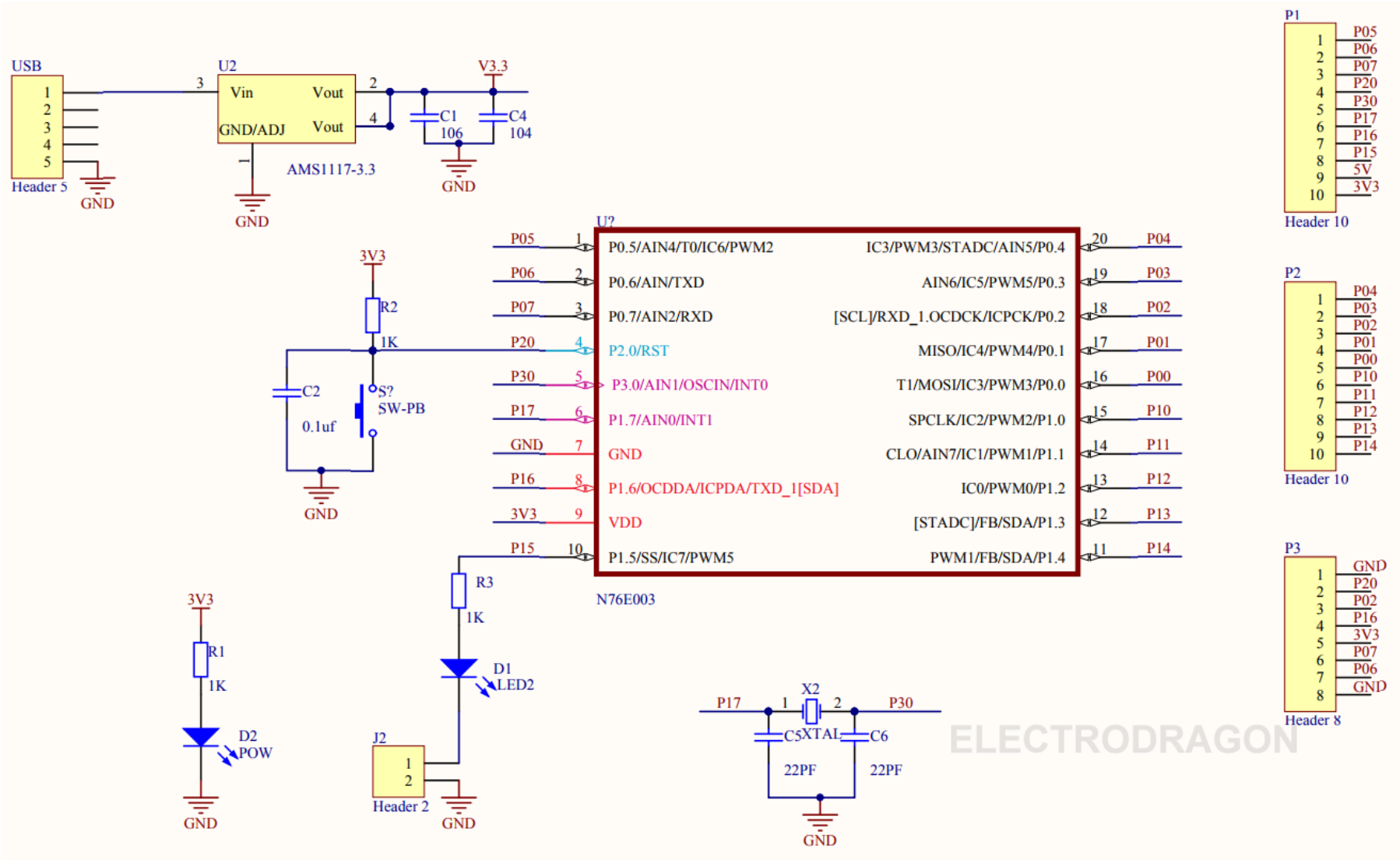
About the N76E006 Test Board

Shown below are the actual photo, the PCB layout and the schematic of the unofficial cheap N76E003 test/development board. This is the board I will be using in this tutorial series.

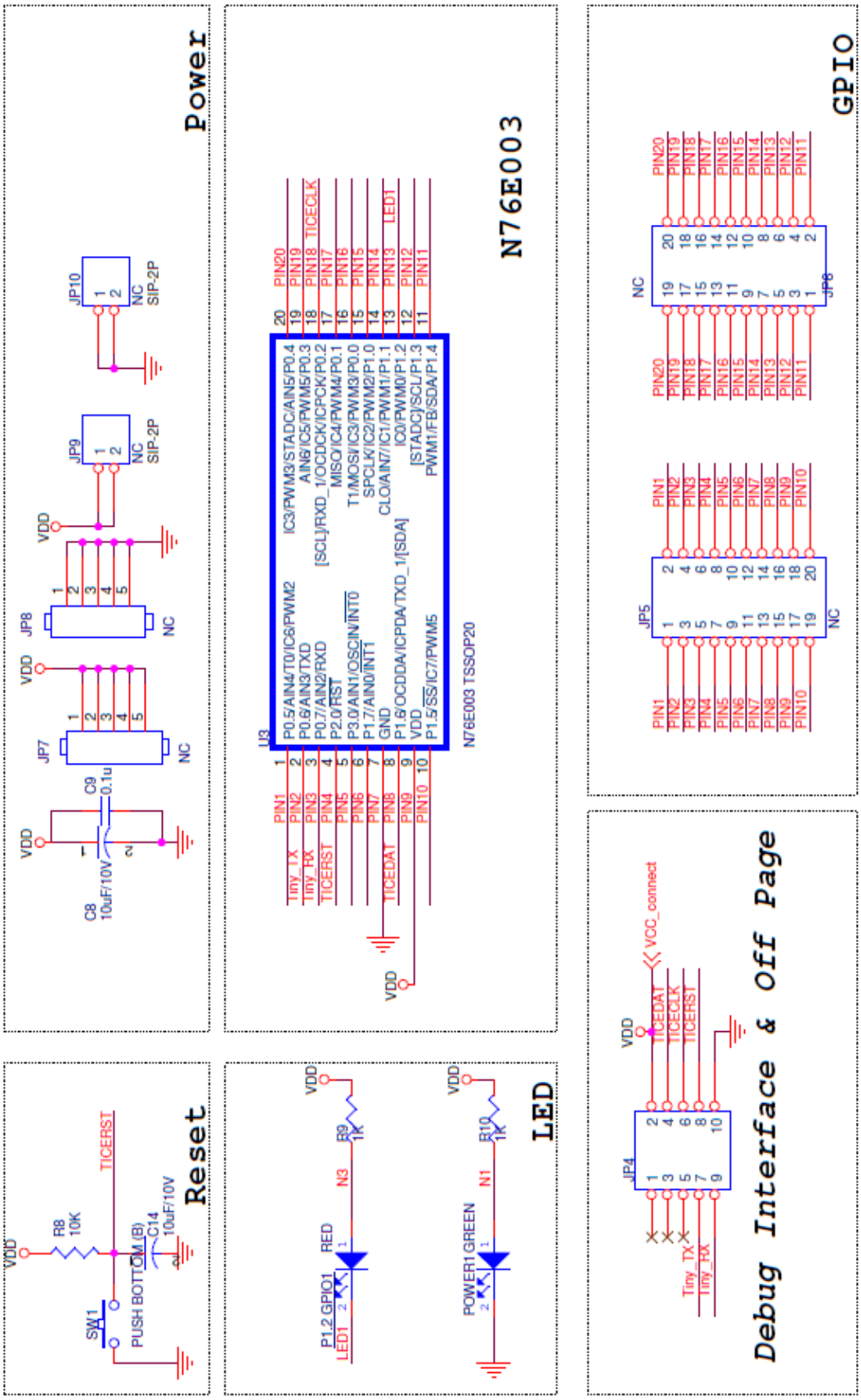


There is nothing much about the 30mm x 43.5mm board. Everything is visible and neat. However, having the schematic alongside the board layout in possession is a great advantage. There are two sidewise header that bring out the GPIO pins and positive supply rails. There is another header opposite to the USB port. This header is for connecting a Nulink programmer-debugger interface and it also has a serial port interface brought out straight from the N76E003 chip. This serial port is useful for quickly debugging/testing stuffs with a serial port monitor. There is an LED connected with P15 pin via a computer jumper. The only thing that is wrong in this board is the crystal resonator part. N76E003 has an external clock input pin but it is meant to be used with active oscillators/crystal modules. In the official, SDK there is no such points to connect an external crystal resonator. The internal high frequency oscillator is accurate enough for most cases.

At this point, I would like to thank Electro Dragon for these images because they are the only ones who shared these resources online.


USB
Header 5
GND
U2
Vin
Vout
GND/ADJ
Vout
AMS1117-3.3
V3.3
C1
106
C4
104
GND
GND
U?
P0.5/AIN4/T0/IC6/PWM2
P0.6/AIN/TXD
P0.7/AIN2/RXD
P2.0/RST
P3.0/AIN1/OSCIN/INT0
P1.7/AIN0/INT1
GND
P1.6/OCDDA/ICPDA/TXD_1[SDA]
VDD
P1.5/SS/IC7/PWM5
IC3/PWM3/STADC/AIN5/P0.4
AIN6/IC5/PWM5/P0.3
[SCL]/RXD_1.OCDCK/ICPCK/P0.2
MISO/IC4/PWM4/P0.1
T1/MOSI/IC3/PWM3/P0.0
SPCLK/IC2/PWM2/P1.0
CLO/AIN7/IC1/PWM1/P1.1
IC0/PWM0/P1.2
[STADC]/FB/SDA/P1.3
PWM1/FB/SDA/P1.4
N76E003
P05
P06
P07
P20
P30
P17
GND
P16
3V3
P15
P04
P03
P02
P01
P00
P10
P11
P12
P13
P14
3V3
R2
1K
C2
0.1uf
S?
SW-PB
GND
3V3
R1
1K
D2
POW
GND
R3
1K
D1
LED2
J2
Header 2
GND
P17
X2
P30
C5
XTAL
C6
22PF
22PF
GND
P1
Header 10
P2
Header 10
P3
Header 8
ELECTRODRAGON

Shown below is the official SDK board's schematic:



Coding Nuvoton N76E003

When entering a new environment, things are not very easy at first. It takes times to get acquainted with new the new environment, new tools and new stuffs. New means everything different from the ones that you have been using prior to its introduction. In many occasions, I had trouble playing with new microcontrollers due to this. However, after some decent play offs things unfolded themselves.

President JFK's speech on lunar landing should be an inspirational note for everyone who is trying to do something new or something that he/she has never done before:

"We choose to go to the Moon in this decade and do the other things, not because they are easy, but because they are hard; because that goal will serve to organize and measure the best of our energies and skills, because that challenge is one that we are willing to accept, one we are unwilling to postpone, and one we intend to win, and the others, too."

Both Keil and IAR are excellent tools for coding Nuvoton MCUs. I have used both and everything is same in both cases. Literally there is no difference at all. I won't recommend which one to use and I leave the choice to the readers. However, there are some areas where you may find difficulty porting codes of one compiler to the other. The table below summarizes some of these differences.

Component	***Keil***	***IAR***

Interrupt Vector	Interrupt vector number, e.g.
	`void EXT_INT0(void)`
	`interrupt 0`
	`{`
	`}`
		Interrupt vector address, e.g.
		`#pragma vector = 0x00`
		`__interrupt void EXTI0(void)`
		`{`
		`}`

String Array for LCD Libraries	`static char txt[] = {"MICROARENA"};`	`const char txt[] = {"MICROARENA"};`

Adding Assembly Code	`#pragma asm`
	`  nop;`
	`#pragma endasm`
	or simply:
	`nop;`
		`asm("nop");`

N76E003 Header File	`#include "N76E003_iar.h"`	`#include "N76E003.h"`
Memory Specifier	`unsigned char code *a;` `unsigned char xdata *b;`	`unsigned char __code *a;` `unsigned char __xdata *b;`

Apart from these facts, you'll notice that the N76E003 BSPs don't contain any software-based delay libraries unlike other compilers. Nuvoton BSPs are rather equipped with accurate timer-based delay libraries but personally I like software delays since they offer more flexibility, cross-environment compatibility, less coding and don't use anything except CPU cycles. Don't worry, I have developed a reasonably accurate software delay library.

Two more things I would like to highlight here. Firstly, both Keil and IAR compiler can throw some errors during code compilations. Most of these errors are due to BSP definitions. One such error is in the line below:

```
#define set_P0S_6
BIT_TMP=EA;TA=0xAA;TA=0x55;SFRS=0x01;P0S|=SET_BIT6;TA=0xAA;TA=0x55;SFRS=0x00;EA=BIT_TMP
```

If you try to use ***set_P0S_6*** definition, IAR sometimes throws an error because it can't find ***BIT_TMP***. However, there are other similar definitions that don't throw such error and in Keil you won't notice something like this. Such things are nasty illogical surprizes. We have to understand that the BSPs are still in development. Always remember that the datasheet is your friend. I suggest that when you try out the examples I have shown here, you read the relevant parts of the datasheet to enhance learning and understanding.

The other thing to note is the fact that not always we have the luxury to avoid register-level coding and so when needed we must have the right knowledge to use them. We can also use bit-level manipulations as shown below:

```
//For setting a bit of a register
#define bit_set(reg, bit_val)                                   reg |= (1 << bit_val)

//For clearing a bit of a register
#define bit_clr(reg, bit_val)                                   reg &= (~(1 << bit_val))

//For toggling a bit of a register
#define bit_tgl(reg, bit_val)                                   reg ^= (1 << bit_val)

//For extracting the bit state of a register
#define get_bit(reg, bit_val)                                   (reg & (1 << bit_val))

//For extracting masked bits of a register
#define get_reg(reg, msk)                                       (reg & msk)
```

Sometimes but not always, we have to code things the old ways. Sometimes mixing assembly code with C code becomes a necessity. For instance, the software delay library uses this concept.

There are other aspects to consider too like case sensitivity and coding conventions. It is wise to choose interrupt-driven methods over polling-based ones. Codes should be included in hierarchical orders. Like such there are tons of stuffs to make your code smart and error-free. The best source of knowledge of such things and much more are app notes of various manufacturers.

Whenever making a new library, add the followings in your library's source code along with other header files of your choice to avoid errors and nasty surprizes:

```
#include "N76E003.h" //for Keil compiler or "N76E003_IAR.h" for IAR compiler
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
```

Additionally, I have made a set of files called "***Extended_Functions***". Here I added all the functions that we will need almost every time when we deal with common internal hardware like timers, ADC,

etc. These files are like repositories of all the additional functions that I made for some internal hardware – something that Nuvoton didn't provide and something that makes coding lot easier.

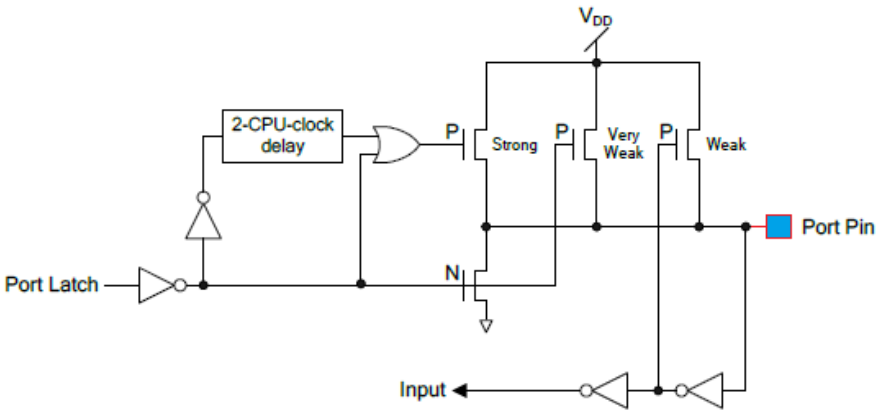
Here I'm sharing two videos to demonstrate how to code and add custom-made library files in both Keil C51 and IAR Embedded-Workbench.

https://youtu.be/m1roxhCM6T8

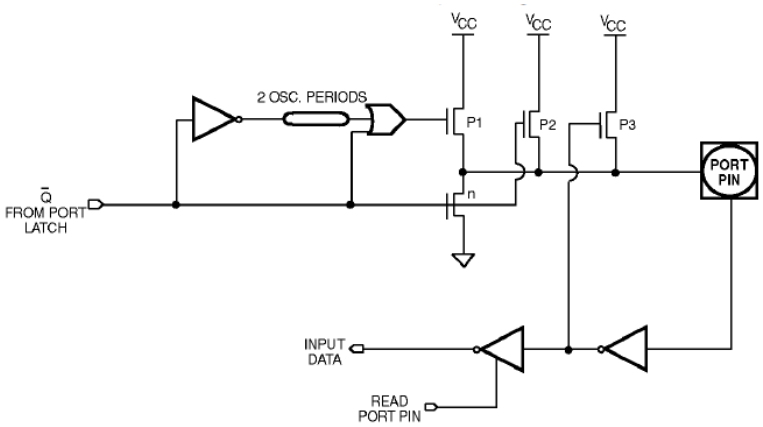
https://youtu.be/p_dvXF4zmqI

General Purpose Input-Output (GPIO)

GPIOs are the most common hardware that we use in a microcontroller. Since N76E003 is based on 8051 architecture, we should be getting some similarities with the old school 8051s. Shown below is the hardware schematic of N76E003's GPIO block:



On close inspection, we can realize that this structure has striking resemblance with the GPIO structure of a typical 8051 microcontroller as shown below:



Thus, we can expect similar behaviour.

There are four GPIO modes and these are as follows:

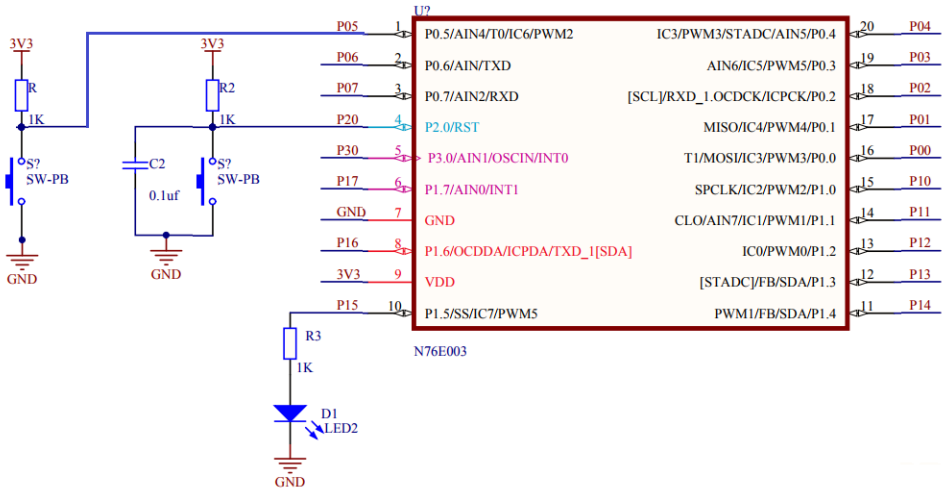
PxM1.n	*PxM2.n*	*I/O Type*	*Description*
0	0	Quasi-Bidirectional	An I/O in this mode is both an input and output. It behaves just like the ordinary I/O of a typical 8051 micro.
0	1	Push – Pull I/O	This mode is same as the first one but with stronger current sourcing capability and is recommended when making outputs.
1	0	Input Only Mode	This mode is intended for low power high impedance inputs unlike other modes.
1	1	Open-drain I/O	As the name suggests, it is same as Quasi bidirectional mode but with open-drain output that can sink current only.

PxM1.n and PxM2.n bits decide these modes. For most cases, we can stick to push-pull and input modes as they are the most commonly used ones.

Code

```
#include "N76E003_iar.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"


void setup(void);


void main(void)
{
  setup();

  while(1)
  {
    if(P05 != 0x00)
    {
      Timer0_Delay1ms(900);
    }

    set_P15;
    Timer0_Delay1ms(100);
    clr_P15;
    Timer0_Delay1ms(100);
  };
}


void setup(void)
{
  P15_PushPull_Mode;
  P05_Input_Mode;
}
```

Schematic



Explanation

The ***Function_define*** BSP header file states GPIO modes as follows:

```
//------------------- Define Port as Quasi mode ------------------------
#define P00_Quasi_Mode              P0M1&=~SET_BIT0;P0M2&=~SET_BIT0
....
//------------------- Define Port as Push Pull mode --------------------
#define P00_PushPull_Mode           P0M1&=~SET_BIT0;P0M2|=SET_BIT0
....
//------------------- Define Port as Input Only mode -------------------
#define P00_Input_Mode              P0M1|=SET_BIT0;P0M2&=~SET_BIT0
....
//-------------------Define Port as Open Drain mode --------------------
#define P00_OpenDrain_Mode          P0M1|=SET_BIT0;P0M2|=SET_BIT0
....
//--------- Define all port as quasi mode ---------
#define Set_All_GPIO_Quasi_Mode     P0M1=0;P0M1=0;P1M1=0;P1M2=0;P3M1=0;P3M2=0;
```

Like any definitions, we can straight call these in our coding and avoid clumsy register coding.

Similarly, ***SFR_Macro*** BSP header file defines the bit-level setting of all N76E003 registers. To set the logic level of GPIO pins we can use the following definitions:

```
//------------------- Define Port Pin Logic High State  ------------------
#define set_P00 P00 = 1
....
//------------------- Define Port Pin Logic Low State  -------------------
#define clr_P00 P00 = 0
....
```

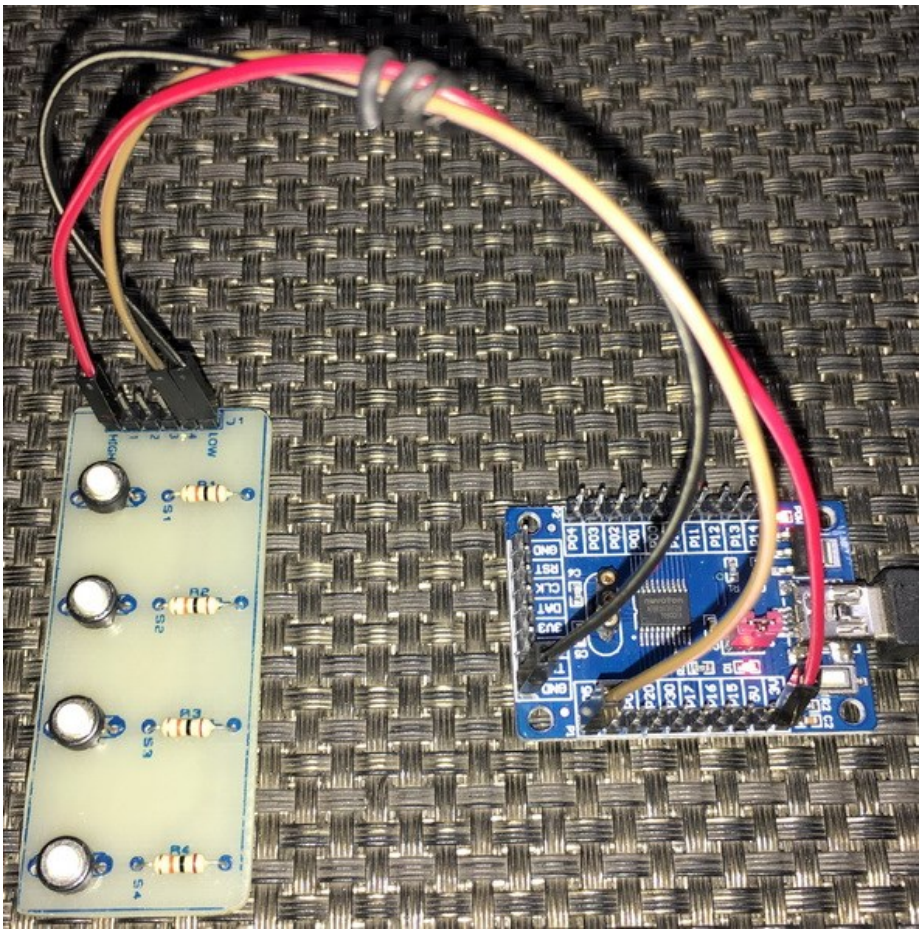
However, these don't restrict us from using classical register-level coding. ***N76E003*** header file states all the registers present in it.

For port/pin reading I didn't see any function definition as like one I already discussed. Thus, there are two ways to do it on your own. The following as two examples of such:

```
//--------- Bit-level Read ---------
if (P04 != 0)
{
}
else
{
}

//--------- Port-level Read ---------
if ((P0 & SET_BIT4) != 0)
{
}
else
{
}
```

The demo here is a simple one. The onboard LED connected to P15 pin is toggled at a fixed interval. When a button connected to P05 is pressed the off time of the LED is increased, affecting toggle rate.

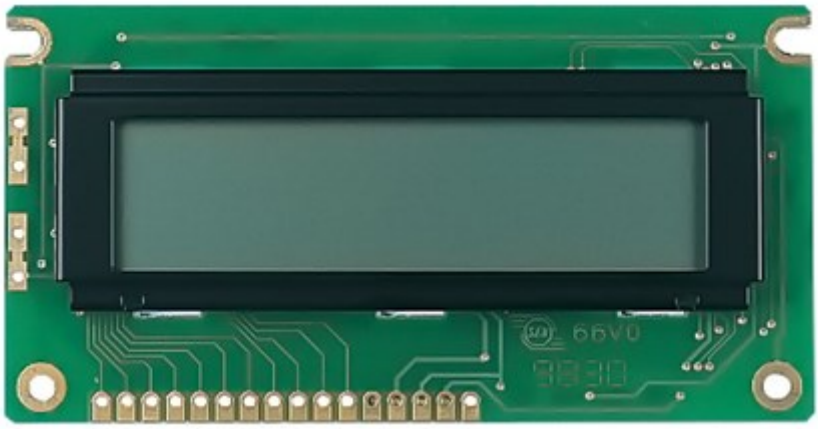
Demo



Demo video: https://youtu.be/Y215BXukqBk

Driving 2x16 LCD

Driving alphanumeric/text LCDs requires no special hardware as simple manipulation of GPIO pins and understanding of their working principle are all that are needed.



Code

lcd.h

```
#define LCD_GPIO_init()                     do{P00_PushPull_Mode;
P01_PushPull_Mode; P10_PushPull_Mode; P11_PushPull_Mode; P12_PushPull_Mode;
P13_PushPull_Mode;}while(0)

#define LCD_RS_HIGH                         set_P00
#define LCD_RS_LOW                          clr_P00

#define LCD_EN_HIGH                         set_P01
#define LCD_EN_LOW                          clr_P01

#define LCD_DB4_HIGH                        set_P10
#define LCD_DB4_LOW                         clr_P10

#define LCD_DB5_HIGH                        set_P11
#define LCD_DB5_LOW                         clr_P11

#define LCD_DB6_HIGH                        set_P12
#define LCD_DB6_LOW                         clr_P12

#define LCD_DB7_HIGH                        set_P13
#define LCD_DB7_LOW                         clr_P13

#define clear_display                       0x01
#define goto_home                           0x02

#define cursor_direction_inc                (0x04 | 0x02)
#define cursor_direction_dec                (0x04 | 0x00)
#define display_shift                       (0x04 | 0x01)
#define display_no_shift                    (0x04 | 0x00)
```

```
#define display_on                          (0x08 | 0x04)
#define display_off                         (0x08 | 0x02)
#define cursor_on                           (0x08 | 0x02)
#define cursor_off                          (0x08 | 0x00)
#define blink_on                            (0x08 | 0x01)
#define blink_off                           (0x08 | 0x00)

#define _8_pin_interface                    (0x20 | 0x10)
#define _4_pin_interface                    (0x20 | 0x00)
#define _2_row_display                      (0x20 | 0x08)
#define _1_row_display                      (0x20 | 0x00)
#define _5x10_dots                          (0x20 | 0x40)
#define _5x7_dots                           (0x20 | 0x00)

#define DAT                                 1
#define CMD                                 0


void LCD_init(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);
void toggle_EN_pin(void);
```

lcd.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "lcd.h"


void LCD_init(void)
{
    Timer0_Delay1ms(10);

    LCD_GPIO_init();


    Timer0_Delay1ms(100);

    toggle_EN_pin();

    LCD_RS_LOW;

    LCD_DB7_LOW;
    LCD_DB6_LOW;
    LCD_DB5_HIGH;
    LCD_DB4_HIGH;

    toggle_EN_pin();
```

```
    LCD_DB7_LOW;
    LCD_DB6_LOW;
    LCD_DB5_HIGH;
    LCD_DB4_HIGH;

    toggle_EN_pin();

    LCD_DB7_LOW;
    LCD_DB6_LOW;
    LCD_DB5_HIGH;
    LCD_DB4_HIGH;

    toggle_EN_pin();

    LCD_DB7_LOW;
    LCD_DB6_LOW;
    LCD_DB5_HIGH;
    LCD_DB4_LOW;

    toggle_EN_pin();

    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_send(unsigned char value, unsigned char mode)
{
    switch(mode)
    {
        case DAT:
        {
            LCD_RS_HIGH;
            break;
        }
        case CMD:
        {
            LCD_RS_LOW;
            break;
        }
    }

    LCD_4bit_send(value);
}


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

    temp = ((lcd_data & 0x80) >> 7);

    switch(temp)
```

```
    {
        case 1:
        {
            LCD_DB7_HIGH;
            break;
        }
        default:
        {
            LCD_DB7_LOW;
            break;
        }
    }

    temp = ((lcd_data & 0x40) >> 6);

    switch(temp)
    {
        case 1:
        {
            LCD_DB6_HIGH;
            break;
        }
        default:
        {
            LCD_DB6_LOW;
            break;
        }
    }

    temp = ((lcd_data & 0x20) >> 5);

    switch(temp)
    {
        case 1:
        {
            LCD_DB5_HIGH;
            break;
        }
        default:
        {
            LCD_DB5_LOW;
            break;
        }
    }

    temp = ((lcd_data & 0x10) >> 4);

    switch(temp)
    {
        case 1:
        {
            LCD_DB4_HIGH;
            break;
        }
        default:
        {
```

```
            LCD_DB4_LOW;
            break;
        }
    }

    toggle_EN_pin();

    temp = ((lcd_data & 0x08) >> 3);

    switch(temp)
    {
        case 1:
        {
            LCD_DB7_HIGH;
            break;
        }
        default:
        {
            LCD_DB7_LOW;
            break;
        }
    }

    temp = ((lcd_data & 0x04) >> 2);

    switch(temp)
    {
        case 1:
        {
            LCD_DB6_HIGH;
            break;
        }
        default:
        {
            LCD_DB6_LOW;
            break;
        }
    }

    temp = ((lcd_data & 0x02) >> 1);

    switch(temp)
    {
        case 1:
        {
            LCD_DB5_HIGH;
            break;
        }
        default:
        {
            LCD_DB5_LOW;
            break;
        }
    }

    temp = ((lcd_data & 0x01));
```

```
    switch(temp)
    {
        case 1:
        {
            LCD_DB4_HIGH;
            break;
        }
        default:
        {
            LCD_DB4_LOW;
            break;
        }
    }

    toggle_EN_pin();
}


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


void LCD_putchar(char char_data)
{
    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);
    }
}


void toggle_EN_pin(void)
{
```

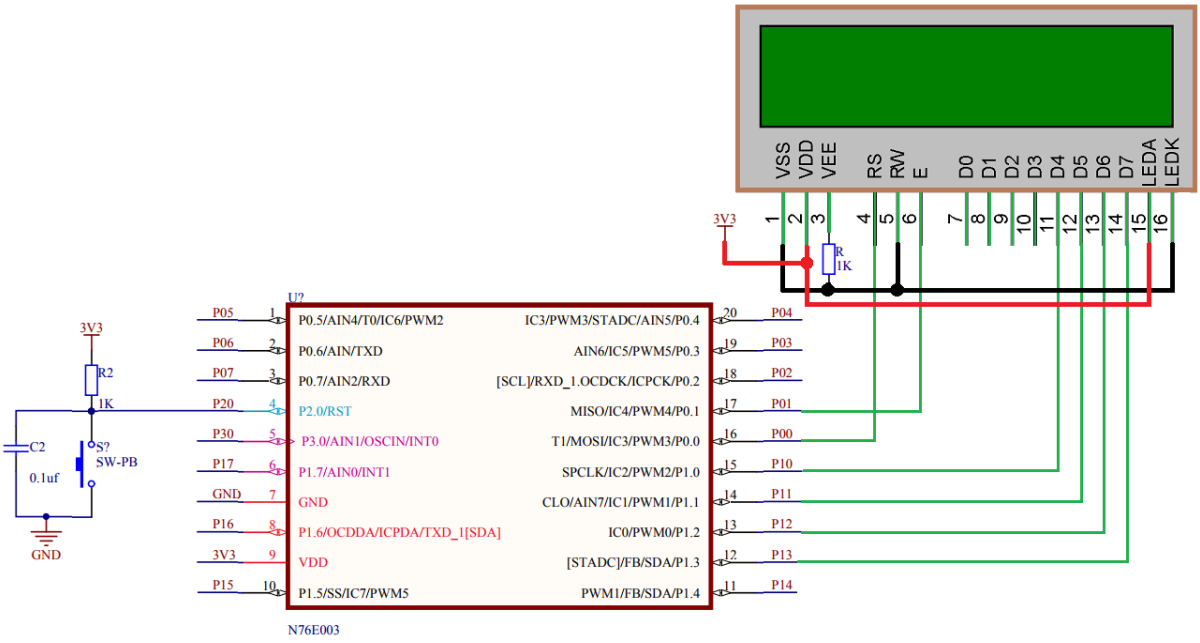
```
    LCD_EN_HIGH;
    Timer0_Delay1ms(4);
    LCD_EN_LOW;
    Timer0_Delay1ms(4);
}
```

main.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "lcd.h"


void show_value(unsigned char value);


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

    const char txt1[] = {"MICROARENA"};
    const char txt2[] = {"SShahryiar"};
    const char txt3[] = {"Nuvoton 8-bit uC"};
    const char txt4[] = {"N76E003"};

    LCD_init();

    LCD_clear_home();

    LCD_goto(3, 0);
    LCD_putstr(txt1);
    LCD_goto(3, 1);
    LCD_putstr(txt2);
    Timer3_Delay100ms(30);

    LCD_clear_home();

    for(s = 0; s < 16; s++)
    {
        LCD_goto(s, 0);
        LCD_putchar(txt3[s]);
        Timer0_Delay1ms(90);
    }

    Timer3_Delay100ms(20);

    for(s = 0; s < 7; s++)
    {
        LCD_goto((4 + s), 1);
        LCD_putchar(txt4[s]);
        Timer0_Delay1ms(90);
    }
```

```
    Timer3_Delay100ms(30);

    s = 0;
    LCD_clear_home();

    LCD_goto(3, 0);
    LCD_putstr(txt1);

    while(1)
    {
        show_value(s);
        s++;
        Timer3_Delay100ms(4);
    };
}


void show_value(unsigned char value)
{
   unsigned char ch = 0x00;

   ch = ((value / 100) + 0x30);
   LCD_goto(6, 1);
   LCD_putchar(ch);

   ch = (((value / 10) % 10) + 0x30);
   LCD_goto(7, 1);
   LCD_putchar(ch);

   ch = ((value % 10) + 0x30);
   LCD_goto(8, 1);
   LCD_putchar(ch);
}
```

Schematic

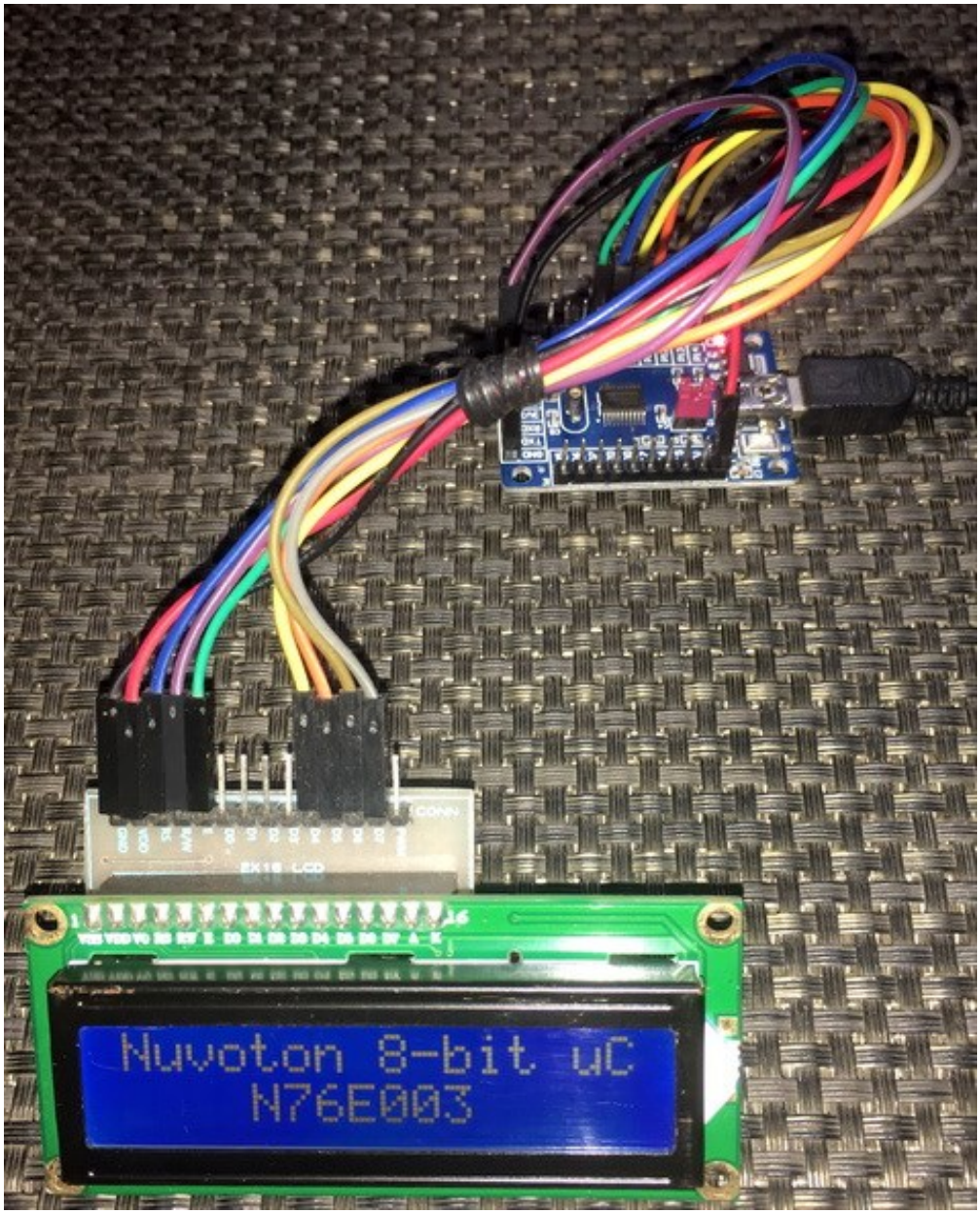


Explanation

There is nothing to explain here. The LCD driver is based on simple manipulation of GPIO pins. The codes for the LCD are coded using all available info on LCD datasheet - just initialization and working principle. If you need to change GPIO pins just edit the following lines in the LCD header file:

```
#define LCD_GPIO_init()                    do{P00_PushPull_Mode;
P01_PushPull_Mode; P10_PushPull_Mode; P11_PushPull_Mode; P12_PushPull_Mode;
P13_PushPull_Mode;}while(0)

#define LCD_RS_HIGH                        set_P00
#define LCD_RS_LOW                         clr_P00

#define LCD_EN_HIGH                        set_P01
#define LCD_EN_LOW                         clr_P01

#define LCD_DB4_HIGH                       set_P10
#define LCD_DB4_LOW                        clr_P10

#define LCD_DB5_HIGH                       set_P11
#define LCD_DB5_LOW                        clr_P11

#define LCD_DB6_HIGH                       set_P12
#define LCD_DB6_LOW                        clr_P12

#define LCD_DB7_HIGH                       set_P13
#define LCD_DB7_LOW                        clr_P13
```

Demo



Demo video: https://youtu.be/RsetFrAffok

Driving 2x16 LCD with Software SPI

One problem with alphanumeric LCDs and GLCDs is the number of GPIOs needed to connect so with host micros. For a small micro like N76E003, each GPIO pin is like a gem and we can't afford to use too many GPIO pins for an LCD. The solution to this problem is to use SPI/I2C-based LCD drivers that significantly reduce GPIO pin requirement. Implementing software-based SPI/I2C for such LCD drivers is also both easy and universal since these solutions don't need hardware SPI/I2C ports. Since the SPI/I2C functionality is software emulated, any set of GPIO pins can be used – another advantage.

In this segment, we will be driving a 2x16 LCD with CD4094B Serial-In-Parallel-Out (SIPO) shift register using software SPI. The same idea can be used for other similar shift registers like 74HC595. There are other ways of using SPI-based LCDs but the aforementioned are the cheapest ways.

Code

LCD_3_Wire.h

```
#define LCD_GPIO_init()                         do{P02_PushPull_Mode;
P03_PushPull_Mode; P04_PushPull_Mode;}while(0)

#define LCD_SDO_HIGH()                           set_P00
#define LCD_SDO_LOW()                            clr_P00

#define LCD_SCK_HIGH()                           set_P01
#define LCD_SCK_LOW()                            clr_P01

#define LCD_STB_HIGH()                           set_P02
#define LCD_STB_LOW()                            clr_P02

#define DAT                                      1
#define CMD                                      0

#define clear_display                            0x01
#define goto_home                                0x02

#define cursor_direction_inc                     (0x04 | 0x02)
#define cursor_direction_dec                     (0x04 | 0x00)
#define display_shift                            (0x04 | 0x01)
```

```
#define display_no_shift                        (0x04 | 0x00)

#define display_on                              (0x08 | 0x04)
#define display_off                             (0x08 | 0x02)
#define cursor_on                               (0x08 | 0x02)
#define cursor_off                              (0x08 | 0x00)
#define blink_on                                (0x08 | 0x01)
#define blink_off                               (0x08 | 0x00)

#define _8_pin_interface                        (0x20 | 0x10)
#define _4_pin_interface                        (0x20 | 0x00)
#define _2_row_display                          (0x20 | 0x08)
#define _1_row_display                          (0x20 | 0x00)
#define _5x10_dots                              (0x20 | 0x40)
#define _5x7_dots                               (0x20 | 0x00)


extern unsigned char data_value;


void SIPO(void);
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_3_Wire.c

```
#include "N76E003_iar.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "LCD_3_Wire.h"


unsigned char data_value;


void SIPO(void)
{
    unsigned char bit_value = 0x00;
    unsigned char clk = 0x08;
    unsigned char temp = 0x00;

    temp = data_value;
    LCD_STB_LOW();

    while(clk > 0)
    {
        bit_value = ((temp & 0x80) >> 0x07);
        bit_value &= 0x01;
```

```
        switch(bit_value)
        {
            case 0:
            {
                LCD_SDO_LOW();
                break;
            }
            default:
            {
                LCD_SDO_HIGH();
                break;
            }
        }

        LCD_SCK_HIGH();

        temp <<= 0x01;
        clk--;

        LCD_SCK_LOW();
    };

    LCD_STB_HIGH();
}


void LCD_init(void)
{
    Timer0_Delay1ms(10);

    LCD_GPIO_init();


    Timer0_Delay1ms(10);

    data_value = 0x08;
    SIPO();
    Timer0_Delay1ms(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 |= 0x08;
    SIPO();
    Timer0_Delay1ms(2);
    data_value &= 0xF7;
    SIPO();
```

```
    Timer0_Delay1ms(2);
}


void LCD_send(unsigned char value, unsigned char mode)
{
    switch(mode)
    {
        case DAT:
        {
            data_value |= 0x04;
            break;
        }
        default:
        {
            data_value &= 0xFB;
            break;
        }
    }

    SIPO();
    LCD_4bit_send(value);
}


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

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

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


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


void LCD_putchar(char char_data)
{
    if((char_data >= 0x20) && (char_data <= 0x7F))
```

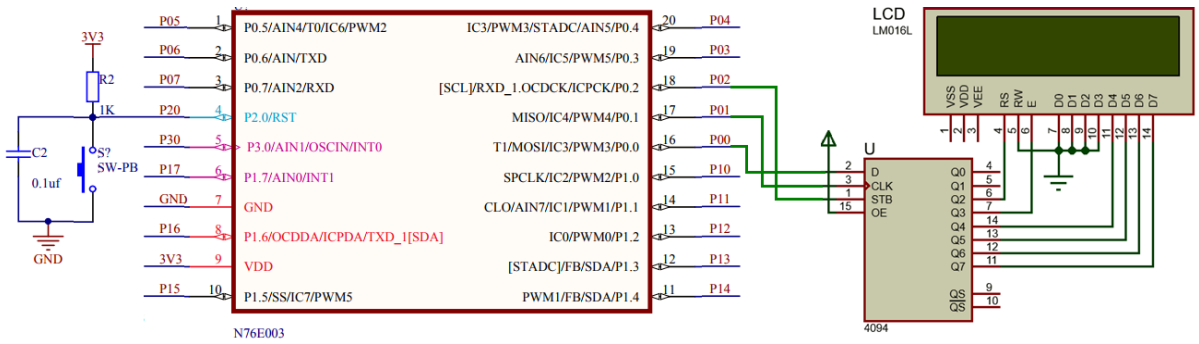
```
    {
        LCD_send(char_data, DAT);
    }
}


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);
    }
}
```

main.c

```
#include "N76E003_iar.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "LCD_3_Wire.h"


void show_value(unsigned char value);


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

  static char txt1[] = {"MICROARENA"};
  static char txt2[] = {"SShahryiar"};
  static char txt3[] = {"Nuvoton 8-bit uC"};
  static char txt4[] = {"N76E003"};

  LCD_init();

  LCD_clear_home();

  LCD_goto(3, 0);
  LCD_putstr(txt1);
  LCD_goto(3, 1);
  LCD_putstr(txt2);
  Timer3_Delay100ms(30);
```

```
  LCD_clear_home();

  for(s = 0; s < 16; s++)
  {
    LCD_goto(s, 0);
    LCD_putchar(txt3[s]);
    Timer0_Delay1ms(90);
  }

  Timer3_Delay100ms(20);

  for(s = 0; s < 7; s++)
  {
    LCD_goto((4 + s), 1);
    LCD_putchar(txt4[s]);
    Timer0_Delay1ms(90);
  }

  Timer3_Delay100ms(30);

  s = 0;
  LCD_clear_home();

  LCD_goto(3, 0);
  LCD_putstr(txt1);

  while(1)
  {
    show_value(s);
    s++;
    Timer3_Delay100ms(4);
  };
}


void show_value(unsigned char value)
{
   unsigned char ch = 0x00;

   ch = ((value / 100) + 0x30);
   LCD_goto(6, 1);
   LCD_putchar(ch);

   ch = (((value / 10) % 10) + 0x30);
   LCD_goto(7, 1);
   LCD_putchar(ch);

   ch = ((value % 10) + 0x30);
   LCD_goto(8, 1);
   LCD_putchar(ch);
}
```

Schematic



Explanation

The code demoed here is same as the last LCD code and so there is not much to explain. The GPIO operations of the LCD are handled using a CD4094B ***Serial-In-Parallel-Out (SIPO)*** shift register. This shift register here acts like an output expander. With just three GPIOs we are able to interface a 4-bit LCD that needs at least six GPIOs to work.

The ***SIPO*** function shown below simulates software-based SPI.

```
void SIPO(void)
{
    unsigned char bit_value = 0x00;
    unsigned char clk = 0x08;
    unsigned char temp = 0x00;

    temp = data_value;
    LCD_STB_LOW();

    while(clk > 0)
    {
        bit_value = ((temp & 0x80) >> 0x07);
        bit_value &= 0x01;

        switch(bit_value)
        {
            case 0:
            {
                LCD_SDO_LOW();
                break;
            }
            default:
            {
                LCD_SDO_HIGH();
                break;
            }
        }

        LCD_SCK_HIGH();

        temp <<= 0x01;
```

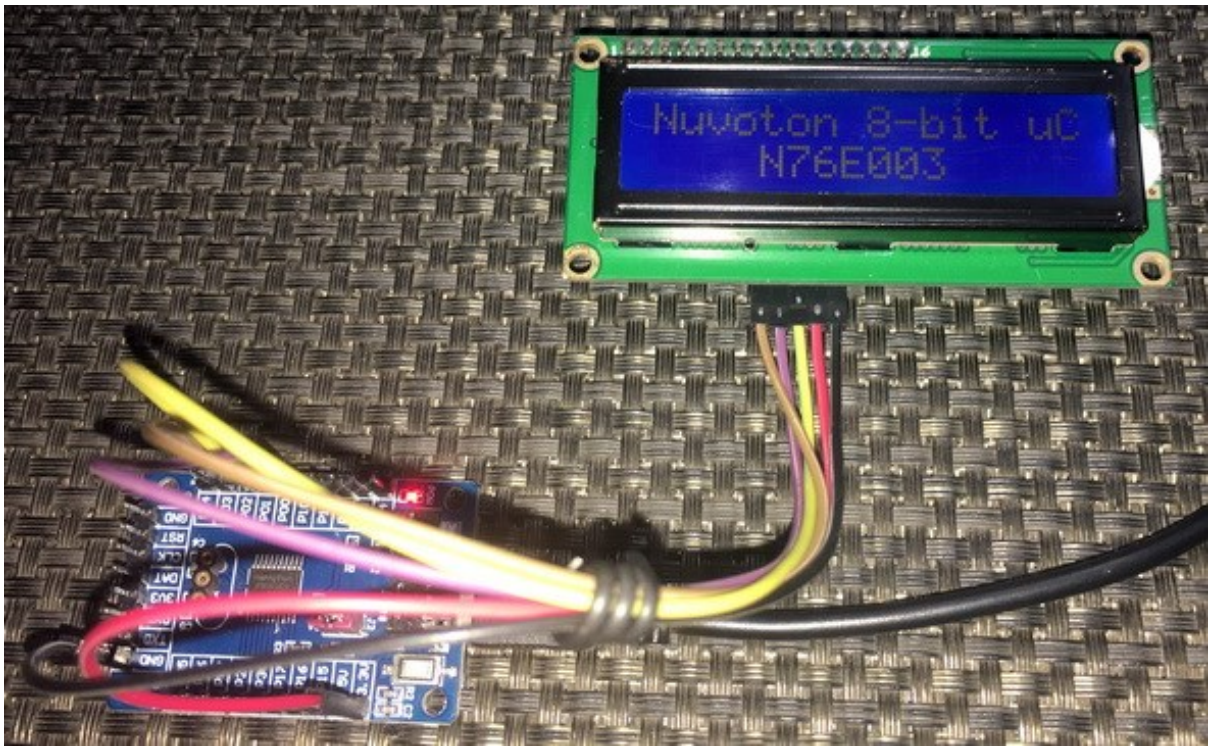
```
        clk--;

        LCD_SCK_LOW();
    };

    LCD_STB_HIGH();
}
```

To change pins, change the following the lines in the ***LCD_3_Wire*** header file:

```
#define LCD_GPIO_init()                          do{P02_PushPull_Mode;
P03_PushPull_Mode; P04_PushPull_Mode;}while(0)

#define LCD_SDO_HIGH()                            set_P00
#define LCD_SDO_LOW()                             clr_P00

#define LCD_SCK_HIGH()                            set_P01
#define LCD_SCK_LOW()                             clr_P01

#define LCD_STB_HIGH()                            set_P02
#define LCD_STB_LOW()                             clr_P02
```

Lastly, I have code two versions of this LCD library – one with BSP-based delays and the other with software delays. Technically there's no big change. The software-based one frees up a hardware timer or two.

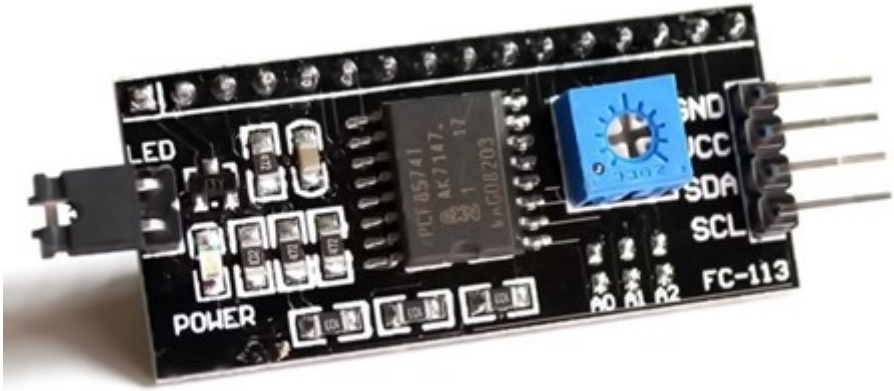
Demo

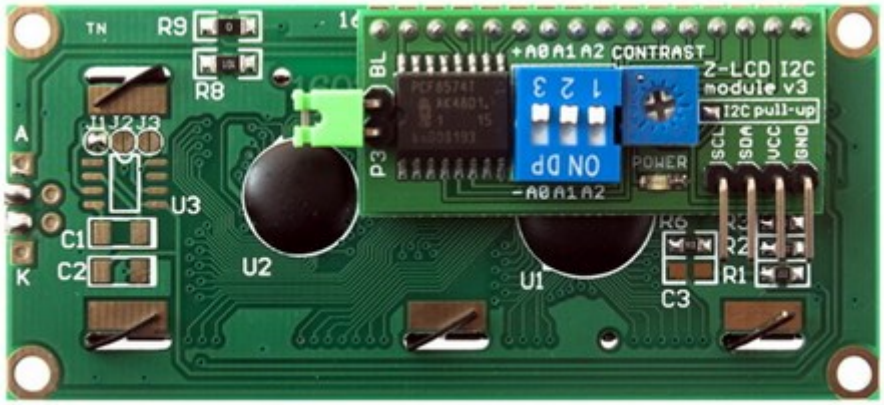


Demo video: https://youtu.be/F0-aeRM5VE4

Driving 2x16 LCD with Software I2C

We have already seen in the last segment how to use software SPI with a shift register to drive a 2x16 LCD. In this segment, we will explore the same concept with software I2C and PCF8574 I2C port expander IC. There is a popular readymade module for such task and I used it here. The advantage of I2C-based LCD over SPI-based LCD driver is the lesser number of GPIOs required compared to SPI-based LCD. However, it is slower than SPI-based drivers.





Code

SW_I2C.h

```
#define SDA_DIR_OUT()    P03_PushPull_Mode
#define SDA_DIR_IN()     P03_Input_Mode

#define SCL_DIR_OUT()    P04_PushPull_Mode
#define SCL_DIR_IN()     P04_Input_Mode

#define SDA_HIGH()       set_P03
#define SDA_LOW()        clr_P03

#define SCL_HIGH()       set_P04
#define SCL_LOW()        clr_P04

#define SDA_IN()         P03
```

```
#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 "N76E003_iar.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "SW_I2C.h"


void SW_I2C_init(void)
{
    SDA_DIR_OUT();
    SCL_DIR_OUT();
    Timer0_Delay100us(1);
    SDA_HIGH();
    SCL_HIGH();
}


void SW_I2C_start(void)
{
    SDA_DIR_OUT();
    SDA_HIGH();
    SCL_HIGH();
    Timer3_Delay10us(4);
    SDA_LOW();
    Timer3_Delay10us(4);
    SCL_LOW();
}


void SW_I2C_stop(void)
{
    SDA_DIR_OUT();
    SDA_LOW();
    SCL_LOW();
    Timer3_Delay10us(4);
    SDA_HIGH();
    SCL_HIGH();
    Timer3_Delay10us(4);
}
```

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

    SDA_DIR_IN();

    while(i > 0)
    {
        SCL_LOW();
        Timer3_Delay10us(2);
        SCL_HIGH();
        Timer3_Delay10us(2);
        j <<= 1;

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

        Timer3_Delay10us(1);
        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;
        Timer3_Delay10us(2);
        SCL_HIGH();
        Timer3_Delay10us(2);
        SCL_LOW();
        Timer3_Delay10us(2);
        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;
        }
    }

    Timer3_Delay10us(2);
    SCL_HIGH();
    Timer3_Delay10us(2);
    SCL_LOW();
}


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

    SDA_DIR_IN();

    SDA_HIGH();
    Timer3_Delay10us(1);
    SCL_HIGH();
    Timer3_Delay10us(1);

    while(SDA_IN() != 0x00)
    {
```

```
        timeout++;

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

    SCL_LOW();
    return 0;
}
```

PCF8574.h

```
#include "SW_I2C.h"


#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 "N76E003_iar.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "PCF8574.h"


void PCF8574_init(void)
{
    SW_I2C_init();
    Timer0_Delay1ms(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.h"


#define clear_display                           0x01
#define goto_home                               0x02

#define cursor_direction_inc                    (0x04 | 0x02)
#define cursor_direction_dec                    (0x04 | 0x00)
#define display_shift                           (0x04 | 0x01)
#define display_no_shift                        (0x04 | 0x00)

#define display_on                              (0x08 | 0x04)
#define display_off                             (0x08 | 0x02)
#define cursor_on                               (0x08 | 0x02)
#define cursor_off                              (0x08 | 0x00)
#define blink_on                                (0x08 | 0x01)
#define blink_off                               (0x08 | 0x00)

#define _8_pin_interface                        (0x20 | 0x10)
#define _4_pin_interface                        (0x20 | 0x00)
#define _2_row_display                          (0x20 | 0x08)
#define _1_row_display                          (0x20 | 0x00)
#define _5x10_dots                              (0x20 | 0x40)
#define _5x7_dots                               (0x20 | 0x00)

#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 "N76E003_iar.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "LCD_2_Wire.h"


static unsigned char bl_state;
static unsigned char data_value;


void LCD_init(void)
{
  PCF8574_init();
  Timer0_Delay1ms(10);

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

  Timer0_Delay1ms(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);
  Timer0_Delay1ms(1);
  data_value &= 0xF9;
  PCF8574_write(data_value);
  Timer0_Delay1ms(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);
  Timer0_Delay1ms(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') ;
}
```

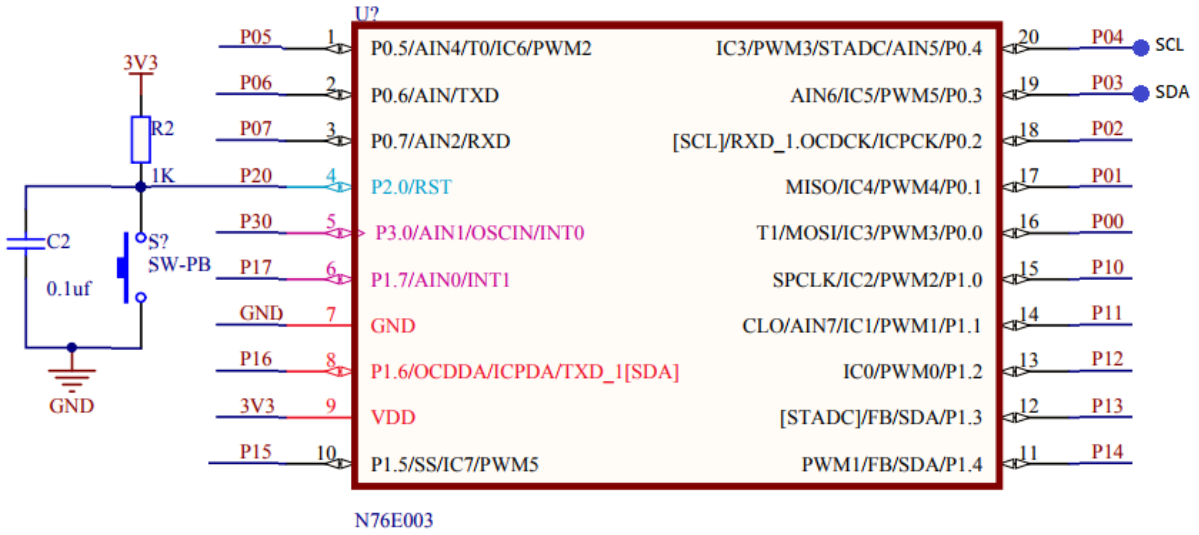
```
void LCD_putchar(char char_data)
{
  if((char_data >= 0x20) && (char_data <= 0x7F))
  {
    LCD_send(char_data, DAT);
  }
}


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);
  }
}
```

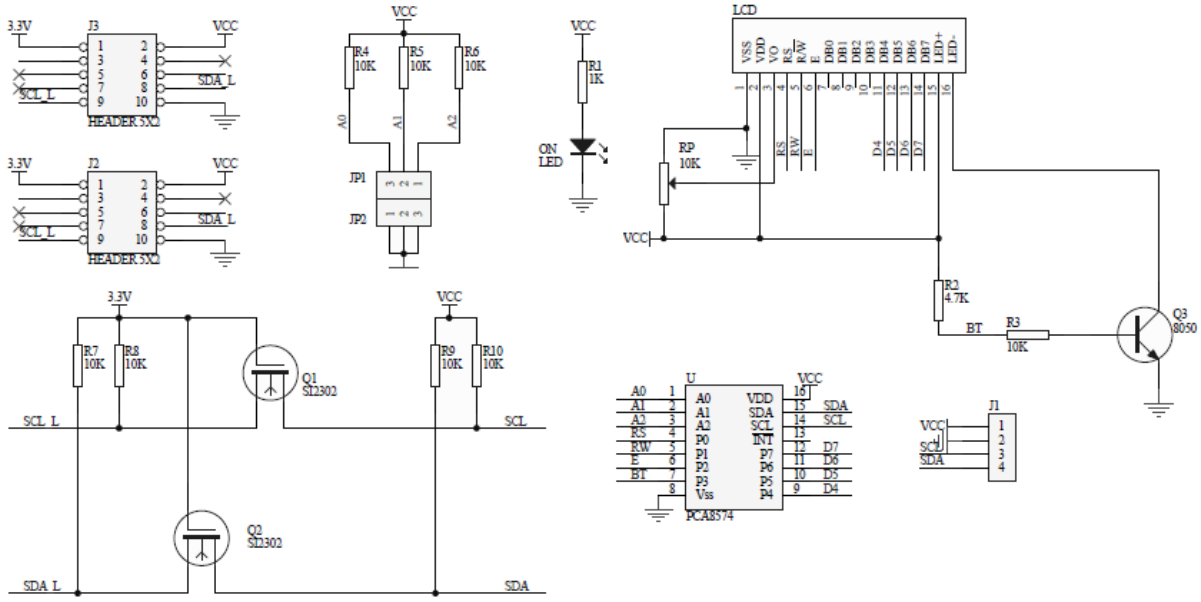
main.c

```
#include "N76E003_iar.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "LCD_2_Wire.h"


void show_value(unsigned char value);


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

  static char txt1[] = {"MICROARENA"};
  static char txt2[] = {"SShahryiar"};
  static char txt3[] = {"Nuvoton 8-bit uC"};
  static char txt4[] = {"N76E003"};

  LCD_init();

  LCD_clear_home();

  LCD_goto(3, 0);
  LCD_putstr(txt1);
```

```
  LCD_goto(3, 1);
  LCD_putstr(txt2);
  Timer3_Delay100ms(30);

  LCD_clear_home();

  for(s = 0; s < 16; s++)
  {
    LCD_goto(s, 0);
    LCD_putchar(txt3[s]);
    Timer0_Delay1ms(90);
  }

  Timer3_Delay100ms(20);

  for(s = 0; s < 7; s++)
  {
    LCD_goto((4 + s), 1);
    LCD_putchar(txt4[s]);
    Timer0_Delay1ms(90);
  }

  Timer3_Delay100ms(30);

  s = 0;
  LCD_clear_home();

  LCD_goto(3, 0);
  LCD_putstr(txt1);

  while(1)
  {
    show_value(s);
    s++;
    Timer3_Delay100ms(4);
  };
}


void show_value(unsigned char value)
{
  unsigned char ch = 0x00;

  ch = ((value / 100) + 0x30);
  LCD_goto(6, 1);
  LCD_putchar(ch);

  ch = (((value / 10) % 10) + 0x30);
  LCD_goto(7, 1);
  LCD_putchar(ch);

  ch = ((value % 10) + 0x30);
  LCD_goto(8, 1);
  LCD_putchar(ch);
}
```

Schematic





Explanation

Just like the last example, software method is used to emulate I2C protocol using ordinary GPIOs. There are three parts of the code – first the software I2C driver, second the driver library for PCF8574 I2C 8-bit port expander and lastly the LCD driver itself. The LCD driver is same as the other LCD drivers in this document. I kept the code modular so that it is easy to understand the role of each piece of code. The I2C driver (***SW_I2C***) implements software I2C which is used by the PCF8574 driver. Thus, the port expander driver is dependent on the ***SW_I2C*** driver and the LCD driver is dependent on the port expander driver, and in cases like such we must find add libraries according to the order of dependency.

The advantage of keeping things modular is to easily modify things in a fast and trouble-free manner while keeping things ready for other deployments. In my codes I try to avoid repetitive and

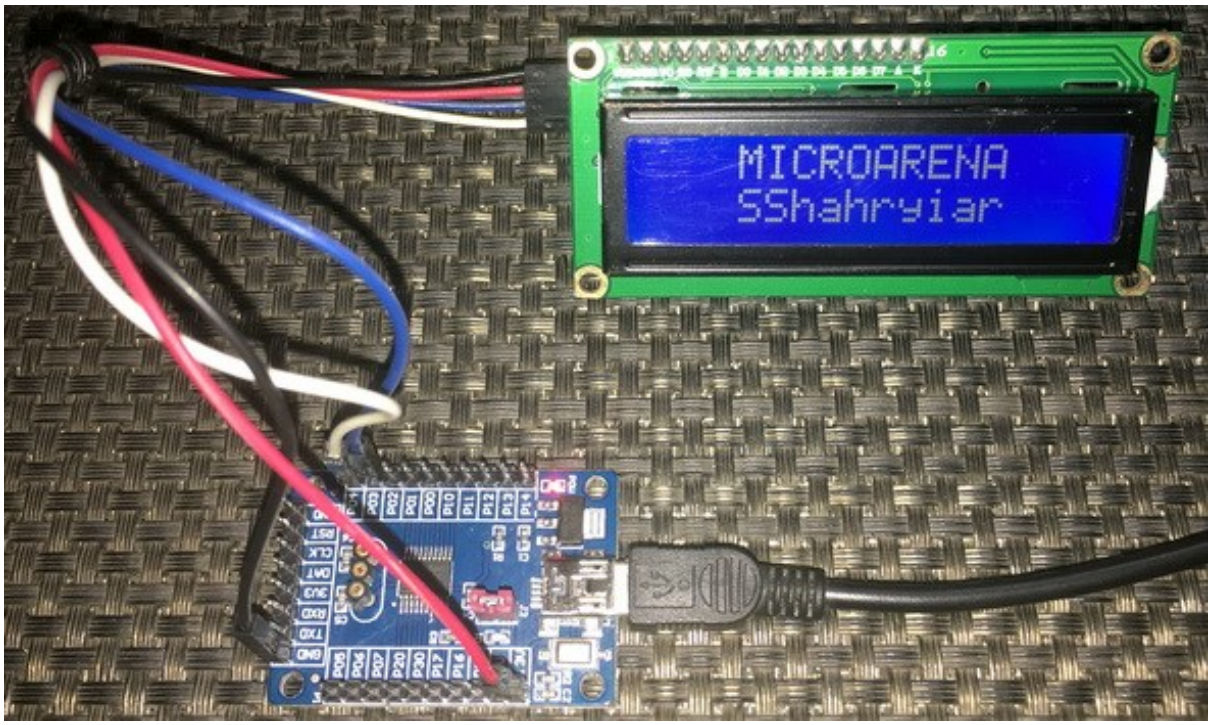
meaningless stuffs with meaningful definitions. For instance, just change the following lines to change pin configurations without going through the whole code:

```
#define SDA_DIR_OUT()    P03_PushPull_Mode
#define SDA_DIR_IN()     P03_Input_Mode

#define SCL_DIR_OUT()    P04_PushPull_Mode
#define SCL_DIR_IN()     P04_Input_Mode

#define SDA_HIGH()       set_P03
#define SDA_LOW()        clr_P03

#define SCL_HIGH()       set_P04
#define SCL_LOW()        clr_P04

#define SDA_IN()         P03
```

Likewise, the ***SW_I2C*** functions are not implemented inside the LCD or port expander driver files so that they can be used for other I2C devices.

I have code two versions of this LCD library just like the SPI-based ones – one with BSP-based delays and the other with software delays.

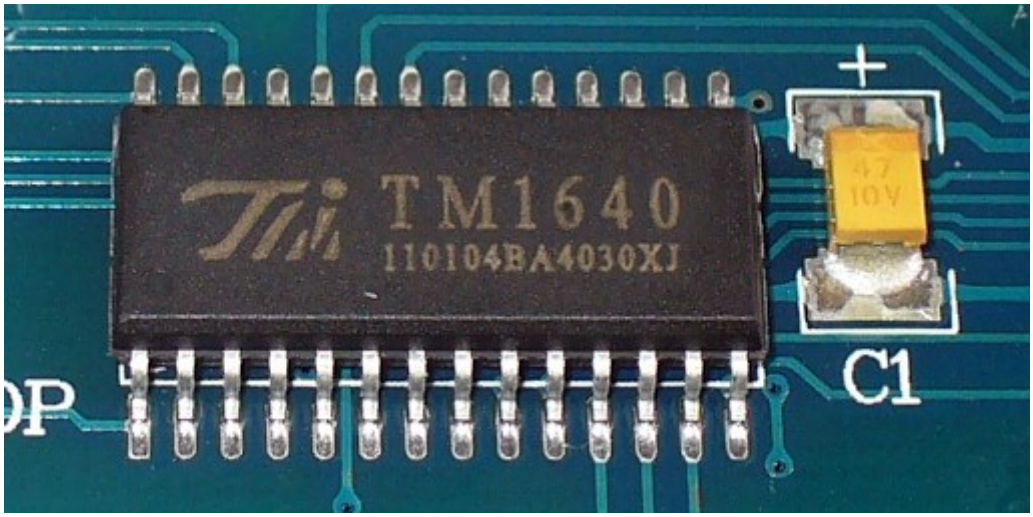
Demo



Demo video: https://youtu.be/vIr6JuSGRxg

Driving seven Segments by Bit-banging TM1640

Seven segment displays take up lot of GPIO pins when they are required to be interfaced with a host micro. There are several driver ICs like MAX7219, TM1640, 74HC594, etc to overcome this issue. TM1640 from Titan Micro Electronics does not support standard I2C or SPI communication protocol unlike most other driver ICs. Thus, to interface it with our host N76E003 micro, we need to apply bit-banging method just like the LCD examples.



Code

fonts.h

```
const unsigned char fonts[11] =
{
  0x00, // (32)   <space>
  0x3F, // (48)   0
  0x06, // (49)   1
  0x5B, // (50)   2
  0x4F, // (51)   3
  0x66, // (52)   4
  0x6D, // (53)   5
  0x7D, // (54)   6
  0x27, // (55)   7
  0x7F, // (56)   8
  0x6F, // (57)   9
};
```

TM1640.h

```
#define TM1640_GPIO_init()                                  do{P03_PushPull_Mode;
P04_PushPull_Mode;}while(0)

#define DIN_pin_HIGH()                                      set_P03
#define DIN_pin_LOW()                                       clr_P03

#define SCLK_pin_HIGH()                                     set_P04
#define SCLK_pin_LOW()                                      clr_P04
```

```
#define no_of_segments                                  16

#define auto_address                                    0x40
#define fixed_address                                   0x44
#define normal_mode                                     0x40
#define test_mode                                       0x48

#define start_address                                   0xC0

#define brightness_5_pc                                 0x88
#define brightness_10_pc                                0x89
#define brightness_25_pc                                0x8A
#define brightness_60_pc                                0x8B
#define brightness_70_pc                                0x8C
#define brightness_75_pc                                0x8D
#define brightness_80_pc                                0x8E
#define brightness_100_pc                               0x8F
#define display_off                                     0x80
#define display_on                                      0x8F


void TM1640_init(unsigned char brightness_level);
void TM1640_start(void);
void TM1640_stop(void);
void TM1640_write(unsigned char value);
void TM1640_send_command(unsigned char value);
void TM1640_send_data(unsigned char address, unsigned char value);
void TM1640_clear_display(void);
```

TM1640.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "TM1640.h"


void TM1640_init(unsigned char brightness_level)
{
  TM1640_GPIO_init();


  Timer0_Delay1ms(10);

  DIN_pin_HIGH();
  SCLK_pin_HIGH();

  TM1640_send_command(auto_address);
  TM1640_send_command(brightness_level);
  TM1640_clear_display();
}
```

```
void TM1640_start(void)
{
  DIN_pin_HIGH();
  SCLK_pin_HIGH();
  Timer3_Delay10us(1);
  DIN_pin_LOW();
  Timer3_Delay10us(1);
  SCLK_pin_LOW();
}


void TM1640_stop(void)
{
  DIN_pin_LOW();
  SCLK_pin_LOW();
  Timer3_Delay10us(1);
  SCLK_pin_HIGH();
  Timer3_Delay10us(1);
  DIN_pin_HIGH();
}


void TM1640_write(unsigned char value)
{
  unsigned char s = 0x08;

  while(s > 0)
  {
    SCLK_pin_LOW();

    if((value & 0x01) == 0x01)
    {
     DIN_pin_HIGH();
    }
    else
    {
     DIN_pin_LOW();
    }

    SCLK_pin_HIGH();

    value >>= 0x01;
    s--;
  };
}


void TM1640_send_command(unsigned char value)
{
  TM1640_start();
  TM1640_write(value);
  TM1640_stop();
}


void TM1640_send_data(unsigned char address, unsigned char value)
```

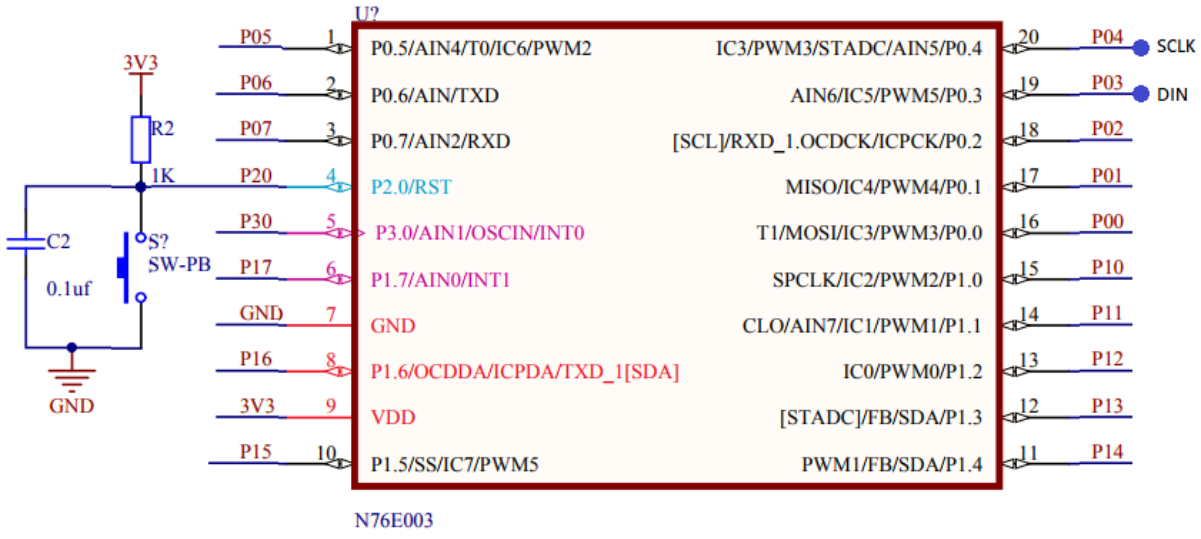
```
{
  TM1640_send_command(fixed_address);

  TM1640_start();

  TM1640_write((0xC0 | (0x0F & address)));
  TM1640_write(value);

  TM1640_stop();
}


void TM1640_clear_display(void)
{
  unsigned char s = 0x00;

  for(s = 0x00; s < no_of_segments; s++)
  {
    TM1640_send_data(s, 0);
  };
}
```

main.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "font.h"
#include "TM1640.h"


void display_data(unsigned char segment, signed int value);


void main(void)
{
  unsigned int i = 0;
  unsigned int j = 999;

  TM1640_init(brightness_75_pc);;

  while(1)
  {
    display_data(0, i++);
    display_data(4, j--);
    Timer3_Delay100ms(4);
  };
}


void display_data(unsigned char segment, signed int value)
{
    unsigned char ch = 0;
```

```
    if((value > 99) && (value <= 999))
    {
      ch = (value / 100);
      TM1640_send_data((2 + segment), fonts[1 + ch]);

      ch = ((value / 10) % 10);
      TM1640_send_data((1 + segment), fonts[1 + ch]);

      ch = (value % 10);
      TM1640_send_data(segment, fonts[1 + ch]);
    }

    else if((value > 9) && (value <= 99))
    {
      TM1640_send_data((2 + segment), 0);

      ch = (value / 10);
      TM1640_send_data((1 + segment), fonts[1 + ch]);

      ch = (value % 10);
      TM1640_send_data(segment, fonts[1 + ch]);
    }

    else
    {
      TM1640_send_data((2 + segment), 0);

      TM1640_send_data((1 + segment), 0);

      ch = (value % 10);
      TM1640_send_data(segment, fonts[1 + ch]);
    }
}
```

Schematic

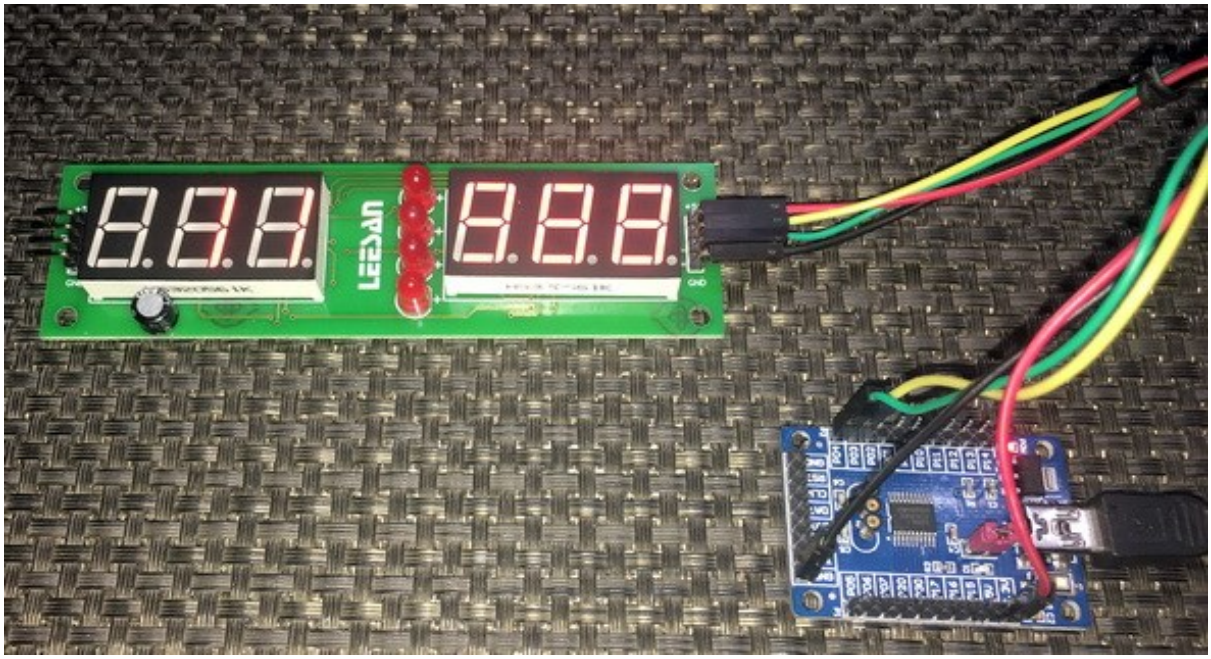


Explanation

Like the LCD libraries demoed previously, TM1640 is driven with GPIO bit-banging. Please read the datasheet of TM1640 to fully understand how the codes are implemented. It uses two pins just like I2C but don't be fooled as it doesn't support I2C protocol. It uses a protocol of its own. To change pin configuration, just change the following lines of code:

```
#define TM1640_GPIO_init()                                  do{P03_PushPull_Mode;
P04_PushPull_Mode;}while(0)

#define DIN_pin_HIGH()                                      set_P03
#define DIN_pin_LOW()                                       clr_P03

#define SCLK_pin_HIGH()                                     set_P04
#define SCLK_pin_LOW()                                      clr_P04
```

Demo



Demo video: https://youtu.be/2ufeAQkt5Jk

External Interrupt (EXTI)

External interrupt is a key GPIO feature in input mode. It momentarily interrupts regular program flow just like other interrupts and does some tasks before resuming interrupted task. In traditional 8051s, there are two external interrupts with dedicated and separate interrupt vector addresses. The same applies to N76E003. Highlighted below in the N76E003's interrupt vector table are the interrupt vector addresses/numbers of these two external interrupts:

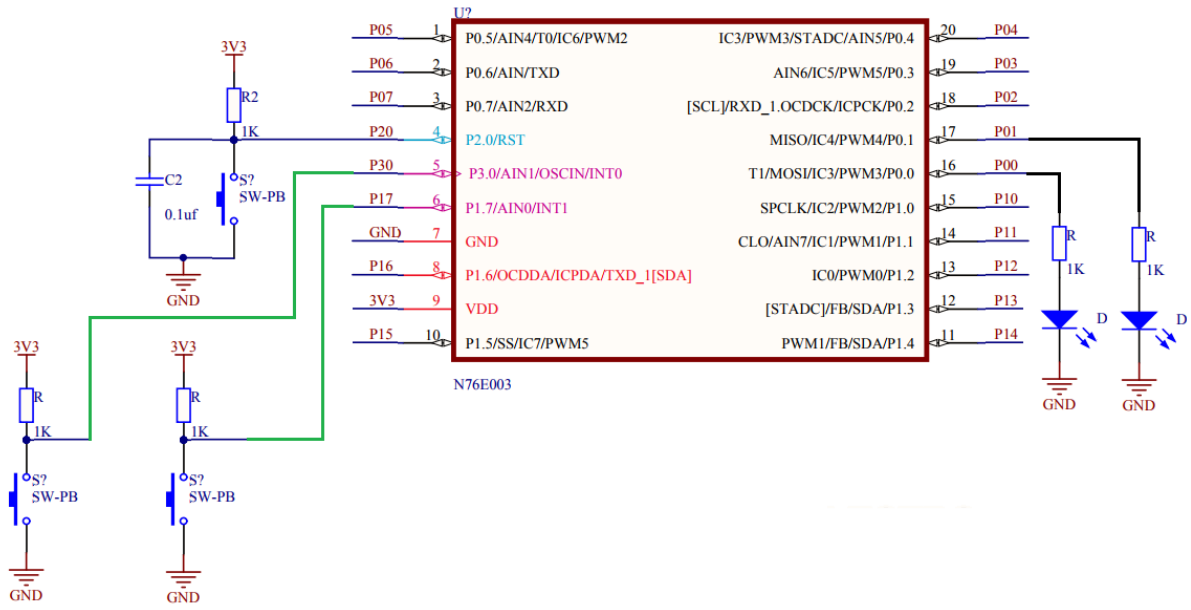
Source	Vector Address	Vector Number	Source	Vector Address	Vector Number
Reset	0000H	-	SPI interrupt	004BH	9
External interrupt 0	0003H	0	WDT interrupt	0053H	10
Timer 0 overflow	000BH	1	ADC interrupt	005BH	11
External interrupt 1	0013H	2	Input capture interrupt	0063H	12
Timer 1 overflow	001BH	3	PWM interrupt	006BH	13
Serial port 0 interrupt	0023H	4	Fault Brake interrupt	0073H	14
Timer 2 event	002BH	5	Serial port 1 interrupt	007BH	15
I^2C status/timer-out interrupt	0033H	6	Timer 3 overflow	0083H	16
Pin interrupt	003BH	7	Self Wake-up Timer interrupt	008BH	17
Brown-out detection interrupt	0043H	8			

Code

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"


void setup(void);


void EXT_INT0(void)
interrupt 0
{
  set_P00;
}


void EXT_INT1(void)
interrupt 2
{
  set_P01;
}


void main(void)
{
  setup();
```

```
  while(1)
  {
    Timer0_Delay1ms(1000);
    clr_P00;
    clr_P01;
  };
}


void setup(void)
{
  P00_PushPull_Mode;
  P01_PushPull_Mode;
  P17_Input_Mode;
  P30_Input_Mode;
  set_P1S_7;
  set_P3S_0;
  set_IT0;
  set_IT1;
  set_EX0;
  set_EX1;
  set_EA;
}
```

Schematic



Explanation

The setup for this demo is simple. There are two LEDs and two buttons connected with a N76E003 chip as per schematic. The buttons are connected with external interrupt pins. Obviously, these pins are declared as input pins. Additionally, internal input Schmitt triggers of these pins are used to ensure

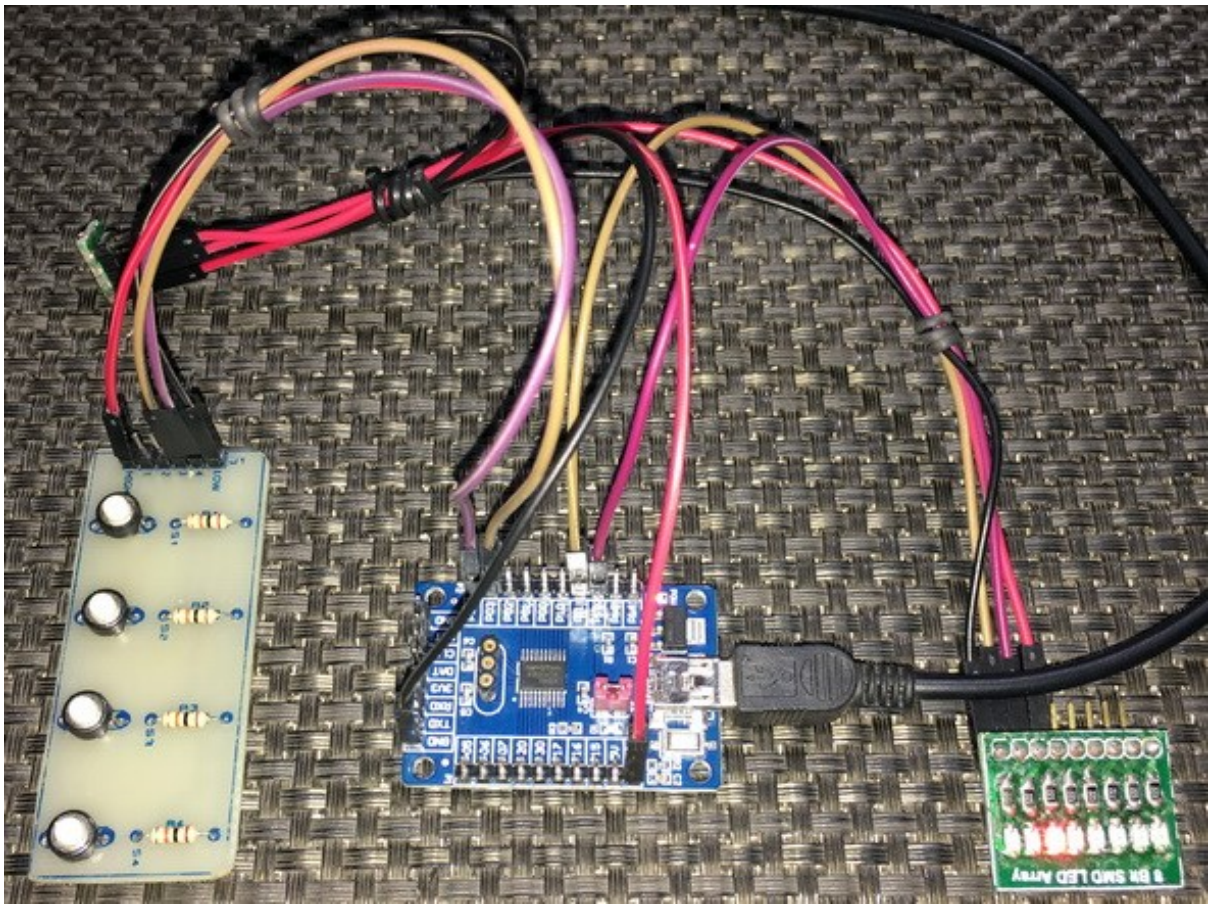
noise cancellation. Both interrupts are enabled along with their respective interrupt hardware. Finally, global interrupt is set. Optionally interrupt priority can be applied.

```
P17_Input_Mode;
P30_Input_Mode;

set_P1S_7;
set_P3S_0;

set_IT0;
set_IT1;

set_EX0;
set_EX1;
set_EA;
```

Since we enabled two interrupts with different interrupt vectors, there will be two interrupt subroutine functions. Each of these functions will briefly turn on LEDs assigned to them. The LEDs are turned off in the main function. Thus, the LEDs mark which interrupt occurred.

```
void EXT_INT0(void)
interrupt 0
{
  set_P00;
}


void EXT_INT1(void)
interrupt 2
{
  set_P01;
}
```

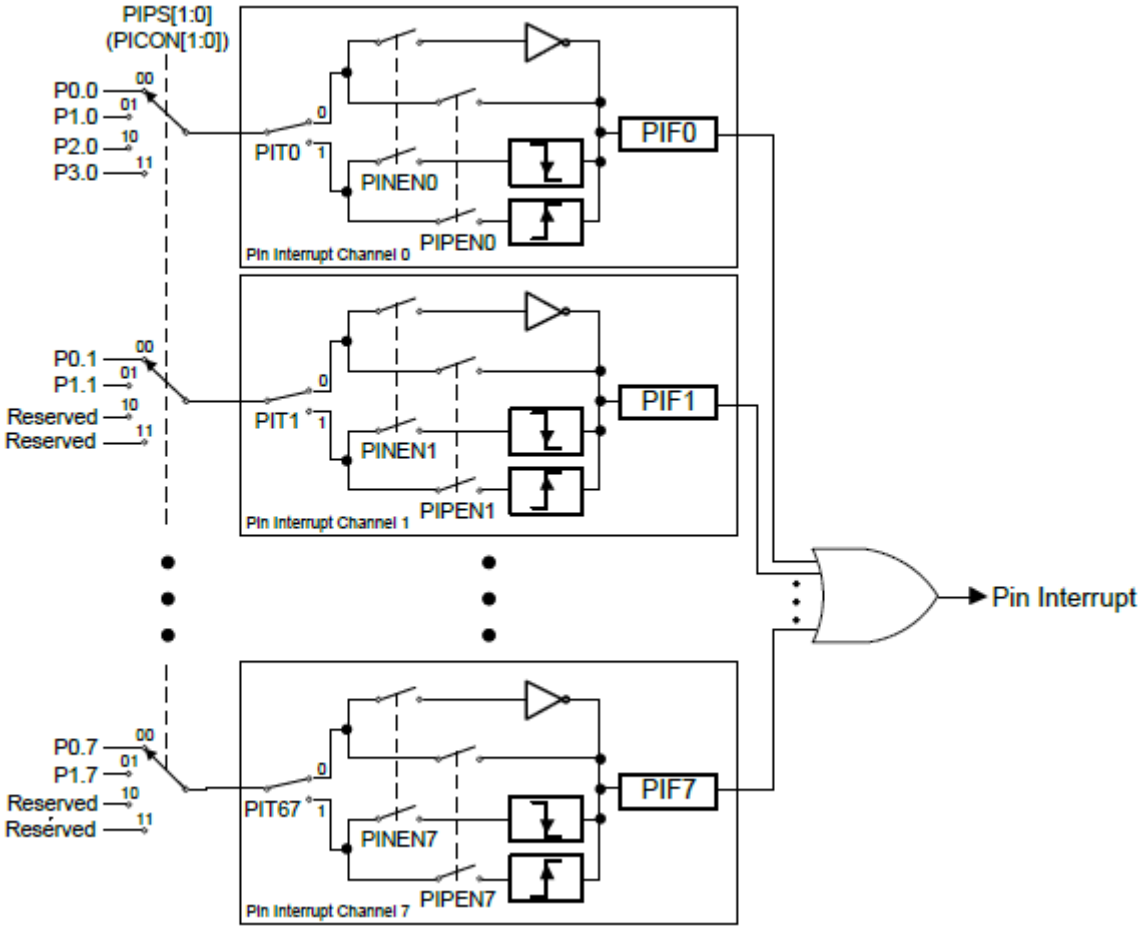
Demo



Demo video: https://youtu.be/fFpBpoNMNyI

Pin Interrupt – Interfacing Rotary Encoder

Apart from dedicated external interrupts, N76E003 is equipped with pin interrupt facility - a feature that can be found in almost every microcontroller of modern times. With pin interrupt, any GPIO can be made to behave like external interrupt. However, unlike external interrupts, a single hardware interrupt channel and therefore one vector address is used for mapping a maximum of eight different GPIO pins. These pins need not to be on the same GPIO port. When interrupt occurs, we need to assert from which pin it originated. This feature becomes very useful when interfacing keypads and buttons.



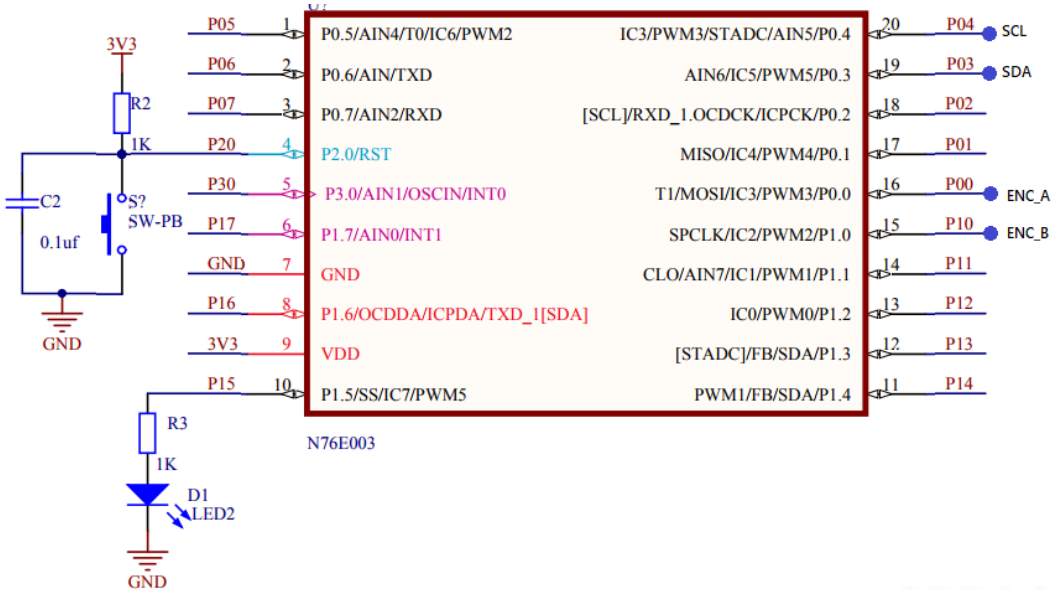
Code

```
#include "N76E003_IAR.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "soft_delay.h"
#include "LCD_2_Wire.h"


signed char encoder_value = 0;


void setup(void);
```

```
void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned int value);


#pragma vector = 0x3B
__interrupt void PIN_INT(void)
{
  clr_EA;

  if(PIF == 0x01)
  {
    if((P1 & 0x03) == 0x02)
    {
      encoder_value++;
    }

    if(encoder_value > 99)
    {
      encoder_value = 0;
    }
  }

  if(PIF == 0x02)
  {
    if((P1 & 0x03) == 0x01)
    {
      encoder_value--;
    }

    if(encoder_value < 0)
    {
      encoder_value = 99;
    }
  }

  PIF = 0x00;

  P15 = ~P15;
}


void main(void)
{
  setup();

  while(1)
  {
    set_EA;
    lcd_print(14, 0, encoder_value);
    delay_ms(40);
  }
}


void setup(void)
{
  P10_Input_Mode;
```

```
  P11_Input_Mode;

  P15_PushPull_Mode;

  Enable_BIT0_LowLevel_Trig;
  Enable_BIT1_LowLevel_Trig;

  Enable_INT_Port1;

  set_EPI;

  LCD_init();
  LCD_clear_home();
  LCD_goto(0, 0);
  LCD_putstr("ENC Count:");
}


void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned int value)
{
  LCD_goto(x_pos, y_pos);
  LCD_putchar((value / 10) + 0x30);
  LCD_goto((x_pos + 1), y_pos);
  LCD_putchar((value % 10) + 0x30);
}
```

Schematic



Explanation

Pin interrupt is not same as dedicated external interrupt but still it is very useful in a number of cases. In this demo, two pin interrupts are used to decode a rotary encoder. Probably this is the simplest method of decoding a rotary encoder.

Setting up pin interrupt is very easy. We need to set the pin interrupt pins are inputs. We can optionally use the internal Schmitt triggers. Then we decide the edge to detect and which ports to check for pin interrupt. Finally, we set the pin interrupt hardware.

```
P10_Input_Mode;
P11_Input_Mode;

Enable_BIT0_LowLevel_Trig;
Enable_BIT1_LowLevel_Trig;

Enable_INT_Port1;

set_EPI;
```

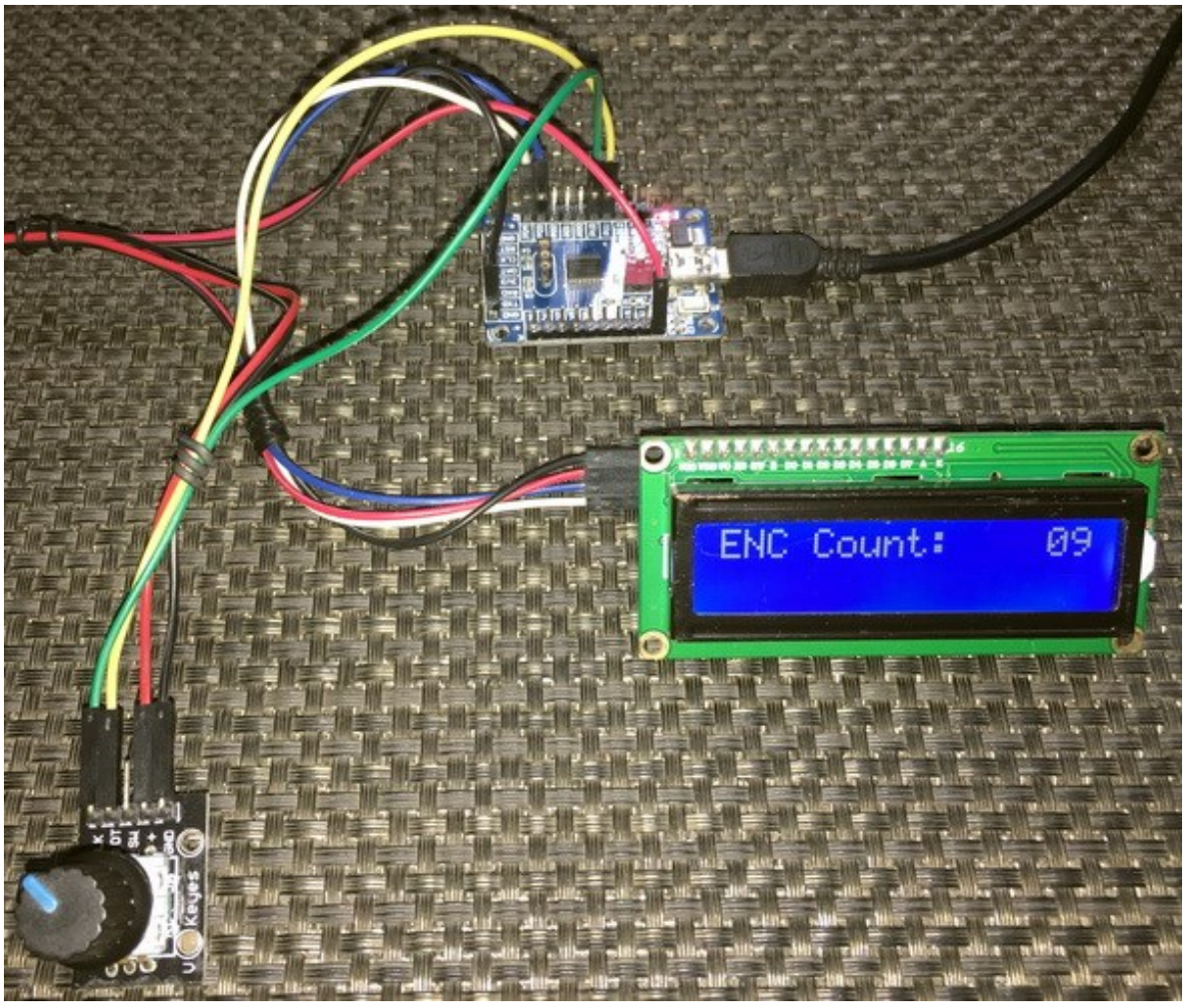
Inside the pin interrupt function, we need to check which pin shot the interrupt by checking respective flags. Encoder count is incremented/decremented based on which flag got shot first and the logic state of the other pin. Since here a rotary encoder is interfaced with pin interrupt facility of N76E003, we have to ensure that the micro doesn't detect any further or false interrupts while already processing one interrupt condition. This is why the global interrupt is disabled every time the code enters the pin interrupt function. This is restarted in the main. Similarly, to ensure proper working we have clear the interrupt flags before exiting the function. P15 is toggled with interrupt to visually indicate the rotation of the encoder.

```
#pragma vector = 0x3B
__interrupt void PIN_INT(void)
{
  clr_EA;

  if(PIF == 0x01)
  {
    if((P1 & 0x03) == 0x02)
    {
      encoder_value++;
    }

    if(encoder_value > 99)
    {
      encoder_value = 0;
    }
  }

  if(PIF == 0x02)
  {
    if((P1 & 0x03) == 0x01)
    {
      encoder_value--;
    }

    if(encoder_value < 0)
    {
      encoder_value = 99;
    }
```

```
  }

  PIF = 0x00;

  P15 = ~P15;
}
```

The main code just shows the encoder count. When the encoder is rotated in one direction, the count increases while rotating it in the opposite direction causes the encoder count to decrease.

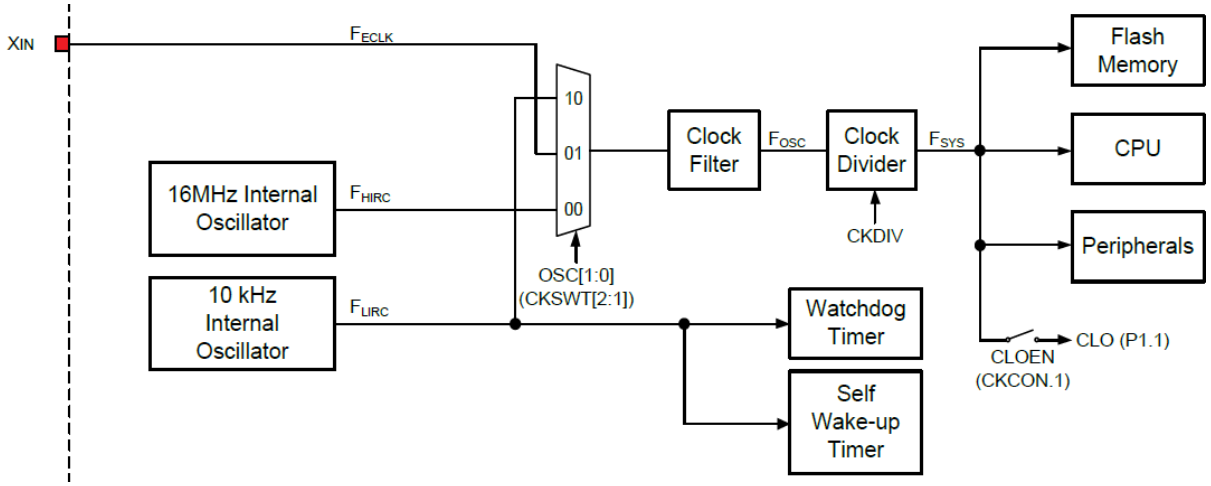
Demo



Demo video: https://youtu.be/nPEzTYj70ys

Clock System

The clock system of N76E003 is very straight forward and very flexible. To begin with, there are three clock sources, a clock selector and a common clock divider block apart from other blocks. Shown below is the block diagram of N76E003's clock system:



The three sources are as follows:

Source	*Max Clock Speed*	*Reliability/Accuracy*	*Purpose*
External Clock (**ECLK**)	**16 MHz** – External clock signal generator like crystal oscillator modules/electronic circuitry	**High** - (dependent on oscillator accuracy and type)	Main system clock when high timing precision is needed
High Speed Internal RC Oscillator (**HIRC**)	**16 MHz** – Internal high frequency RC oscillator with 1% tolerance	**Reasonable** - (±1% for most typical temperature ranges and ±2% for extreme temperatures)	General purpose main clock with 1% tolerance or backup clock source when external clock is present
Low Speed Internal RC Oscillator (**LIRC**)	**10 kHz** - Internal low frequency RC oscillator with 10% tolerance	**Low** - (±10% for most typical temperature ranges and ±35% for extreme temperatures)	Usually used for low power idle modes when high timing precision is not required, not recommend for RTC and can't be disabled

Once a given clock source is set, it becomes the clock for all systems. The only exception here is the watchdog timer and the self-wake-up timer which are only run by the LIRC.

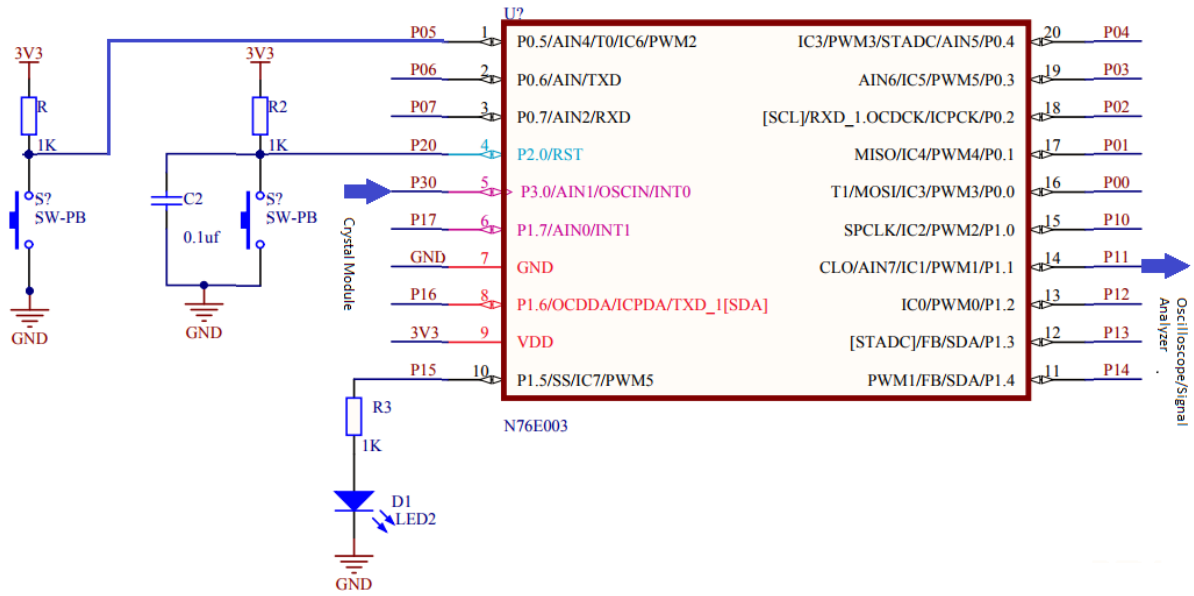
Code

```
#include "N76E003_iar.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"


#define HIRC    0
#define LIRC    1
#define ECLK    2


void set_clock_source(unsigned char clock_source);
void disable_clock_source(unsigned char clock_source);
void set_clock_division_factor(unsigned char value);


void main(void)
{
  signed char i = 30;

  P11_PushPull_Mode;
  P15_PushPull_Mode;

  set_clock_division_factor(0);
  set_clock_source(HIRC);

  set_CLOEN;

  while(i > 0)
  {
    clr_P15;
    Timer0_Delay1ms(100);
    set_P15;
    Timer0_Delay1ms(100);
    i--;
  }

  set_clock_source(ECLK);
  disable_clock_source(HIRC);

  i = 30;

  while(i > 0)
  {
    clr_P15;
    Timer0_Delay1ms(100);
    set_P15;
    Timer0_Delay1ms(100);
    i--;
  }

  set_clock_source(LIRC);
  disable_clock_source(HIRC);
```

```
  while(1)
  {
    clr_P15;
    Timer0_Delay1ms(1);
    set_P15;
    Timer0_Delay1ms(1);
  };
}


void set_clock_source(unsigned char clock_source)
{
  switch(clock_source)
  {
    case LIRC:
    {
      set_OSC1;
      clr_OSC0;

      break;
    }

    case ECLK:
    {
      set_EXTEN1;
      set_EXTEN0;

      while((CKSWT & SET_BIT3) == 0);

      clr_OSC1;
      set_OSC0;

      break;
    }

    default:
    {
      set_HIRCEN;

      while((CKSWT & SET_BIT5) == 0);

      clr_OSC1;
      clr_OSC0;

      break;
    }
  }

  while((CKEN & SET_BIT0) == 1);
}


void disable_clock_source(unsigned char clock_source)
{
  switch(clock_source)
```

```
  {
    case HIRC:
    {
      clr_HIRCEN;
      break;
    }

    default:
    {
      clr_EXTEN1;
      clr_EXTEN0;
      break;
    }
  }
}


void set_clock_division_factor(unsigned char value)
{
  CKDIV = value;
}
```

Schematic

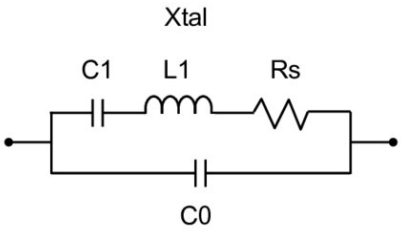


Explanation

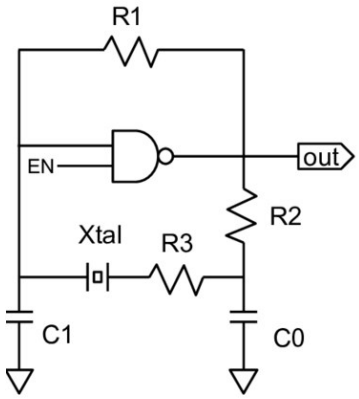
The very first thing to note is the absence of two ***OSC*** pins unlike other micros and yet a crystal resonator is connected with P30 and P17. This configuration is not correct. There is only ***OSCIN*** pin. This is because N76E003 can only be driven with active clock sources like crystal modules, external electronic circuitry, etc. The HIRC clock source is accurate enough for most purposes and there is literally no need for external clock. I have done most of the experiments with HIRC and I'm satisfied with it.

Many people don't understand the difference between a crystal oscillator module and a crystal resonator. Both are based on quartz crystals but the oscillator one has internal electronics to generate clock pulses precisely while the resonator just contains the quartz crystal. Crystal modules are accurate compared to resonators because the internal electronics in them take care of the effects of temperature. Resonators are therefore called passive clock crystals while the clock modules are termed active clocks.

Crystal Resonator



Crystal Oscillator



Here to test all three clock sources, I used two things – first the onboard LED and second the clock output pin. Different clock sources are enabled briefly one after another and the onboard LED is blinked. The blinking rate of the LED is an indirect indicator of clock speed. The clock output too is monitored with an oscilloscope/signal analyser for clock speeds. HIRC is turned on first, then ECLK and finally LIRC. By default, both HIRC and LIRC are turned on during power on. When switching between clock sources, we should poll if the new clock source is stable prior to using it and disable the one that we don't need.

I have coded the following three for setting up the clock system. Their names suggest their purposes.

```
void set_clock_source(unsigned char clock_source);
void disable_clock_source(unsigned char clock_source);
void set_clock_division_factor(unsigned char value);
```

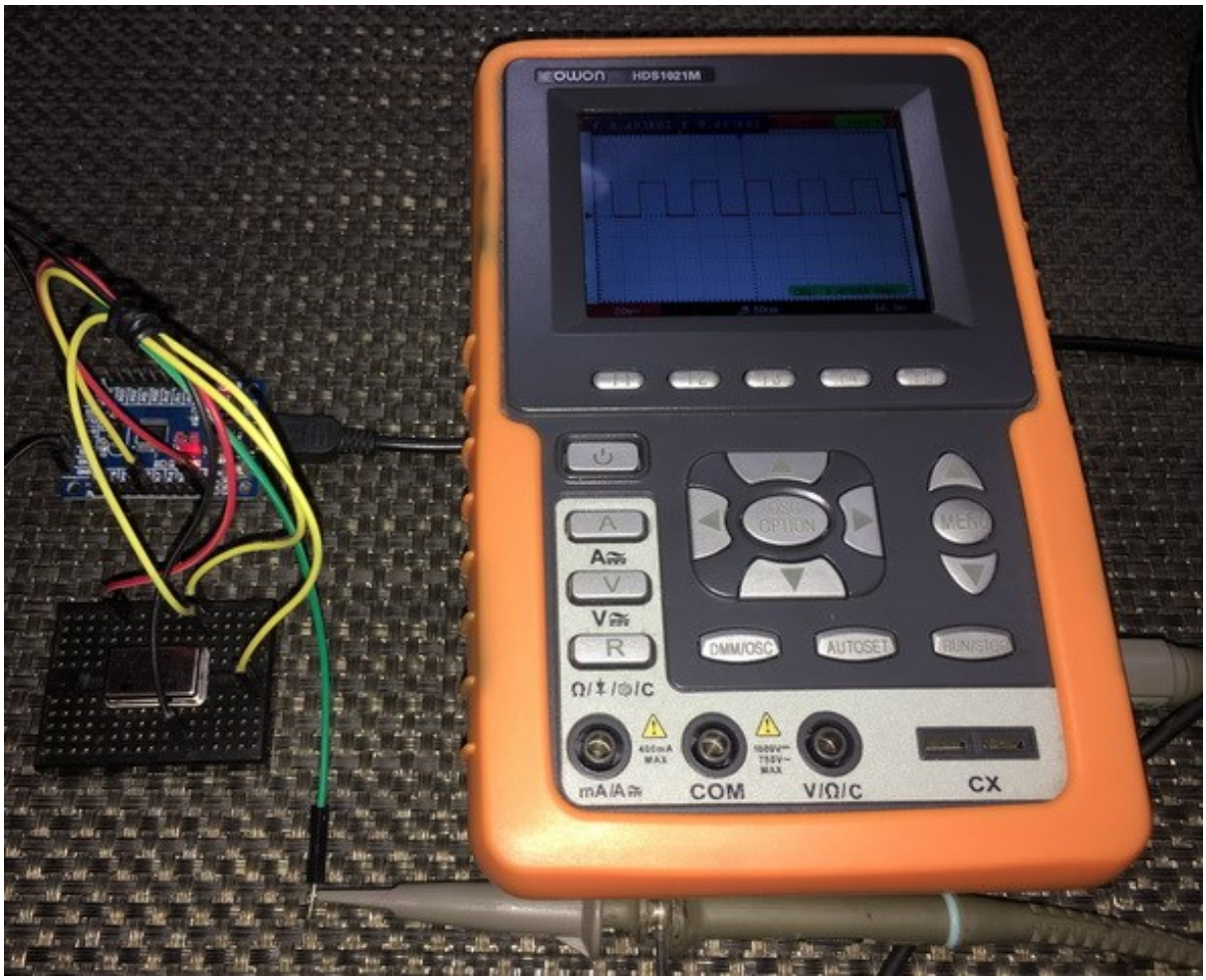
These three functions will be all that you'll ever need to configure the clock system without any hassle. The first two are most important as they select clock source and disabled the one that is not need. If you are still confused about setting the system clock then you can avoid the clock division function and straight use the following function:

```
void set_clock_frequency(unsigned long F_osc, unsigned long F_sys)
{
  F_osc = (F_osc / (2 * F_sys));
```

```
  if((F_osc >= 0x00) && (F_osc <= 0xFF))
  {
    CKDIV = ((unsigned char)F_osc);
  }
}
```

This function takes two parameters – the frequency of the clock source and the frequency of the system after clock division.

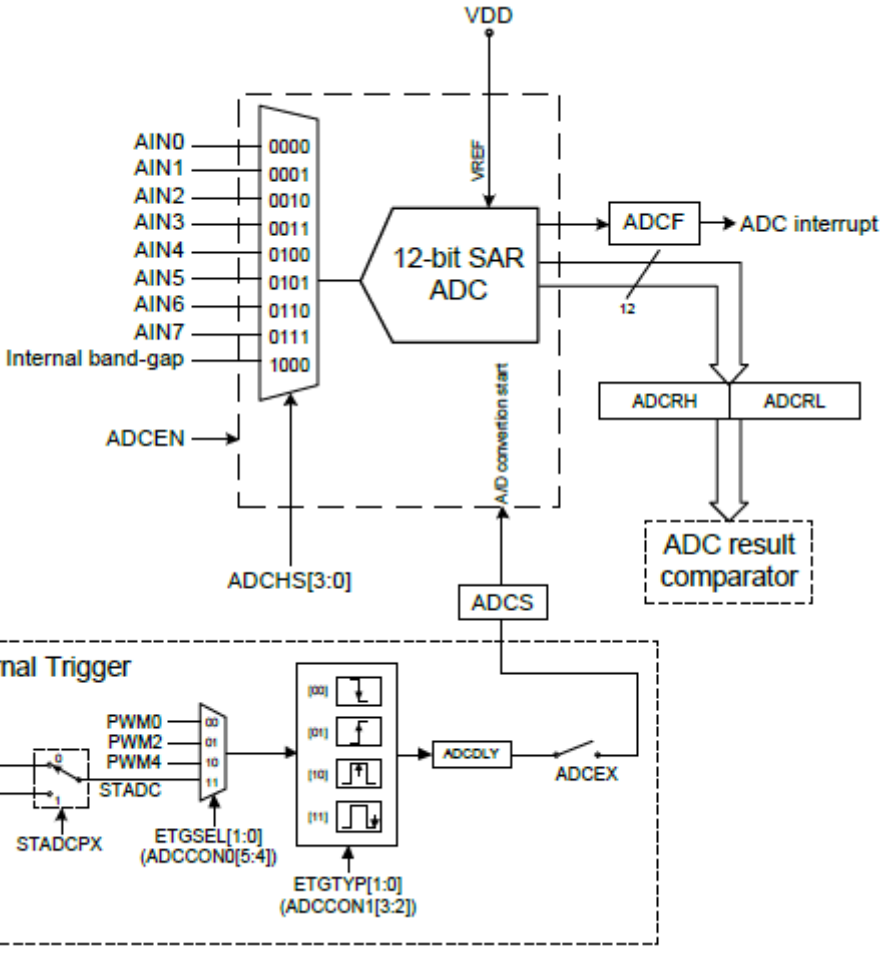
Demo

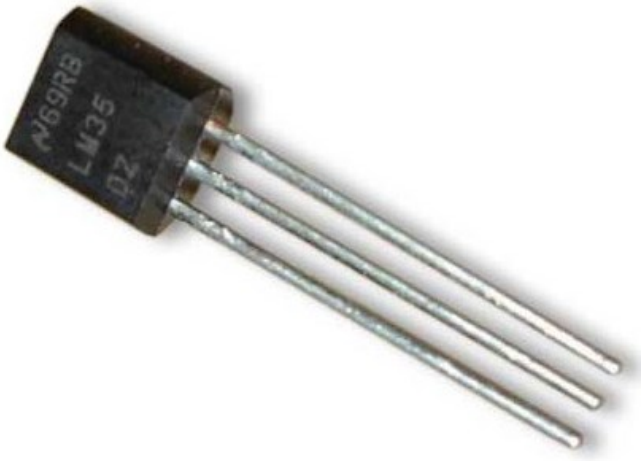


Demo video: https://youtu.be/C_01oeoqAVE

12-Bit ADC – LM35 Thermometer

Most 8051s don't have any embedded ADC but N76E003 comes with a 12-bit SAR ADC. This is also one area where N76E003 differs a lot from STM8S003. The 12-bit resolution is the factor. N76E003 has eight single-ended ADC inputs along with a bandgap voltage generator and a built-in comparator. The ADC can be triggered internally with software or by external hardware pins/PWM. Everything is same as the ADCs of other microcontrollers and there's not much difference.



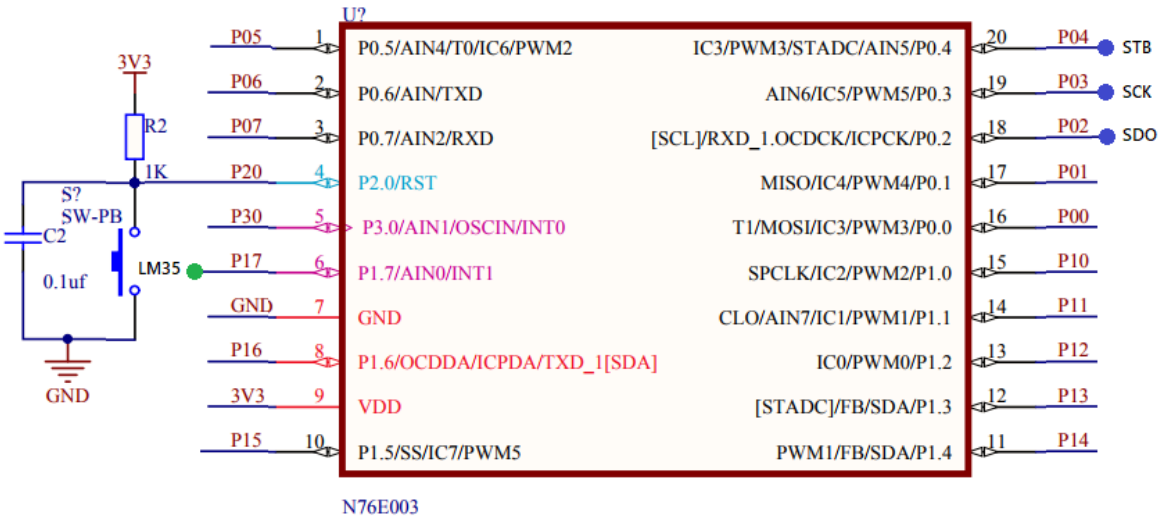


Code

```
#include "N76E003_IAR.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "LCD_3_Wire.h"


#define scalar          0.12412


void setup(void);
unsigned int ADC_read(void);
void lcd_print_i(unsigned char x_pos, unsigned char y_pos, unsigned int
value);
void lcd_print_f(unsigned char x_pos, unsigned char y_pos, unsigned int
value);


void main(void)
{
  unsigned int temp = 0;
  unsigned int adc_count = 0;

  setup();

  while(1)
  {
    adc_count = ADC_read();
    temp = ((unsigned int)(((float)adc_count) / scalar));
    lcd_print_i(12, 0, adc_count);
    lcd_print_f(11, 1, temp);
    Timer0_Delay1ms(600);
  }
}


void setup(void)
{
  LCD_init();
  LCD_clear_home();
  LCD_goto(0, 0);
  LCD_putstr("ADC Count:");
  LCD_goto(0, 1);
  LCD_putstr("Tmp/deg C:");

  Enable_ADC_AIN0;
}


unsigned int ADC_read(void)
{
  register unsigned int value = 0x0000;
```

```
  clr_ADCF;
  set_ADCS;
  while(ADCF == 0);

  value = ADCRH;
  value <<= 4;
  value |= ADCRL;

  return value;
}


void lcd_print_i(unsigned char x_pos, unsigned char y_pos, unsigned int value)
{
  LCD_goto(x_pos, y_pos);
  LCD_putchar((value / 1000) + 0x30);
  LCD_goto((x_pos + 1), y_pos);
  LCD_putchar(((value % 1000) / 100) + 0x30);
  LCD_goto((x_pos + 2), y_pos);
  LCD_putchar(((value % 100) / 10) + 0x30);
  LCD_goto((x_pos + 3), y_pos);
  LCD_putchar((value % 10) + 0x30);
}

void lcd_print_f(unsigned char x_pos, unsigned char y_pos, unsigned int value)
{
  LCD_goto(x_pos, y_pos);
  LCD_putchar((value / 1000) + 0x30);
  LCD_goto((x_pos + 1), y_pos);
  LCD_putchar(((value % 1000) / 100) + 0x30);
  LCD_goto((x_pos + 2), y_pos);
  LCD_putchar('.');
  LCD_goto((x_pos + 3), y_pos);
  LCD_putchar(((value % 100) / 10) + 0x30);
  LCD_goto((x_pos + 4), y_pos);
  LCD_putchar((value % 10) + 0x30);
}
```

Schematic



Explanation

In this demo, one ADC channel (***AIN0***) is used to read a LM35 temperature sensor. Polling method is used to read the ADC.

Enabling the ADC is simply done by coding the following line:

```
Enable_ADC_AIN0;
```

In the background of this, ADC channel selection and other parameters are set. If you want more control over the ADC then you must set the ADC registers on your own. Most of the times that can be avoided.

Reading the ADC needs some attention because the ADC data registers are not aligned like other registers and we just need 12-bits, not 8/16-bits.

ADCRH – ADC Result High Byte

7	6	5	4	3	2	1	0
ADCR[11:4]							
R							

Address: C3H — Reset value: 0000 0000b

ADCRL – ADC Result Low Byte

7	6	5	4	3	2	1	0
-	-	-	-	ADCR[3:0]			
-	-	-	-	R			

Address: C2H — Reset value: 0000 0000b

Notice that the we must extract ADC from ***ADCCRH*** and from the low four bits of ***ADCCRL***. To handle this issue the follow function is devised:

```
unsigned int ADC_read(void)
{
  register unsigned int value = 0x0000;
```

```
  clr_ADCF;
  set_ADCS;
  while(ADCF == 0);

  value = ADCRH;
  value <<= 4;
  value |= ADCRL;

  return value;
}
```

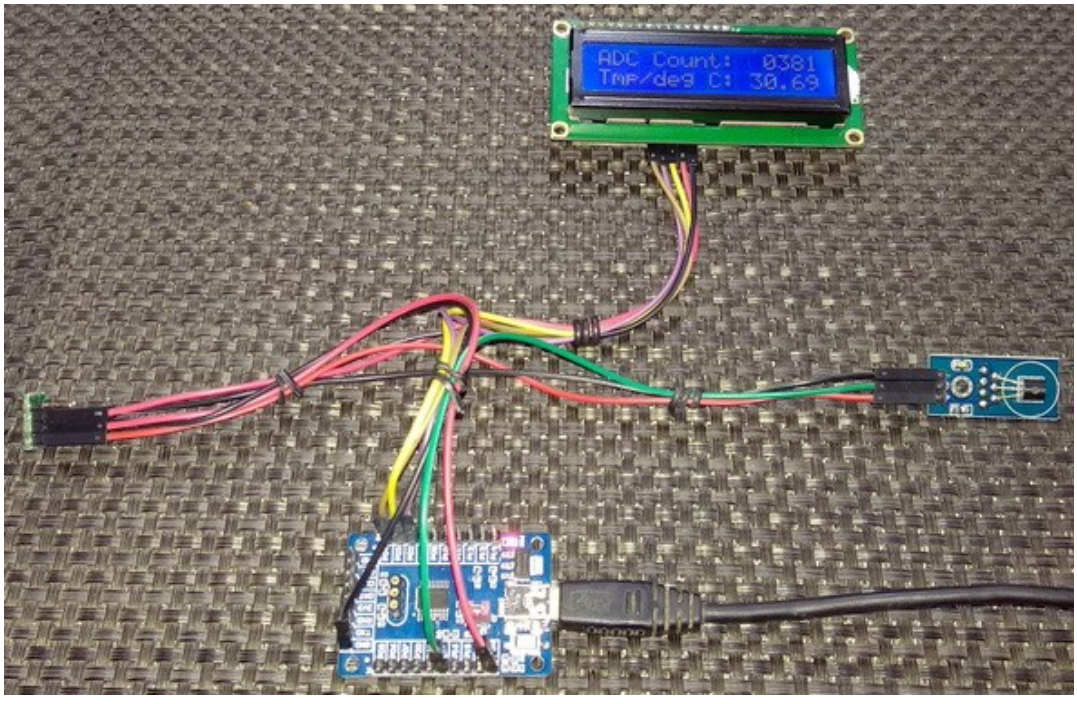
When this function is called, the ADC conversion completion flag is cleared and the ADC is software-triggered. The conversion completion flag is polled. When this flag status is changed, the ADC data registers are read. Finally, the value is returned.

LM35 gives 10mV output for each degree of temperature. Therefore, for 26°C the sensor is supposed to give 260mV output. At 3.3V the ADC count will 4095 while at 0V the ADC count will be 0 count. Thus, 0.806mV equals one count and so 260mV should give:

$$\frac{260}{0.806} = 322 \text{ counts.}$$

In the display, we have to show 26.00°C i.e. 2600 considering no decimal point and fully integer value. Thus, to transform 322 to 2600, we have to divide the result with a scalar (0.12412). The main code just does that after reading the ADC. The ADC count and temperature are then both shown in the LCD.

Demo



Demo video: https://youtu.be/j-8qGBzRxAQ

ADC Interrupt – LDR-based Light Sensor

Like any other interrupts, ADC interrupt is a very interrupt. In the last example we saw polling-based ADC readout. In this segment, we will see how to use interrupt-based method to extract ADC data. The concept of ADC interrupt is simply to notify that an ADC data has been made ready for reading once triggered.

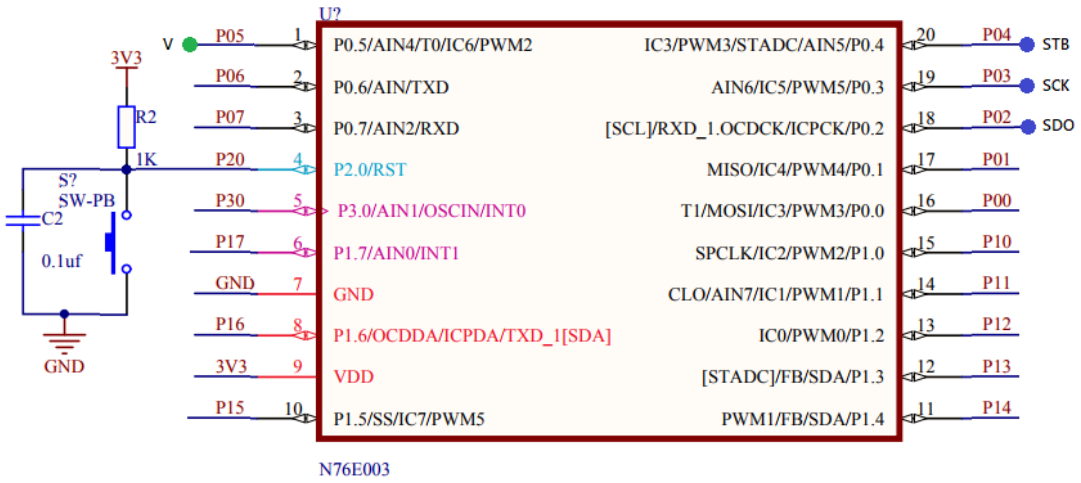
Code

```
#include "N76E003_IAR.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "soft_delay.h"
#include "LCD_2_Wire.h"


//LDR Definitions//

#define LDR_constant                    100000.0
#define R_fixed                          10000.0
#define VDD                                 4095


unsigned int adc_value = 0x0000;


void setup(void);
unsigned int ADC_read(void);
void lcd_print_i(unsigned char x_pos, unsigned char y_pos, unsigned int
value);
unsigned int measure_light_intensity(void);


#pragma vector = 0x5B
__interrupt void ADC_ISR(void)
{
  adc_value = ADC_read();
  clr_ADCF;
}
```

```c
void main(void)
{
  unsigned int lux = 0;

  setup();

  while(1)
  {
    set_ADCS;
    lux = measure_light_intensity();
    lcd_print_i(12, 0, adc_value);
    lcd_print_i(12, 1, lux);
    delay_ms(400);
  }
}


void setup(void)
{
  LCD_init();
  LCD_clear_home();
  LCD_goto(0, 0);
  LCD_putstr("ADC Count:");
  LCD_goto(0, 1);
  LCD_putstr("Lux Value:");

  Enable_ADC_AIN4;
  set_EADC;
  set_EA;
}


unsigned int ADC_read(void)
{
  register unsigned int value = 0x0000;

  value = ADCRH;
  value <<= 4;
  value |= ADCRL;

  return value;
}


void lcd_print_i(unsigned char x_pos, unsigned char y_pos, unsigned int value)
{
  LCD_goto(x_pos, y_pos);
  LCD_putchar((value / 1000) + 0x30);
  LCD_goto((x_pos + 1), y_pos);
  LCD_putchar(((value % 1000) / 100) + 0x30);
  LCD_goto((x_pos + 2), y_pos);
  LCD_putchar(((value % 100) / 10) + 0x30);
  LCD_goto((x_pos + 3), y_pos);
  LCD_putchar((value % 10) + 0x30);
}
```

```
unsigned int measure_light_intensity(void)
{
  float lux = 0;

  lux = adc_value;

  lux = (LDR_constant * ((VDD / (R_fixed * lux))) - 0.1);

  if((lux >= 0) && (lux <= 9999))
  {
    return ((unsigned int)lux);
  }
  else
  {
    return 0;
  }
}
```

Schematic



Explanation

Setting the ADC in interrupt is not much different from the previous example except for the interrupt parts.

```
Enable_ADC_AIN4;
set_EADC;
set_EA;
```

We have to enable both the ADC and global interrupts.

The reading process is also same:

```
unsigned int ADC_read(void)
{
  register unsigned int value = 0x0000;
```

```
  value = ADCRH;
  value <<= 4;
  value |= ADCRL;

  return value;
}
```

The ADC is triggered in the main with the following line of code since we are using software-based triggering:

```
set_ADCS;
```

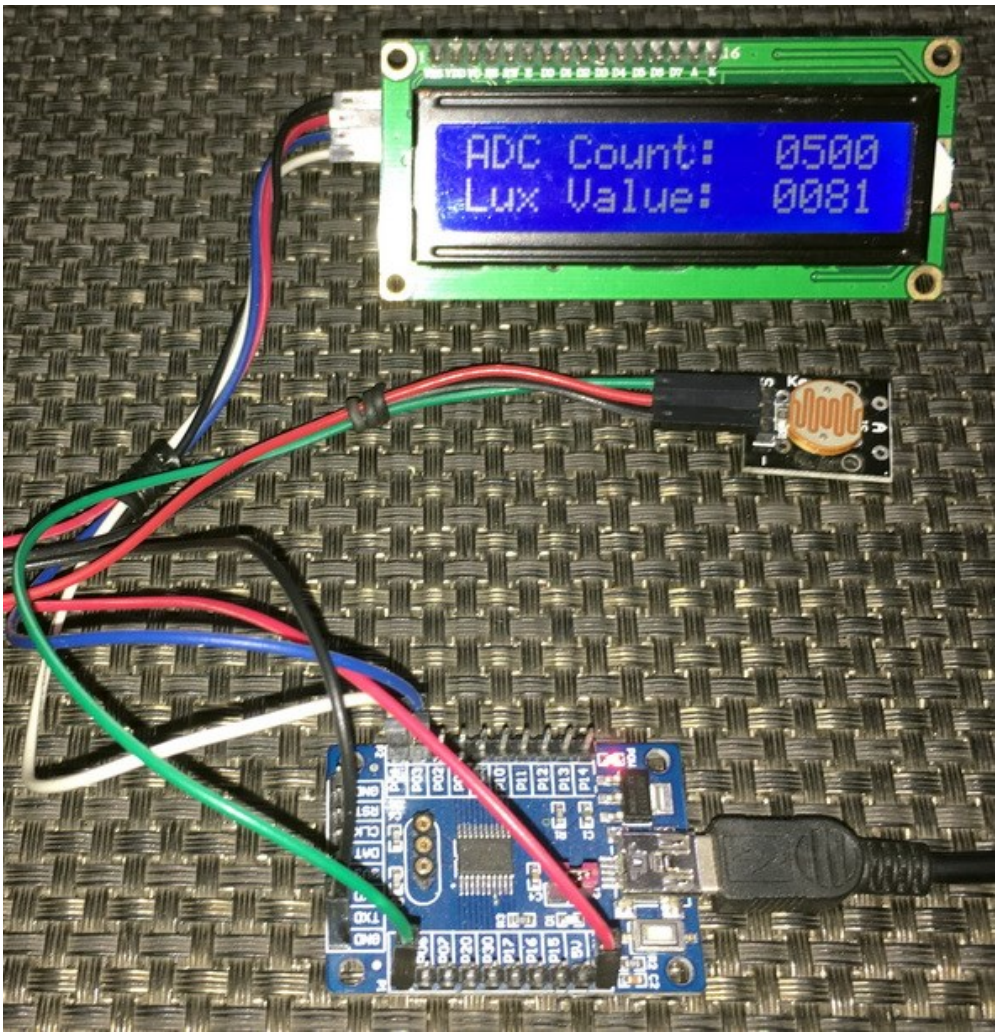
Now instead of reading the ADC in the main by polling, the ADC is read in the interrupt.

```
#pragma vector = 0x5B
__interrupt void ADC_ISR(void)
{
  adc_value = ADC_read();
  clr_ADCF;
}
```

When ADC interrupt occurs, we must clear ADC interrupt flag and read the ADC data registers. In this way, the main code is made free from polling and free for other tasks.

The demo here is a rudimentary LDR-based light intensity meter. Light falling on the LDR changes its resistance. Using voltage divider method, we can back calculate the resistance of the LDR and use this info to measure light intensity.

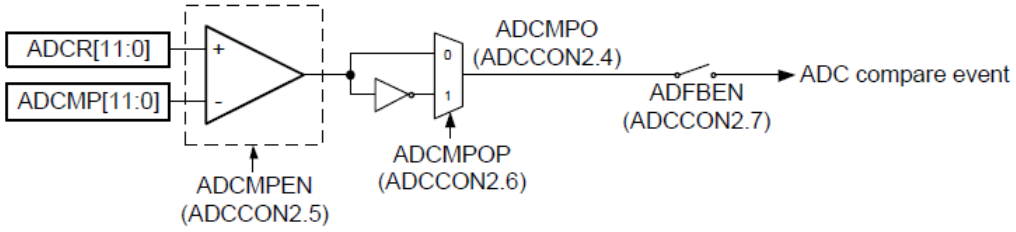
Demo



Demo video: https://youtu.be/3TcjGDZ4koE

ADC Comparator

Many micros are equipped with on-chip analogue comparator. N76E003 has an embedded ADC comparator. The main feature of this comparator is the range it offers. Unlike comparators of other micros in which only a few selectable set points can set, the ADC comparator of N76E003 can be set in any range from 0 count to the max ADC count of 4095. This allows us to easily implement it for many applications like low battery alarm, SMPSs, over voltage sense, overload detection, etc. It must be noted however, it is not a true comparator because a true comparator has nothing to do with ADC.



The comparator block is situated at the output of the ADC and so it is just a single block like the ADC but all eight channels share it. Thus, when it is needed to compare multiple channels, it should be reset and reconfigured.

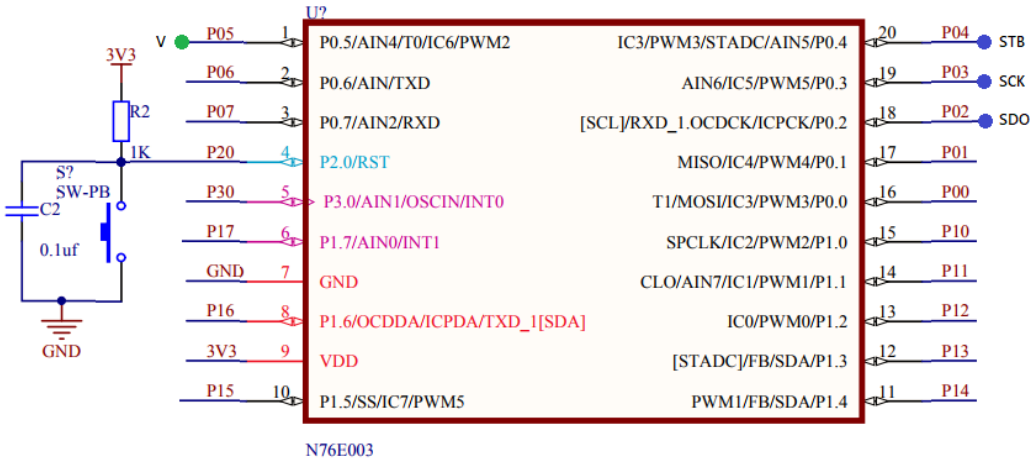
Code

```
#include "N76E003_IAR.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "soft_delay.h"
#include "LCD_2_Wire.h"


unsigned int adc_value = 0x0000;


void setup(void);
unsigned int ADC_read(void);
void set_ADC_comparator_value(unsigned int value);
void lcd_print_i(unsigned char x_pos, unsigned char y_pos, unsigned int
value);


#pragma vector = 0x5B
__interrupt void ADC_ISR(void)
{
  adc_value = ADC_read();
  clr_ADCF;
}


void main(void)
{
  setup();
```

```
  while(1)
  {
    set_ADCS;
    lcd_print_i(12, 0, adc_value);

    LCD_goto(12, 1);

    if((ADCCON2 & 0x10) != 0x00)
    {
      LCD_putstr("HIGH");
      set_P15;
    }
    else
    {
      LCD_putstr(" LOW");
      clr_P15;
    }

    delay_ms(400);
  }
}


void setup(void)
{
  P15_PushPull_Mode;

  LCD_init();
  LCD_clear_home();
  LCD_goto(0, 0);
  LCD_putstr("ADC Count:");
  LCD_goto(0, 1);
  LCD_putstr("Cmp State:");

  Enable_ADC_BandGap;
  Enable_ADC_AIN4;

  set_ADC_comparator_value(1023);
  set_ADCMPEN;

  set_EADC;
  set_EA;
}


unsigned int ADC_read(void)
{
  register unsigned int value = 0x0000;

  value = ADCRH;
  value <<= 4;
  value |= ADCRL;

  return value;
}
```

```
void set_ADC_comparator_value(unsigned int value)
{
  ADCMPH = ((value & 0x0FF0) >> 4);
  ADCMPL = (value & 0x000F);
}


void lcd_print_i(unsigned char x_pos, unsigned char y_pos, unsigned int value)
{
  LCD_goto(x_pos, y_pos);
  LCD_putchar((value / 1000) + 0x30);
  LCD_goto((x_pos + 1), y_pos);
  LCD_putchar(((value % 1000) / 100) + 0x30);
  LCD_goto((x_pos + 2), y_pos);
  LCD_putchar(((value % 100) / 10) + 0x30);
  LCD_goto((x_pos + 3), y_pos);
  LCD_putchar((value % 10) + 0x30);
}
```

Schematic



Explanation

To configure the ADC comparator block, we just need to specify two things – the reference value with which the comparison should to be done and the polarity of comparison. After setting these up, we have to enable the comparator block. These are done as follows:

```
set_ADC_comparator_value(1023);
set_ADCMPEN;
```

Note in the code snippet above, I didn't code anything regarding polarity because by default the polarity is set as such that the comparator's output will change state when the input voltage is greater than or equal to the set ADC count level of 1023.

ADCCON2 – ADC Control 2

7	6	5	4	3	2	1	0
ADFBEN	ADCMPOP	ADCMPEN	ADCMPO	-	-	-	ADCDLY.8
R/W	R/W	R/W	R	-	-	-	R/W

Address: E2H Reset value: 0000 0000b

Bit	Name	Description
7	ADFBEN	**ADC compare result asserting Fault Brake enable** 0 = ADC asserting Fault Brake Disabled. 1 = ADC asserting Fault Brake Enabled. Fault Brake is asserted once its compare result ADCMPO is 1. Meanwhile, PWM channels output Fault Brake data. PWMRUN (PWMCON0.7) will also be automatically cleared by hardware. The PWM output resumes when PWMRUN is set again.
6	ADCMPOP	**ADC comparator output polarity** 0 = ADCMPO is 1 if ADCR[11:0] is greater than or equal to ADCMP[11:0]. 1 = ADCMPO is 1 if ADCR[11:0] is less than ADCMP[11:0].
5	ADCMPEN	**ADC result comparator enable** 0 = ADC result comparator Disabled. 1 = ADC result comparator Enabled.
4	ADCMPO	**ADC comparator output value** This bit is the output value of ADC result comparator based on the setting of ACMPOP. This bit updates after every A/D conversion complete.
3:1	-	**Reserved**
0	ADCDLY.8	**ADC external trigger delay counter bit 8** See ADCDLY register.

Just like ADC reading, we have to take care of the bit positions for the comparator set point. It is coded as follows.

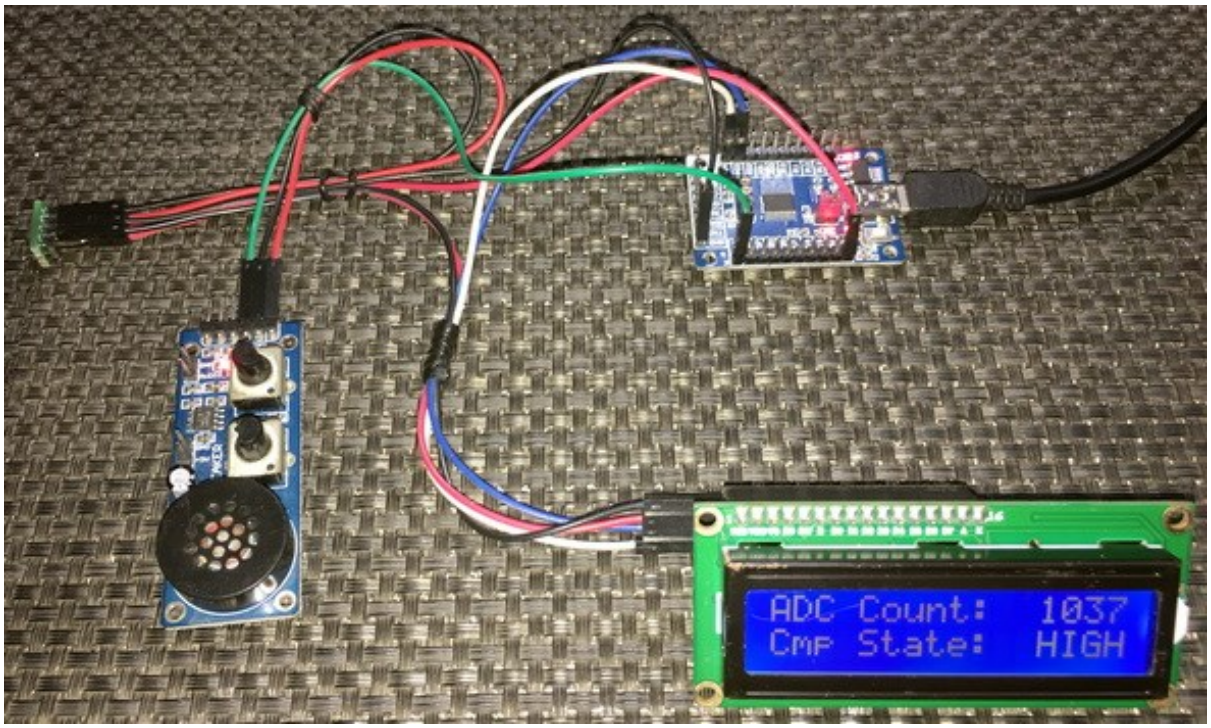
```
void set_ADC_comparator_value(unsigned int value)
{
  ADCMPH = ((value & 0x0FF0) >> 4);
  ADCMPL = (value & 0x000F);
}
```

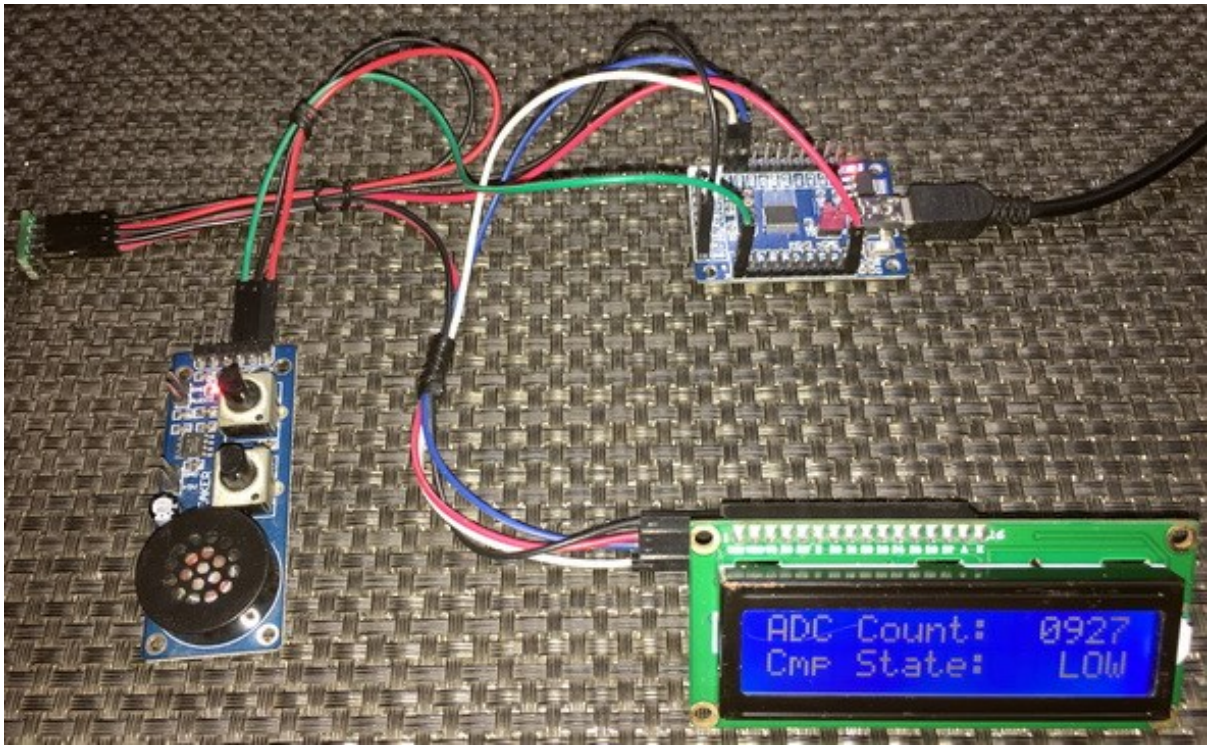
In this demo, I also used the bandgap voltage as reference source:

```
Enable_ADC_BandGap;
```

The rest of the code is similar to the ADC interrupt code.

Demo

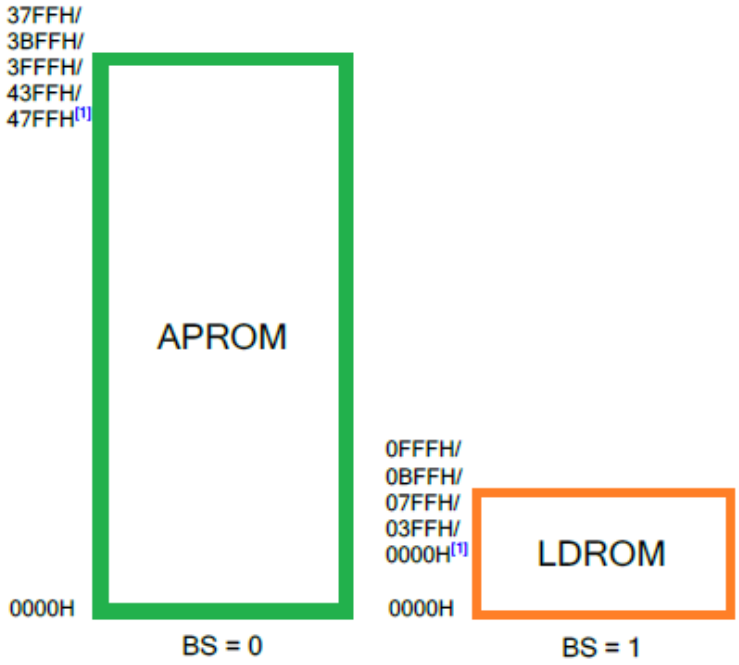




Demo video: https://youtu.be/Nn--zrezoqc

Data Flash – Using APROM as EEPROM

In most standard 8051s, there is no dedicated memory space as EEPROM. This is unlike other microcontrollers of modern era. EEPROM memory is needed for the storage of critical data that need to be retained even in power down state. In N76E003, there are two types of ROM memory. EEPROM-like storage can be achieved by using ***APROM (Application ROM)***. APROM is the actual flash memory where store our application codes. ***User Code Loader ROM*** or ***LDROM*** of N76E003 microcontroller, is another ROM where we can keep a bootloader and configuration codes. Sizes of these ROMs can be varied according to configuration bits.



[1] The logic boundary addresses of APROM and LDROM are defined by CONFIG1[2:0].

Code

Flash.h

```
#define  CID_READ            0x0B
#define  DID_READ            0x0C

#define  PAGE_ERASE_AP       0x22
#define  BYTE_READ_AP        0x00
#define  BYTE_PROGRAM_AP     0x21
#define  PAGE_SIZE           128u

#define  ERASE_FAIL          0x70
#define  PROGRAM_FAIL        0x71
#define  IAPFF_FAIL          0x72
#define  IAP_PASS            0x00


void enable_IAP_mode(void);
void disable_IAP_mode(void);
void trigger_IAP(void);
```

```
unsigned char write_data_to_one_page(unsigned int u16_addr, const unsigned
char *pDat, unsigned char num);
void write_data_flash(unsigned int u16_addr, unsigned char *pDat, unsigned int
num);
void read_data_flash(unsigned int u16_addr, unsigned char *pDat, unsigned int
num);
```

Flash.c

```
#include "N76E003_iar.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "soft_delay.h"
#include "Flash.h"


static unsigned char EA_Save_bit;


void enable_IAP_mode(void)
{
    EA_Save_bit = EA;
    clr_EA;
    TA = 0xAA;
    TA = 0x55;
    CHPCON |= 0x01 ;
    TA = 0xAA;
    TA = 0x55;
    IAPUEN |= 0x01;
    EA = EA_Save_bit;
}


void disable_IAP_mode(void)
{
    EA_Save_bit = EA;
    clr_EA;
    TA = 0xAA;
    TA = 0x55;
    IAPUEN &= ~0x01;
    TA = 0xAA;
    TA = 0x55;
    CHPCON &=~ 0x01;
    EA = EA_Save_bit;
}


void trigger_IAP(void)
{
    EA_Save_bit = EA;
    clr_EA;
    TA = 0xAA;
    TA = 0x55;
    IAPTRG |= 0x01;
```

```
    EA = EA_Save_bit;
}


unsigned char write_data_to_one_page(unsigned int u16_addr, const unsigned
char *pDat, unsigned char num)
{
    unsigned char i = 0;
    unsigned char offset = 0;
    unsigned char __code *pCode;
    unsigned char __xdata *xd_tmp;

    enable_IAP_mode();
    offset = (u16_addr & 0x007F);
    i = (PAGE_SIZE - offset);

    if(num > i)
    {
      num = i;
    }

    pCode = (unsigned char __code *)u16_addr;

    for(i = 0; i < num; i++)
    {
        if(pCode[i] != 0xFF)
        {
          break;
        }
    }

    if(i == num)
    {
        IAPCN = BYTE_PROGRAM_AP;
        IAPAL = u16_addr;
        IAPAH = (u16_addr >> 8);

        for(i = 0; i < num; i++)
        {
          IAPFD = pDat[i];
          trigger_IAP();
          IAPAL++;
        }

        for(i = 0; i < num; i++)
        {
          if(pCode[i] != pDat[i])
          {
                   break;
          }
        }

        if(i != num)
        {
          goto WriteDataToPage20;
        }
```

```
    }

    else
    {
      WriteDataToPage20:
      pCode = (unsigned char __code *)(u16_addr & 0xFF80);
      for(i = 0; i < 128; i++)
      {
           xd_tmp[i] = pCode[i];
      }

      for(i = 0; i < num; i++)
      {
           xd_tmp[offset + i] = pDat[i];
      }

      do
      {
           IAPAL = (u16_addr & 0xFF80);
           IAPAH = (u16_addr >> 8);
           IAPCN = PAGE_ERASE_AP;
           IAPFD = 0xFF;
           trigger_IAP();
           IAPCN =BYTE_PROGRAM_AP;

           for(i = 0; i < 128; i++)
           {
                IAPFD = xd_tmp[i];
                trigger_IAP();
                IAPAL++;
           }

           for(i = 0; i < 128; i++)
           {
                if(pCode[i] != xd_tmp[i])
                {
                     break;
                }
           }
      }while(i != 128);

    }

    disable_IAP_mode();

    return num;
}


void write_data_flash(unsigned int u16_addr, unsigned char *pDat,unsigned int
num)
{
    unsigned int CPageAddr = 0;
    unsigned int EPageAddr = 0;
    unsigned int cnt = 0;
```

```
    CPageAddr = (u16_addr >> 7);
    EPageAddr = ((u16_addr + num) >> 7);

    while(CPageAddr != EPageAddr)
    {
      cnt = write_data_to_one_page(u16_addr, pDat, 128);
      u16_addr += cnt;
      pDat += cnt;
      num -= cnt;
      CPageAddr = (u16_addr >> 7);
    }

    if(num)
    {
      write_data_to_one_page(u16_addr, pDat, num);
    }
}


void read_data_flash(unsigned int u16_addr, unsigned char *pDat, unsigned int
num)
{
    unsigned int i = 0;

    for(i = 0; i < num; i++)
    {
        pDat[i] = *(unsigned char __code *)(u16_addr+i);
    }
}
```

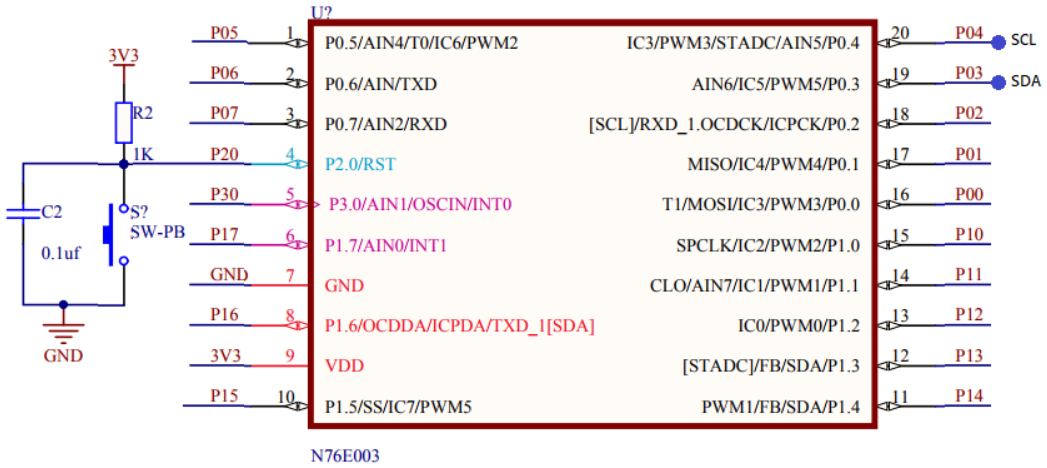
main.c

```
#include "N76E003_IAR.h"
#include "Common.h"
#include "Delay.h"
#include "soft_delay.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "LCD_2_Wire.h"
#include "Flash.h"


#define BASE_ADDRESS        3700


void lcd_print_c(unsigned char x_pos, unsigned char y_pos, unsigned char
value);
void lcd_print_i(unsigned char x_pos, unsigned char y_pos, unsigned int
value);


void main (void)
{
  unsigned char s = 0;
  unsigned char val[1] = {0};
  unsigned char ret_val[1] = {0};
```

```
  P15_PushPull_Mode;

  LCD_init();
  LCD_clear_home();

  clr_P15;
  LCD_goto(0, 0);
  LCD_putstr("R Addr:");
  LCD_goto(0, 1);
  LCD_putstr("R Data:");

  for(s = 0; s <= 9; s++)
  {
    read_data_flash((s + BASE_ADDRESS), ret_val, 1);
    delay_ms(10);
    lcd_print_i(11, 0, (s + BASE_ADDRESS));
    lcd_print_c(13, 1, ret_val[0]);
    delay_ms(600);
  }

  delay_ms(2000);

  set_P15;
  LCD_goto(0, 0);
  LCD_putstr("W Addr:");
  LCD_goto(0, 1);
  LCD_putstr("W Data:");

  for(s = 0; s <= 9; s++)
  {
    val[0] = s;
    write_data_flash((s + BASE_ADDRESS), val, 1);
    delay_ms(10);
    lcd_print_i(11, 0, (s + BASE_ADDRESS));
    lcd_print_c(13, 1, val[0]);
    delay_ms(600);
  }

  while(1)
  {
  };
}


void lcd_print_c(unsigned char x_pos, unsigned char y_pos, unsigned char
value)
{
  LCD_goto(x_pos, y_pos);
  LCD_putchar((value / 100) + 0x30);
  LCD_goto((x_pos + 1), y_pos);
  LCD_putchar(((value % 10) / 10) + 0x30);
  LCD_goto((x_pos + 2), y_pos);
  LCD_putchar((value % 10) + 0x30);
}
```

```
void lcd_print_i(unsigned char x_pos, unsigned char y_pos, unsigned int value)
{
  LCD_goto(x_pos, y_pos);
  LCD_putchar((value / 10000) + 0x30);
  LCD_goto((x_pos + 1), y_pos);
  LCD_putchar(((value % 10000) / 1000) + 0x30);
  LCD_goto((x_pos + 2), y_pos);
  LCD_putchar(((value % 1000) / 100) + 0x30);
  LCD_goto((x_pos + 3), y_pos);
  LCD_putchar(((value % 100) / 10) + 0x30);
  LCD_goto((x_pos + 4), y_pos);
  LCD_putchar((value % 10) + 0x30);
}
```

Schematic



Explanation

To write and read data from flash we can use the following functions:

```
void write_data_flash(unsigned int u16_addr, unsigned char *pDat, unsigned int num);
void read_data_flash(unsigned int u16_addr, unsigned char *pDat, unsigned int num);
```

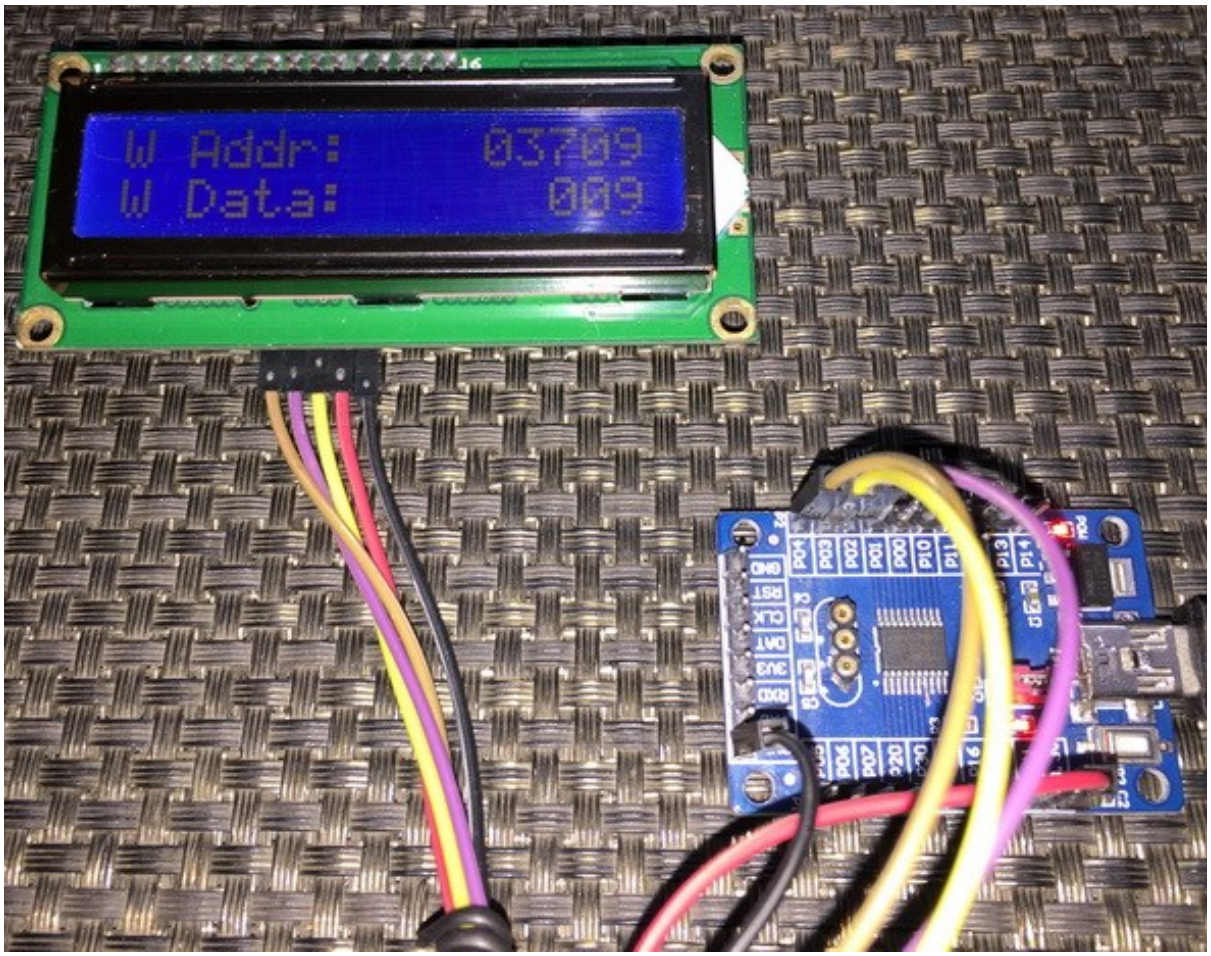
Both of these functions are pointer-based functions. The first parameter of these function is the physical location of data, the second argument is the data pointer itself and the last argument is the number of bytes to read/write. Try to use the upper addresses of the flash where application code usually doesn't reside. Reading the flash doesn't require any involvement of IAP while writing does require IAP. The functions are self-explanatory and can be used readily without any changes since they are provided in the BSP examples.

In the demo, ten data bytes are saved in ten separate locations starting from location 3700 to 3709. Prior to that these locations are read. On first start up, these locations have no saved data and so they show up garbage values (205/255 usually). When a valid user data is saved, the location is updated with it. When reset or powered down, the newly written data bytes are retained and reread when powered up again.

Try not to frequently write on the flash since flash memories wear on frequent writes. Use a RAM-based buffer and try to write one page of flash when the buffer is full. In this way, flash memory wear-and-tear is slightly reduced.

Be careful about saving locations. Don't use locations that hold application code. Try to use unoccupied empty flash locations or upper flash addresses. Use the ***NuMicro ISP Programming Tool*** for finding empty locations.

Demo



Demo video: https://youtu.be/b2iH-DCTfqU

Overview of N76E003 Timers

Since N76E003 is based on 8051 architecture, many of its hardware features will have similarities with a standard 8051 microcontroller. The timers of N76E003 have such similarities but keep in mind that similarities don't mean same. Strictly speaking in N76E003, there six timers and these are:

- Timer 0 and Timer 1

  These are both 16-bit general purpose timers with four modes of operation. With these modes we can allow the timers to operate as general purpose 8-bit auto-reloading timers, 13-bit and 16-bit timers. Most commonly the 16-bit timer mode (Mode 1) is used as it is the simplest mode available. Additionally, we can take outputs of these timers. Unlike typical 8051s, we can use chose from ***Fsys***, fixed and prescaled ***Fsys*** *(Fsys / 12)* or external input as timer clock sources.

- Timer 2

  Timer 2 is different from Timers 0 and 1 in many aspects. Mainly it is used for waveform captures. It can also be used in compare mode to generate timed events or compare-match-based PWM. It has auto-reloading feature unlike Timers 0 and 1.

- Timer 3

  Timer 3 is an auto-reloading 16-bit timer with no output and ***Fsys*** as clock source. Like Timer 1, Timer 3 can be used for serial communication hardware (UART).

- Self-Wake Up Timer (WKT)

  The self-wake-up timer or simply wake-up timer is a special event timer that is not available in standard 8051s. The purpose of this timer is to wake-up or restore the normal working of a N76E003 chip from low power modes after a certain period of time. It can also be used to trigger timed events or trigger other hardware.

- Watchdog Timer

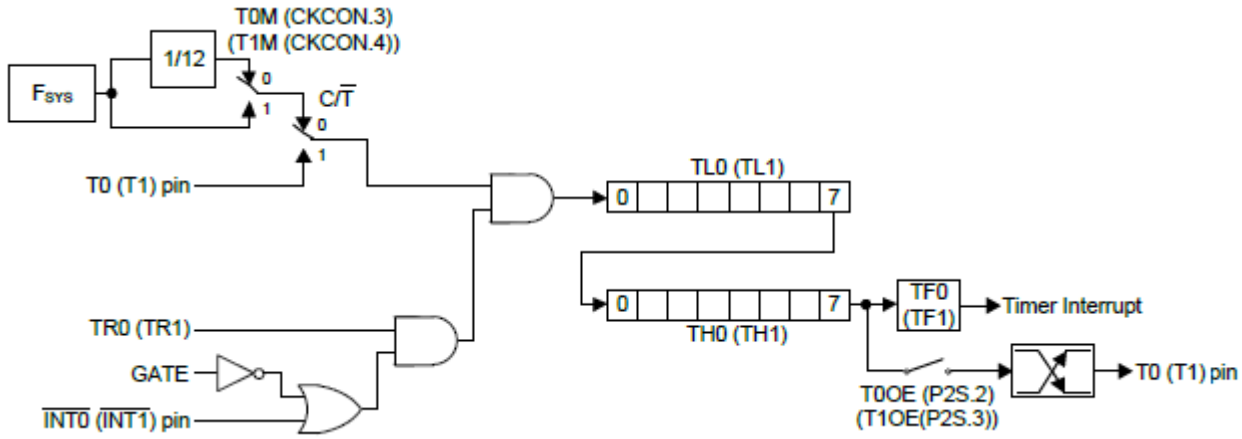
  The watchdog timer of N76E003 can be used either as a 6-bit general purpose timer or as a standard watchdog timer. Most commonly it is not used as a general-purpose timer because that is not what it is meant to be. In watchdog mode, it helps in recovering a stuck N76E003 micro by resetting it.

While using BSPs be careful about timer-based delay routines and UART routines. These built-in routines reset and reconfigure timers according to their purposes. Make sure that there is no conflicting issue when using them. To avoid such conflict, either use software delay routines or one dedicate timer for a particular job. For instance, use Timer 1 for delays and Timer 3 for UART. You have to know what you are doing and with which hardware.

Timer 0 – Time base Generation

Time-base generation is one of the most basic function of a timer. By using time-bases, we can avoid software delays and unwanted loops when we need to wait or check something after a fixed amount of time. BSP-based delay functions are based on this principle.

Here in this section, we will see how to create a time-base to toggle P15 LED.



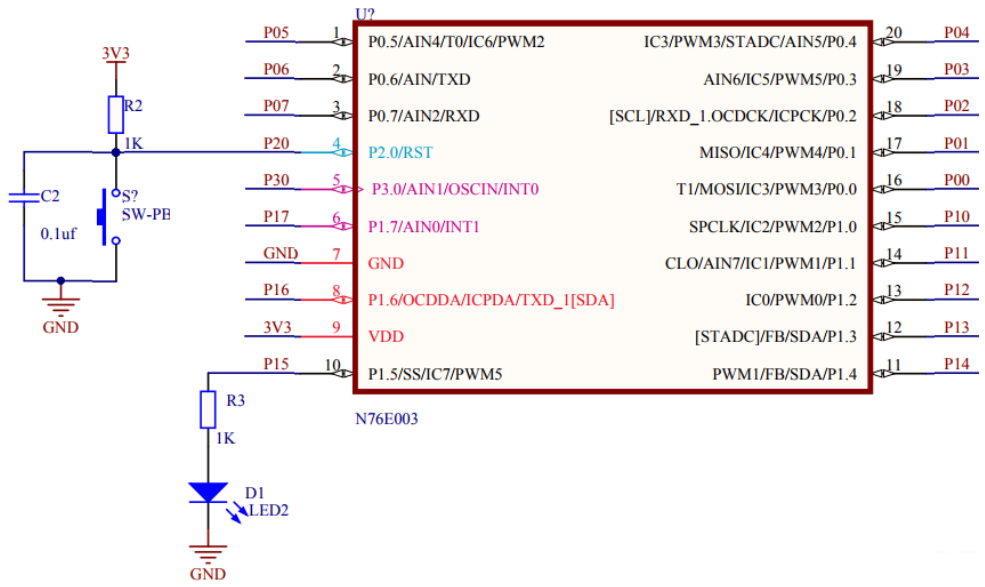
Code

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"


#define HIRC    0
#define LIRC    1
#define ECLK    2


void setup(void);
void set_clock_source(unsigned char clock_source);
void disable_clock_source(unsigned char clock_source);
void set_clock_division_factor(unsigned char value);
void set_Timer_0(unsigned int value);
unsigned int get_Timer_0(void);


void main(void)
{
  setup();

  while(1)
  {
    if(get_Timer_0() < 32767)
    {
      P15 = 1;
```

```
    }
    else
    {
      P15 = 0;
    }
  };
}


void setup(void)
{
  disable_clock_source(ECLK);
  set_clock_source(HIRC);
  set_clock_division_factor(80);

  P15_PushPull_Mode;

  set_T0M;
  TIMER0_MODE1_ENABLE;
  set_Timer_0(0);
  set_TR0;
}


void set_clock_source(unsigned char clock_source)
{
switch(clock_source)
{
    case LIRC:
    {
      set_OSC1;
      clr_OSC0;

      break;
    }

    case ECLK:
    {
      set_EXTEN1;
      set_EXTEN0;

      while((CKSWT & SET_BIT3) == 0);

      clr_OSC1;
      set_OSC0;

      break;
    }

    default:
    {
      set_HIRCEN;

      while((CKSWT & SET_BIT5) == 0);

      clr_OSC1;
```

```
      clr_OSC0;

      break;
    }
  }

  while((CKEN & SET_BIT0) == 1);
}


void disable_clock_source(unsigned char clock_source)
{
  switch(clock_source)
  {
    case HIRC:
    {
      clr_HIRCEN;
      break;
    }

    default:
    {
      clr_EXTEN1;
      clr_EXTEN0;
      break;
    }
  }
}


void set_clock_division_factor(unsigned char value)
{
  CKDIV = value;
}


void set_Timer_0(unsigned int value)
{
  TH0 = ((value && 0xFF00) >> 8);
  TL0 = (value & 0x00FF);
}


unsigned int get_Timer_0(void)
{
  unsigned int value = 0x0000;

  value = TH0;
  value <<= 8;
  value |= TL0;

  return value;
}
```

Schematic



Explanation

For demoing time-base generation, we don't need any additional hardware since we will just be blinking P15 LED without software-based delays.

Before I start explaining the code example, I would like to discuss how Timer 0 and 1 works in Mode 1. In Mode 1, these timers behave like up-counting 16-bit timers. This means they can count (contents of TL and TH registers) from 0 to 65535 with every pulse/clock tick. There is no down-counting feature in these timers.

The source of these pulse/clock ticks can be from system clock, prescaled system clock or from external inputs. When external inputs are used, these timers can be used like counters. Unlike conventional 8051, there are options to take output from timers. The outputs change state with overflows/rollovers, i.e. when the timer counts resets from 65535 to 0.
In the setup function, we do a couple of things. First, we setup the system clock source to 100kHz with HIRC as clock source. P15 is set as an output. Timer 0 is used and so there a 0 in timer setups. Timer 0 is clocked with system clock, i.e. 100kHz and so one tick is:

$$\text{Timer Tick} = \frac{1}{Fsys} = \frac{1}{100kHz} = 10\mu s$$

In Mode 1, the timer will reset/overflow after:

$$65536 \times 10\mu s \approx 655ms$$

provided that the initial count of the timer was set to 0. This is what we are doing in the set up. After setting the timer's count and the mode of operation, the timer is started.

```
disable_clock_source(ECLK);
set_clock_source(HIRC);
set_clock_division_factor(80);

P15_PushPull_Mode;

set_T0M;
TIMER0_MODE1_ENABLE;
set_Timer_0(0);
set_TR0;
```

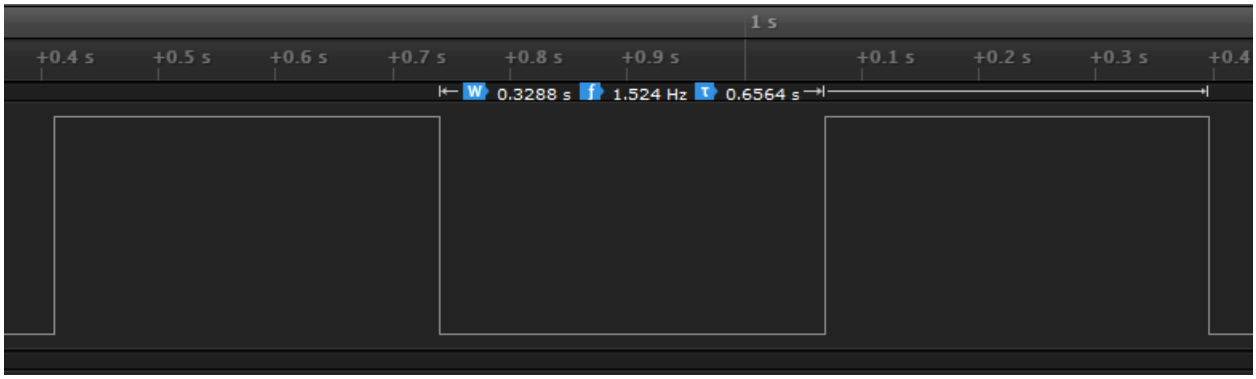
Functions ***set_Timer_0*** and ***get_Timer_0*** writes Timer 0's count and reads Timer 0's count respectively.

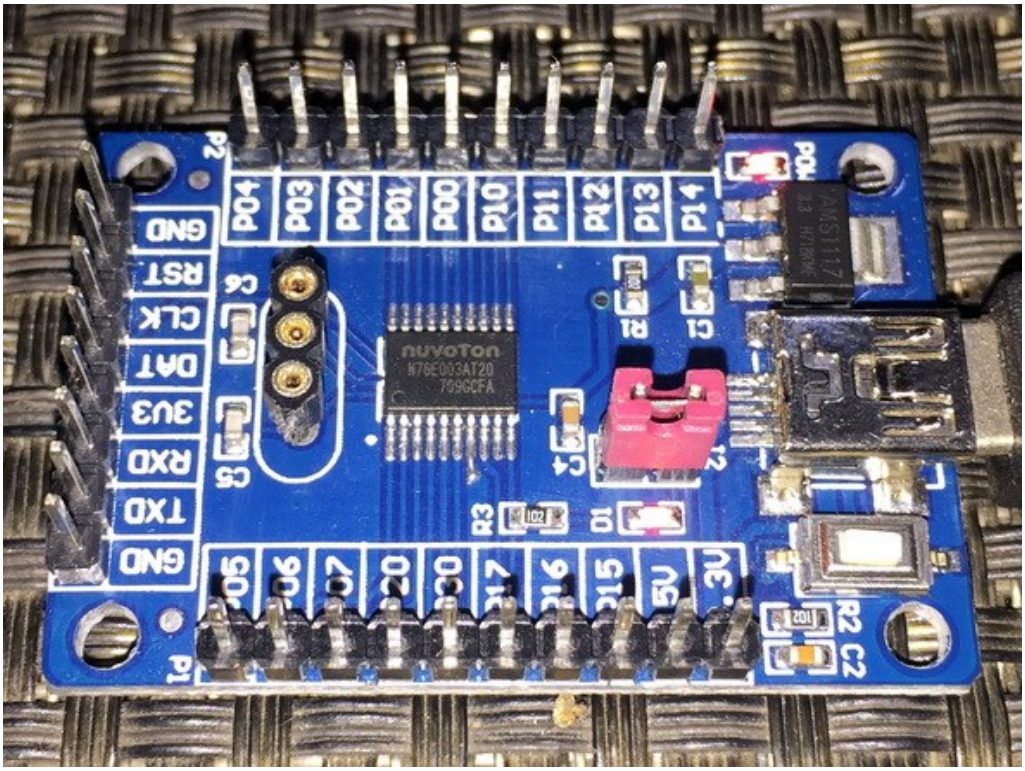
```
void set_Timer_0(unsigned int value);
unsigned int get_Timer_0(void);
```

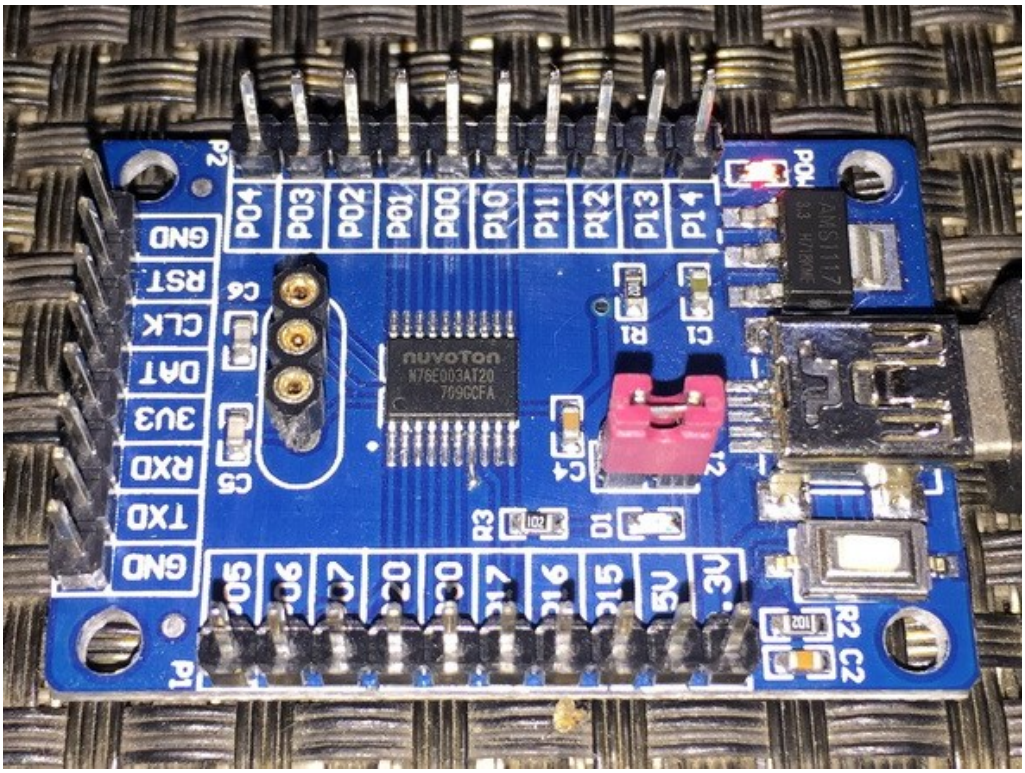
It is needed to toggle the state of the P15 LED and yeah, that is done at every half count of the timer, i.e. at every 327ms. In the main, the timer's count is checked using ***get_Timer_0*** function. When the timer's count is less than 32767 counts, P15 LED is held high. P15 LED is set low when the timer's count is greater than this 32767. In this way the toggling effect is achieved without software delays. However, the timer's count is frequently polled.

```
if(get_Timer_0() < 32767)
{
   P15 = 1;
}
else
{
   P15 = 0;
}
```

Demo



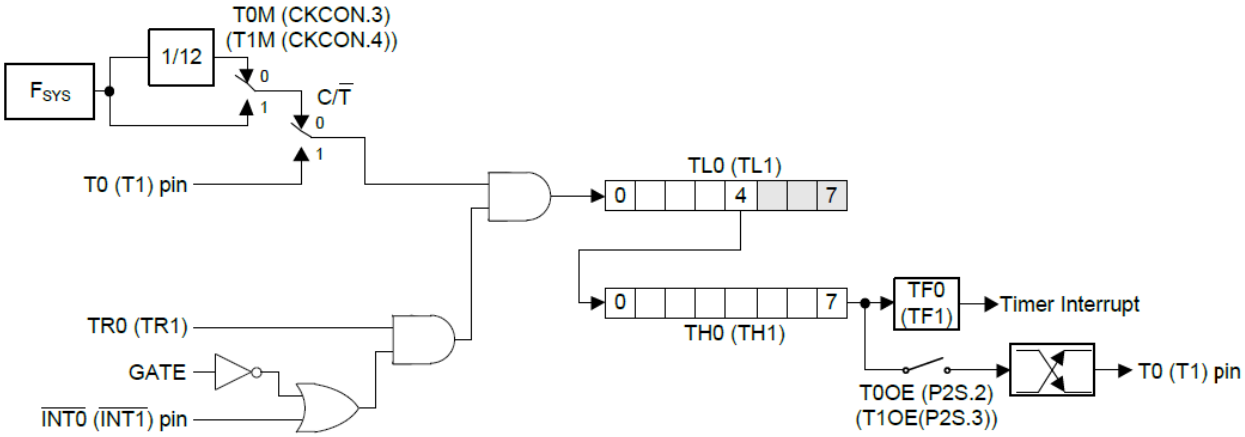




Demo video: https://youtu.be/IJvFhdL24B8

Timer 1 – Stopwatch

We have already seen the operation of general-purpose timers in Mode 1. In this segment, we will how to use these timers in Mode 0. Everything is same between Mode 0 and 1. However, the timer counts or resolutions vary in these modes. Mode 0 makes general purpose timers behave as 13-bit timer/counters.



Check the block diagram shown above and compare with the one previously shown. Note the greyed-out part in this diagram. The upper bits of TL registers are not used in this mode. Between Mode 0 and 1, personally Mode 1 is easier than Mode 0.

Here Timer 1 in Mode 0 is used to create a digital stopwatch.

Code

```
#include "N76E003_iar.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "LCD_3_Wire.h"


#define HIRC    0
#define LIRC    1
#define ECLK    2


unsigned char toggle = 0;
unsigned int ms = 0;

unsigned char sec = 0;
unsigned char min = 0;
unsigned char hrs = 0;


void setup(void);
```

```
void set_clock_source(unsigned char clock_source);
void disable_clock_source(unsigned char clock_source);
void set_clock_division_factor(unsigned char value);
void set_Timer_1_for_Mode_0(unsigned int value);
unsigned int get_Timer_1_for_Mode_0(void);
void print_C(unsigned char x_pos, unsigned char y_pos, unsigned char value);
void print_I(unsigned char x_pos, unsigned char y_pos, unsigned int value);


#pragma vector = 0x1B
__interrupt void Timer1_ISR (void)
{
    set_Timer_1_for_Mode_0(0x1D64);

    ms++;

    if(ms == 499)
    {
      toggle = ~toggle;
    }

    if(ms > 999)
    {
      ms = 0;
      sec++;

      if(sec > 59)
      {
        sec = 0;
        min++;

        if(min > 59)
        {
          min = 0;
          hrs++;

          if(hrs > 23)
          {
            hrs = 0;
          }
        }
      }
    }
}


void main(void)
{
  static char txt[] = {"Nu Stopwatch"};

  setup();

  LCD_goto(2, 0);
  LCD_putstr(txt);

  while(1)
```

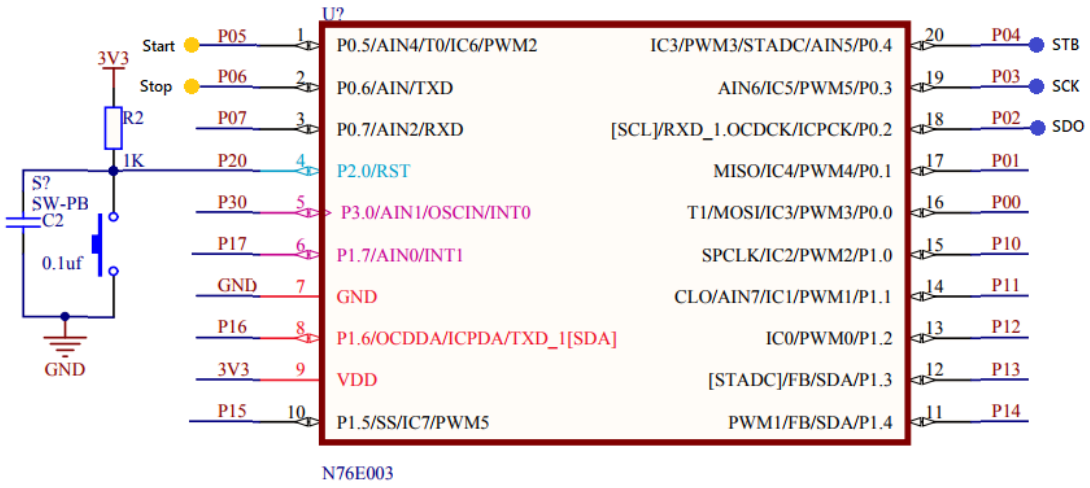
```
{
  if(P05 == 1)
  {
    set_ET1;
    set_EA;
    set_TR1;
    set_Timer_1_for_Mode_0(0x1D64);
  }

  if(P06 == 1)
  {
    clr_ET1;
    clr_EA;
    clr_TR1;
    toggle = 0;
  }

  if((P05 == 1) && (P06 == 1))
  {
    clr_ET1;
    clr_EA;
    clr_TR1;

    ms = 0;
    sec = 0;
    min = 0;
    hrs = 0;

    toggle = 0;
    set_Timer_1_for_Mode_0(0x1D64);
  }


  print_C(2, 1, hrs);
  print_C(5, 1, min);
  print_C(8, 1, sec);
  print_I(11, 2, ms);

  if(!toggle)
  {
     LCD_goto(4, 1);
     LCD_putchar(':');
     LCD_goto(7, 1);
     LCD_putchar(':');
     LCD_goto(10, 1);
     LCD_putchar(':');
  }
  else
  {
     LCD_goto(4, 1);
     LCD_putchar(' ');
     LCD_goto(7, 1);
     LCD_putchar(' ');
     LCD_goto(10, 1);
     LCD_putchar(' ');
  }
```

```
  };
}


void setup(void)
{
  disable_clock_source(ECLK);
  set_clock_source(HIRC);
  set_clock_division_factor(1);

  P05_Input_Mode;
  P06_Input_Mode;

  clr_T1M;
  TIMER1_MODE0_ENABLE;
  set_Timer_1_for_Mode_0(0x1D64);

  LCD_init();
  LCD_clear_home();
}


void set_clock_source(unsigned char clock_source)
{
switch(clock_source)
{
    case LIRC:
    {
      set_OSC1;
      clr_OSC0;

      break;
    }

    case ECLK:
    {
      set_EXTEN1;
      set_EXTEN0;

      while((CKSWT & SET_BIT3) == 0);

      clr_OSC1;
      set_OSC0;

      break;
    }

    default:
    {
      set_HIRCEN;

      while((CKSWT & SET_BIT5) == 0);

      clr_OSC1;
      clr_OSC0;
```

```
      break;
    }
  }

  while((CKEN & SET_BIT0) == 1);
}


void disable_clock_source(unsigned char clock_source)
{
  switch(clock_source)
  {
    case HIRC:
    {
      clr_HIRCEN;
      break;
    }

    default:
    {
      clr_EXTEN1;
      clr_EXTEN0;
      break;
    }
  }
}


void set_clock_division_factor(unsigned char value)
{
  CKDIV = value;
}


void set_Timer_1_for_Mode_0(unsigned int value)
{
  TL1 = (value & 0x1F);
  TH1 = ((value & 0xFFE0) >> 5);
}


unsigned int get_Timer_1_for_Mode_0(void)
{
  unsigned char hb = 0x00;
  unsigned char lb = 0x00;
  unsigned int value = 0x0000;


  value = TH1;
  value <<= 8;
  value |= TL1;

  lb = (value & 0x001F);
  hb = ((value & 0xFFE0) >> 5);

  value = hb;
```

```
  value <<= 8;
  value |= lb;

  return value;
}


void print_C(unsigned char x_pos, unsigned char y_pos, unsigned char value)
{
  LCD_goto(x_pos, y_pos);
  LCD_putchar((value / 10) + 0x30);
  LCD_goto((x_pos + 1), y_pos);
  LCD_putchar((value % 10) + 0x30);
}


void print_I(unsigned char x_pos, unsigned char y_pos, unsigned int value)
{
  LCD_goto(x_pos, y_pos);
  LCD_putchar((value / 100) + 0x30);
  LCD_goto((x_pos + 1), y_pos);
  LCD_putchar(((value % 100) / 10) + 0x30);
  LCD_goto((x_pos + 2), y_pos);
  LCD_putchar((value % 10) + 0x30);
}
```

Schematic



Explanation

Firstly, let us inspect the setup code:

```
disable_clock_source(ECLK);
set_clock_source(HIRC);
set_clock_division_factor(1);

P05_Input_Mode;
P06_Input_Mode;
```

```
clr_T1M;
TIMER1_MODE0_ENABLE;
set_Timer_1_for_Mode_0(0x1D64);
```

The system clock is set to 8MHz using HIRC. As per schematic and code, two buttons labelled ***Start*** and ***Stop*** are connected with P05 and P06 respectively. Timer 1 is set up in Mode 1 with prescaled system clock source (Fsys / 12). Thus, it has an input clock frequency of 666.67kHz. Thus, one timer tick is 1.5µs. To get the timer overflow and reset every 1 millisecond we will need:

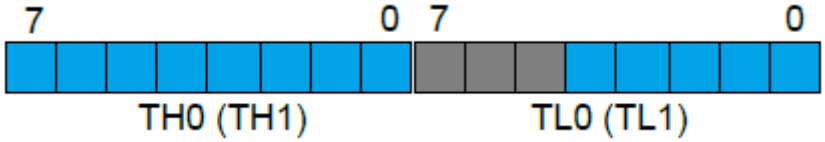
$$\frac{1ms}{1.5\mu s} \approx 667 \text{ counts}$$

Since the timer is an up-counting timer and has a resolution of 13-bits, it will overflow after 8191 counts. Thus, to get 667 counts, we need to set is at:

$$(8191 - 667) = 7524 \text{ or } 0x1D64 \text{ counts}$$

The timer is, therefore, set at this count value.
Now how do we set the timer in Mode 1?

Check the register diagram below. The grey part is what is ignored. Remember that TL and TH are 16-bits as a whole but out of that we will just be using 13-bits.



In our case, the count 0x1D64 is segmented as the following in binary:

$$0001\ 1101\ 0110\ 0100$$

This number is masked as follows:

All of these is done by the following function:

```
void set_Timer_1_for_Mode_0(unsigned int value)
{
  TL1 = (value & 0x1F);
  TH1 = ((value & 0xFFE0) >> 5);
}
```

Reading the time is doing just the opposite of writing and this is accomplished by the following function:

```
unsigned int get_Timer_1_for_Mode_0(void)
{
  unsigned char hb = 0x00;
  unsigned char lb = 0x00;
  unsigned int value = 0x0000;


  value = TH1;
  value <<= 8;
  value |= TL1;

  lb = (value & 0x001F);
  hb = ((value & 0xFFE0) >> 5);

  value = hb;
  value <<= 8;
  value |= lb;

  return value;
}
```

Unlike the previous example, here interrupt is used to keep time. The timer run bit, global interrupt and Timer 1 interrupt are all enabled when the start button is pressed and are all disabled when the stop button is pressed or the stopwatch is reset by pressing both buttons.

```
if(P05 == 1)
{
   set_ET1;
   set_EA;
   set_TR1;
   set_Timer_1_for_Mode_0(0x1D64);
}

if(P06 == 1)
{
   clr_ET1;
   clr_EA;
   clr_TR1;
   toggle = 0;
}

if((P05 == 1) && (P06 == 1))
{
   clr_ET1;
   clr_EA;
   clr_TR1;

   ms = 0;
   sec = 0;
   min = 0;
   hrs = 0;
```

```
   toggle = 0;
   set_Timer_1_for_Mode_0(0x1D64);
}
```

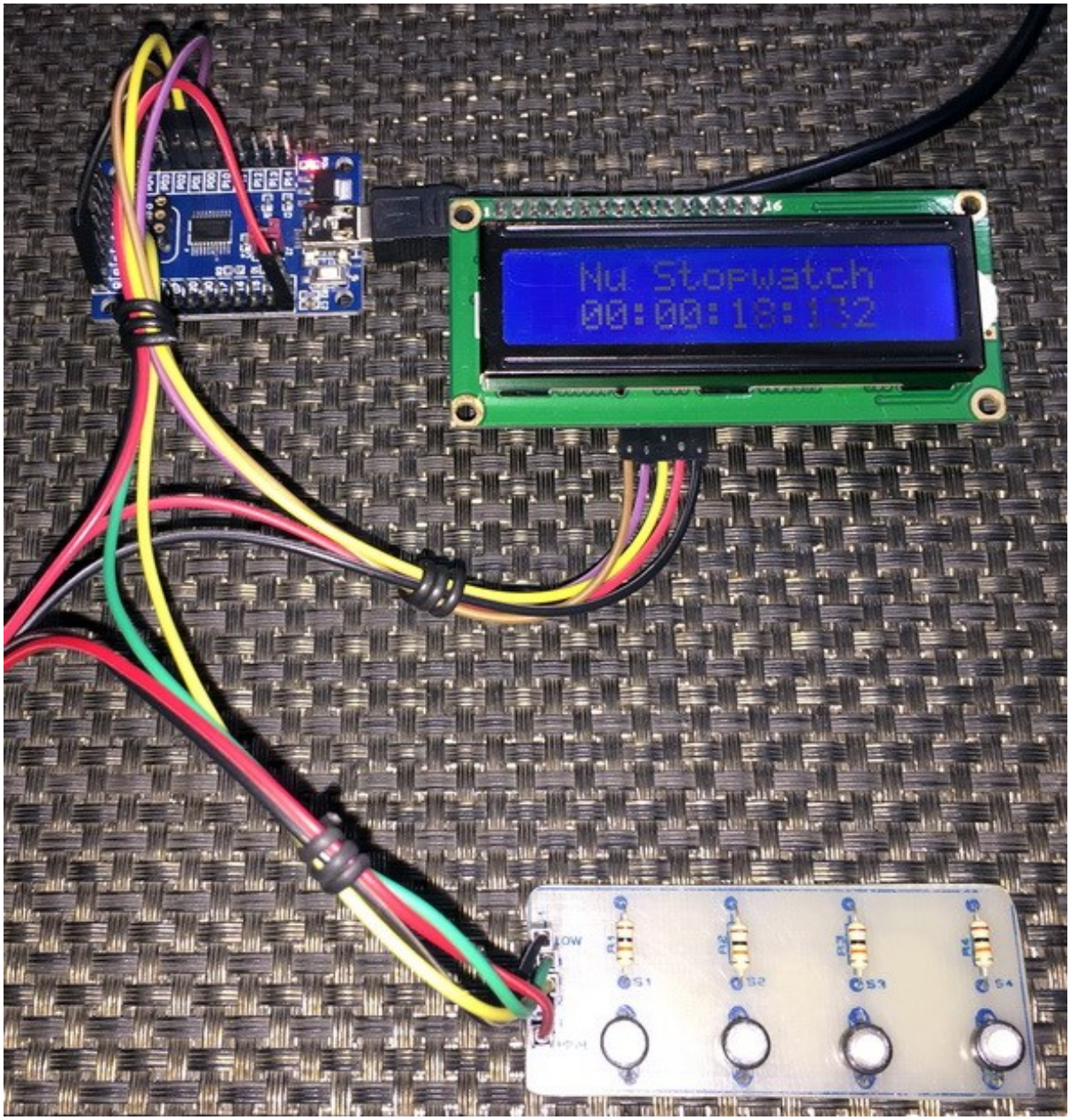
Inside the interrupt subroutine, the time keeping is done:

```
#pragma vector = 0x1B
__interrupt void Timer1_ISR (void)
{
    set_Timer_1_for_Mode_0(0x1D64);

    ms++;

    if(ms == 499)
    {
      toggle = ~toggle;
    }

    if(ms > 999)
    {
      ms = 0;
      sec++;

      if(sec > 59)
      {
        sec = 0;
        min++;

        if(min > 59)
        {
          min = 0;
          hrs++;

          if(hrs > 23)
          {
            hrs = 0;
          }
        }
      }
}
```

At every timer overflow interrupt the timer's counter is reset to 0x1D64 to make sure that there are 667 counts before the overflow.

The time data is displayed on an LCD screen.

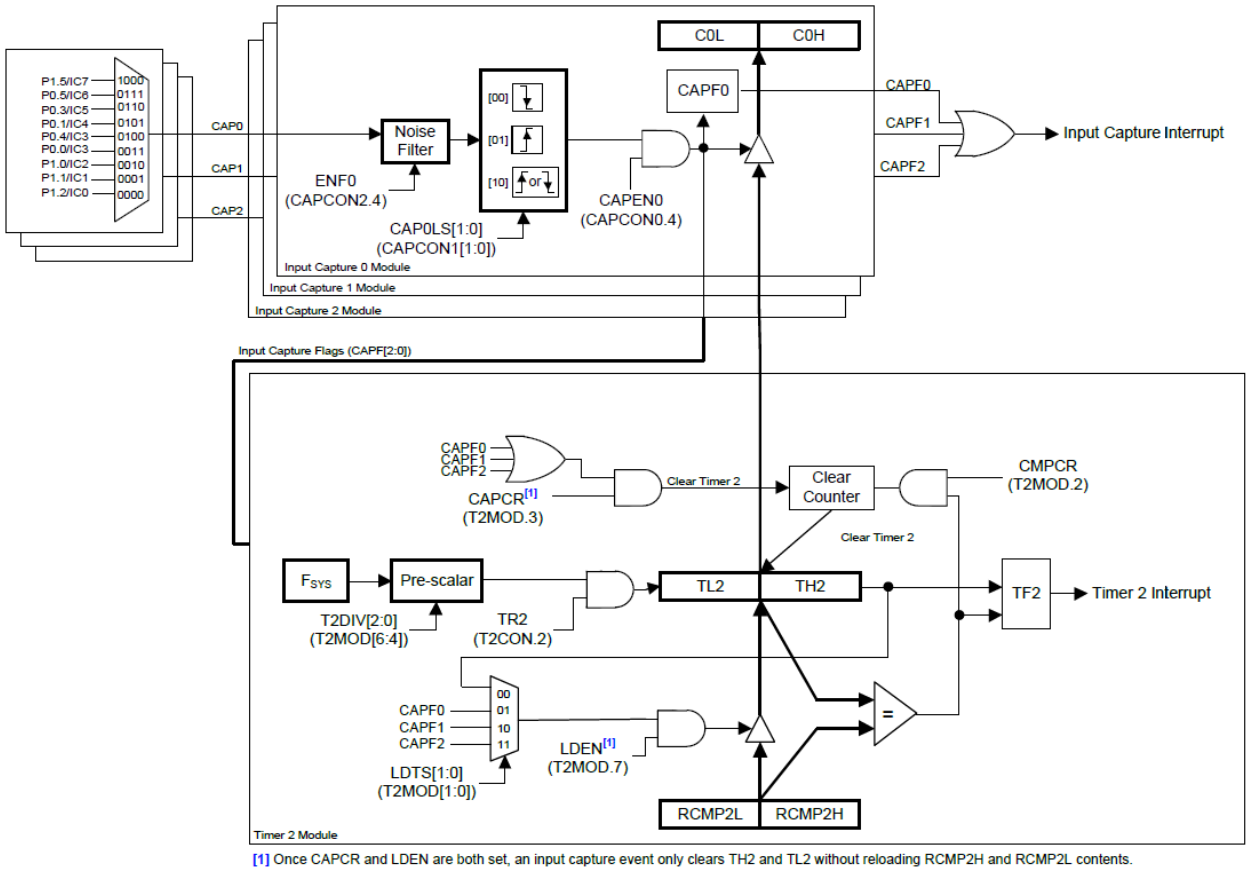
Demo



Demo video: https://youtu.be/egzuPC8Yox4

Timer 2 Input Capture – Frequency Counter

Timer 2, as I stated before, is different in many areas from the other timers of N76E003. In my opinion, it is best suited for measuring pulse widths, their periods and thereby frequencies. In other words, it is the best wave capturing tool that we have in N76E003s.



[1] Once CAPCR and LDEN are both set, an input capture event only clears TH2 and TL2 without reloading RCMP2H and RCMP2L contents.

Unlike conventional 8051s, there varieties of option to capture wave or generate compare-match events.

In this segment, we will see how to make a frequency counter with Timer 2's capture facility. We will also see how to generate Timer 1's output.

Code

```
#include "N76E003_iar.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "LCD_3_Wire.h"


#define HIRC    0
#define LIRC    1
#define ECLK    2
```

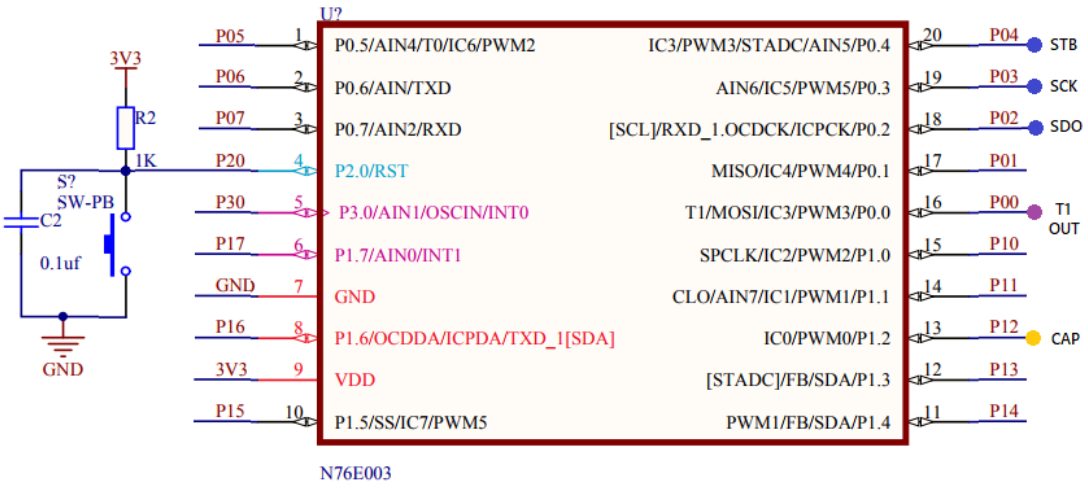
```
#define timer_clock_speed  8000000.0


unsigned long overflow = 0;
unsigned long pulse_time = 0;
unsigned long start_time = 0;
unsigned long end_time = 0;


void setup(void);
void set_clock_source(unsigned char clock_source);
void disable_clock_source(unsigned char clock_source);
void set_clock_division_factor(unsigned char value);
void setup_GPIOs(void);
void setup_Timer_1(void);
void setup_Timer_2(void);
void setup_capture(void);
void set_Timer_1(unsigned int value);
void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned long value);


#pragma vector = 0x1B
__interrupt void Timer_1_ISR(void)
{
  set_Timer_1(0);
}


#pragma vector = 0x2B
__interrupt void Timer_2_ISR(void)
{
  clr_TF2;
  overflow++;
}


#pragma vector = 0x63
__interrupt void Input_Capture_ISR(void)
{
  if((CAPCON0 & SET_BIT0) != 0)
  {
    clr_CAPF0;
    end_time = C0H;
    end_time <<= 8;
    end_time |= C0L;
    pulse_time = ((overflow << 16) - start_time + end_time);
    start_time = end_time;
    overflow = 0;
  }
}


void main(void)
{
  register float f = 0.0;
```

```
  setup();

  LCD_init();
  LCD_clear_home();
  LCD_goto(1, 0);
  LCD_putstr("Nu Freq. Meter");
  LCD_goto(0, 1);
  LCD_putstr("Freq./Hz:");

  while(1)
  {
    f = (timer_clock_speed / ((float)pulse_time));
    lcd_print(11, 1, ((unsigned long)f));
    Timer0_Delay1ms(100);
  };
}


void setup(void)
{
  disable_clock_source(ECLK);
  set_clock_source(HIRC);
  set_clock_division_factor(1);
  setup_GPIOs();
  setup_capture();
  setup_Timer_1();
  setup_Timer_2();
  set_EA;
}


void set_clock_source(unsigned char clock_source)
{
  switch(clock_source)
  {
    case LIRC:
    {
      set_OSC1;
      clr_OSC0;

      break;
    }

    case ECLK:
    {
      set_EXTEN1;
      set_EXTEN0;

      while((CKSWT & SET_BIT3) == 0);

      clr_OSC1;
      set_OSC0;

      break;
    }
```

```
    default:
    {
      set_HIRCEN;

      while((CKSWT & SET_BIT5) == 0);

      clr_OSC1;
      clr_OSC0;

      break;
    }
  }

  while((CKEN & SET_BIT0) == 1);
}


void disable_clock_source(unsigned char clock_source)
{
  switch(clock_source)
  {
    case HIRC:
    {
       clr_HIRCEN;
       break;
    }

    default:
    {
       clr_EXTEN1;
       clr_EXTEN0;
        break;
    }
  }
}


void set_clock_division_factor(unsigned char value)
{
  CKDIV = value;
}


void setup_GPIOs(void)
{
  P00_PushPull_Mode;
  P12_Input_Mode;
}


void setup_Timer_1(void)
{
  set_T1M;
  TIMER1_MODE1_ENABLE;
  set_Timer_1(0);
```

```
  P2S |= SET_BIT3;
  set_TR1;
  set_ET1;
}


void setup_Timer_2(void)
{
  T2CON &= ~SET_BIT0;
  T2MOD = 0x00;
  set_TR2;
  set_ET2;
}


void setup_capture(void)
{
  IC0_P12_CAP0_FallingEdge_Capture;
  set_ECAP;
}


void set_Timer_1(unsigned int value)
{
  TH1 = ((value && 0xFF00) >> 8);
  TL1 = (value & 0x00FF);
}


void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned long value)
{
  LCD_goto(x_pos, y_pos);
  LCD_putchar((value / 10000) + 0x30);
  LCD_goto((x_pos + 1), y_pos);
  LCD_putchar(((value % 10000) / 1000) + 0x30);
  LCD_goto((x_pos + 2), y_pos);
  LCD_putchar(((value % 1000) / 100) + 0x30);
  LCD_goto((x_pos + 3), y_pos);
  LCD_putchar(((value % 100) / 10) + 0x30);
  LCD_goto((x_pos + 4), y_pos);
  LCD_putchar((value % 10) + 0x30);
}
```

Schematic



Explanation

Before explaining the capture part, I would like to put emphasis on how to generate Timer 1's output. The setup for timer is nothing different from other timer settings in Mode 1 except for the part shown below:

```
P2S |= SET_BIT3;
```

This code sets the timer output. Note that in this example, the system clock speed and hence the clock speed of all hardware sub-systems is set to 8MHz with HIRC and clock divider. Thus, Timer 1's output will be high for:

$$\frac{65536}{8MHz} = 8.192\text{ms}$$

and low for the same amount of time. So, the total time period of Timer 1's output is roughly 16ms. Therefore, the frequency of this output is about 60Hz.

P00 is the Timer 1's output pin and P12 is the capture pin of Timer 2. When P00 and P12 are shorted together, the frequency counter made here with Timer 2 and its capture unit will read 60Hz.

So how it is done by Timer 2 and the capture unit?

Here in this example, we don't need the compare-match feature for the measurement of waveform timings and so all compare match features are disabled. We need to reload the timer when it overflows. Since the timer's counter registers are not manipulated, they are set to zeros by default and so the reload count is zero. These are reflected by the first two lines of the timer's setup:

```
void setup_Timer_2(void)
{
  T2CON &= ~SET_BIT0;
  T2MOD = 0x00;
```

```
  set_TR2;
  set_ET2;
}
```

Next, Timer 2 is started with its interrupt enabled.

Finally, the capture pin and its channel to be used are enabled using BSP-based definition. Capture interrupt is also enabled.

```
void setup_capture(void)
{
  IC0_P12_CAP0_FallingEdge_Capture;
  set_ECAP;
}
```

Note that this is the confusing part of the code. There are three capture channels (***CAP0***, ***CAP1*** and ***CAP2***). These channels share 9 input capture GPIO pins. The pins are cleverly multiplexed. Detection edge selections can be also done. The BSP definition-names state these parameters. For example, the definition used here means:

"*input capture pin 0 connected to P12 GPIO pin is connected to capture channel 0 to detect falling edge transitions*".

Once the hardware setup is complete, the game now resides in the capture interrupt part:

```
#pragma vector = 0x63
__interrupt void Input_Capture_ISR(void)
{
  if((CAPCON0 & SET_BIT0) != 0)
  {
    clr_CAPF0;
    end_time = C0H;
    end_time <<= 8;
    end_time |= C0L;
    pulse_time = ((overflow << 16) - start_time + end_time);
    start_time = end_time;
    overflow = 0;
  }
}
```

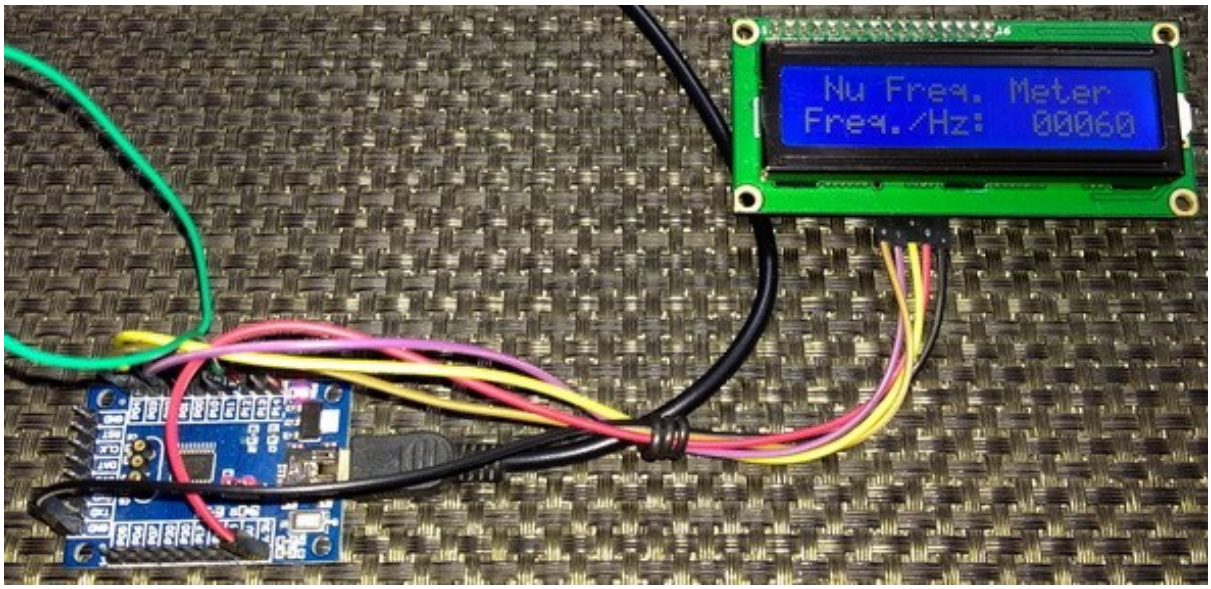
Since there is one interrupt vector for all three capture channels, we have check first which capture channel caused the interrupt and clear that particular interrupt flag soon. Capture interrupt, in this case, occurs when a falling edge is detected by CAP0 channel. When such an interrupt occurs, the time of its happening, i.e. the counter value of Timer 2 at that instance is stored. This marks the first falling edge event. To measure period or frequency of a waveform, another such falling edge is needed. Therefore, when another falling edge is detected by the capture hardware, the process repeats. This results in the time capture of two events.

We know the frequency of the timer's clock and so we can find the value of its each tick. We have two captures in form of two Timer 2 counts. We can find the difference between them and thereby compute the time period (***pulse_time***).

Since the time period of the captured waveform is calculated, its frequency can be deduced. This is done in the main loop.

```
f = (timer_clock_speed / ((float)pulse_time));
```

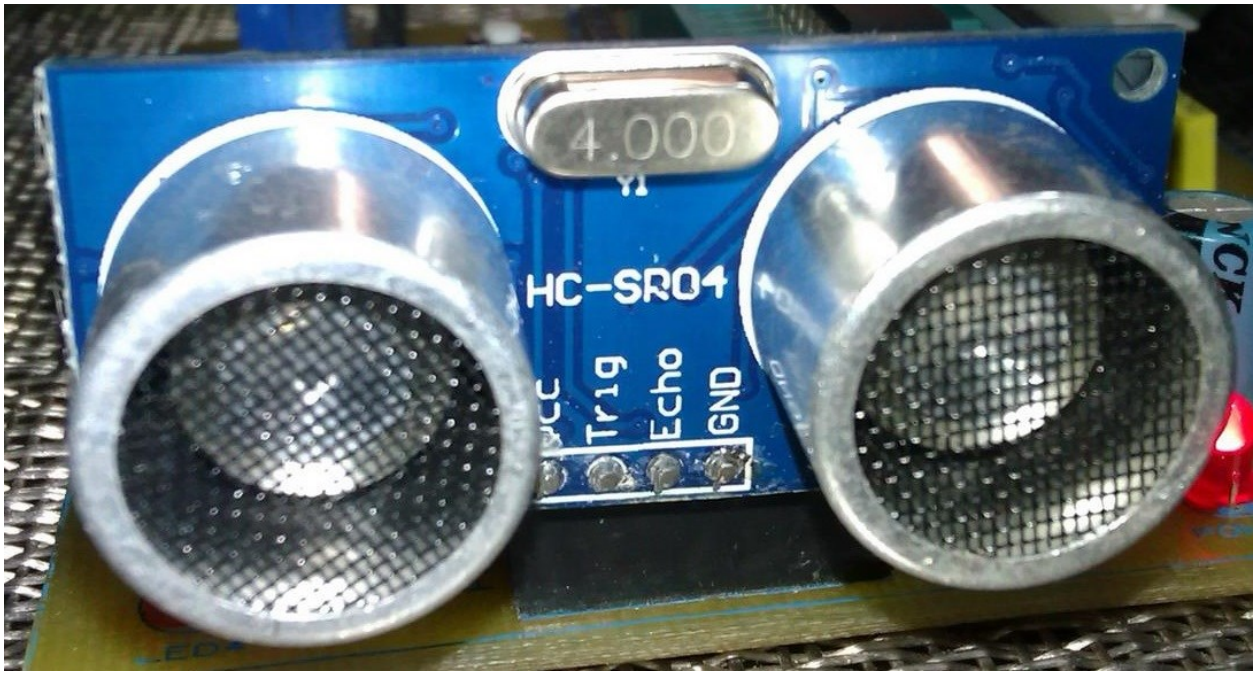
Demo



Demo video: https://youtu.be/33lRj7zE1JM

Timer 2 Pulse Width Capture – Interfacing HC-SR04 SONAR

In the last segment, it was demonstrated how to capture a waveform and compute its frequency using Timer 2's capture hardware. This segment is basically an extension of the last segment. Here, it will be shown how we can use two capture channels connected to the same capture pin to find out the pulse width of a pulse. For demoing this idea, a HC-SR04 ultrasonic SONAR sensor is used. The SONAR sensor gives a pulse output that varies in width according to the distance between the sensor and a target.



Code

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "LCD_3_Wire.h"


#define HIRC    0
#define LIRC    1
#define ECLK    2


unsigned int pulse_width = 0;


void setup(void);
void set_clock_source(unsigned char clock_source);
void disable_clock_source(unsigned char clock_source);
void set_system_clock_frequency(unsigned long F_osc, unsigned long F_sys);
void setup_GPIOs(void);
void setup_Timer_2(void);
```

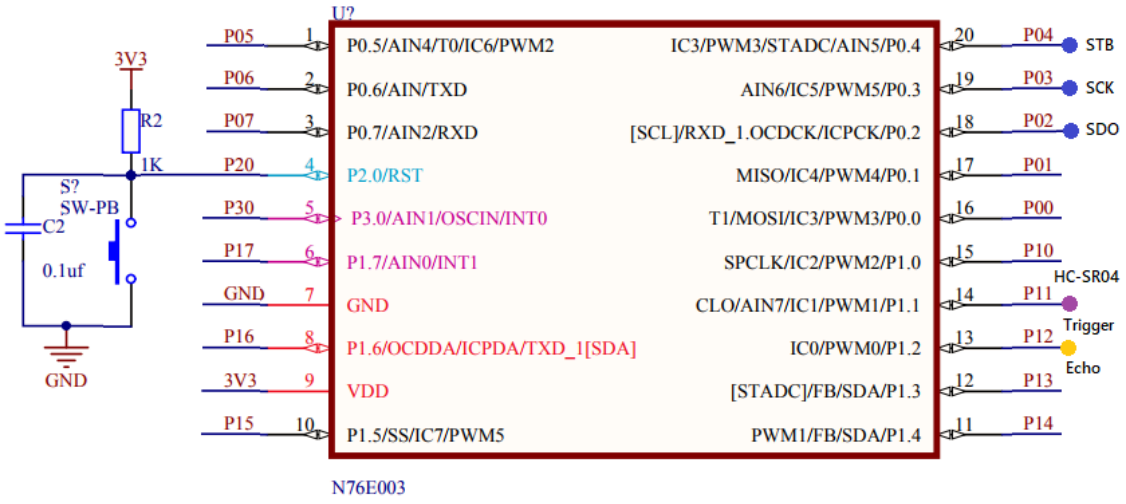
```
void setup_capture(void);
void set_Timer_2(unsigned int value);
void set_Timer_2_reload_compare(unsigned int value);
void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned int value);


void Input_Capture_ISR(void)
interrupt 12
{
  if(CAPCON0 & 0x01)
  {
    clr_CAPF0;
  }

  if(CAPCON0 & 0x02)
  {
    clr_CAPF1;

    pulse_width = C1H;
    pulse_width <<= 8;
    pulse_width |= C1L;
  }
}


void main(void)
{
  unsigned int range = 0;

  LCD_init();
  LCD_clear_home();
  LCD_goto(0, 0);
  LCD_putstr("Pamge/cm:");
  LCD_goto(0, 1);
  LCD_putstr("Pulse/us:");

  setup();

  while(1)
  {
    set_P11;
    Timer3_Delay10us(1);
    clr_P11;

    range = ((unsigned int)(((float)pulse_width) / 58.0));

    lcd_print(11, 0, range);
    lcd_print(11, 1, pulse_width);
    Timer0_Delay1ms(900);
  };
}


void setup(void)
{
  disable_clock_source(ECLK);
```

```
  set_clock_source(HIRC);
  set_system_clock_frequency(16, 16);
  setup_GPIOs();
  setup_capture();
  setup_Timer_2();
  set_EA;
}


void set_clock_source(unsigned char clock_source)
{
  switch(clock_source)
  {
    case LIRC:
    {
      set_OSC1;
      clr_OSC0;

      break;
    }

    case ECLK:
    {
      set_EXTEN1;
      set_EXTEN0;

      while((CKSWT & SET_BIT3) == 0);

      clr_OSC1;
      set_OSC0;

      break;
    }

    default:
    {
      set_HIRCEN;

      while((CKSWT & SET_BIT5) == 0);

      clr_OSC1;
      clr_OSC0;

      break;
    }
  }

  while((CKEN & SET_BIT0) == 1);
}


void disable_clock_source(unsigned char clock_source)
{
  switch(clock_source)
  {
    case HIRC:
```

```
    {
       clr_HIRCEN;
       break;
    }

    default:
    {
       clr_EXTEN1;
       clr_EXTEN0;
       break;
    }
  }
}


void set_system_clock_frequency(unsigned long F_osc, unsigned long F_sys)
{
  F_osc = (F_osc / (0x02 * F_sys));

  if((F_osc >= 0x00) && (F_osc <= 0xFF))
  {
    CKDIV = ((unsigned char)F_osc);
  }
}


void setup_GPIOs(void)
{
  P11_PushPull_Mode;
  P12_Input_Mode;
}


void setup_Timer_2(void)
{
  set_Timer_2_reload_compare(0);
  set_Timer_2(0);
  set_LDEN;
  T2MOD |= 0x01;
  T2MOD |= 0x20;
  set_TR2;
}


void setup_capture(void)
{
  CAPCON0 = 0x30;
  CAPCON1 = 0x01;
  CAPCON2 = 0x30;
  CAPCON3 = 0x00;
  CAPCON4 = 0x00;
  set_ECAP;
}


void set_Timer_2(unsigned int value)
```

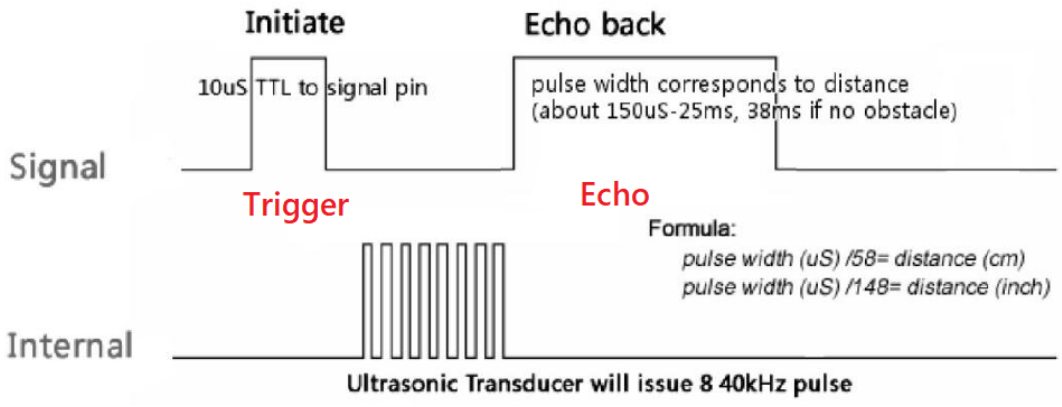
```
{
  TL2 = (value & 0x00FF);
  TH2 = ((value & 0xFF00) >> 0x08);
}


void set_Timer_2_reload_compare(unsigned int value)
{
  RCMP2L = (value & 0x00FF);
  RCMP2H = ((value & 0xFF00) >> 0x08);
}


void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned int value)
{
  LCD_goto(x_pos, y_pos);
  LCD_putchar((value / 10000) + 0x30);
  LCD_goto((x_pos + 1), y_pos);
  LCD_putchar(((value % 10000) / 1000) + 0x30);
  LCD_goto((x_pos + 2), y_pos);
  LCD_putchar(((value % 1000) / 100) + 0x30);
  LCD_goto((x_pos + 3), y_pos);
  LCD_putchar(((value % 100) / 10) + 0x30);
  LCD_goto((x_pos + 4), y_pos);
  LCD_putchar((value % 10) + 0x30);
}
```

Schematic



Explanation

HC-SR04 SONAR sensor has two pins apart from power supply pins. These pins are labelled "***Echo***" and "***Trigger***". When the trigger pin of a HC-SR04 is set high for about 10µs, it acknowledges this short duration pulse in the trigger pin as a command from its host micro to measure and return distance data.



P11 is setup as the trigger pin and P12 is set up as the echo pin. The concepts of input capture are same as in the last capture example but there are a few minor changes. Firstly, the system clock speed is set to full 16MHz using HIRC. Secondly, the setup of Timer 2 is slightly changed. Auto-reloading is enabled this time. Timer 2's clock is prescaled to 1MHz. This is done so in order to make sure that the capture has a resolution of 1µs and the timer doesn't overflow while taking measurements. The maximum possible width of a pulse from HC-SR04 is 38ms when no obstacle is detected by it but with this timer setup we can measure pulse widths up to 65ms. Timer 2 is also set up as to reset its count when a CAP0 event occurs.

```
void setup_Timer_2(void)
{
  set_Timer_2_reload_compare(0);
  set_Timer_2(0);
  set_LDEN;
  T2MOD |= 0x01;
  T2MOD |= 0x20;
  set_TR2;
}
```

In order to measure pulse widths, we can use one capture channel in both edge capture mode or use two capture channels connected to the same input capture GPIO pin detecting different edges. The latter is used here.

```
void setup_capture(void)
{
  CAPCON0 = 0x30;
  CAPCON1 = 0x01;
  CAPCON2 = 0x30;
  CAPCON3 = 0x00;
  CAPCON4 = 0x00;
  set_ECAP;
}
```

I didn't use BSP-based definitions for setting up captures in this example because of some limitation. BSP-based definitions reset selections. This is why I configured the input capture control registers manually. Two capture channels having different edge detections are used and both of these channels share P12 or IC0 pin.

Now when HC-SR04 is triggered, it will give out a pulse. We have to measure the width of that pulse. A pulse is defined by a rising edge and a falling edge. CAP0 channel is set to detect the rising edge of the pulse and reset Timer 2 to 0 count. CAP1 channel is used to detect the falling edge of the same pulse and read Timer 2's count. This count represents pulse width in microseconds.

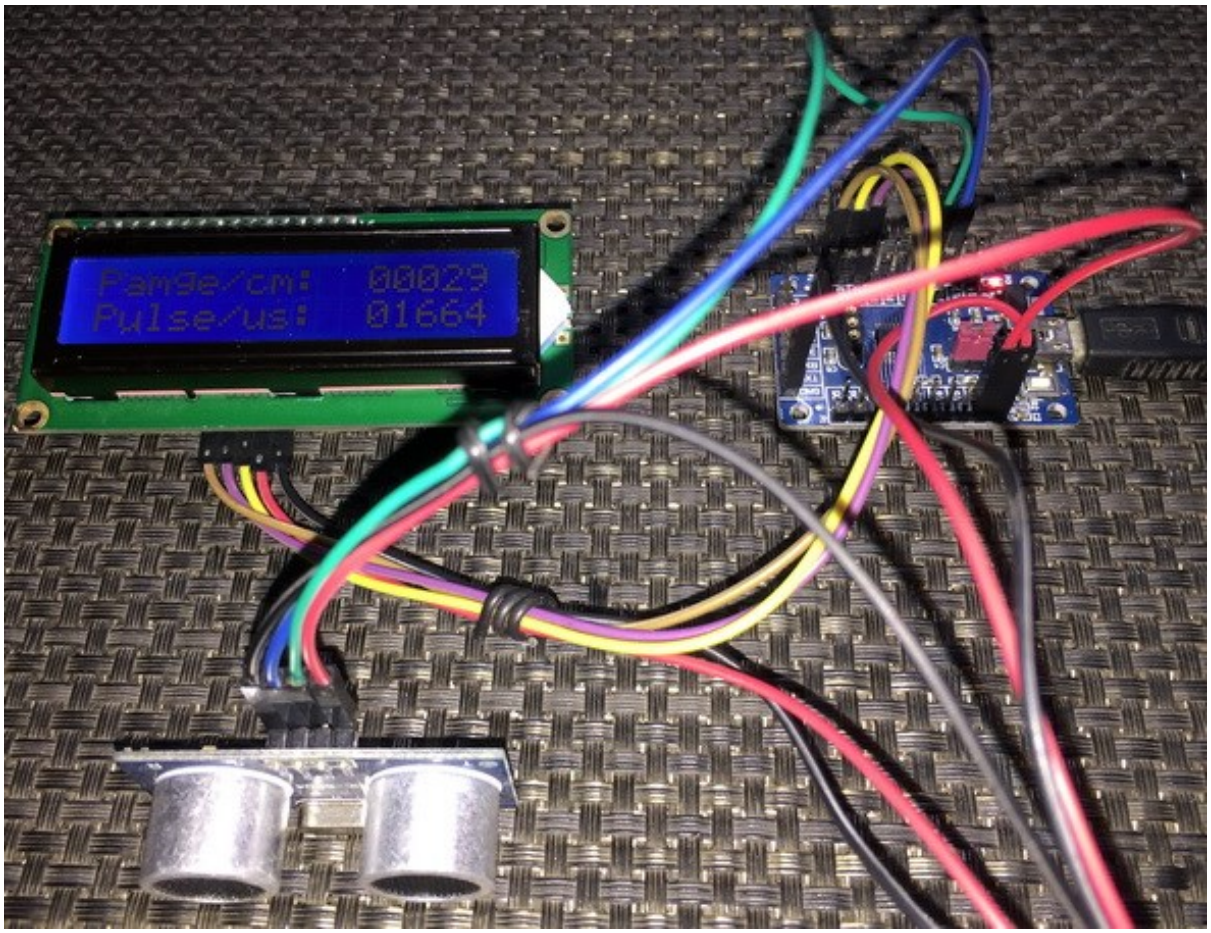
```
void Input_Capture_ISR(void)
interrupt 12
{
  if(CAPCON0 & 0x01)
  {
    clr_CAPF0;
  }

  if(CAPCON0 & 0x02)
  {
    clr_CAPF1;

    pulse_width = C1H;
    pulse_width <<= 8;
    pulse_width |= C1L;
  }
}
```

In the main loop, distance is calculated using this measured pulse width time and the formula given in HC-SR04's datasheet.

```
range = ((unsigned int)(((float)pulse_width) / 58.0));
```

The distance found between an object and HC-SR04 is displayed on a text LCD.

Demo



Demo video: https://youtu.be/icGIjSAh8SM

Timer 3 – Driving 7 Segments, LED and Scanning Keypad

We all know that N76E003 is a cool chip but it lacks GPIOs unlike other chips of similar capabilities. Thus, we must be miser while using GPIOs. When it is needed to interface several inputs and outputs with a microcontroller using fewer pins, we often take the assistance of logic ICs like shift registers, counters, multiplexers, etc. We have already used this technique while making the 3-wire LCD interface. An LCD has its own controller(s) to take care of projecting data on the screen once data has been feed to it. However, that's not the case with seven segment displays and keypads when used without any specialized driver IC like TM1640/MAX7219. It is, then, a job for a host micro to do the scanning and data manipulation periodically with affecting other tasks. This can be achieved easily with a timer.

In this segment, we will see how the aforementioned is done with Timer 3. For the demo, I used a salvaged temperature controller's I/O unit. The I/O unit consists of two 4-digit seven segment displays, four LEDs and four push buttons. It is made with a 74HC164 Serial-In-Parallel-Out (SPIO) shift register and a 74HC145 BCD-to-Decimal decoder. In order to use it in real-time, its displays, LEDs and buttons should be scanned and updated at a fast rate without hindering any part of an application running in main loop.

Code

```
#include "N76E003_iar.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"


#define HIRC                        0
#define LIRC                        1
#define ECLK                        2

#define GATE_HIGH                   set_P15
#define GATE_LOW                    clr_P15

#define CLK_HIGH                    set_P16
#define CLK_LOW                     clr_P16

#define A_HIGH                      set_P00
#define A_LOW                       clr_P00

#define B_HIGH                      set_P01
#define B_LOW                       clr_P01

#define C_HIGH                      set_P02
#define C_LOW                       clr_P02

#define D_HIGH                      set_P03
#define D_LOW                       clr_P03

#define SW                          P17

#define top_seg                     4
#define bot_seg                     0

#define HIGH                        1
#define LOW                         0


const unsigned char num[0x0A] = {0xED, 0x21, 0x8F, 0xAB, 0x63, 0xEA, 0xEE,
0xA1, 0xEF, 0xEB};
unsigned char data_values[0x09] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00};


unsigned char SW_in = 0;
unsigned char n = 0;


void setup(void);
void setup_GPIOs(void);
void setup_Timer_3(void);
void set_Timer_3(unsigned int value);
unsigned int get_Timer_3(void);
void write_74HC164(register unsigned char value);
```

```
void write_74HC145(register unsigned char channel);
void show_LEDs(unsigned char LED1_state, unsigned char LED2_state, unsigned
char LED3_state, unsigned char LED4_state);
void show_numbers(signed int value, unsigned char pos);


#pragma vector = 0x83
__interrupt void Timer3_ISR(void)
{
  write_74HC164(data_values[n]);
  write_74HC145(n);

  n++;
  if(n > 9)
  {
    n = 0;
  }

  clr_TF3;
}


void main (void)
{

  unsigned int i = 0;
  unsigned int j = 9999;

  setup();

  while(1)
  {
    switch(SW_in)
    {
     case 1:
     {
         show_LEDs(1, 0, 0, 0);
         break;
     }
     case 2:
     {
         show_LEDs(0, 1, 0, 0);
         break;
     }
     case 3:
     {
         show_LEDs(0, 0, 1, 0);
         break;
     }
     case 4:
     {
         show_LEDs(0, 0, 0, 1);
         break;
     }
    }
```

```
    SW_in = 0x00;

    i++;
    j--;

    if(i > 9999)
    {
     i = 0;
     j = 9999;
    }

    show_numbers(i, bot_seg);
    show_numbers(j, top_seg);

    Timer1_Delay10ms(40);
    show_LEDs(0, 0, 0, 0);
  };
}


void setup(void)
{
  setup_GPIOs();
  setup_Timer_3();
}


void setup_GPIOs(void)
{
  P00_PushPull_Mode;
  P01_PushPull_Mode;
  P02_PushPull_Mode;
  P03_PushPull_Mode;
  P15_PushPull_Mode;
  P16_PushPull_Mode;
  P17_Input_Mode;
}


void setup_Timer_3(void)
{
  set_Timer_3(0xF9C0);
  set_ET3;
  set_EA;
  set_TR3;
}


void set_Timer_3(unsigned int value)
{
  RL3 = (value & 0x00FF);
  RH3 = ((value && 0xFF00) >> 8);
}


unsigned int get_Timer_3(void)
```

```
{
  unsigned int value = 0x0000;

  value = RH3;
  value <<= 8;
  value |= RL3;

  return value;
}


void write_74HC164(register unsigned char value)
{
  register unsigned char s = 0x08;

  while(s > 0)
  {
    if((value & 0x80) != 0x00)
    {
        GATE_HIGH;
    }
    else
    {
        GATE_LOW;
    }

    CLK_HIGH;
    CLK_LOW;

    value <<= 1;
    s--;
  };
}


void write_74HC145(register unsigned char channel)
{
  P0 = 0x00;

  switch(channel)
  {
    case 0:
    {
      asm("nop");

      if(SW == LOW)
      {
          SW_in = 1;
      }
      break;
  }

    case 1:
    {
      P0 = 0x01;
      break;
```

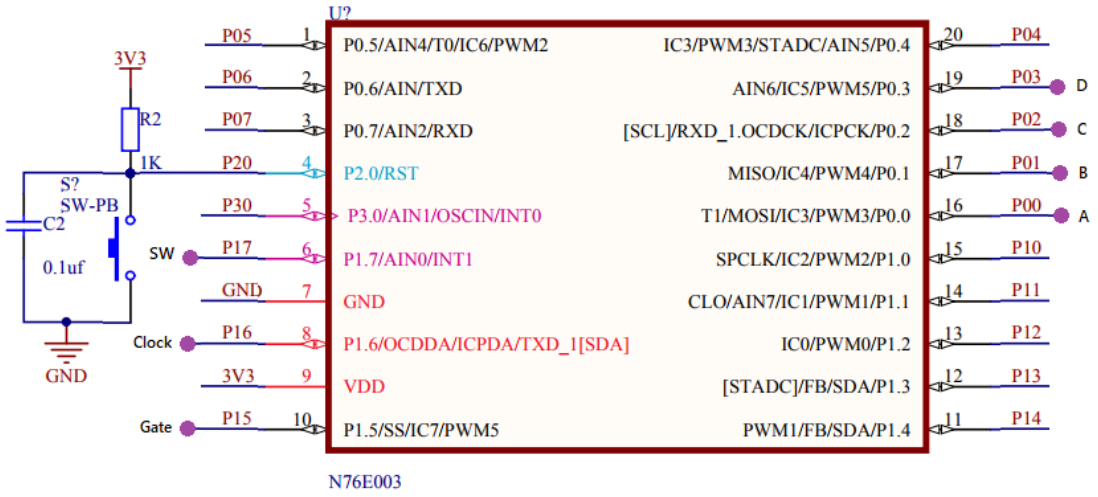
```
    }

    case 2:
    {
      P0 = 0x02;
      break;
    }

    case 3:
    {
      P0 = 0x03;
      break;
    }

    case 4:
    {
      P0 = 0x04;
      break;
    }

    case 5:
    {
      P0 = 0x05;
      break;
    }

    case 6:
    {
      P0 = 0x06;
      break;
    }

    case 7:
    {
      P0 = 0x07;
      asm("nop");

      if(SW == LOW)
      {
          SW_in = 2;
      }
      break;
    }

    case 8:
    {
      P0 = 0x08;
      asm("nop");

      if(SW == LOW)
      {
          SW_in = 3;
      }
      break;
    }
```

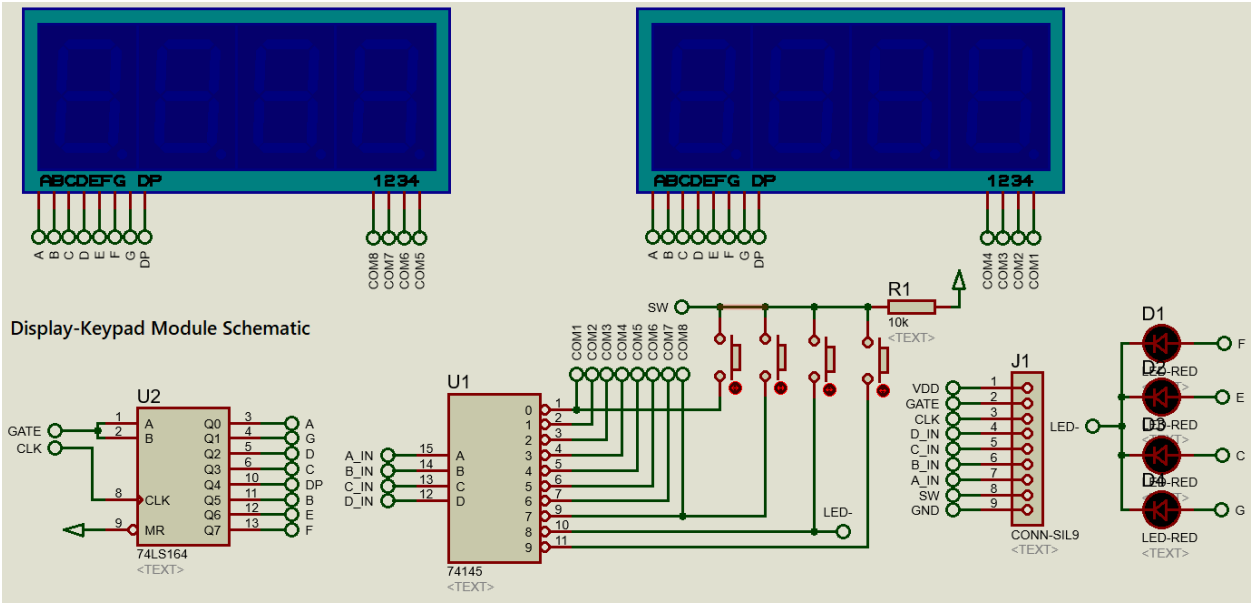
```
    case 9:
    {
      P0 = 0x09;
      asm("nop");

      if(SW == LOW)
      {
          SW_in = 4;
      }
      break;
    }
  }
}


void show_LEDs(unsigned char LED1_state, unsigned char LED2_state, unsigned
char LED3_state, unsigned char LED4_state)
{
  switch(LED1_state)
  {
    case HIGH:
    {
       data_values[8] |= 0x80;
       break;
    }
    case LOW:
    {
       data_values[8] &= 0x7F;
       break;
    }
  }

  switch(LED2_state)
  {
    case HIGH:
    {
       data_values[8] |= 0x40;
       break;
    }
    case LOW:
    {
       data_values[8] &= 0xBF;
       break;
    }
  }

  switch(LED3_state)
  {
    case HIGH:
    {
       data_values[8] |= 0x08;
       break;
    }
    case LOW:
    {
       data_values[8] &= 0xF7;
```

```
      break;
    }
  }

  switch(LED4_state)
  {
    case HIGH:
    {
       data_values[8] |= 0x02;
       break;
    }
    case LOW:
    {
       data_values[8] &= 0xFD;
       break;
    }
  }
}


void show_numbers(signed int value, unsigned char pos)
{
  register unsigned char ch = 0x00;

  if((value >= 0) && (value <= 9))
  {
    ch = (value % 10);
    data_values[(0 + pos)] = num[ch];
    data_values[(1 + pos)] = 0x00;
    data_values[(2 + pos)] = 0x00;
    data_values[(3 + pos)] = 0x00;
  }
  else if((value > 9) && (value <= 99))
  {
    ch = (value % 10);
    data_values[(0 + pos)] = num[ch];
    ch = ((value / 10) % 10);
    data_values[(1 + pos)] = num[ch];
    data_values[(2 + pos)] = 0x00;
    data_values[(3 + pos)] = 0x00;
  }
  else if((value > 99) && (value <= 999))
  {
    ch = (value % 10);
    data_values[(0 + pos)] = num[ch];
    ch = ((value / 10) % 10);
    data_values[(1 + pos)] = num[ch];
    ch = ((value / 100) % 10);
    data_values[(2 + pos)] = num[ch];
    data_values[(3 + pos)] = 0x00;
  }
  else if((value > 999) && (value <= 9999))
  {
    ch = (value % 10);
    data_values[(0 + pos)] = num[ch];
    ch = ((value / 10) % 10);
```

```
    data_values[(1 + pos)] = num[ch];
    ch = ((value / 100) % 10);
    data_values[(2 + pos)] = num[ch];
    ch = (value / 1000);
    data_values[(3 + pos)] = num[ch];
  }
}
```

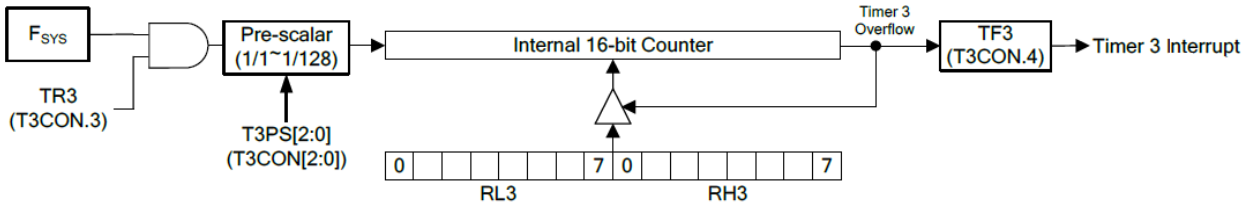
Schematic





Explanation

Timer 3's internal hardware is very simple. It is an up-counting timer and is run by using system clock as clock source. There is a prescalar to reduce the system clock input. The main feature of Timer 3 is it auto-reload feature. This feature basically allows us to forget reloading its 16-bit counter unlike Timer 0 and 1. There no external input or output option for this timer. Owing to all these, it is most suitable for time-base generation and serial communication.



In this demo, Timer 3 is set to interrupt every 100µs.

$$\text{Timer 3 ISR} = \frac{(65536 - Reload\ Counter\ Value)}{Timer\ 3\ Clock\ Frequency} = \frac{(0\text{xFFFF} - 0\text{xF9C0})}{16MHz} = 100\mu s$$

The following code sets up Timer 3 as discussed:

```
void setup_Timer_3(void)
{
  set_Timer_3(0xF9C0);
  set_ET3;
  set_EA;
  set_TR3;
}
```

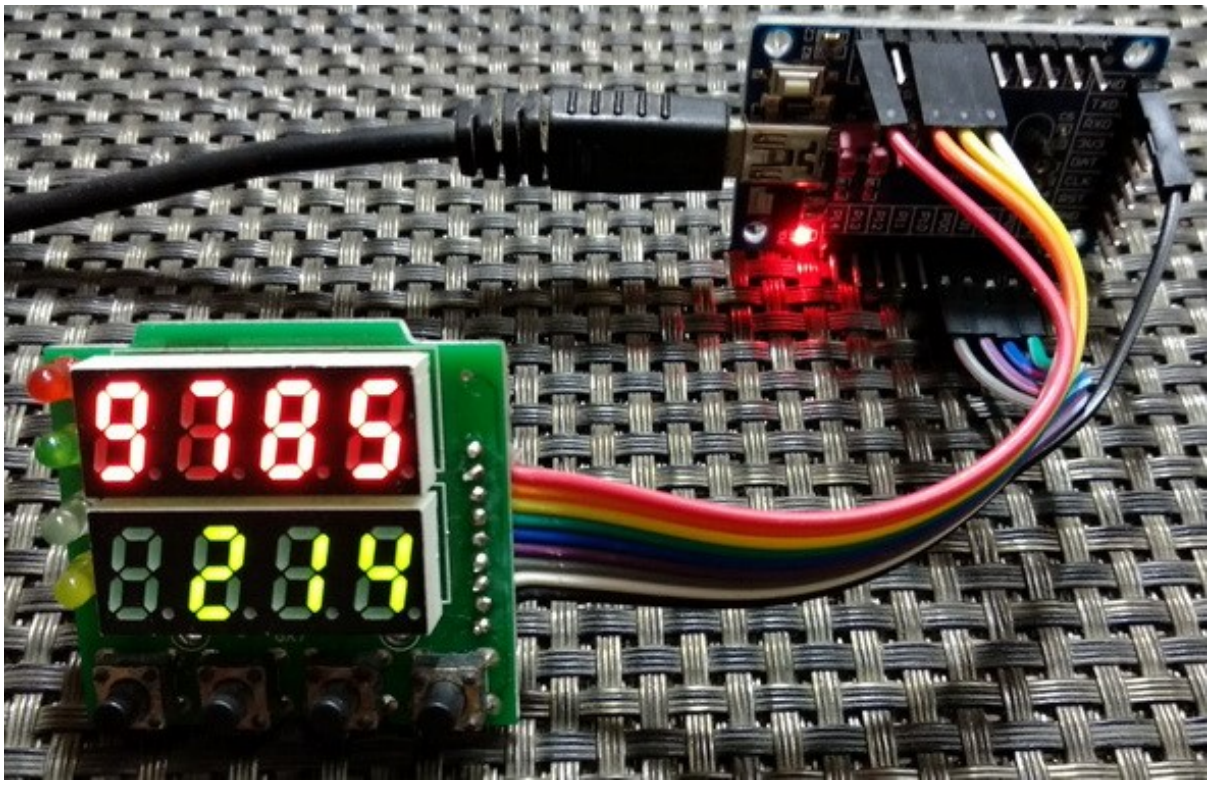
Note that Timer 3's input clock and system clock frequency is 16MHz since no prescalar is used.

Now what we are doing inside the timer interrupt? According to the schematic and objective of this project, we need to update the seven segment displays and read the 4-bit keypad so fast that it looks as if everything is being done without any delay and in real time.

```
#pragma vector = 0x83
__interrupt void Timer3_ISR(void)
{
  write_74HC164(data_values[n]);
  write_74HC145(n);

  n++;
  if(n > 9)
  {
    n = 0;
  }

  clr_TF3;
}
```

Inside timer ISR, both logic ICs are updated. Firstly, the number to be displayed is sent to the 74HC164 IC and then the seven-segment display to show the number is updated by writing the 74HC145 IC. At every interrupt, one seven segment display is updated. There are 8 such displays and so it takes less than a millisecond to update all these displays. Everything seems to complete in the blink of an eye. During this time the keypad is also scanned in the main. With different keys, different LEDs light up.

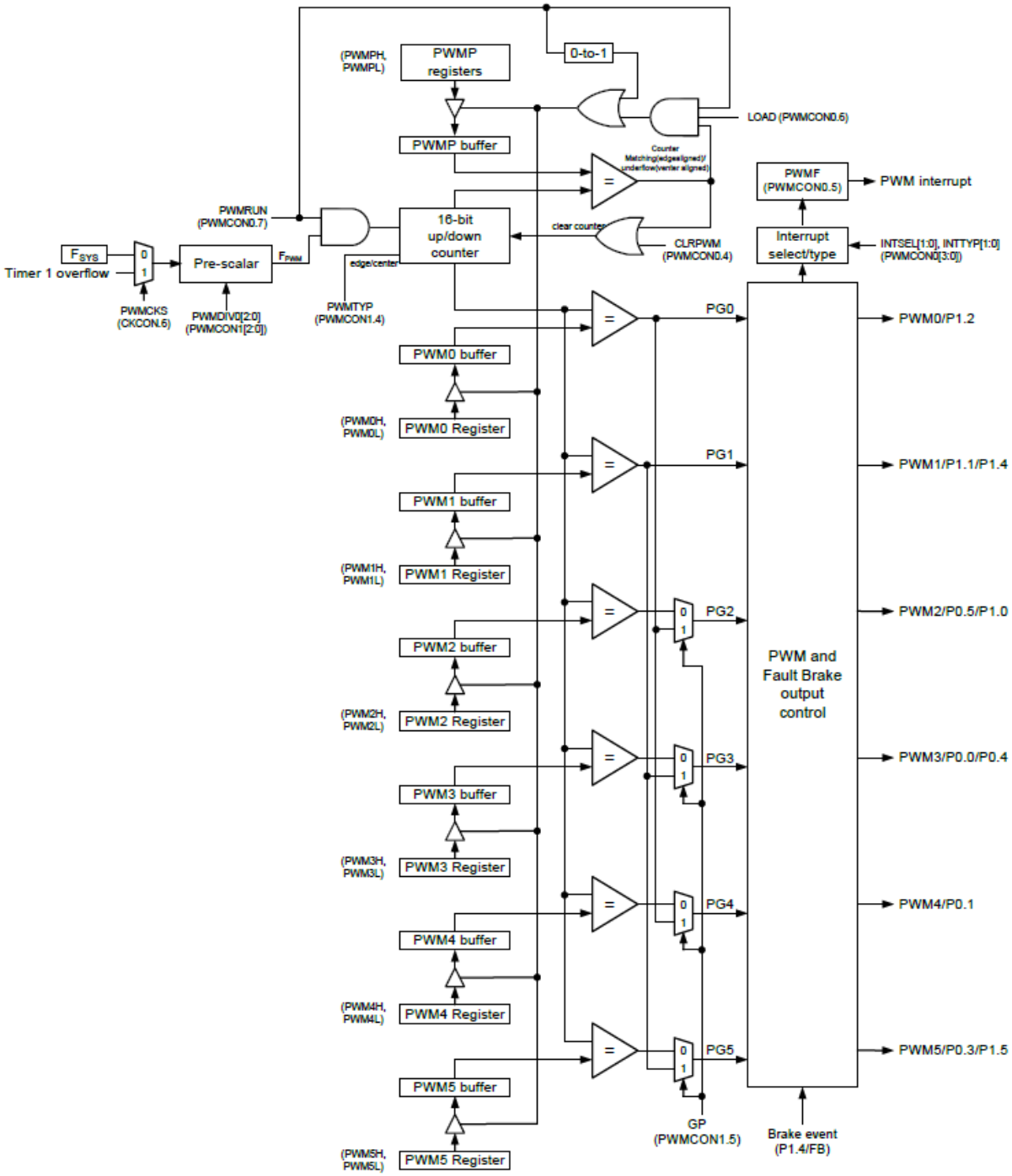
Demo



Demo video: https://youtu.be/Gd6BecOFtxk

Simple PWM – RGB LED Fading

PWM hardware is another basic requirement for any modern-era microcontroller. We can use PWM for a number of applications like motor control, light control, switch-mode power supplies (SMPSs), etc. With PWM we can also simulate digital-to-analogue converter (DAC). Fortunately, N76E003 comes with a separate PWM block that is not a part of any internal timer. It has all the feature that you can imagine. It can be used to generate simple independent 6 single channel PWMs. It can also be used to generate complementary and interdependent PWMs with dead time feature.



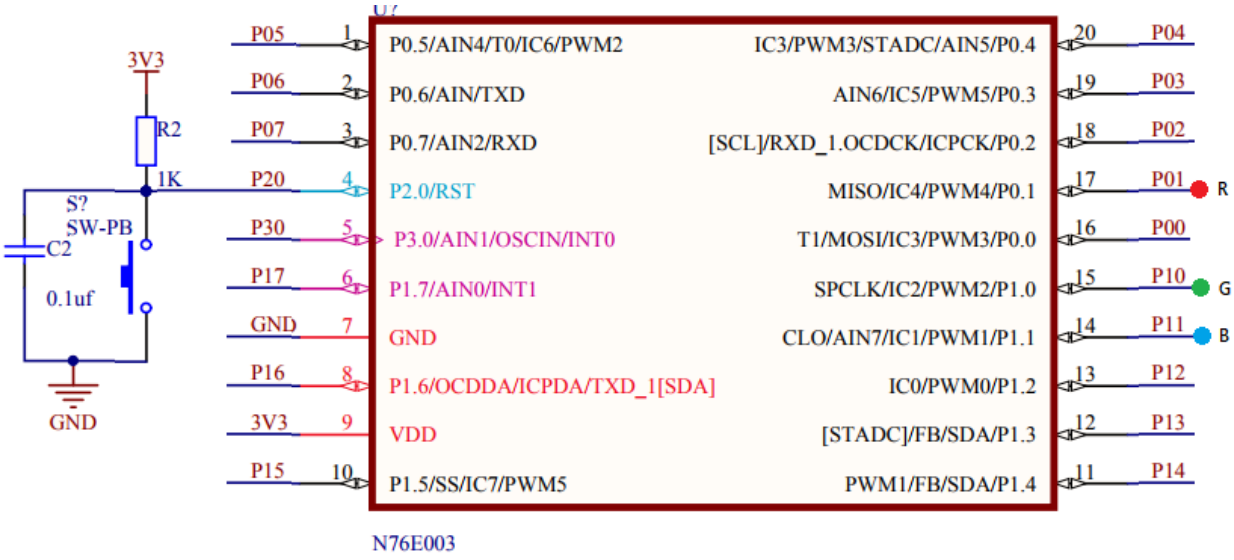
Code

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "soft_delay.h"


unsigned int R_value[10] = {20, 150, 250, 360, 440, 560, 680, 820, 900, 1020};
unsigned int G_value[10] = {440, 560, 680, 820, 900, 1020, 20, 150, 250, 360};
unsigned int B_value[10] = {900, 1020, 20, 150, 250, 360, 440, 560, 680, 820};


void set_PWM_period(unsigned int value);
void set_PWM0(unsigned int value);
void set_PWM1(unsigned int value);
void set_PWM2(unsigned int value);
void set_PWM3(unsigned int value);
void set_PWM4(unsigned int value);
void set_PWM5(unsigned int value);


void main(void)
{
    signed int i = 0;
    signed char j = 0;

    P01_PushPull_Mode;
    P10_PushPull_Mode;
    P11_PushPull_Mode;

    PWM1_P11_OUTPUT_ENABLE;
    PWM2_P10_OUTPUT_ENABLE;
    PWM4_P01_OUTPUT_ENABLE;

    PWM_IMDEPENDENT_MODE;
    PWM_EDGE_TYPE;
    set_CLRPWM;
    PWM_CLOCK_FSYS;
    PWM_CLOCK_DIV_64;
    PWM_OUTPUT_ALL_NORMAL;
    set_PWM_period(1023);
    set_PWMRUN;

    while(1)
    {
        for(i = 0; i < 1024; i += 10)
        {
            set_PWM1(i);
            delay_ms(20);
        }
        for(i = 1023; i > 0; i -= 10)
        {
            set_PWM1(i);
```

```
            delay_ms(20);
        }

        for(i = 0; i < 1024; i += 10)
        {
            set_PWM2(i);
            delay_ms(20);
        }
        for(i = 1023; i > 0; i -= 10)
        {
            set_PWM2(i);
            delay_ms(20);
        }

        for(i = 0; i < 1024; i += 10)
        {
            set_PWM4(i);
            delay_ms(20);
        }
        for(i = 1023; i > 0; i -= 10)
        {
            set_PWM4(i);
            delay_ms(20);
        }

        delay_ms(600);

        for(i = 0; i <=9; i++)
        {
            for(j = 0; j <= 9; j++)
            {
                set_PWM4(R_value[j]);
                set_PWM1(G_value[j]);
                set_PWM2(B_value[j]);
                delay_ms(200);
            }
            for(j = 9; j >= 0; j--)
            {
                set_PWM4(R_value[j]);
                set_PWM1(G_value[j]);
                set_PWM2(B_value[j]);
                delay_ms(200);
            }
        }

        delay_ms(600);
    }
}


void set_PWM_period(unsigned int value)
{
  PWMPL = (value & 0x00FF);
  PWMPH = ((value & 0xFF00) >> 8);
}
```

```c
void set_PWM0(unsigned int value)
{
  PWM0L = (value & 0x00FF);
  PWM0H = ((value & 0xFF00) >> 8);
  set_LOAD;
}


void set_PWM1(unsigned int value)
{
  PWM1L = (value & 0x00FF);
  PWM1H = ((value & 0xFF00) >> 8);
  set_LOAD;
}


void set_PWM2(unsigned int value)
{
  PWM2L = (value & 0x00FF);
  PWM2H = ((value & 0xFF00) >> 8);
  set_LOAD;
}


void set_PWM3(unsigned int value)
{
  PWM3L = (value & 0x00FF);
  PWM3H = ((value & 0xFF00) >> 8);
  set_LOAD;
}


void set_PWM4(unsigned int value)
{
  set_SFRPAGE;
  PWM4L = (value & 0x00FF);
  PWM4H = ((value & 0xFF00) >> 8);
  clr_SFRPAGE;
  set_LOAD;
}


void set_PWM5(unsigned int value)
{
  set_SFRPAGE;
  PWM5L = (value & 0x00FF);
  PWM5H = ((value & 0xFF00) >> 8);
  clr_SFRPAGE;
  set_LOAD;
}
```

Schematic



Explanation

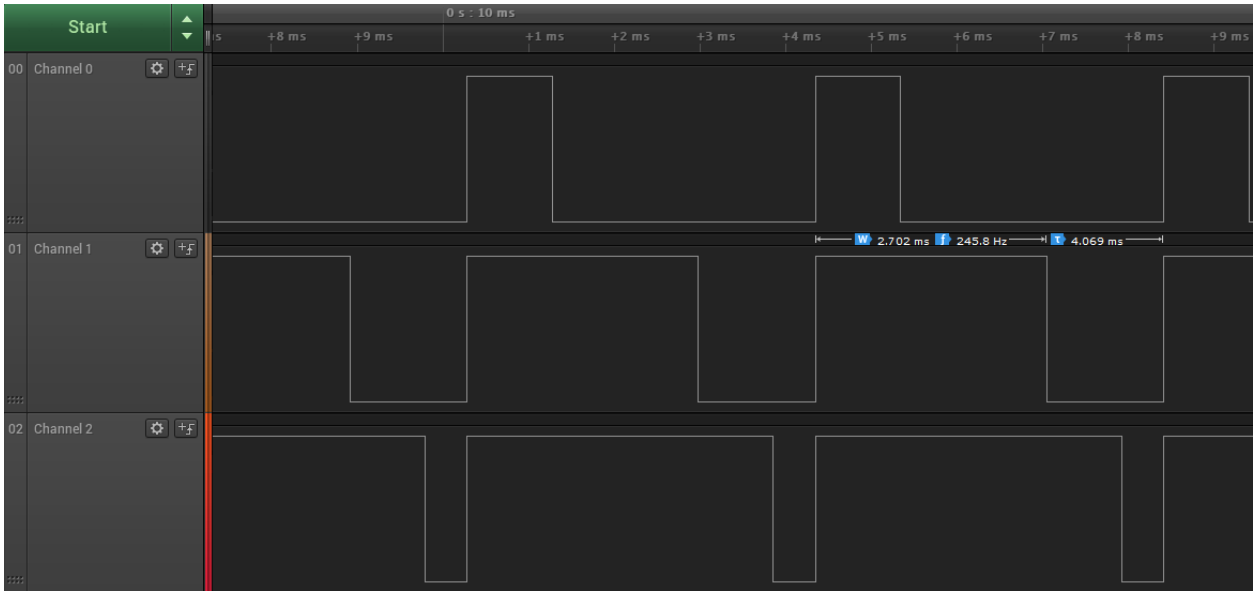
PWM generated by the N76E003's PWM hardware is based on compare-match principle and this is evident from its block diagram. There are 6 PWM channels and they can be used independently or as interdependent groups to form complimentary PWMs. PWMs generated by N76E003 can be edge-aligned or centre-aligned PWMs as per user's requirement. The PWM block can be clocked directly by the system clock or by Timer 1 overflow. Additionally, there is a prescalar unit to reduce input clock source.

For demonstrating N76E003's PWM feature, I used an RGB LED. Three independent PWMs were generated using PWM channels 1,2 and 4. Let us look into the process of setting up the PWM channels and the PWM hardware. We must firstly set the PWM GPIOs as push-pull GPIOs since PWM is an output function of GPIO pins. We must also enable the PWM output channels we want to use.

```
P01_PushPull_Mode;
P10_PushPull_Mode;
P11_PushPull_Mode;
PWM1_P11_OUTPUT_ENABLE;
PWM2_P10_OUTPUT_ENABLE;
PWM4_P01_OUTPUT_ENABLE;
```

Secondly, the PWM hardware is set up according to our needs. Since we need independent PWMs here, independent PWM mode is selected. Edge-align PWM is selected since it is most easy to understand and use. The PWM channels are reset before using them. The PWM outputs are non-inverted and so they are characterized as normal PWMs. Here, I use system clock as the clock source for the PWM block but it is prescaled/divided by 64 to get an effective PWM clock of 250kHz. The PWM resolution is set to 10-bits when we set the period count or maximum PWM value/duty cycle. Setting the period count yields in setting up the internal PWM counter. This gives a PWM of approximately 245Hz frequency or 4ms period. Finally, the PWM hardware is enabled after setting up all these.

```
PWM_IMDEPENDENT_MODE;
PWM_EDGE_TYPE;
set_CLRPWM;
PWM_CLOCK_FSYS;
PWM_CLOCK_DIV_64;
PWM_OUTPUT_ALL_NORMAL;
set_PWM_period(1023);
set_PWMRUN;
```

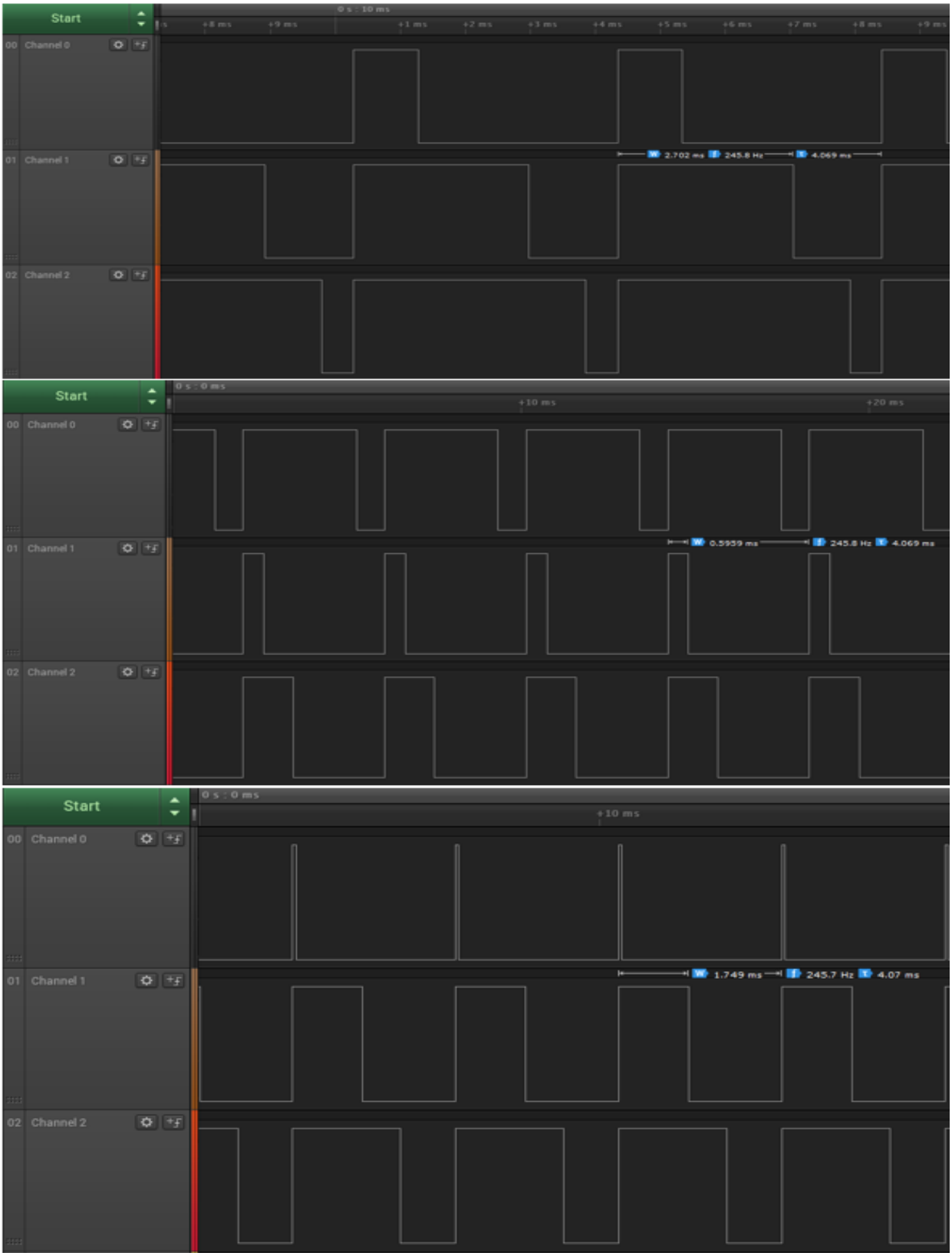


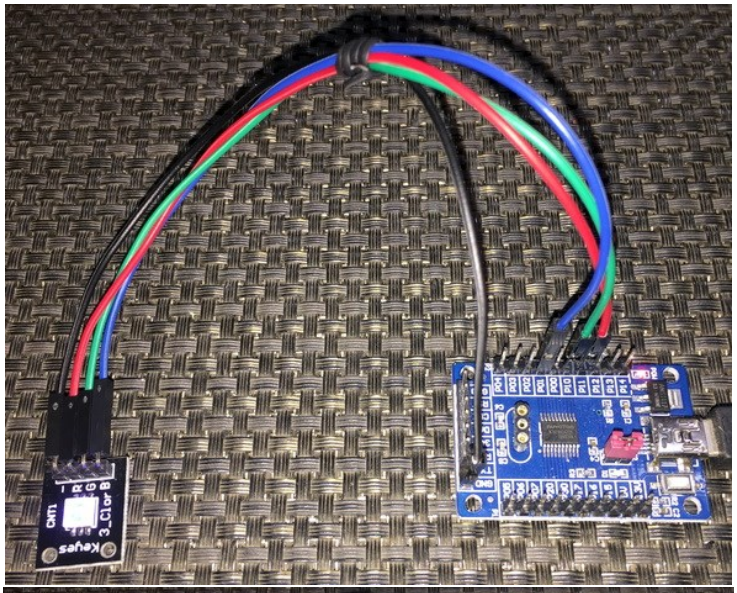
To change PWM duty cycle, functions like the one shown below is called. After altering the duty cycle or compare value, the new value is loaded and run.

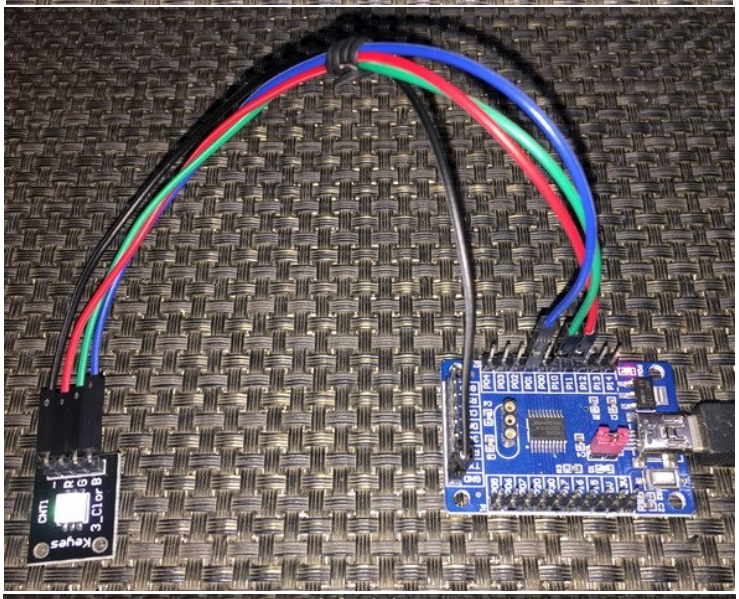
```
void set_PWMn(unsigned int value)
{
  PWMnL = (value & 0x00FF);
  PWMnH = ((value & 0xFF00) >> 8);
  set_LOAD;
}
```

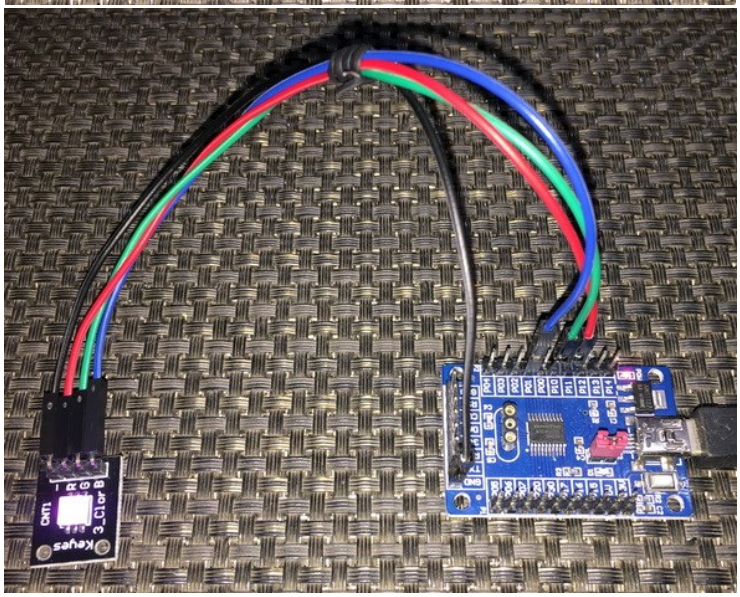
In the demo, the RGB LED fades different colours as a symbol of changing PWMs.

Demo


Start
00 Channel 0
01 Channel 1
02 Channel 2
2.702 ms
245.8 Hz
4.069 ms
Start
00 Channel 0
01 Channel 1
02 Channel 2
0.5959 ms
245.8 Hz
4.069 ms
Start
00 Channel 0
01 Channel 1
02 Channel 2
1.749 ms
245.7 Hz
4.07 ms





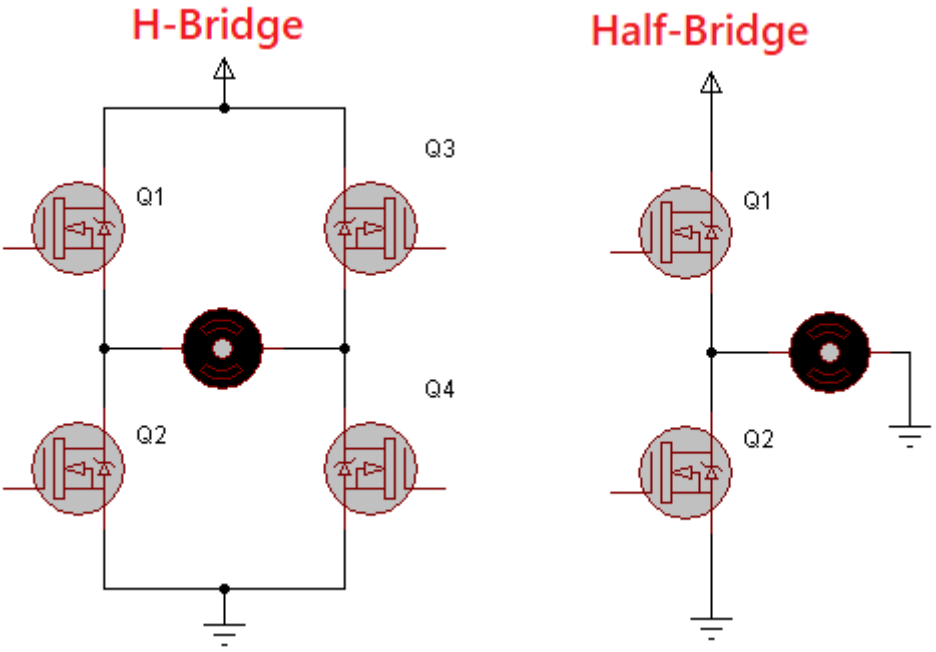


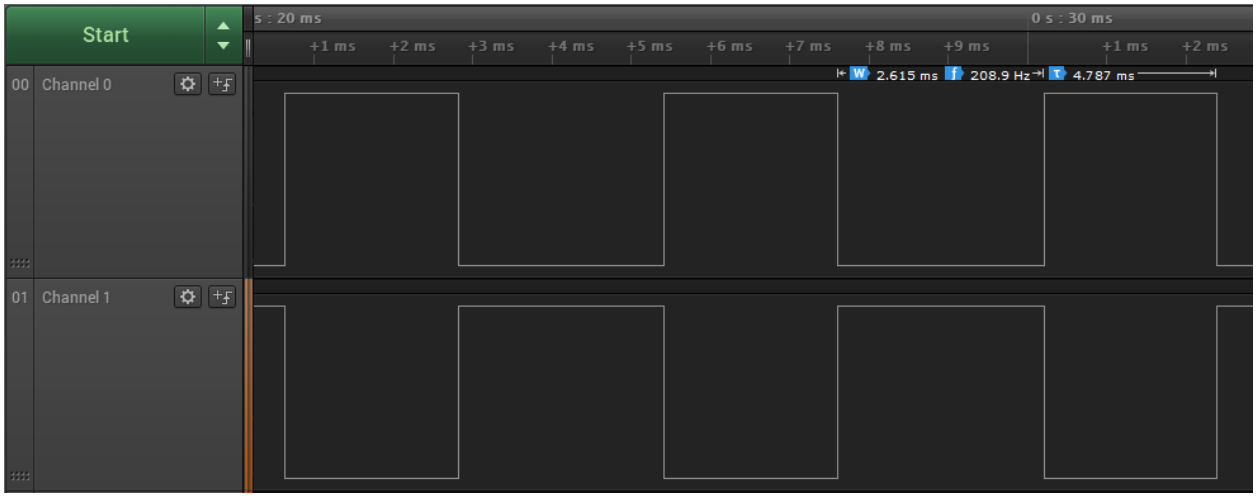
Demo video: https://youtu.be/vomQy0alnPU

Complementary PWM with Dead Time

In the last section, we saw how we can generate independent PWM. Now we will see complimentary or group PWM with dead-time. We will also see the things that were skipped in the previous segment.

Complementary PWM with dead-time feature is a must-have feature of any PWM drives in today's embedded-system industry. Unlike the simple PWM we saw previously, this kind of PWM has most usage. Complementary PWM with dead-time is used in applications where we need to design motor controllers, SMPSs, inverters, etc with H-bridges or half-bridges. In such applications, PWMs need to operate in groups while ensuring that both PWMs don't turn on at the same time. One PWM in a group should be the opposite/antiphase of the other in terms of duty-cycle or waveshape.





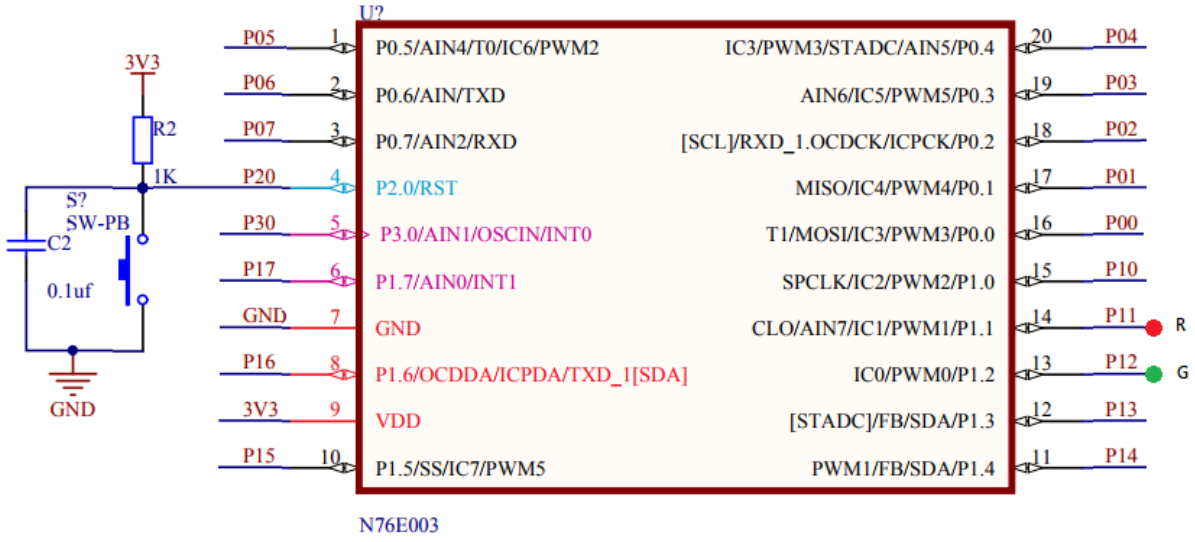
Code

```c
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"


void set_PWM_period(unsigned int value);
void set_PWM0(unsigned int value);
void set_PWM1(unsigned int value);
void set_PWM2(unsigned int value);
void set_PWM3(unsigned int value);
void set_PWM4(unsigned int value);
void set_PWM5(unsigned int value);
void set_PWM_dead_time(unsigned int value);


void main(void)
{
  signed int i = 0;

  P11_PushPull_Mode;
  P12_PushPull_Mode;

  PWM0_P12_OUTPUT_ENABLE;
  PWM1_P11_OUTPUT_ENABLE;

  PWM_COMPLEMENTARY_MODE;
  PWM_CENTER_TYPE;
  set_CLRPWM;
  PWM_CLOCK_FSYS;
  PWM_CLOCK_DIV_64;
  PWM0_OUTPUT_INVERSE;
  PWM1_OUTPUT_INVERSE;
  set_PWM_period(600);
  set_PWM_dead_time(40);
  PWM01_DEADTIME_ENABLE;
  set_PWMRUN;

  while(1)
  {
    for(i = 0; i < 600; i++)
    {
      set_PWM0(i);
      Timer0_Delay1ms(5);
    }
    for(i = 600; i > 0; i--)
    {
      set_PWM0(i);
      Timer0_Delay1ms(5);
    }
  };
}
```

```
void set_PWM_period(unsigned int value)
{
  PWMPL = (value & 0x00FF);
  PWMPH = ((value & 0xFF00) >> 8);
}


void set_PWM0(unsigned int value)
{
  PWM0L = (value & 0x00FF);
  PWM0H = ((value & 0xFF00) >> 8);
  set_LOAD;
}


void set_PWM1(unsigned int value)
{
  PWM1L = (value & 0x00FF);
  PWM1H = ((value & 0xFF00) >> 8);
  set_LOAD;
}


void set_PWM2(unsigned int value)
{
  PWM2L = (value & 0x00FF);
  PWM2H = ((value & 0xFF00) >> 8);
  set_LOAD;
}


void set_PWM3(unsigned int value)
{
  PWM3L = (value & 0x00FF);
  PWM3H = ((value & 0xFF00) >> 8);
  set_LOAD;
}


void set_PWM4(unsigned int value)
{
  set_SFRPAGE;
  PWM4L = (value & 0x00FF);
  PWM4H = ((value & 0xFF00) >> 8);
  clr_SFRPAGE;
  set_LOAD;
}


void set_PWM5(unsigned int value)
{
  set_SFRPAGE;
  PWM5L = (value & 0x00FF);
  PWM5H = ((value & 0xFF00) >> 8);
  clr_SFRPAGE;
  set_LOAD;
```

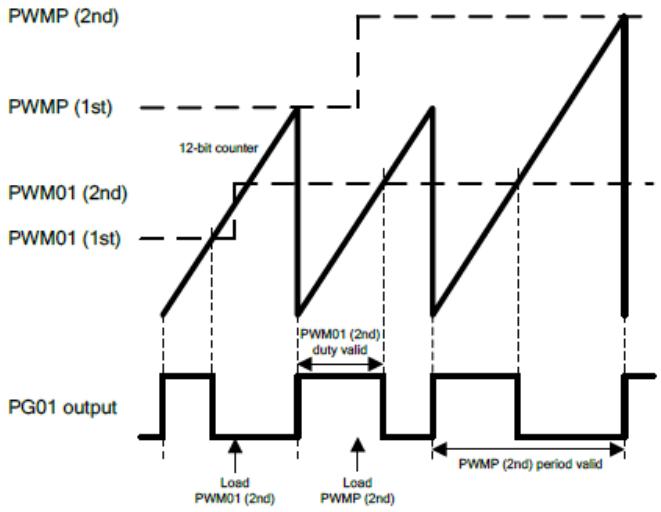
```
}


void set_PWM_dead_time(unsigned int value)
{
  unsigned char hb = 0;
  unsigned char lb = 0;

  lb = (value & 0x00FF);
  hb = ((value & 0x0100) >> 8);
  BIT_TMP = EA;

  EA = 0;
  TA = 0xAA;
  TA = 0x55;
  PDTEN &= 0xEF;
  PDTEN |= hb;
  PDTCNT = lb;
  EA = BIT_TMP;
}
```

Schematic



Explanation

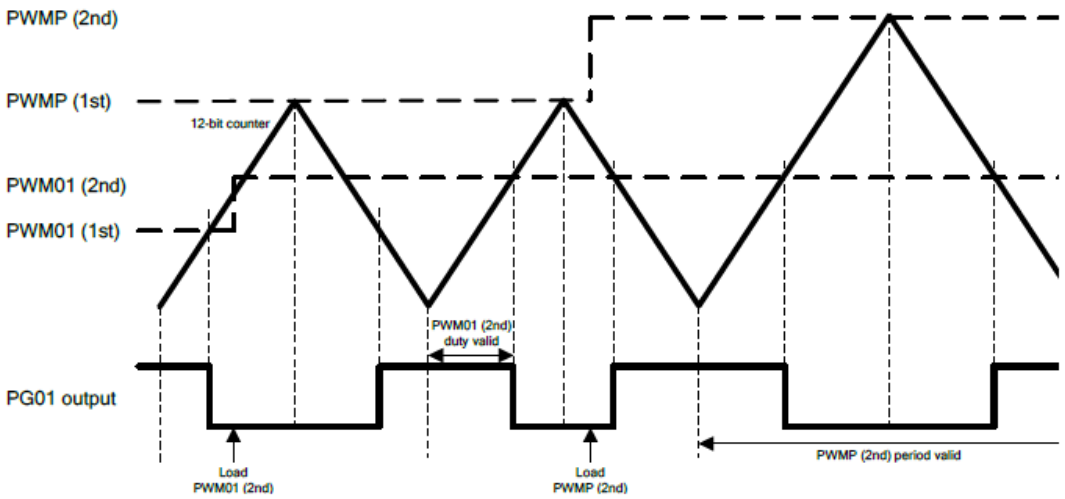
As discussed, there are two types of PWM in terms of count alignment. These are shown below:



PWM frequency = $\frac{F_{PWM}}{\{PWMPH,PWMPL\}+1}$ (F_{PWM} is the PWM clock source frequency divided by PWMDIV).

PWM high level duty = $\frac{\{PWMnH,PWMnL\}}{\{PWMPH,PWMPL\}+1}$.

Edge-Aligned PWM



PWM frequency = $\frac{F_{PWM}}{2\times\{PWMPH,PWMPL\}}$ (F_{PWM} is the PWM clock source frequency divided by PWMDIV).

PWM high level duty = $\frac{\{PWMnH,PWMnL\}}{\{PWMPH,PWMPL\}}$.

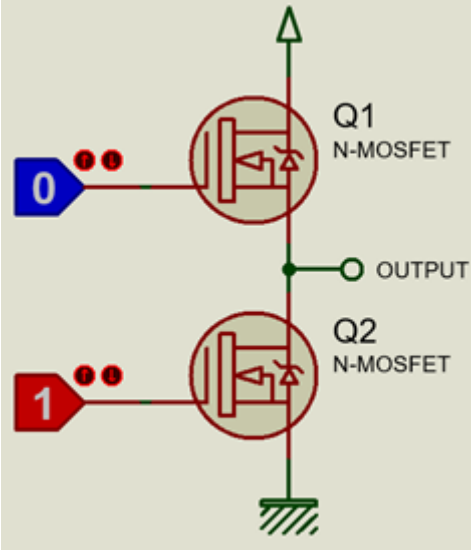
Centre-Aligned PWM

The first type was demonstrated in the simple PWM example. The second is demonstrated here. From the perspective of a general user, the difference in them don't affect much. Make sure that you check the formulae for each type before using.

The set up for complimentary PWM with dead time is no different from the simple PWM set up. As mentioned, there are a few minor differences. First the mode is set for complimentary PWM mode. Secondly, centre-aligned PWM is used here. The outputs are also set as inverted outputs, i.e. 100% duty cycle means minimum PWM value/count. The dead time period is set and implemented.

```
P11_PushPull_Mode;
P12_PushPull_Mode;

PWM0_P12_OUTPUT_ENABLE;
PWM1_P11_OUTPUT_ENABLE;

PWM_COMPLEMENTARY_MODE;
PWM_CENTER_TYPE;
set_CLRPWM;
PWM_CLOCK_FSYS;
PWM_CLOCK_DIV_64;
PWM0_OUTPUT_INVERSE;
PWM1_OUTPUT_INVERSE;
set_PWM_period(600);
set_PWM_dead_time(40);
PWM01_DEADTIME_ENABLE;
set_PWMRUN;
```

Now what is dead time in complimentary PWMs? Well simply is a short duration delay that is inserted between the polarity shifts of two PWMs in a group.



Consider a half-bridge MOSFET configuration as shown above. Perhaps it is the best way to describe the concept of dead-time. Surely, nobody would ever want to turn on both MOSFETs simultaneously in any condition and also during transition. Doing so would lead to a temporary short circuit between voltage source and ground, and would also lead to unnecessary heating of the MOSFETs and even permanent damage. By applying dead-time this can be avoided. In half/H-bridges, complimentary

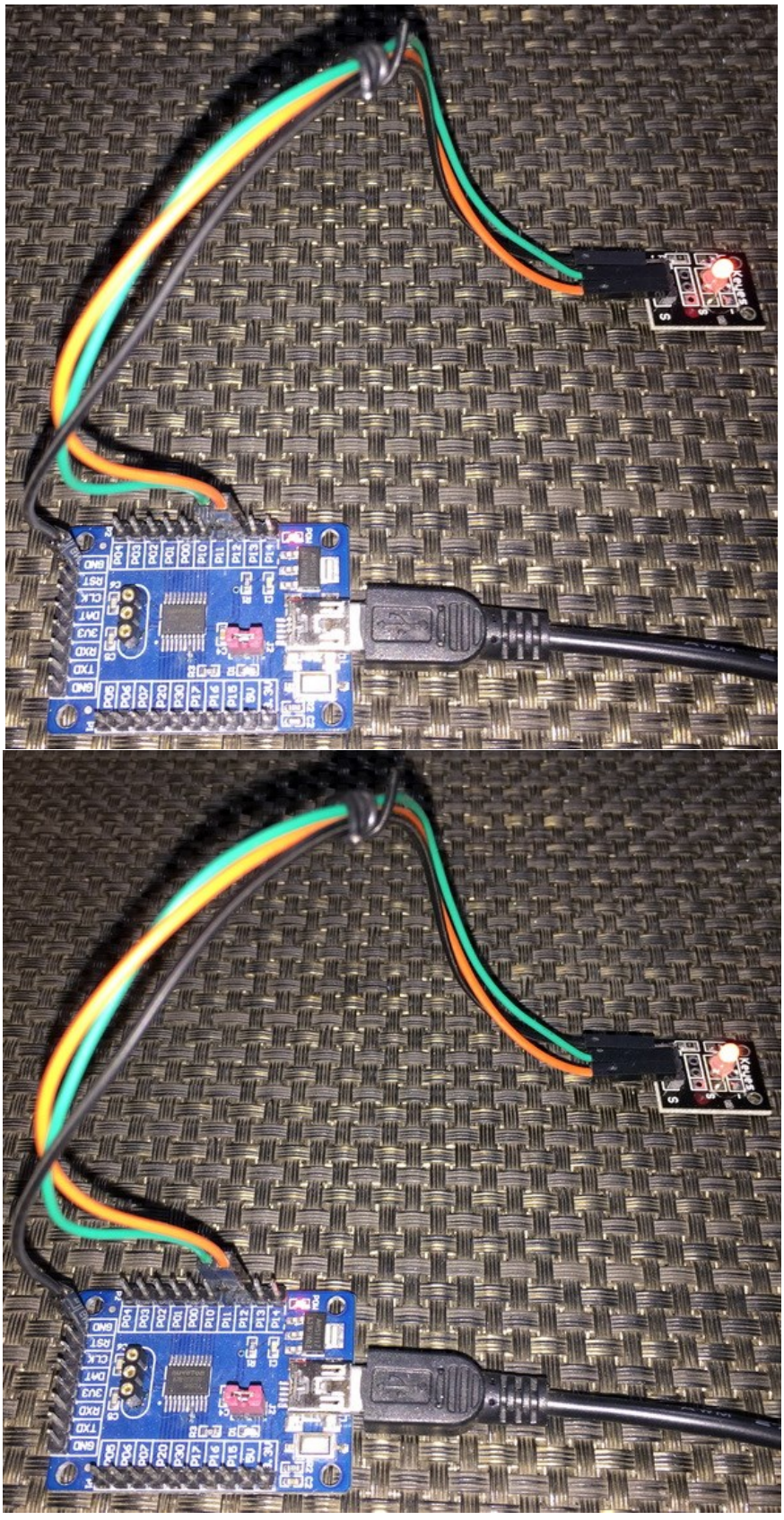
PWMs ensure that when one MOSFET is on, the other is off. However, at the edges of PWM polarity shifts, i.e. when one is rising while the other is falling, there is short but certain such short-circuit duration. If a dead time is inserted between the transitions, it will ensure that one MOSFET is only turned on when the other has been turned off fully.

Setting dead time requires us to disable TA protection. The function for setting dead time is shown below:

```
void set_PWM_dead_time(unsigned int value)
{
  unsigned char hb = 0;
  unsigned char lb = 0;

  lb = (value & 0x00FF);
  hb = ((value & 0x0100) >> 8);
  BIT_TMP = EA;

  EA = 0;
  TA = 0xAA;
  TA = 0x55;
  PDTEN &= 0xEF;
  PDTEN |= hb;
  PDTCNT = lb;
  EA = BIT_TMP;
}
```

Since complimentary PWMs work in groups, changing the duty cycle of one PWM channel will affect the other and so we don't need to change the duty cycles of both PWMs individually. This is why only one PWM channel is manipulated in the code.

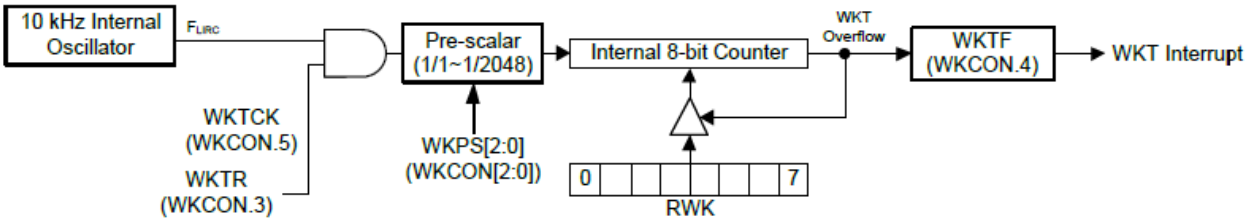
Demo



Demo video: https://youtu.be/0APOUon1IVE

Wakeup Timer and Power Modes

One of key feature of many modern era microcontrollers is a way to wake up a micro once it went to low power, sleep or idle mode. Like STM8s, N76E003 has this feature. The wakeup timer (WKT) is not a complex hardware. As the block diagram below shows it is just a timer-counter with ***LIRC*** as clock source. When the counter overflows, an interrupt is triggered. This interrupt wakes up the N76E003 chip.



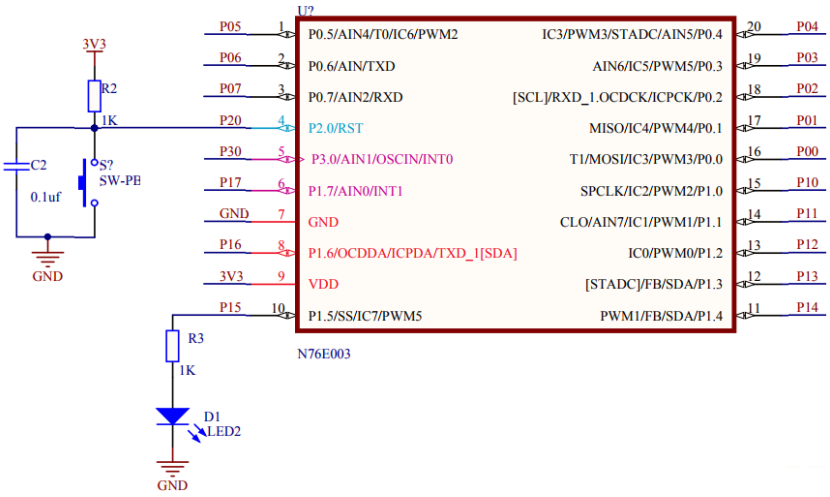
Wakeup timer is particularly very useful when used with low power modes. In many applications, we need to measure data on a fixed time basis while at other times when measurements are not taken, idle times are passed. These idle periods consume power and so if we use a wakeup timer with low power sleep mode, we can save energy. This is a big requirement for portable battery-operated devices. Examples of such portable devices include data-loggers, energy meters and smart watches.

Code

```
#include "N76E003_iar.h"
#include "Common.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "soft_delay.h"


#pragma vector = 0x8B
__interrupt void WKT_ISR(void)
{
  clr_WKTR;
  clr_WKTF;
}


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

  P15_PushPull_Mode;

  WKCON = 0x03;
  RWK = 0X00;
  set_EWKT;
  set_EA;

  while(1)
  {
```

```
    for(s = 0; s < 9; s++)
    {
      P15 = ~P15;
      delay_ms(100);
    }

    set_WKTR;
    set_PD;

    for(s = 0; s <= 9; s++)
    {
      P15 = ~P15;
      delay_ms(300);
    }

    set_WKTR;
    set_PD;
  };
}
```

Schematic



Explanation

In the demo demonstrated here, P15 LED is toggle fast for nine times. This marks the start of the demo. After this toggling has been done, both the wakeup timer and the power down modes are turned on.

```
set_WKTR;
set_PD;
```

During ***Power Down*** time, all operations inside the micro are suspended. There is also another low power mode called ***Idle Mode***. Both are similar but in idle mode peripherals are kept active.

PCON – Power Control

7	6	5	4	3	2	1	0
SMOD	SMOD0	-	POF	GF1	GF0	PD	IDL
R/W	R/W	-	R/W	R/W	R/W	R/W	R/W

Address: 87H Reset value: see Table 6-2. SFR Definitions and Reset Values

Bit	Name	Description
1	PD	**Power-down mode** Setting this bit puts CPU into Power-down mode. Under this mode, both CPU and peripheral clocks stop and Program Counter (PC) suspends. It provides the lowest power consumption. After CPU is woken up from Power-down, this bit will be automatically cleared via hardware and the program continue executing the interrupt service routine (ISR) of the very interrupt source that woke the system up before. After return from the ISR, the device continues execution at the instruction, which follows the instruction that put the system into Power-down mode. Note that If IDL bit and PD bit are set simultaneously, CPU will enter Power-down mode. Then it does not go to Idle mode after exiting Power-down.
0	IDL	**Idle mode** Setting this bit puts CPU into Idle mode. Under this mode, the CPU clock stops and Program Counter (PC) suspends but all peripherals keep activated. After CPU is woken up from Idle, this bit will be automatically cleared via hardware and the program continue executing the ISR of the very interrupt source that woke the system up before. After return from the ISR, the device continues execution at the instruction which follows the instruction that put the system into Idle mode.

After the first set of LED toggles and power down mode, it takes about 1.5 seconds (according to the prescalar value of 64 and 256 counts) for the wakeup timer to get triggered.

$$Wakeup\ time = \frac{Prescalar \times Counter\ Value}{LIRC\ Frequency} = \frac{64 \times 256}{10000} \approx 1.5s$$ *(considering variations in LIRC)*

When the interrupt kicks in, the MCU is brought back to working mode and the wakeup timer is disabled. Again, the P15 LED is toggled. This time the LED is toggled at a slower rate, indicating the continuation of the rest of the tasks after power down. After this the process is repeated with power down and wake up timer reenabled.
Setting up the wakeup timer requires the same stuffs that we need during a timer configuration, i.e. a prescalar value, counter value and interrupt. These three settings are done by the following lines of code:

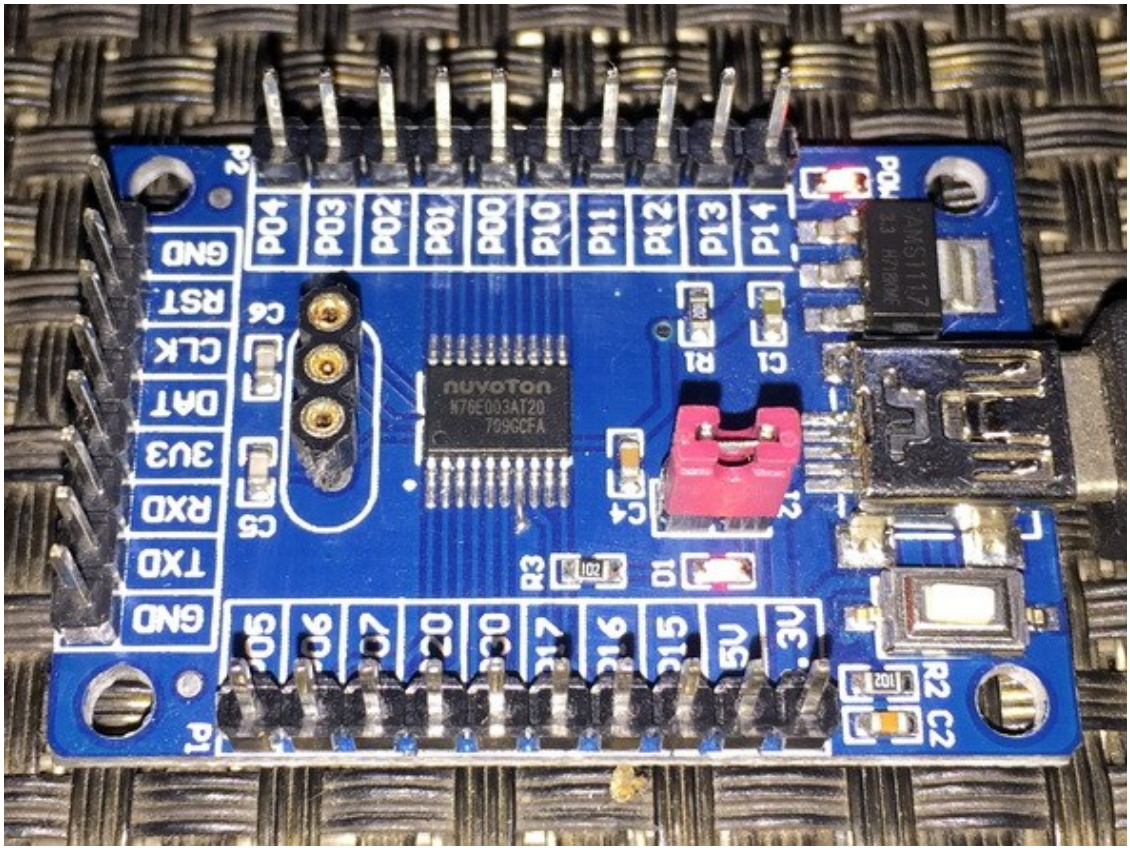
```
WKCON = 0x03;
RWK = 0X00;
set_EWKT;
set_EA;
```

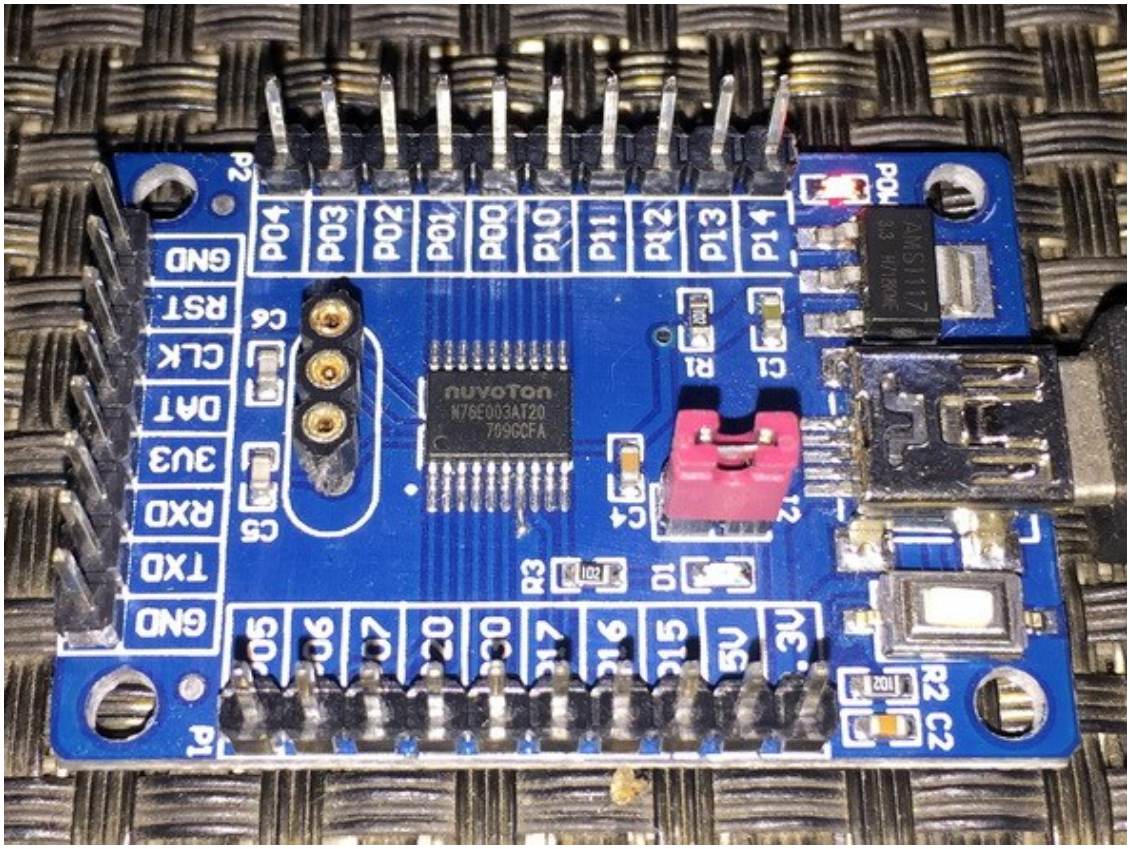
Since interrupt is used, WKT interrupt will be triggered when its counter rolls over.

```
#pragma vector = 0x8B
__interrupt void WKT_ISR(void)
{
  clr_WKTR;
  clr_WKTF;
}
```

When WKT interrupt is triggered, its interrupt flag should be cleared in the software.

Demo

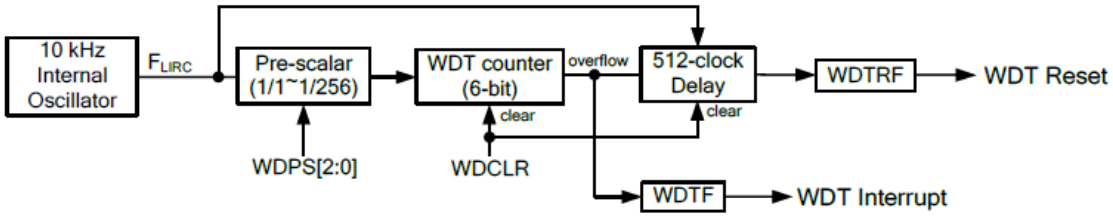




Demo video: https://youtu.be/ZyQZmJB3FRI

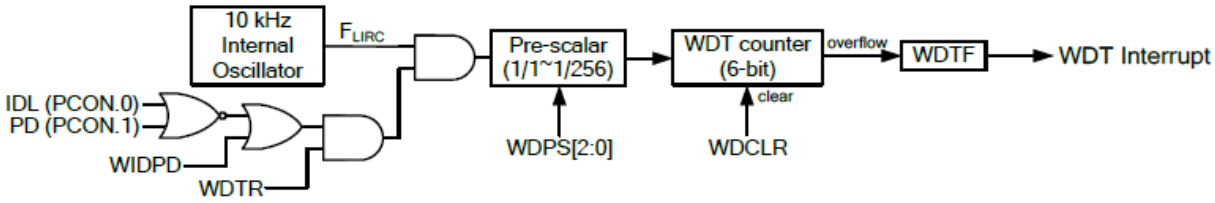
Watchdog Timer

The watchdog timer (WDT) of N76E003 is just a reset-issuing timer and the purpose of this timer is to recover a N76E003 micro from an unanticipated event/loop that may result in unresponsive or erratic behaviour. It is clocked with LIRC oscillator and this makes it independent from main clock (HIRC or ECLK) failure.



LIRC is prescaled and feed to a counter. When the counter overflows, WDT interrupt is issued, furthermore a reset is also generated based on delay. In normal running condition, the counter must be periodically reset to 0 count to avoid reset. If for some reason this is not done then a reset will be triggered.

The watchdog timer of N76E003 can also be used as a 6-bit general-purpose timer. However, this usually not used as such. The only difference between using it as timer and as a watchdog timer is the reset part.

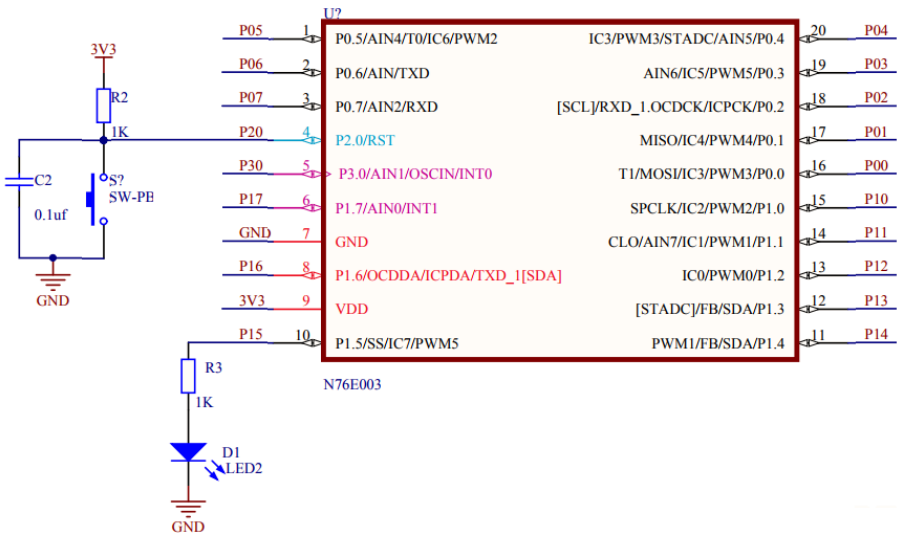


Code

```
#include "N76E003.h"
#include "Common.h"
#include "Delay.h"
#include "SFR_Macro.h"
#include "Function_define.h"


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

    P15_PushPull_Mode;

    Timer0_Delay1ms(1000);

    for(s = 0; s <= 9; s++)
    {
        P15 = ~P15;
        Timer0_Delay1ms(60);
```

```
    }

    TA = 0xAA;
    TA = 0x55;
    WDCON = 0x07;
    set_WDCLR;
    while((WDCON | ~SET_BIT6) == 0xFF);
    EA = 1;
    set_WDTR;

    while(1)
    {
        for(s = 0; s <= 9; s++)
        {
            P15 = ~P15;
            Timer0_Delay1ms(200);
        }
        while(1);
    }
}
```

Schematic



Explanation

For demoing WDT, again the P15 LED is used. The code starts with it toggling states ten times. This marks the beginning of the code and the interval prior to WDT configuration.

Next the WDT is setup. Note that the WDT is ***Timed-Access (TA)*** protected. TA protection protects crucial hardware like brownout detection circuit and WDT hardware from erratic writes.

TA – Timed Access

7	6	5	4	3	2	1	0
TA[7:0]							
W							

Address: C7H Reset value: 0000 0000b

Bit	Name	Description
7:0	TA[7:0]	**Timed access** The timed access register controls the access to protected SFRs. To access protected bits, user should first write AAH to the TA and immediately followed by a write of 55H to TA. After these two steps, a writing permission window is opened for 4 clock cycles during this period that user may write to protected SFRs.

For using the TA hardware, we need not to follow any special procedure. To enable access to some hardware that are TA protected like the WDT all we have to do is to write the followings to TA register:

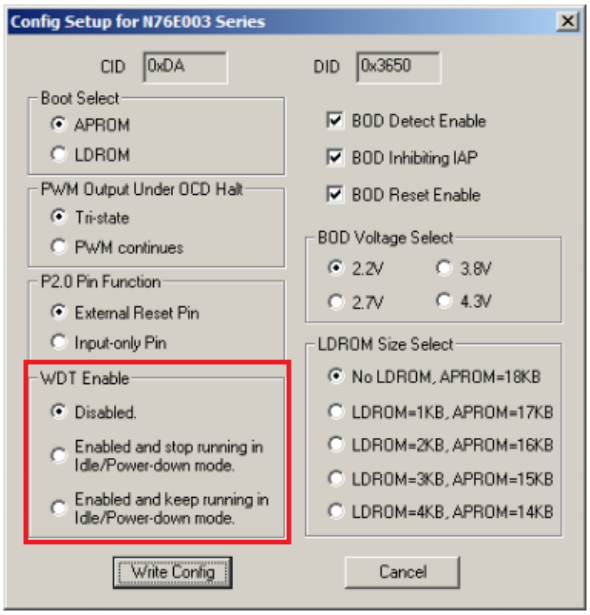
```
TA = 0xAA;
TA = 0x55;
```

Without TA protection access, any changes to made to the registers of the TA protected hardware is unaffected.

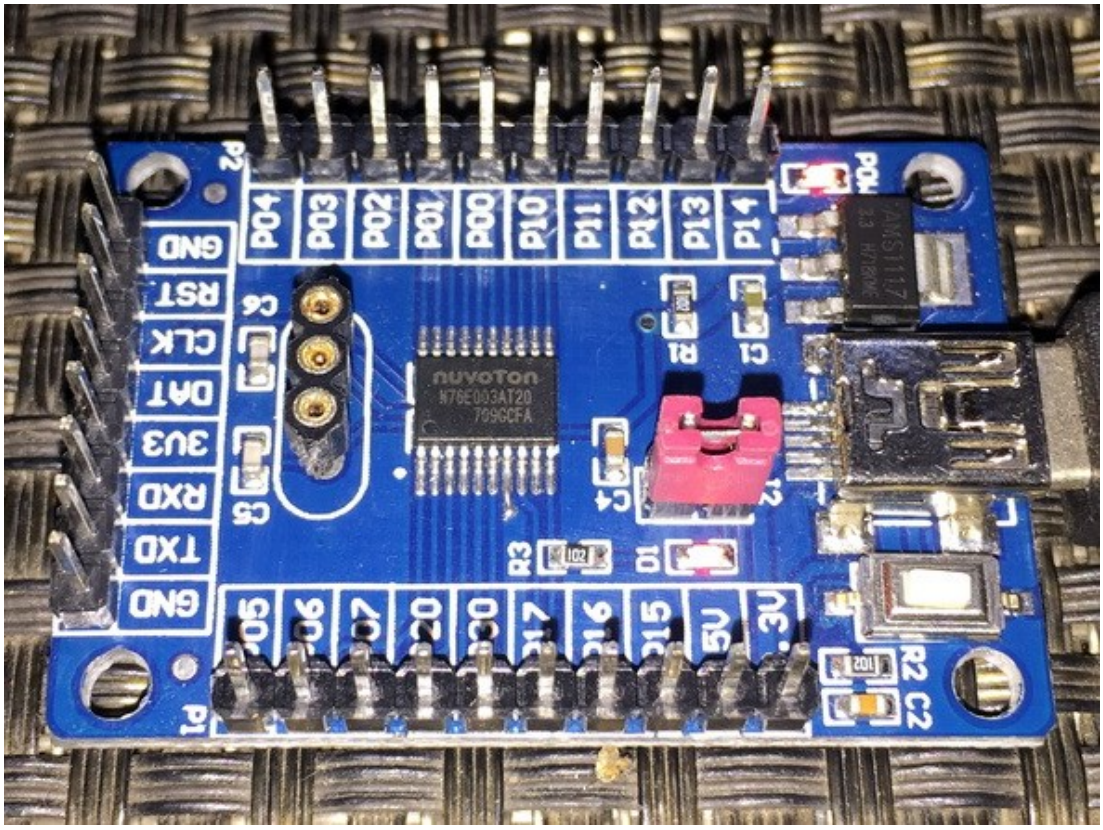
After enabling access to WDT, we can set it up. Setting the WDT requires us mainly to setup LIRC clock prescalar. Once the prescalar is set we are good to go for enabling the WDT.

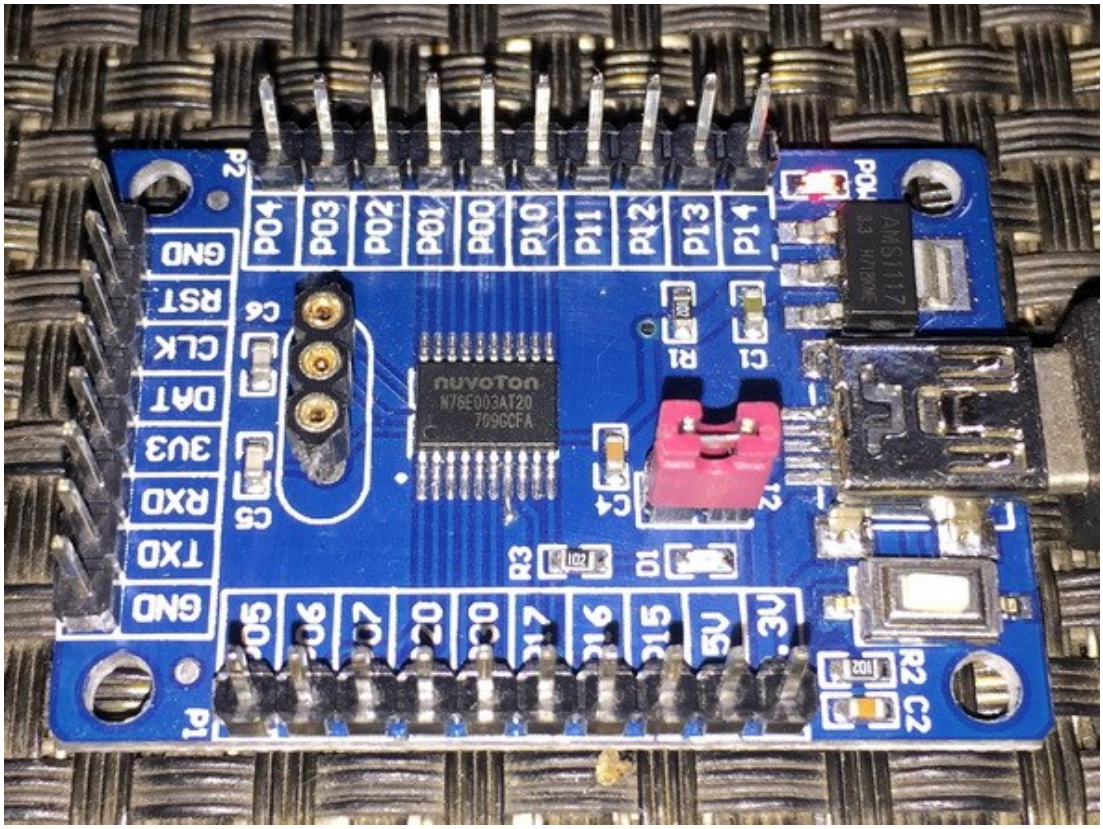
```
WDCON = 0x07;
set_WDCLR;
while((WDCON | ~SET_BIT6) == 0xFF);
set_EA;
set_WDTR;
```

Before all these can be done, we have to enable the WDT hardware either through ***In-Application Programming (IAP)*** or by configuration bits during programming.



Demo





Demo video: https://youtu.be/h8LeHpnA2vY

Communication Peripheral Overview

N76E003 packs basic serial communication peripherals for I2C, SPI and UART. These interfaces enable us to interface external EEPROM and flash memories, real-time clocks (RTC), sensors, etc.

Comm.	*Description*	*I/O*	*Max. Speed*	*Max. Distance*	*Max. Possible Number of Devices in a Bus*
UART	Asynchronous serial point-to-point communication	2	115.2kbps	15m	2 (Point-to-Point)
SPI	Short-range synchronous master-slave serial communication	3/4	4Mbps	0.1m	Virtually unlimited
I2C	Short-range synchronous master-slave serial communication using one data and one clock line	2	1Mbps	0.5m	127
RS-485	Asynchronous differential two wire serial communication	2	115.2kbps	1.2km	Several

N76E003 has two hardware UARTs, one SPI and one I2C peripheral. The UART interfaces can be further expanded to implement RS-485 and RS-422 communication protocols. With ordinary GPIOs and optionally with hardware timers we can implement software-based UART, SPI and I2C too. However, software methods are slow and resource-hungry. Software-based approach is, on the other hand, needed when we have to interface devices that don't use any of these communication topologies. For example, the one-wire protocol used by DHT series relative humidity and temperature sensors needs to be implemented by using an ordinary GPIO pin. Same goes for typical text and graphical LCDs.

Comm.	*No. of Peripheral Blocks*	*No of GPIOs Used*	*Default GPIO Pins*	*Alternative GPIO Pins*
UART	2 UART0 UART1	2	TXD0 = P06 RXD0 = P07 TXD1 = P16 RXD1 = P02	N/A
SPI	1	3 or 4	MOSI = P00 MISO = P01 SCK = P10 SS = P15	N/A SS = Other GPIO pin if hardware-based SS pin is not used
I2C	1	2	SCL = P13 SDA = P14	SCL = P02 SDA = P16

Note that there is a conflict between UART1 and alternative I2C pins. There are also similar conflicts with other hardware peripherals like PWMs, ADCs, etc since N76E003 packs lots of stuff in such a small form-factor. Every single GPIO pin is precious. Try to keep things with default GPIO pins in order to reduce conflicts with other hardware peripherals, coding and to maximize effective use of GPIOs. Likewise bear in mind that timers have also such conflicts as they are used to implement both hardware-based delays and the UART peripherals. You have to know for sure what you are doing, what do you want and with which stuffs you wish to accomplish your task.

Serial Communication - UART

To date serial communication (UART) is perhaps the simplest and widely used form of communication in use. UART block is only block available in most microcontrollers that can be easily interfaced with a computer or a phone. Most communication modules like Bluetooth, Wi-Fi, GSM modems, GPS modules, etc are interfaced using UART. It has incredible range and is also the backbone of other communication methods like RS-485, RS-422, etc.

To learn more about UART visit the following link:

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

Code

HMC1022.h

```
#define Get_Angular_Measurement                     0x31
#define Start_Calibration                           0xC0
#define End_Calibration                             0xC1
#define Set_Magnetic_Declination_High_Byte          0x03
#define Set_Magnetic_Declination_Low_Byte           0x04

#define no_of_data_bytes_returned                   0x08

#define calibration_LED                             P15


void read_heading(void);
void calibrate_compass(void);
void factory_reset(void);
void set_declination_angle(unsigned long angle);
```

HMC1022.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "soft_delay.h"
#include "HMC1022.h"


unsigned char done = 0;
unsigned char data_bytes[no_of_data_bytes_returned] = {0x00, 0x00, 0x00, 0x00,
0x00, 0x00};


void read_heading(void)
{
     unsigned char s = 0;
     unsigned long CRC = 0;

     Send_Data_To_UART1(Get_Angular_Measurement);

     for(s = 0; s < no_of_data_bytes_returned; s++)
     {
        data_bytes[s] = Receive_Data_From_UART1();
        if(s < (no_of_data_bytes_returned - 1))
        {
            CRC += data_bytes[s];
        }
     }

     CRC = (CRC & 0xFF);

     if(CRC == data_bytes[7])
     {
       done = 1;
     }
}


void calibrate_compass(void)
{
 unsigned char s = 0x00;

 Send_Data_To_UART1(Start_Calibration);

 for(s = 0; s < 60; s++)
 {
     calibration_LED = 1;
     delay_ms(100);
     calibration_LED = 0;
     delay_ms(900);
 }

 for(s = 0; s < 60; s++)
```

```
 {
     calibration_LED = 1;
     delay_ms(400);
     calibration_LED = 0;
     delay_ms(600);
 }

 Send_Data_To_UART1(End_Calibration);
}


void factory_reset(void)
{
 Send_Data_To_UART1(0xA0);
 Send_Data_To_UART1(0xAA);
 Send_Data_To_UART1(0xA5);
 Send_Data_To_UART1(0xC5);
}


void set_declination_angle(unsigned long angle)
{
 unsigned long hb = 0;
 unsigned char lb = 0;

 lb = (angle & 0x00FF);

 hb = (angle & 0xFF00);
 hb >>= 8;

 Send_Data_To_UART1(Set_Magnetic_Declination_High_Byte);
 Send_Data_To_UART1(hb);

 Send_Data_To_UART1(Set_Magnetic_Declination_Low_Byte);
 Send_Data_To_UART1(lb);
}
```

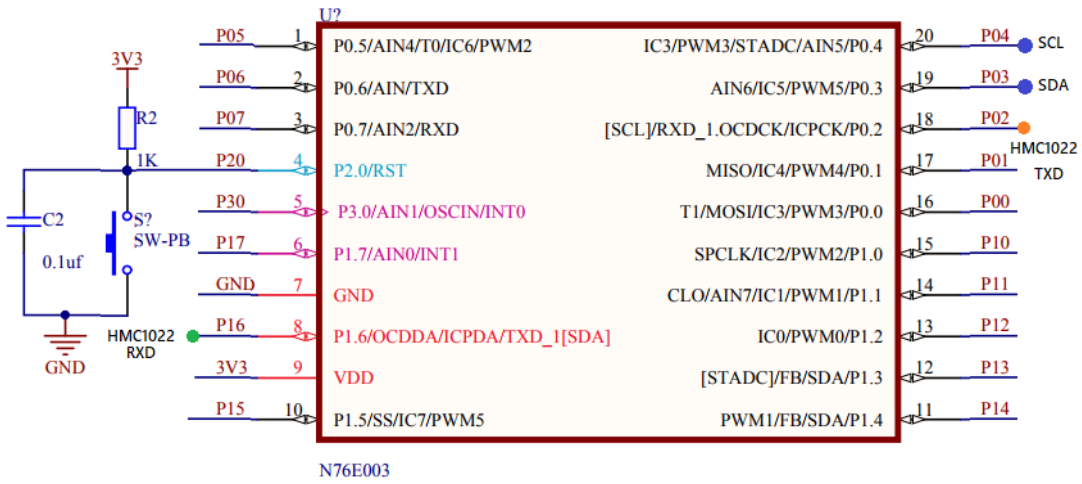
main.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "soft_delay.h"
#include "LCD_2_Wire.h"
#include "HMC1022.h"


const unsigned char symbol[8] =
{
   0x00, 0x06, 0x09, 0x09, 0x06, 0x00, 0x00, 0x00
};


extern unsigned char done;
```

```
extern unsigned char data_bytes[no_of_data_bytes_returned];


void setup(void);
void lcd_symbol(void);


void main(void)
{
  setup();

  while(1)
  {
    read_heading();

    if(done)
     {
       LCD_goto(6, 1);
       LCD_putchar(data_bytes[2]);

       LCD_goto(7, 1);
       LCD_putchar(data_bytes[3]);

       LCD_goto(8, 1);
       LCD_putchar(data_bytes[4]);

       LCD_goto(9, 1);
       LCD_putchar('.');
       LCD_goto(10, 1);
       LCD_putchar(data_bytes[6]);

       done = 0;
     }

     P15 = ~P15;
     delay_ms(200);
  };
}


void setup(void)
{
  P15_PushPull_Mode;

  LCD_init();
  LCD_clear_home();
  lcd_symbol();
  LCD_goto(3, 0);
  LCD_putstr("Heading  N");
  LCD_goto(11, 0);
  LCD_send(0, DAT);

  InitialUART1_Timer3(9600);
}
```

```
void lcd_symbol(void)
{
  unsigned char s = 0;

  LCD_send(0x40, CMD);

  for(s = 0; s < 8; s++)
  {
   LCD_send(symbol[s], DAT);
  }

  LCD_send(0x80, CMD);
}
```

Schematic



Explanation

The following are the commands that HMC1022 acknowledges when sent via UART:

Commands (Hex)	Description
0x31	Get Angular Measurement
0xC0	Start Calibration
0xC1	End Calibration
0xA0-0xAA-0xA5-0xC5	Reset Module to Factory Default
0xA0-0xAA-0xA5-I2C_ADDR	Change Module's I2C Address*
0x03 + 8bit Data (High)	Set Magnetic Declination (High Byte)
0x04 + 8bit Data (Low)	Set Magnetic Declination (Low Byte)

HMC1022 gives heading data when it is requested. The highlighted command is that command.

Once requested it sends out a string of characters that carry heading information. The followings are the response package and its details:

Package Format

Byte 0	Byte 1	Byte 2 … Byte 6	Byte 7
CrLf (New Line)		Data	Checksum

Package Details

Byte	Response (Hex)	Response (ASCII equivalent)	Description
0	0x0D	CR (Carriage Return)	Part of New Line (CR + LF = New Line)
1	0x0A	LF (Line Feed)	Part of New Line (CR + LF = New Line)
2	0x30 ~ 0x33	0 ~ 3	Angle Value (Hundreds; 100s)
3	0x30 ~ 0x39	0 ~ 9	Angle Value (Tens; 10s)
4	0x30 ~ 0x39	0 ~ 9	Angle Value (Units; 1s)
5	0x2E	.	Period/ Decimal Point
6	0x30 ~ 0x39	0 ~ 9	Angle Value (Decimal)
7	0x00 ~ 0xFF		Checksum (Sum of Byte 0 to 6) Lower Byte

From the above info, it is clear that HMC1022 will return 8 bytes when requested for heading. Out of these we need bytes 2, 3,4 and 6. The rest can be ignored for simplicity.

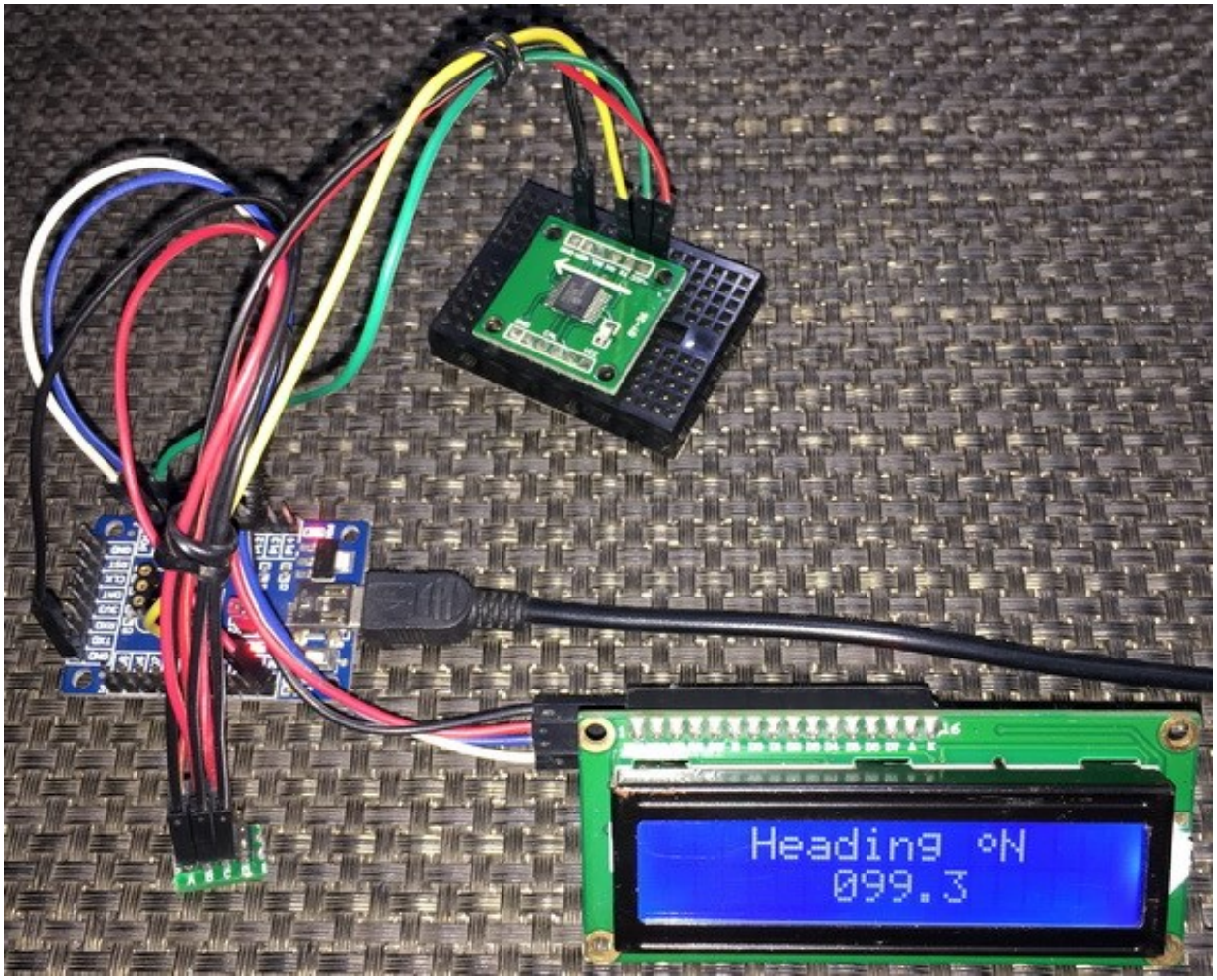
Data from HM1022 is received by the following function:

```
void read_heading(void)
{
     unsigned char s = 0;
     unsigned long CRC = 0;

     Send_Data_To_UART1(Get_Angular_Measurement);

     for(s = 0; s < no_of_data_bytes_returned; s++)
     {
        data_bytes[s] = Receive_Data_From_UART1();
        if(s < (no_of_data_bytes_returned - 1))
        {
            CRC += data_bytes[s];
        }
     }

     CRC = (CRC & 0xFF);

     if(CRC == data_bytes[7])
     {
       done = 1;
     }
}
```

Note that the function ***Send_Data_To_UART1*** and ***Receive_Data_From_UART1*** are BSP functions. Likewise, ***InitialUART1_Timer3(9600)*** function is also a BSP function that initiates UART1 with Timer 3. As with ADC, BSP functions for UART are enough for basic setup. Try to use Timer 3 as it remains mostly free. If you need more control over the UART, you can manually configure the registers.

In the main, the received bytes containing heading info are displayed.

Demo



Demo video: https://youtu.be/Y0jehv58ugE

UART Interrupt

UART, as we saw in the last example, can be used in polling mode but it is wise to use it in interrupt mode. This feature becomes especially important and therefore, a must-have requirement when it come to that fact that the host micro may not always know when to receive data. Unlike SPI/I2C, UART doesn't always follow known data transactions. Take the example of a GSM modem. We and so does our tiny host N76E003 micro don't know for sure when a message would arrive. Thus, when an interrupt due to reception is triggered, it is known to mark the beginning of data reception process. The host micro can relax idle until data is received fully.

Here, we will see how to use UART interrupt to read a LV-Max EZ0 SONAR module.

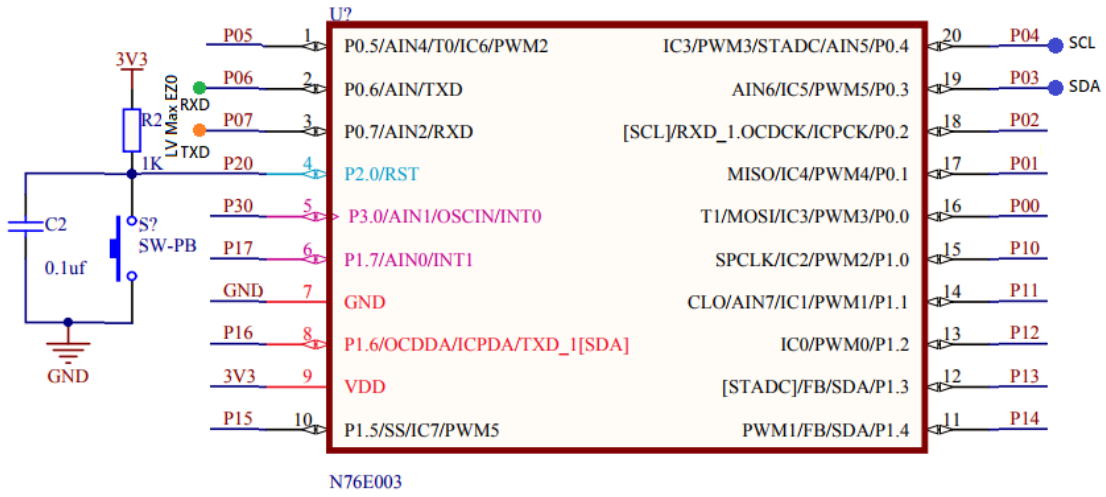
Code

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "soft_delay.h"
#include "LCD_2_Wire.h"


unsigned char received = 0;
unsigned char count = 0;
unsigned char buffer[5] = {0x00, 0x00, 0x00, 0x00, 0x00};
```

```
void setup(void);


void UART0_ISR(void)
interrupt 4
{
    if(RI == 1)
    {
        buffer[count] = SBUF;
        count++;

        if(count >= 4)
        {
            clr_ES;
            received = 1;
        }

        clr_RI;
    }

    P15 = ~P15;
}


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

    setup();

    while(1)
    {
        Send_Data_To_UART0('S');
        delay_ms(40);
        set_ES;

        if(received)
        {

            for(i = 1; i < 4; i++)
            {
              LCD_goto((12 + i), 1);
              LCD_putchar(buffer[i]);
            }

            count = 0;
            received  = 0;
            set_ES;
        }

        delay_ms(40);
    }
}
```

```
void setup(void)
{
    P15_PushPull_Mode;

    LCD_init();
    LCD_clear_home();

    LCD_goto(1, 0);
    LCD_putstr("UART Interrupt");
    LCD_goto(0, 1);
    LCD_putstr("Range/Inch:");

    InitialUART0_Timer3(9600);
    set_EA;
}
```

Schematic



**Note the LV-Max EZ0 SONAR sensor is not directly connected with the N76E003 as shown in the schematic but rather it is connected via a 74HC04 hex inverter. Two of these inverters are used – one for the TX side and the other for the RX side.*

Explanation

This time UART0 is used and is setup using BSP built-in function. The only exception is the interrupt part. The global interrupt is enabled but not the UART interrupt.

```
InitialUART0_Timer3(9600);
set_EA;
```

The UART interrupt is only enabled after commanding the SONAR sensor:

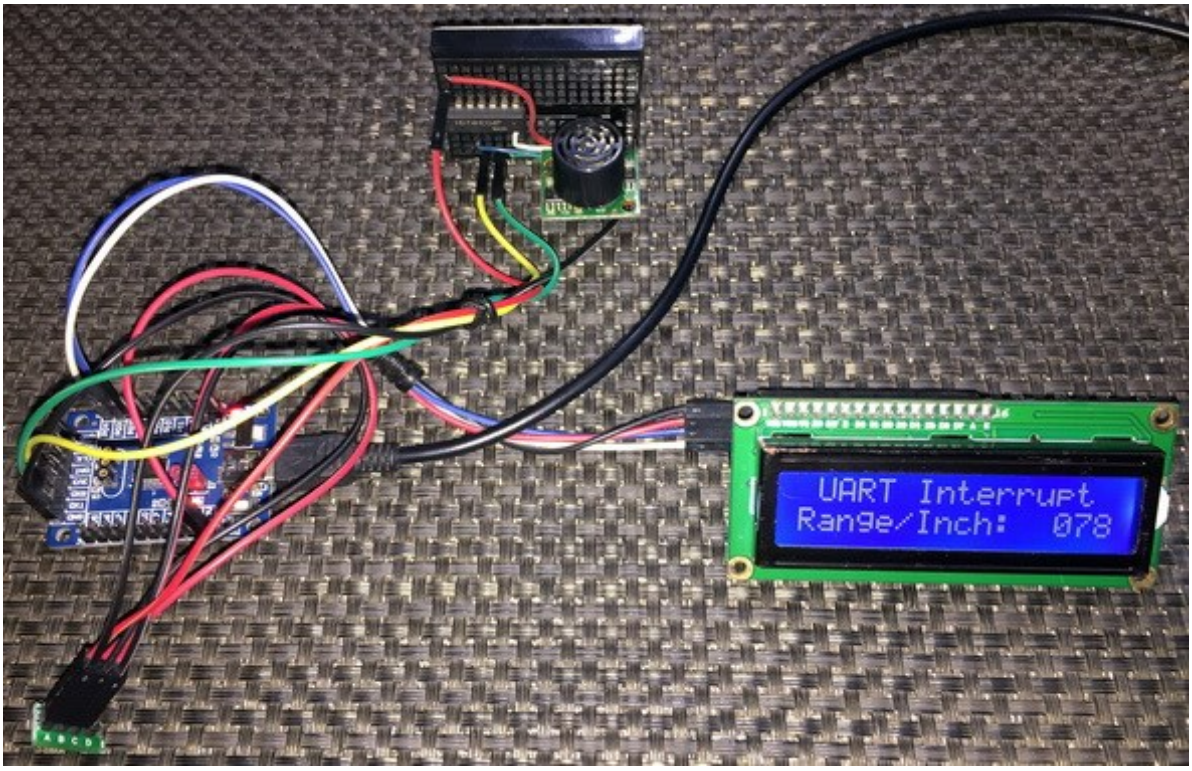
```
Send_Data_To_UART0('S');
delay_ms(40);
set_ES;
```

This is done to avoid unnecessary reads.

The sensor is read inside the interrupt routine. The sensor outputs data as ***Rxxx<CR>***. So, there are 5 bytes to read. The ***"R"*** in the readout works as a preamble or sync byte. The ***"xxx"*** part contains range info in inches. Finally, ***<CR>*** is a carriage return character. Therefore, every time a character is received an interrupt is triggered and the character is saved in an array. When all 5 bytes have been received, the display is updated. On exiting the interrupt, the interrupt flag is cleared. P15 is also toggled to visually indicate that an UART interrupt has been triggered.

```
void UART0_ISR(void)
interrupt 4
{
  if(RI == 1)
  {
     buffer[count] = SBUF;
     count++;

     if(count >= 4)
     {
       clr_ES;
       received = 1;
     }

     clr_RI;
  }

  P15 = ~P15;
}
```

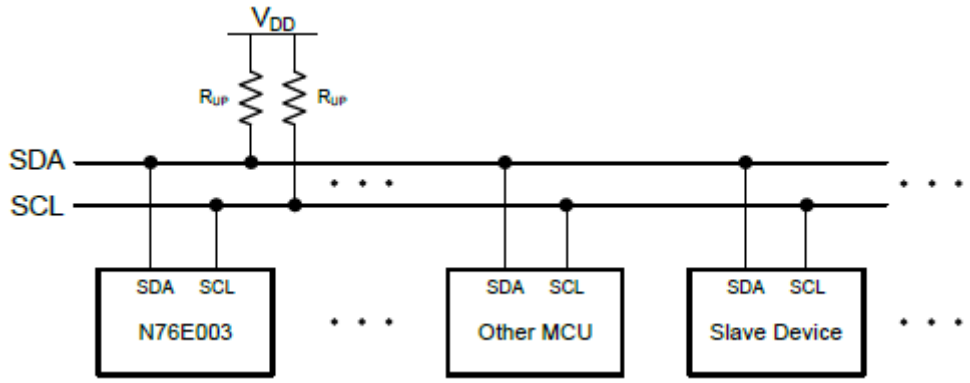
Demo



Demo video: https://youtu.be/SCsPt88HFVk

Inter-Integrated Circuit (I2C) – Interfacing DS1307 I2C RTC

Developed by Philips nearly three decades ago, I2C communication (***Inter-Integrated Circuit***) is widely used in a number of devices and is comparatively easier than SPI. For I2C communication only two wires – *serial data (SDA)* and *serial clock (SCK)* are needed and these two wires form a bus in which we can connect up to 127 devices.



Everything you need to know about I2C can be found in these pages:

- 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

Apart from these N76E003's datasheet explains I2C communication in high detail.

Code

I2C.h

```
#define regular_I2C_pins                0
#define alternate_I2C_pins              1

#define regular_I2C_GPIOs()             do{P13_OpenDrain_Mode;
P14_OpenDrain_Mode; clr_I2CPX;}while(0)

#define alternative_I2C_GPIOs()         do{P02_OpenDrain_Mode;
P16_OpenDrain_Mode; set_I2CPX;}while(0)

#define I2C_GPIO_Init(mode)             do{if(mode !=
0){alternative_I2C_GPIOs();}else{regular_I2C_GPIOs();}}while(0)

#define I2C_CLOCK                       0x27 //Fclk = Fsys / (4*(prescalar + 1))
```

```
#define I2C_ACK                         0
#define I2C_NACK                        1

#define timeout_count                   1000

void I2C_init(void);
void I2C_start(void);
void I2C_stop(void);
unsigned char I2C_read(unsigned char ack_mode);
void I2C_write(unsigned char value);
```

I2C.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "I2C.h"

void I2C_init(void)
{
    I2C_GPIO_Init(regular_I2C_pins);
    I2CLK = I2C_CLOCK;
    set_I2CEN;
}


void I2C_start(void)
{
    signed int t = timeout_count;

    set_STA;
    clr_SI;
    while((SI == 0) && (t > 0))
    {
       t--;
    };
}

void I2C_stop(void)
{
    signed int t = timeout_count;

    clr_SI;
    set_STO;
    while((STO == 1) && (t > 0))
    {
        t--;
    };
}


unsigned char I2C_read(unsigned char ack_mode)
{
```

```
    signed int t = timeout_count;
    unsigned char value = 0x00;

    set_AA;
    clr_SI;
    while((SI == 0) && (t > 0))
    {
        t--;
    };

    value = I2DAT;

    if(ack_mode == I2C_NACK)
    {
        t = timeout_count;
        clr_AA;
        clr_SI;
        while((SI == 0) && (t > 0))
        {
            t--;
        };
    }

    return value;
}


void I2C_write(unsigned char value)
{
    signed int t = timeout_count;

    I2DAT = value;
    clr_STA;
    clr_SI;
    while((SI == 0) && (t > 0))
    {
        t--;
    };
}
```

DS1307.h

```
#define I2C_W                           0
#define I2C_R                           1

#define sec_reg                         0x00
#define min_reg                         0x01
#define hr_reg                          0x02
#define day_reg                         0x03
#define date_reg                        0x04
#define month_reg                       0x05
#define year_reg                        0x06
#define control_reg                     0x07

#define DS1307_addr                     0xD0
#define DS1307_WR                       (DS1307_addr + I2C_W)
```

```
#define DS1307_RD                           (DS1307_addr + I2C_R)


void DS1307_init(void);
unsigned char DS1307_read(unsigned char address);
void DS1307_write(unsigned char address, unsigned char value);
unsigned char bcd_to_decimal(unsigned char value);
unsigned char decimal_to_bcd(unsigned char value);
void get_time(void);
void set_time(void);
```

DS1307.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "DS1307.h"
#include "I2C.h"


struct
{
   unsigned char s;
   unsigned char m;
   unsigned char h;
   unsigned char dy;
   unsigned char dt;
   unsigned char mt;
   unsigned char yr;
}time;


void DS1307_init(void)
{
    I2C_init();
  DS1307_write(control_reg, 0x00);
}


unsigned char DS1307_read(unsigned char address)
{
    unsigned char value = 0x00;

    I2C_start();
    I2C_write(DS1307_WR);
    I2C_write(address);

    I2C_start();
    I2C_write(DS1307_RD);
    value = I2C_read(I2C_NACK);
    I2C_stop();

    return value;
}
```

```
void DS1307_write(unsigned char address, unsigned char value)
{
    I2C_start();
    I2C_write(DS1307_WR);
    I2C_write(address);
    I2C_write(value);
    I2C_stop();
}


unsigned char bcd_to_decimal(unsigned char value)
{
    return ((value & 0x0F) + (((value & 0xF0) >> 0x04) * 0x0A));
}


unsigned char decimal_to_bcd(unsigned char value)
{
    return (((value / 0x0A) << 0x04) & 0xF0) | ((value % 0x0A) & 0x0F);
}


void get_time(void)
{
    time.s = DS1307_read(sec_reg);
    time.s = bcd_to_decimal(time.s);

    time.m = DS1307_read(min_reg);
    time.m = bcd_to_decimal(time.m);

    time.h = DS1307_read(hr_reg);
    time.h = bcd_to_decimal(time.h);

    time.dy = DS1307_read(day_reg);
    time.dy = bcd_to_decimal(time.dy);

    time.dt = DS1307_read(date_reg);
    time.dt = bcd_to_decimal(time.dt);

    time.mt = DS1307_read(month_reg);
    time.mt = bcd_to_decimal(time.mt);

    time.yr = DS1307_read(year_reg);
    time.yr = bcd_to_decimal(time.yr);
}


void set_time(void)
{
    time.s = decimal_to_bcd(time.s);
    DS1307_write(sec_reg, time.s);

    time.m = decimal_to_bcd(time.m);
    DS1307_write(min_reg, time.m);
```

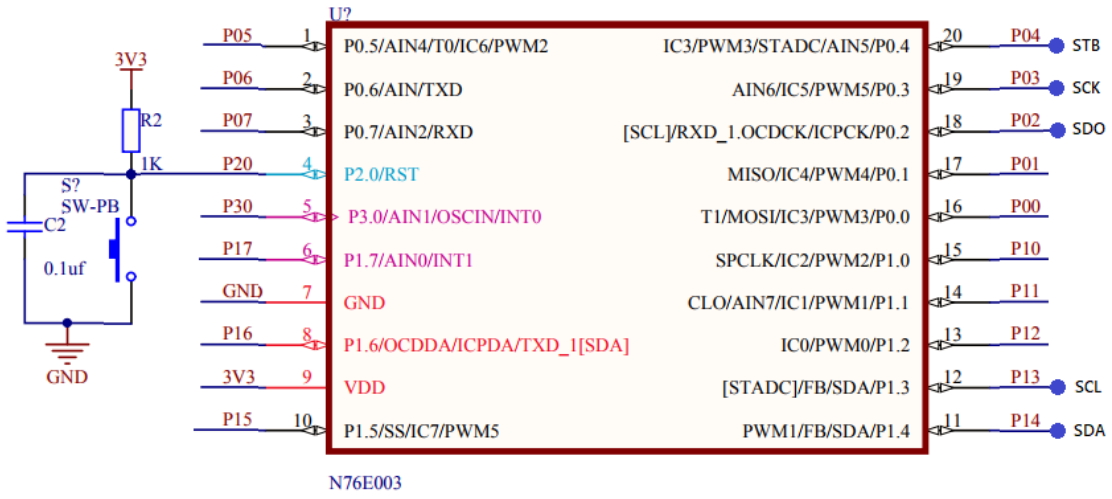
```
    time.h = decimal_to_bcd(time.h);
    DS1307_write(hr_reg, time.h);

    time.dy = decimal_to_bcd(time.dy);
    DS1307_write(day_reg, time.dy);

    time.dt = decimal_to_bcd(time.dt);
    DS1307_write(date_reg, time.dt);

    time.mt = decimal_to_bcd(time.mt);
    DS1307_write(month_reg, time.mt);

    time.yr = decimal_to_bcd(time.yr);
    DS1307_write(year_reg, time.yr);
}
```

main.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "soft_delay.h"
#include "LCD_3_Wire.h"
#include "I2C.h"
#include "DS1307.h"


extern struct
{
   unsigned char s;
   unsigned char m;
   unsigned char h;
   unsigned char dy;
   unsigned char dt;
   unsigned char mt;
   unsigned char yr;
}time;


void show_value(unsigned char x_pos, unsigned char y_pos, unsigned char
value);
void display_time(void);


void main(void)
{
    time.s = 30;
    time.m = 58;
    time.h = 23;

    P15_PushPull_Mode;

    LCD_init();
```

```
    LCD_clear_home();

    LCD_goto(0, 0);

    LCD_putstr("N76E003 I2C RTCC");

    DS1307_init();
    set_time();

    while(1)
    {
        get_time();
        display_time();
    };
}


void show_value(unsigned char x_pos, unsigned char y_pos, unsigned char value)
{
    unsigned char chr = 0;

    chr = ((value / 10) + 0x30);
    LCD_goto(x_pos, y_pos);
    LCD_putchar(chr);

    chr = ((value % 10) + 0x30);
    LCD_goto((x_pos + 1), y_pos);
    LCD_putchar(chr);
}


void display_time(void)
{
    P15 ^= 1;
    LCD_goto(6, 1);
    LCD_putchar(' ');
    LCD_goto(9, 1);
    LCD_putchar(' ');
    delay_ms(400);

    show_value(4, 1, time.h);
    show_value(7, 1, time.m);
    show_value(10, 1, time.s);

    LCD_goto(6, 1);
    LCD_putchar(':');
    LCD_goto(9, 1);
    LCD_putchar(':');
    delay_ms(400);
}
```

Schematic



**Note that the pin naming of the chip in the schematic above is wrong. P13 and P14 are both SDA pins according to this naming when actually P13 is SCL and P14 is SDA pin.*

Explanation

I have coded two files for I2C easy and quick I2C implementation. These are I2C header and source file. These files describe I2C hardware functionality as well as provide higher level functions under the hood of which I2C hardware is manipulated. I have only coded these files for master mode only since barely we would ever need N76E003 slaves. I also didn't implement interrupt-based I2C functionalities as this, in my experience, is also not needed in most of the cases.

```
void I2C_init(void);
void I2C_start(void);
void I2C_stop(void);
unsigned char I2C_read(unsigned char ack_mode);
void I2C_write(unsigned char value);
```

Using hardware I2C requires us to manipulate the following registers apart from clock and GPIO settings.

I2CON	I²C control	C0H	(C7) -	(C6) I2CEN	(C4) STA	(C4) STO	(C3) SI	(C2) AA	(C1) -	(C0) I2CPX	0000 0000b
I2TOC	I²C time-out counter	BFH	-	-	-	-	-	I2TOCEN	DIV	I2TOF	0000 0000b
I2CLK	I²C clock	BEH	I2CLK[7:0]								0000 1001b
I2STAT	I²C status	BDH	I2STAT[7:3]					0	0	0	1111 1000b
I2DAT	I²C data	BCH	I2DAT[7:0]								0000 0000b
SADDR_1	Slave 1 address	BBH	SADDR_1[7:0]								0000 0000b
SADEN_1	Slave 1 address mask	BAH	SADEN_1[7:0]								0000 0000b
SADEN	Slave 0 address mask	B9H	SADEN[7:0]								0000 0000b

I2C GPIOs must be set as open-drain GPIOs as per I2C standard. I2C clock must be set according to the devices connected in the bus. Typically, I2C clock speed ranges from 20 – 400kHz. This clock speed is dependent on system clock speed Fsys. Therefore, before setting I2C clock speed you have to set/know Fsys. I2C clock is decided by the value of ***I2CLK*** register. It is basically a prescalar value in master mode.

$$\text{I2C Bus Clock} = \frac{Fsys}{4(1+I2CLK)}$$

So, if Fsys = 16MHz and I*2CLK = 39 (0x27)*, the I2C bus clock rate is 100kHz.

Note that only in master I2C clock can be setup. In slave mode, the slave(s) synchronizes automatically with bus clock.

I2CON register is they key register for using I2C. It contains all the flags and condition-makers.

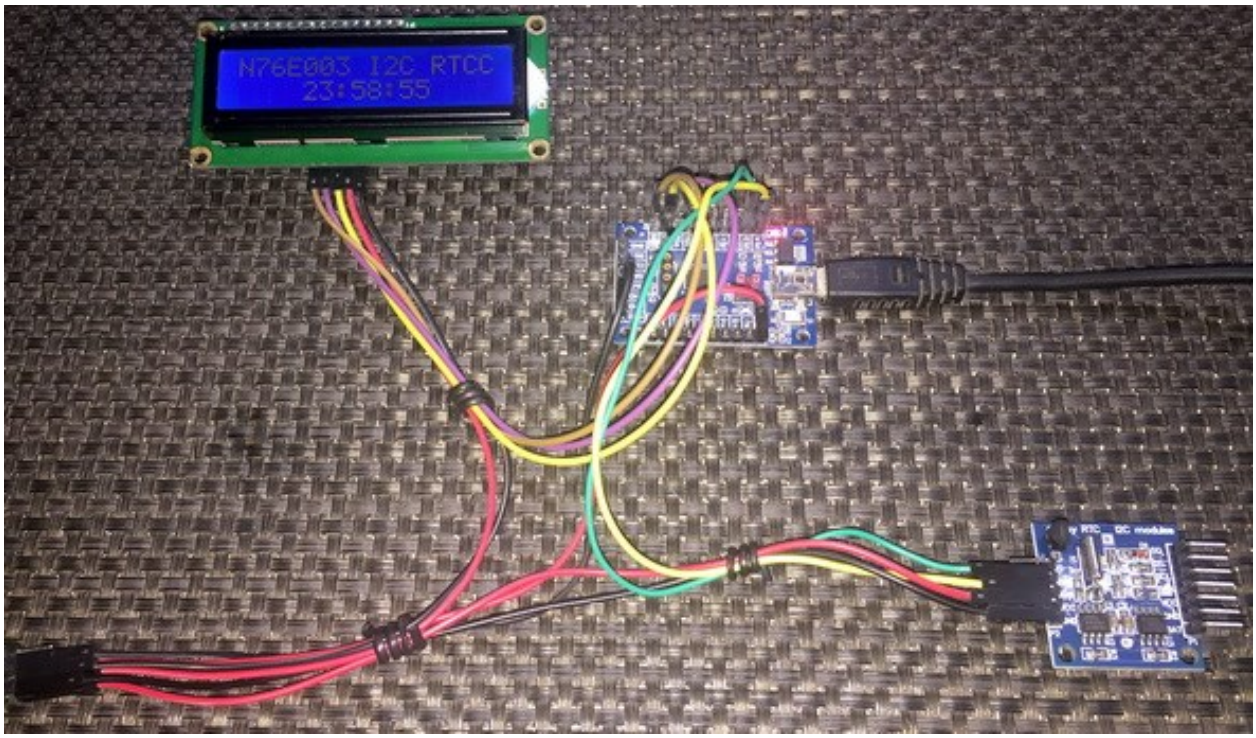
For instance, consider the I2C start condition. In order to make a start condition, we have to set the ***STA*** bit and clear I2C interrupt flag, ***SI***. Since I used polling mode, ***SI*** is polled until it is cleared. This ensures that a start condition has been successfully made. I have also added a software-based timeout feature should there be an issue with the I2C bus. In N76E003, there is a hardware-based approach for timeout too. Note that I didn't care about the I2C status register states as again this is rarely needed. If you want more grip on the I2C you can follow the BSP examples.

```
void I2C_start(void)
{
    signed int t = timeout_count;

    set_STA;
    clr_SI;
    while((SI == 0) && (t > 0))
    {
       t--;
    };
}
```

Finally, the last register that we will need is the I2C data register (***I2DAT***). It is through it we send and receive data from I2C bus.

The demo I have shown here is that of a DS1307 I2C-based real time clock. Perhaps this is the simplest device for learning about and understanding I2C.

Demo

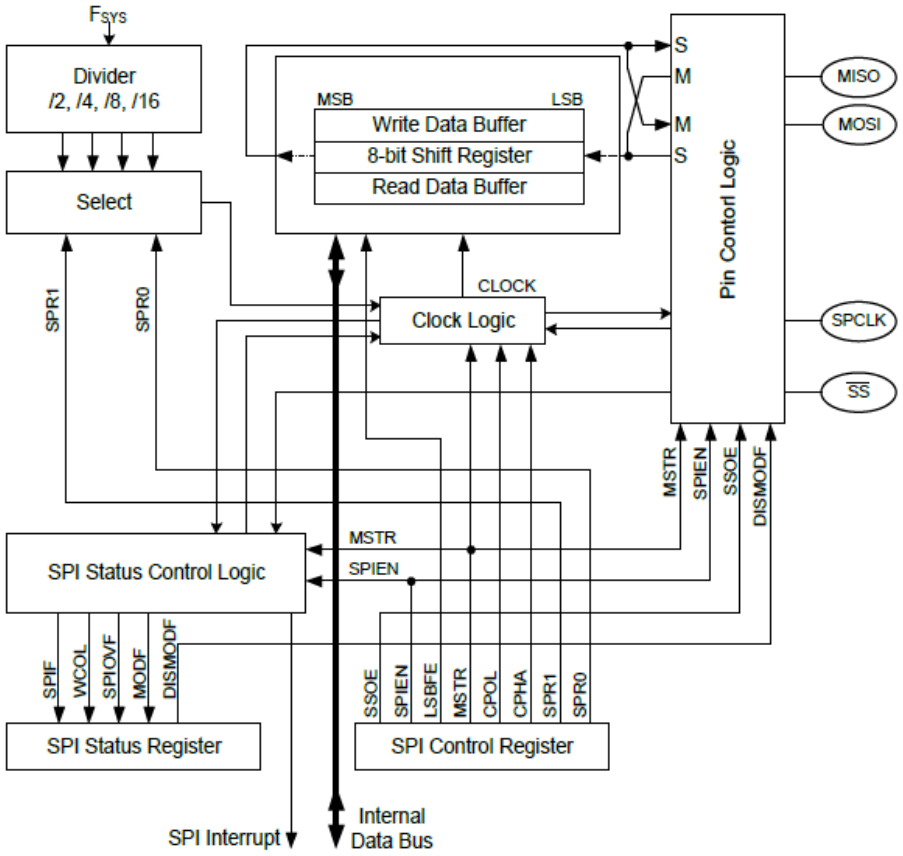


Demo video: https://youtu.be/tsjbRC59m2I

Serial Peripheral Interface (SPI) – Interfacing MAX7219 and MAX6675

SPI just like I2C is another highly popular form of onboard serial communication. Compared to I2C it is fast and robust. However, it requires more GPIO pins than other communication forms. These make it ideal communication interface for TFT displays, OLED displays, flash memories, DACs, etc.

SPI is best realized as a shift register that shifts data in and out with clock pulses. In a SPI bus, there is always one master device which generates clock and selects slave(s). Master sends commands to slave(s). Slave(s) responds to commands sent by the master. The number of slaves in a SPI bus is virtually unlimited. Except the chip selection pin, all SPI devices in a bus can share the same clock and data pins.



In general, if you wish to know more about SPI bus here are some cool links:

- 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://embeddedmicro.com/tutorials/mojo/serial-peripheral-interface-spi
- http://www.circuitbasics.com/basics-of-the-spi-communication-protocol

Code

MAX72xx.h

```
#define
MAX72xx_SPI_GPIO_init()                              do{P00_PushPull_Mode;
P10_PushPull_Mode; P11_PushPull_Mode;}while(0)

#define MAX72xx_CS_OUT_HIGH()                                set_P11
#define MAX72xx_CS_OUT_LOW()                                 clr_P11

#define NOP                                                  0x00
#define DIG0                                                 0x01
#define DIG1                                                 0x02
#define DIG2                                                 0x03
#define DIG3                                                 0x04
#define DIG4                                                 0x05
#define DIG5                                                 0x06
#define DIG6                                                 0x07
#define DIG7                                                 0x08
#define decode_mode_reg                                      0x09
#define intensity_reg                                        0x0A
#define scan_limit_reg                                       0x0B
#define shutdown_reg                                         0x0C
#define display_test_reg                                     0x0F

#define shutdown_cmd                                         0x00
#define run_cmd                                              0x01

#define no_test_cmd                                          0x00
#define test_cmd                                             0x01

#define digit_0_only                                         0x00
#define digit_0_to_1                                         0x01
#define digit_0_to_2                                         0x02
#define digit_0_to_3                                         0x03
#define digit_0_to_4                                         0x04
#define digit_0_to_5                                         0x05
#define digit_0_to_6                                         0x06
#define digit_0_to_7                                         0x07

#define No_decode_for_all                                    0x00
#define Code_B_decode_digit_0                                0x01
#define Code_B_decode_digit_0_to_3                           0x0F
#define Code_B_decode_for_all                                0xFF


void MAX72xx_SPI_HW_Init(unsigned char clk_value);
void MAX72xx_init(void);
void MAX72xx_write(unsigned char address, unsigned char value);
```

MAX72xx.c

```
#include "N76E003_IAR.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "MAX72xx.h"


void MAX72xx_SPI_HW_Init(unsigned char clk_value)
{
  switch(clk_value)
  {
    case 1:
    {
      clr_SPR1;
      set_SPR0;
      break;
    }
    case 2:
    {
      set_SPR1;
      clr_SPR0;
      break;
    }
    case 3:
    {
      set_SPR1;
      set_SPR0;
      break;
    }
    default:
    {
      clr_SPR1;
      clr_SPR0;
      break;
    }
  }

  set_DISMODF;
  set_MSTR;
  clr_CPOL;
  clr_CPHA;
  set_SPIEN;
}


void MAX72xx_init(void)
{
  MAX72xx_SPI_GPIO_init();
  MAX72xx_SPI_HW_Init(0);

  MAX72xx_write(shutdown_reg, run_cmd);
  MAX72xx_write(decode_mode_reg, Code_B_decode_digit_0_to_3);
  MAX72xx_write(scan_limit_reg, digit_0_to_3);
```

```
  MAX72xx_write(intensity_reg, 0x19);
}


void MAX72xx_write(unsigned char address, unsigned char value)
{
  MAX72xx_CS_OUT_LOW();

  SPDR = address;
  while(!(SPSR & SET_BIT7));
  clr_SPIF;

  SPDR = value;
  while(!(SPSR & SET_BIT7));
  clr_SPIF;

  MAX72xx_CS_OUT_HIGH();
}
```

MAX6675.h

```
#define MAX6675_SPI_GPIO_init()     do{P01_Input_Mode; P10_PushPull_Mode;
P12_PushPull_Mode;}while(0)

#define MAX6675_CS_OUT_HIGH()       set_P12
#define MAX6675_CS_OUT_LOW()        clr_P12

#define T_min                       0
#define T_max                       1024

#define count_max                   4096

#define no_of_pulses                16

#define deg_C                       0
#define deg_F                       1
#define tmp_K                       2

#define open_contact                0x04
#define close_contact               0x00

#define scalar_deg_C                0.25
#define scalar_deg_F_1              1.8
#define scalar_deg_F_2              32.0
#define scalar_tmp_K                273.0

#define no_of_samples               16


void MAX6675_SPI_HW_Init(unsigned char clk_value);
void MAX6675_init(void);
unsigned char MAX6675_get_ADC(unsigned int *ADC_data);
float MAX6675_get_T(unsigned int ADC_value, unsigned char T_unit);
```

MAX6675.c

```c
#include "N76E003_IAR.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "MAX6675.h"


void MAX6675_SPI_HW_Init(unsigned char clk_value)
{
  switch(clk_value)
  {
    case 1:
    {
      clr_SPR1;
      set_SPR0;
      break;
    }
    case 2:
    {
      set_SPR1;
      clr_SPR0;
      break;
    }
    case 3:
    {
      set_SPR1;
      set_SPR0;
      break;
    }
    default:
    {
      clr_SPR1;
      clr_SPR0;
      break;
    }
  }

  set_DISMODF;
  set_MSTR;
  clr_CPOL;
  set_CPHA;
  set_SPIEN;
}


void MAX6675_init(void)
{
    MAX6675_SPI_GPIO_init();
    MAX6675_SPI_HW_Init(0);
}


unsigned char MAX6675_get_ADC(unsigned int *ADC_data)
```

```
{
    unsigned char lb = 0;
    unsigned char hb = 0;
    unsigned char samples = no_of_samples;
    unsigned int temp_data = 0;
    unsigned long avg_value = 0;


    while(samples > 0)
    {
         MAX6675_CS_OUT_LOW();

         SPDR = 0x00;
         while(!(SPSR & SET_BIT7));
         hb = SPDR;
         clr_SPIF;

         SPDR = 0x00;
         while(!(SPSR & SET_BIT7));
         lb = SPDR;
         clr_SPIF;

         MAX6675_CS_OUT_HIGH();

         temp_data = hb;
         temp_data <<= 8;
         temp_data |= lb;
         temp_data &= 0x7FFF;

         avg_value += (unsigned long)temp_data;

         samples--;
         Timer0_Delay1ms(10);
    };

    temp_data = (avg_value >> 4);

    if((temp_data & 0x04) == close_contact)
    {
        *ADC_data = (temp_data >> 3);
        return close_contact;
    }
    else
    {
        *ADC_data = (count_max + 1);
        return open_contact;
    }
}


float MAX6675_get_T(unsigned int ADC_value, unsigned char T_unit)
{
    float tmp = 0.0;

    tmp = (((float)ADC_value) * scalar_deg_C);
```

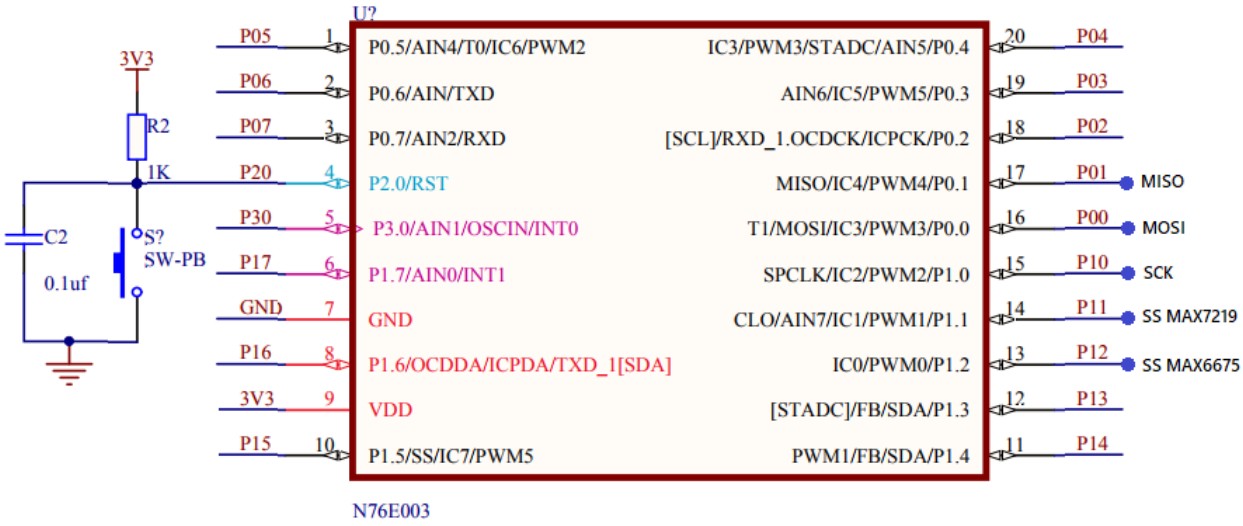
```
    switch(T_unit)
    {
        case deg_F:
        {
             tmp *= scalar_deg_F_1;
             tmp += scalar_deg_F_2;
             break;
        }
        case tmp_K:
        {
            tmp += scalar_tmp_K;
            break;
        }
        default:
        {
            break;
        }
    }

    return tmp;
}
```

main.c

```
#include "N76E003_IAR.h"
#include "Common.h"
#include "Delay.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "soft_delay.h"
#include "MAX72xx.h"
#include "MAX6675.h"


void main(void)
{
  unsigned int ti = 0;
  unsigned int t = 0;

  P15_PushPull_Mode;
  MAX6675_init();
  MAX72xx_init();

  while(1)
  {
      P15 = 1;
      clr_CPOL;
      set_CPHA;
      MAX6675_get_ADC(&ti);
      t = ((unsigned int)MAX6675_get_T(ti, tmp_K));
      delay_ms(100);

      P15 = 0;
      clr_CPOL;
      clr_CPHA;
      MAX72xx_write(DIG3, ((t / 1000) % 10));
```

```
      MAX72xx_write(DIG2, ((t / 100) % 10));
      MAX72xx_write(DIG1, ((t / 10) % 10));
      MAX72xx_write(DIG0, (t % 10));
      delay_ms(100);
  };
}
```

Schematic



Explanation

With SPI we have to deal with two things – first, the initialization and second, the data transfer process. Configuring SPI needs a few things to be set up. These are:

- SPI communication bus speed or simply clock speed
- Clock polarity
- Clock phase
- Orientation of data, i.e. MSB first or LSB first
- Master/slave status
- Optionally hardware slave selection pin use

Most of these can be done by manipulating the following registers:

SPCR – Serial Peripheral Control Register

7	6	5	4	3	2	1	0
SSOE	SPIEN	LSBFE	MSTR	CPOL	CPHA	SPR1	SPR0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Address: F3H, page 0 Reset value: 0000 0000b

SPDR – Serial Peripheral Data Register

7	6	5	4	3	2	1	0
SPDR[7:0]							
R/W							

Address: F5H Reset value: 0000 0000b

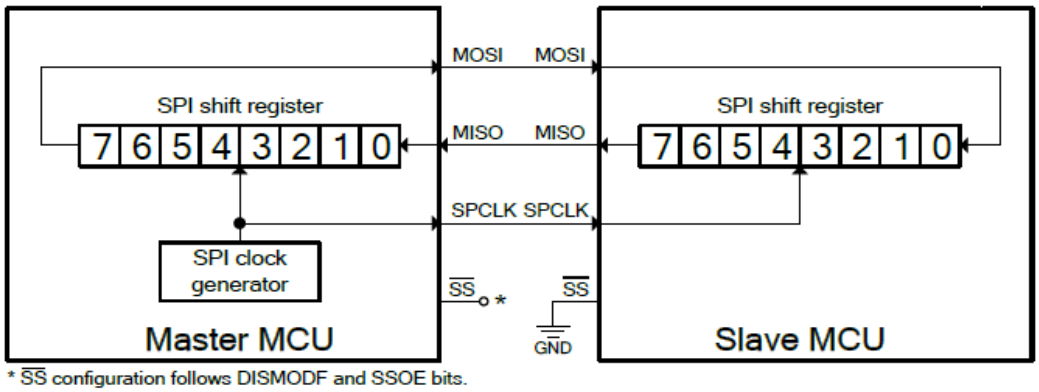
SPCR configures the hardware SPI peripheral of N76E003 while ***SPDR*** is used for transferring data.

To aid in all these I have coded the following functions. Though I didn't use these code snippets in the actual demo code for demonstration purposes, these will most certainly work and reduce coding effort.

```
void SPI_init(unsigned char clk_speed, unsigned char mode)
{
  set_SPI_clock_rate(clk_speed);
  set_SPI_mode(mode);
  set_DISMODF;
  set_MSTR;
  set_SPIEN;
}


unsigned char SPI_transfer(unsigned char write_value)
{
  signed int t = timeout_count;
  register unsigned char read_value = 0x00;

  SPDR = write_value;
  while((!(SPSR & SET_BIT7)) && (t > 0))
  {
    t--;
  };
  read_value = SPDR;
  clr_SPIF;

  return read_value;
}
```

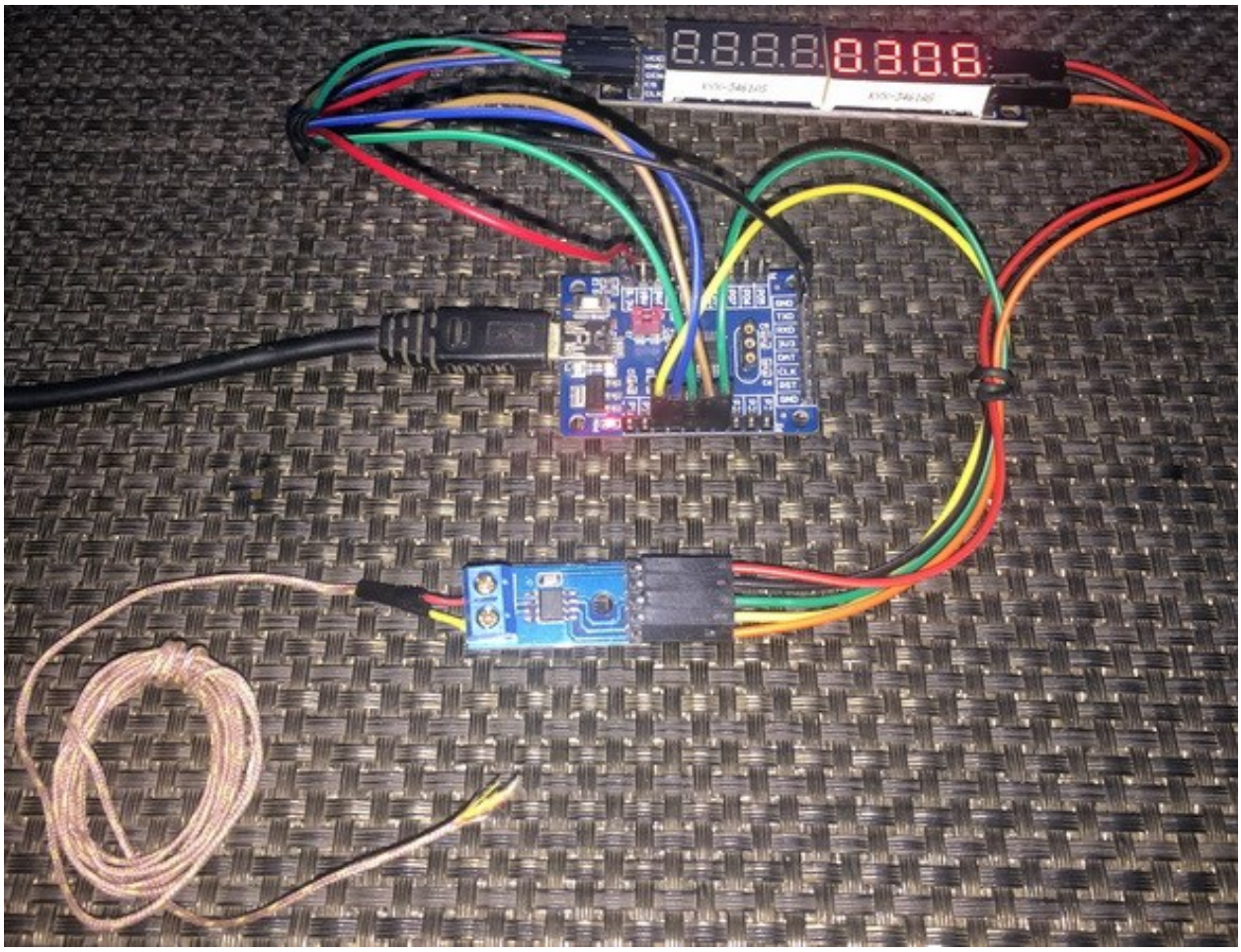
SPI_init function sets up the SPI peripheral using two information – the bus clock speed and SPI data transfer mode. It initializes the SPI hardware as a SPI master device and without hardware salve selection pin. This configuration is mostly used in common interfacings. Only mode and clock speed are the variables. In the demo, it is visible that MAX7219 and MAX6675 work with different SPI modes. I also didn't use the hardware slave selection because there are two devices and one slave pin. Certainly, both external devices are not driven at the same time although both share the same bus lines.

SPI_transfer reads and writes on the SPI bus. Here the master writes to the slave first, waits for successful transmission and reads from the slave later. It is best realized by the following figure:



To demo SPI, MAX6675 and MAX7219 were used because one lacked read feature while the other lacked write feature. In the demo, MAX6675 is read to measure the temperature sensed by the K-type thermocouple attached with it and display the measured temperature in Kelvin on MAX7219-based seven segment display arrays.

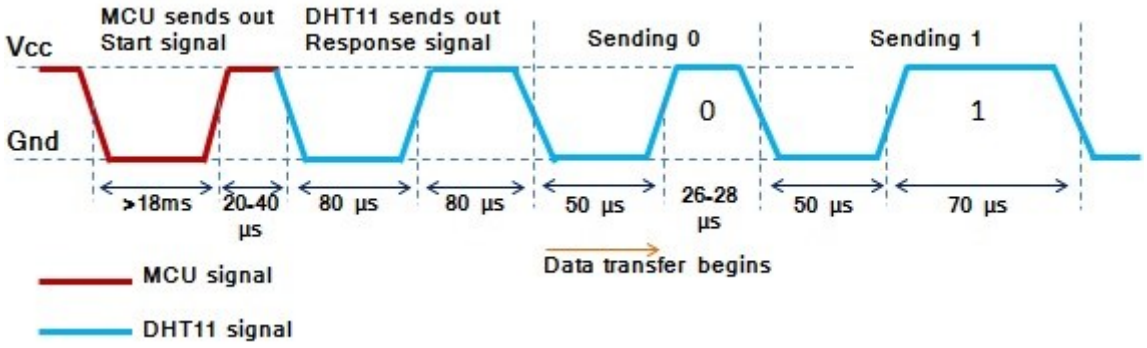
Demo



Demo video: https://youtu.be/0E-nOCFYh54

One Wire (OW) Communication – Interfacing DHT11

One protocol is not a standard protocol like I2C or SPI. It is just a mere method of communicating with a device that uses just one wire. There are a few devices that uses such principle for communication. DHT series sensors, DS18B20s, digital serial number chips, etc. are a few to mention. As I stated before, since one wire communication doesn't belong to a dedicated format of communication, there is no dedicated hardware for it. With just a bidirectional GPIO pin we can implement one wire communication. Optionally we can also use a timer for measuring response pulse widths since the method of sending ones and zeroes over one wire requires the measurement of pulse durations. This way of encoding ones and zeros as a function of pulse width is called time-slotting.



Shown above is the timing diagram for DHT11's time slots. Check the length of pulse high time for one and zero. This is the technique applied in time-slotting mechanism. We have to time pulse lengths in order to identify the logic states.

Code

DHT11.h

```
#define HIGH                    1
#define LOW                     0

#define DHT11_pin_init()        P05_Quasi_Mode

#define DHT11_pin_HIGH()        set_P05
#define DHT11_pin_LOW()         clr_P05

#define DHT11_pin_IN()          P05


void DHT11_init(void);
unsigned char get_byte(void);
unsigned char get_data(void);
```

DHT11.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
```

```c
#include "Common.h"
#include "Delay.h"
#include "soft_delay.h"
#include "DHT11.h"


extern unsigned char values[5];


void DHT11_init(void)
{
   DHT11_pin_init();
   delay_ms(1000);
}


unsigned char get_byte(void)
{
   unsigned char s = 0;
   unsigned char value = 0;

   for(s = 0; s < 8; s++)
   {
      value <<= 1;
      while(DHT11_pin_IN() == LOW);
      delay_us(30);

      if(DHT11_pin_IN() == HIGH)
      {
          value |= 1;
      }

      while(DHT11_pin_IN() == HIGH);
   }
   return value;
}


unsigned char get_data(void)
{
   short chk = 0;
   unsigned char s = 0;
   unsigned char check_sum = 0;

   DHT11_pin_HIGH();
   DHT11_pin_LOW();
   delay_ms(18);
   DHT11_pin_HIGH();
   delay_us(26);

   chk = DHT11_pin_IN();

   if(chk)
   {
      return 1;
   }
```

```
    delay_us(80);

    chk = DHT11_pin_IN();

    if(!chk)
    {
        return 2;
    }
    delay_us(80);

    for(s = 0; s <= 4; s += 1)
    {
        values[s] = get_byte();
    }

    DHT11_pin_HIGH();

    for(s = 0; s < 4; s += 1)
    {
        check_sum += values[s];
    }

    if(check_sum != values[4])
    {
        return 3;
    }
    else
    {
        return 0;
    }
}
```

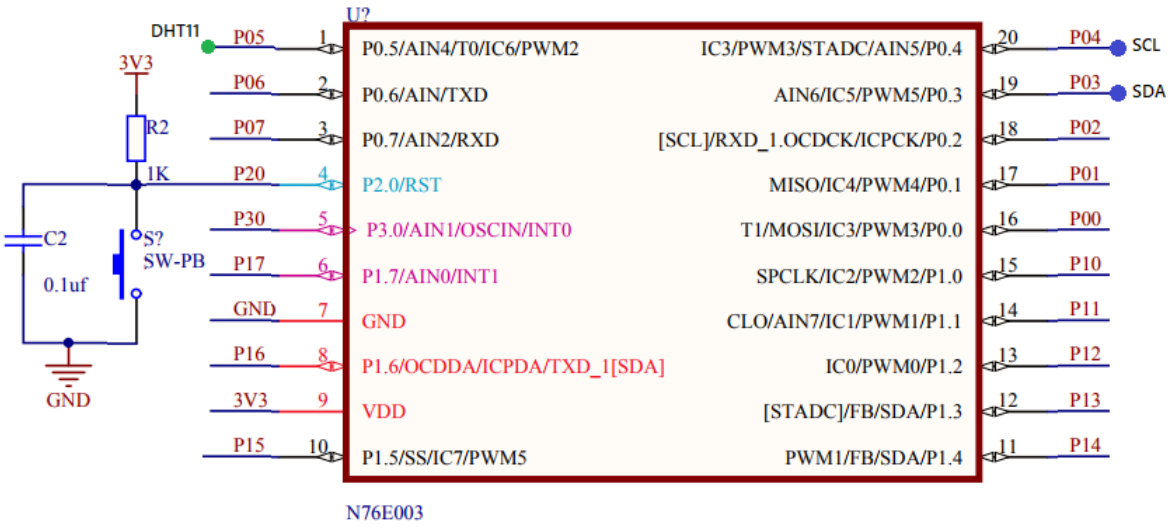
main.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "soft_delay.h"
#include "LCD_2_Wire.h"
#include "DHT11.h"


unsigned char values[5];


const unsigned char symbol[8] =
{
    0x00, 0x06, 0x09, 0x09, 0x06, 0x00, 0x00, 0x00
};


void setup(void);
void lcd_symbol(void);
void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned char value);
```

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

  setup();

  while(1)
  {
    state = get_data();

    switch(state)
    {
            case 1:
            {
            }
            case 2:
            {
                 LCD_clear_home();
                 LCD_putstr("No Sensor Found!");
                 break;
            }
            case 3:
            {
                 LCD_clear_home();
                 LCD_putstr("Checksum Error!");
                 break;
            }
            default:
            {
                 LCD_goto(0, 0);
                 LCD_putstr("R.H/ %:       ");

                 lcd_print(14, 0, values[0]);

                 LCD_goto(0, 1);
                 LCD_putstr("Tmp/");
                 LCD_goto(4, 1);
                 LCD_send(0, DAT);
                 LCD_goto(5, 1);
                 LCD_putstr("C:");

                 if((values[2] & 0x80) == 1)
                 {
                     LCD_goto(13, 1);
                     LCD_putstr("-");
                 }
                 else
                 {
                     LCD_goto(13, 1);
                     LCD_putstr(" ");
                 }

                 lcd_print(14, 1, values[2]);
                 break;
```

```
            }
    }

    delay_ms(1000);
  };
}


void setup(void)
{
  DHT11_init();

  LCD_init();
  LCD_clear_home();
  lcd_symbol();
}


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

  LCD_send(0x40, CMD);

  for(s = 0; s < 8; s++)
  {
   LCD_send(symbol[s], DAT);
  }

  LCD_send(0x80, CMD);
}


void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned char value)
{
 unsigned char chr = 0x00;

 chr = ((value / 10) + 0x30);
 LCD_goto(x_pos, y_pos);
 LCD_putchar(chr);

 chr = ((value % 10) + 0x30);
 LCD_goto((x_pos + 1), y_pos);
 LCD_putchar(chr);
}
```

Schematic



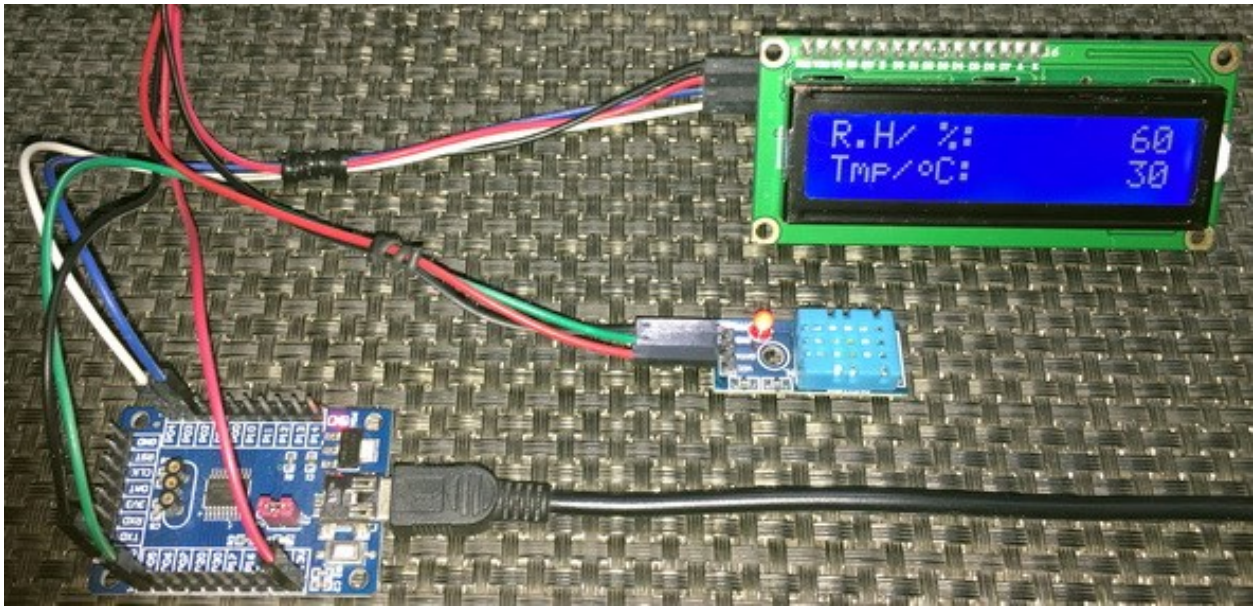
Explanation

As stated earlier, OW basically needs two things – first, the control of a single GPIO pin and second, the measurement of time. I would like to highlight the most important part of the whole code here:

```
unsigned char get_byte(void)
{
   unsigned char s = 0;
   unsigned char value = 0;

   for(s = 0; s < 8; s++)
   {
      value <<= 1;
      while(DHT11_pin_IN() == LOW);
      delay_us(30);

      if(DHT11_pin_IN() == HIGH)
      {
          value |= 1;
      }

      while(DHT11_pin_IN() == HIGH);
   }
   return value;
}
```

From DHT11's timing diagram, we know that a zero is define by a pulse of 26 - 28µs and a logic one is defined by a pulse of 70µs. It is here in the ***get_byte*** function we check the pulse coming from DHT11. The DHT11's pin is polled as an input and after 30µs if the result of this polling is high then it is considered as a logic one or else it is considered otherwise. Once triggered, DHT11 will provide a 5-byte output. All of the bits in these 5 bytes are checked this way. Though it is not the best method to do so, it is easy and simple.

Demo



Demo video: https://youtu.be/1Zetmad3fLU

One Wire (OW) Communication – Interfacing DS18B20

One wire communication, as stated previously, doesn't have a defined protocol. Except for the time-slotting mechanism part, there is nothing in common across the devices using this form of communication. We have seen how to interface a DHT11 relative humidity and temperature sensor previously. In this segment, we will see how to interface a DS18B20 one-wire digital temperature sensor with N76E003. DS18B20 follows and uses a completely different set of rules for communication.

Code

one_wire.h

```
#define DS18B20_GPIO_init()             P06_OpenDrain_Mode

#define DS18B20_IN()                    P06

#define DS18B20_OUT_LOW()               clr_P06
#define DS18B20_OUT_HIGH()              set_P06

#define TRUE                            1
#define FALSE                           0


unsigned char onewire_reset(void);
void onewire_write_bit(unsigned char bit_value);
unsigned char onewire_read_bit(void);
void onewire_write(unsigned char value);
unsigned char onewire_read(void);
```

one_wire.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "soft_delay.h"
#include "one_wire.h"


unsigned char onewire_reset(void)
{
    unsigned char res = FALSE;

    DS18B20_GPIO_init();

    DS18B20_OUT_LOW();
    delay_us(480);
    DS18B20_OUT_HIGH();
    delay_us(60);

    res = DS18B20_IN();
```

```
     delay_us(480);

     return res;
}


void onewire_write_bit(unsigned char bit_value)
{
    DS18B20_OUT_LOW();

    if(bit_value)
    {
        delay_us(104);
        DS18B20_OUT_HIGH();
    }
}


unsigned char onewire_read_bit(void)
{
    DS18B20_OUT_LOW();
    DS18B20_OUT_HIGH();
  delay_us(15);

    return(DS18B20_IN());
}


void onewire_write(unsigned char value)
{
     unsigned char s = 0;

     while(s < 8)
     {
          if((value & (1 << s)))
          {
              DS18B20_OUT_LOW();
                nop;
              DS18B20_OUT_HIGH();
              delay_us(60);
          }

          else
          {
            DS18B20_OUT_LOW();
              delay_us(60);
              DS18B20_OUT_HIGH();
              nop;
          }

          s++;
     }
}


unsigned char onewire_read(void)
```

```
{
    unsigned char s = 0x00;
    unsigned char value = 0x00;

    while(s < 8)
    {
        DS18B20_OUT_LOW();
        nop;
        DS18B20_OUT_HIGH();

        if(DS18B20_IN())
        {
            value |=  (1 << s);
        }

        delay_us(60);

        s++;
    }

    return value;
}
```

DS18B20.h

```
#include "one_wire.h"


#define convert_T                       0x44
#define read_scratchpad                 0xBE
#define write_scratchpad                0x4E
#define copy_scratchpad                 0x48
#define recall_E2                       0xB8
#define read_power_supply               0xB4
#define skip_ROM                        0xCC

#define resolution                      12


void DS18B20_init(void);
float DS18B20_get_temperature(void);
```

DS18B20.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "soft_delay.h"
#include "DS18B20.h"


void DS18B20_init(void)
{
```

```
    onewire_reset();
    delay_ms(100);
}


float DS18B20_get_temperature(void)
{
    unsigned char msb = 0x00;
    unsigned char lsb = 0x00;
    register float temp = 0.0;

    onewire_reset();
    onewire_write(skip_ROM);
    onewire_write(convert_T);

    switch(resolution)
    {
        case 12:
        {
            delay_ms(750);
            break;
        }
        case 11:
        {
            delay_ms(375);
            break;
        }
        case 10:
        {
            delay_ms(188);
            break;
        }
        case 9:
        {
            delay_ms(94);
            break;
        }
    }

    onewire_reset();

    onewire_write(skip_ROM);
    onewire_write(read_scratchpad);

    lsb = onewire_read();
    msb = onewire_read();

    temp = msb;
    temp *= 256.0;
    temp += lsb;


    switch(resolution)
    {
        case 12:
        {
```

```
            temp *= 0.0625;
            break;
        }
        case 11:
        {
            temp *= 0.125;
            break;
        }
        case 10:
        {
            temp *= 0.25;
            break;
        }
        case 9:
        {
            temp *= 0.5;
            break;
        }
    }

    delay_ms(40);

    return (temp);
}
```

main.c

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "soft_delay.h"
#include "DS18B20.h"
#include "LCD_3_Wire.h"


const unsigned char symbol[8] =
{
   0x00, 0x06, 0x09, 0x09, 0x06, 0x00, 0x00, 0x00
};


void lcd_symbol(void);
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);


void main(void)
{
    float t = 0.0;
```

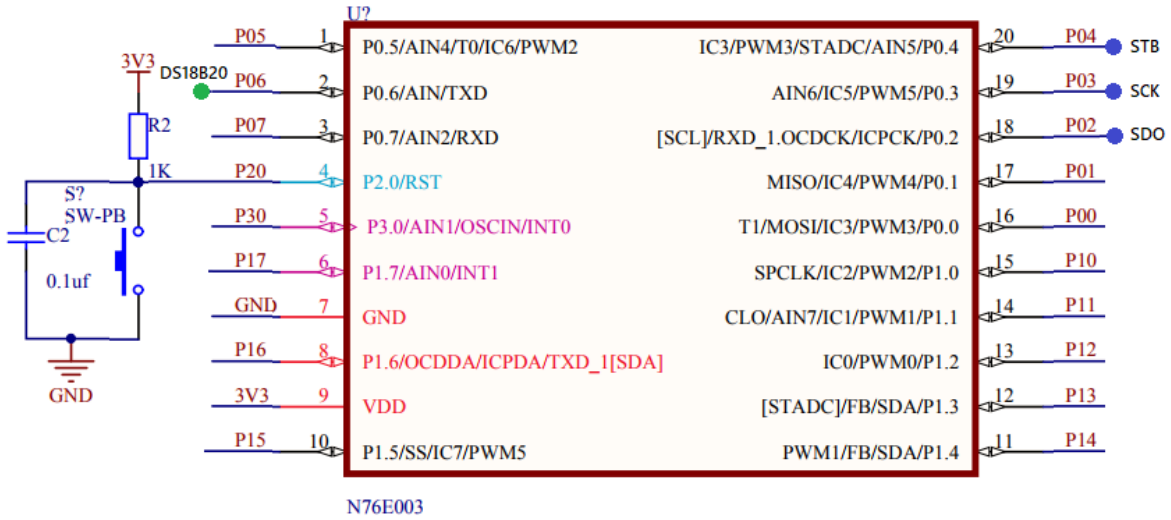
```
    DS18B20_init();
    LCD_init();
    lcd_symbol();

    LCD_goto(0, 0);
    LCD_putstr("N76E003 DS18B20");

    LCD_goto(0, 1);
    LCD_putstr("T/ C");
    LCD_goto(2, 1);
    LCD_send(0, DAT);

    while(1)
    {
       t = DS18B20_get_temperature();
       print_F(9, 1, t, 3);
       delay_ms(100);
    };
}


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

   LCD_send(0x40, CMD);

   for(s = 0; s < 8; s++)
   {
        LCD_send(symbol[s], DAT);
   }

   LCD_send(0x80, CMD);
}


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, '\0'};

    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 = 0x0000;

    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);
    }
}
```

Schematic



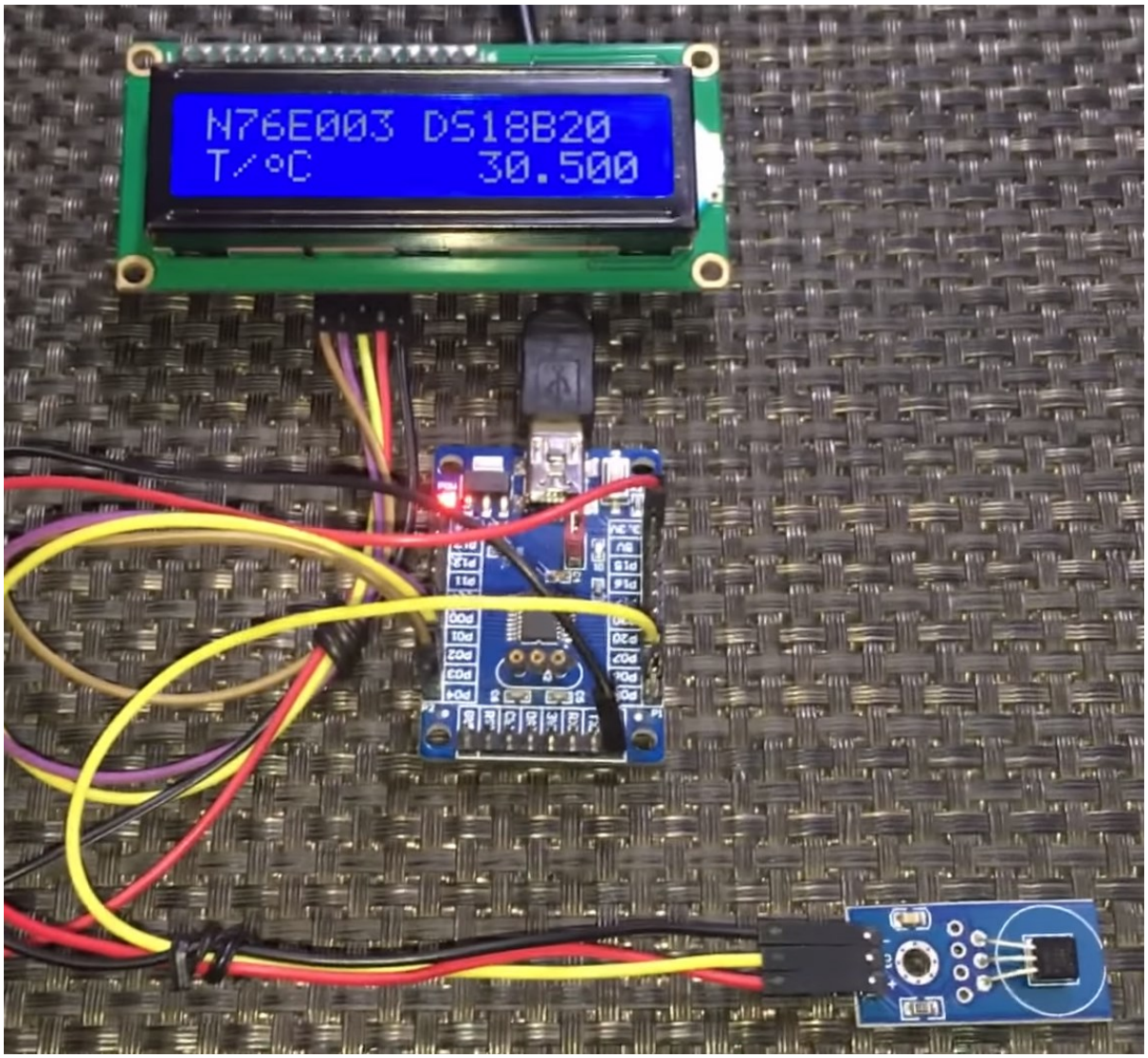
Explanation

One wire communication is detailed in these application notes from Maxim:

https://www.maximintegrated.com/en/app-notes/index.mvp/id/126
https://www.maximintegrated.com/en/app-notes/index.mvp/id/162

These notes are all that are needed for implementing the one wire communication interface for DS18B20. Please go through these notes. The codes are self-explanatory and are implemented from the code examples in these app notes.

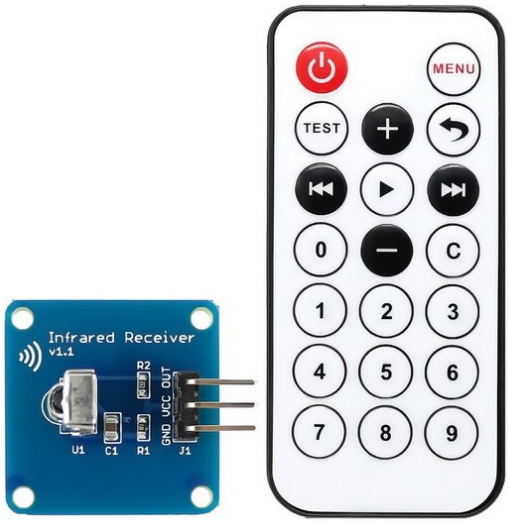
Demo



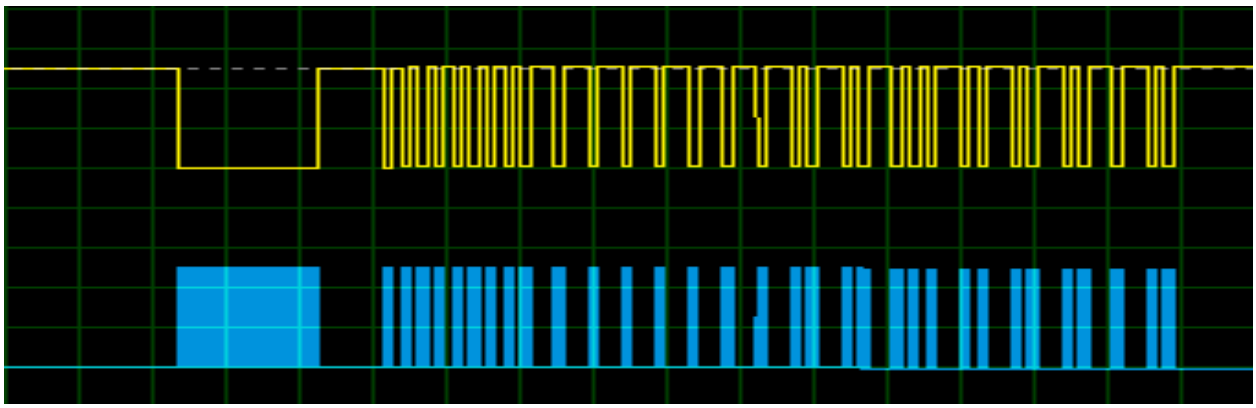
Demo video: https://youtu.be/uOq7etfuqvg

Decoding NEC IR Remote Protocol

IR remote controllers are used in a number of devices for remotely controlling them. Examples of these devices include TV, stereo, smart switches, projectors, etc. IR communication as such is unidirectional and in a way one wire communication. Ideally there is one transmitter and one receiver only.



Data to be sent from a remote transmitter is sent by modulating it with a carrier wave (usually between 36kHz to 40kHz). Modulation ensure safety from data corruption and long-distance transmission. An IR receiver at the receiving end receives and demodulates the sent out modulated data and outputs a stream of pulses. These pulses which vary in pulse widths/timing/position/phase carry the data information that was originally sent by the remote transmitter. How the pulses should behave is governed closely by a protocol or defined set of rules. In most micros, there is no dedicated hardware for decoding IR remote protocols. A micro that needs to decode IR remote data also doesn't know when it will be receiving an IR data stream. Thus, a combination of external interrupt and timer is needed for decoding IR data.



In this segment, we will see how we can use an external interrupt and a timer to easily decode a NEC IR remote. This same method can be applied to any IR remote controller. At this stage, I recommend that you do a study of NEC IR protocol from here. SB-Project's website is an excellent page for info as such.

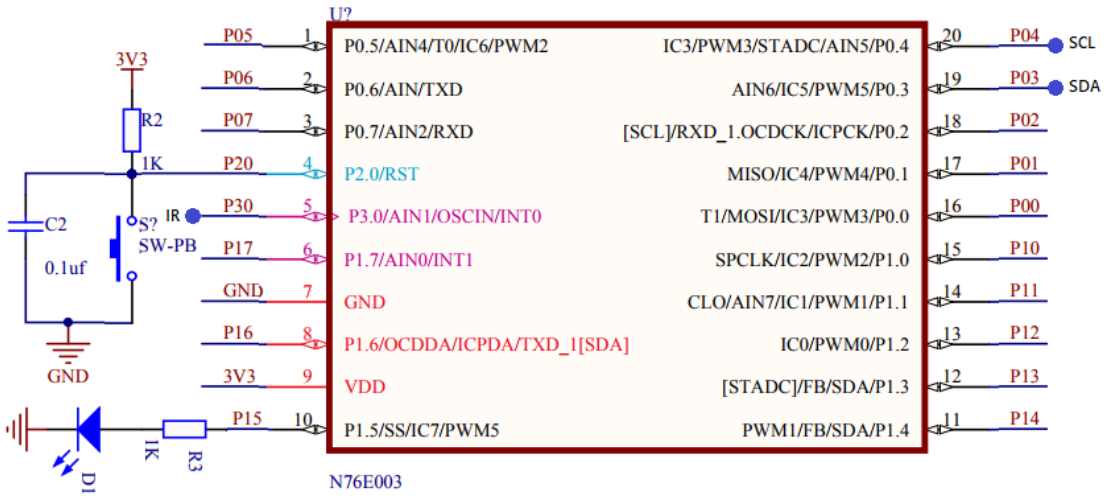
Code

```
#include "N76E003.h"
#include "SFR_Macro.h"
#include "Function_define.h"
#include "Common.h"
#include "Delay.h"
#include "soft_delay.h"
#include "LCD_2_Wire.h"


#define sync_high          22000
#define sync_low           14000
#define one_high            3600
#define one_low             2400
#define zero_high           1800
#define zero_low            1200


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


void setup(void);
void set_Timer_0(unsigned int value);
unsigned int get_Timer_0(void);
void erase_frames(void);
unsigned char decode(unsigned char start_pos, unsigned char end_pos);
void decode_NEC(unsigned char *addr, unsigned char *cmd);
void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned char value);


void EXTI0_ISR(void)
interrupt 0
{
  frames[bits] = get_Timer_0();
  bits++;
  set_TR0;

  if(bits >= 33)
  {
     received = 1;
     clr_EA;
     clr_TR0;
  }
  set_Timer_0(0x0000);
  P15 = ~P15;
}


void main(void)
{
  unsigned char address = 0x00;
  unsigned char command = 0x00;
```

```
  setup();

  while(1)
  {
     if(received)
     {
       decode_NEC(&address, &command);
       lcd_print(13, 0, address);
       lcd_print(13, 1, command);
       delay_ms(100);
       erase_frames();
       set_EA;
     }
  };
}


void setup(void)
{
  erase_frames();
  P15_PushPull_Mode;
  TIMER0_MODE1_ENABLE;
  set_Timer_0(0x0000);
  set_IT0;
  set_EX0;
  set_EA;

  LCD_init();
  LCD_clear_home();
  LCD_goto(0, 0);
  LCD_putstr("Address:");
  LCD_goto(0, 1);
  LCD_putstr("Command:");
}


void set_Timer_0(unsigned int value)
{
  TH0 = ((value && 0xFF00) >> 8);
  TL0 = (value & 0x00FF);
}


unsigned int get_Timer_0(void)
{
  unsigned int value = 0x0000;

  value = TH0;
  value <<= 8;
  value |= TL0;

  return value;
}


void erase_frames(void)
```

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

  set_Timer_0(0x0000);
  received = 0;
  bits = 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);
}


void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned char value)
{
  LCD_goto(x_pos, y_pos);
  LCD_putchar((value / 100) + 0x30);
  LCD_goto((x_pos + 1), y_pos);
  LCD_putchar(((value % 10) / 10) + 0x30);
  LCD_goto((x_pos + 2), y_pos);
  LCD_putchar((value % 10) + 0x30);
}
```

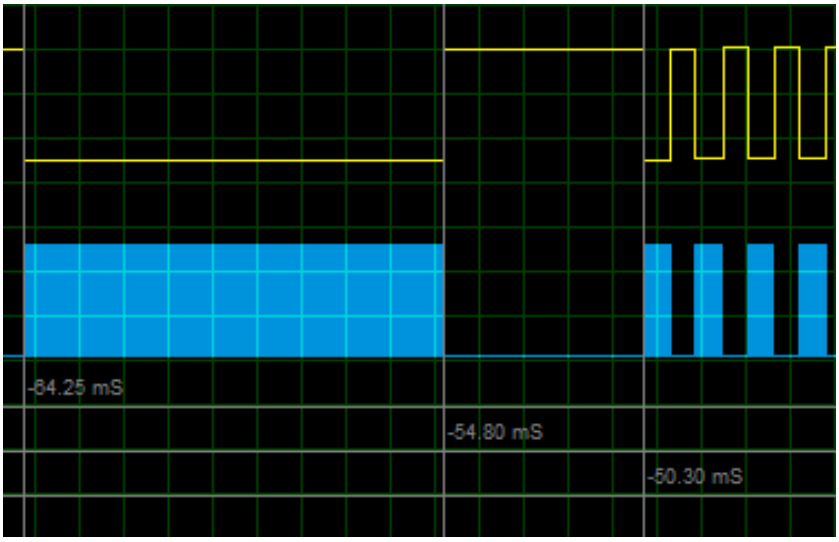
Schematic



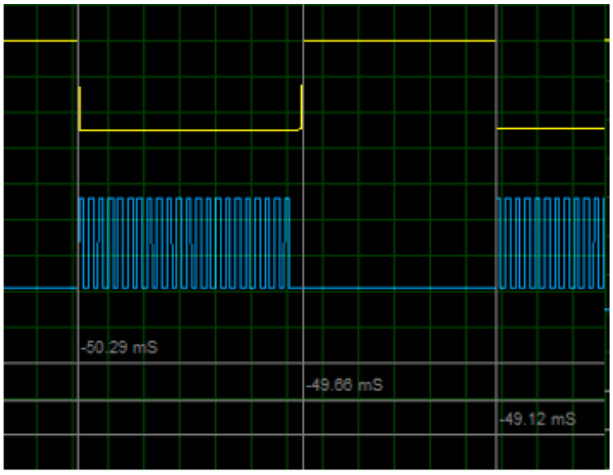
Explanation

A NEC transmission sends out 33 pulses. The first one is a sync bit and the rest 32 contain address and command info. These 32 address and command bits can be divided into 4 groups – address, command, inverted address and inverted command. The first 16 bits contain address and inverted address while the rest contain command and inverted command. The inverted signals can be used against the non-inverted ones for checking signal integrity.

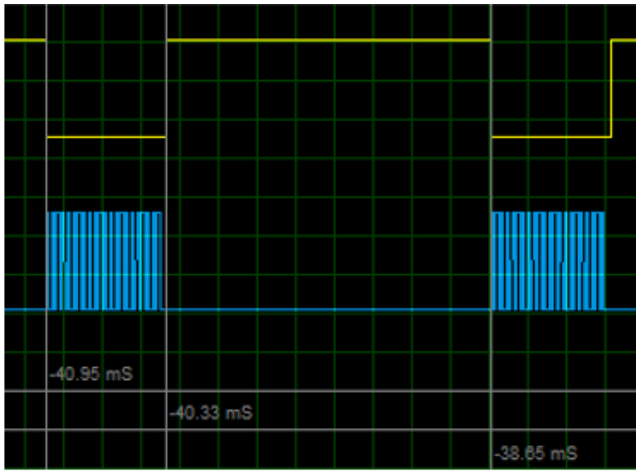
We already know that IR data is received as a steam of pulses. Pulses represent sync bit or other info, ones and zeros. In case of NEC IR protocol, the pulses have variable lengths. Sync bit represented by a pulse of 9ms high and 4.5ms low – total pulse time is 13.5ms. Check the timing diagram below. Please note that in this timing diagram the blue pulses are from the transmitter's output and the yellow ones are those received by the receiver. Clearly the pulses are inverted.



Logic one is represented by a pulse of 560µs high time and 2.25ms of low time – total pulse time is 2.81ms. Likewise, logic zero is represented by a pulse of 560µs high time and 1.12ms of low time – total pulse time is 1.68ms.



Logic 0



Logic 1

These timings can vary slightly about 15 - 25% due to a number of factors like medium, temperature, etc. Therefore, we can assume the following timings:

Signal	*Ideal Pulse Length (ms)*	*Maximum Pulse Length (ms)*	*Minimum Pulse Length (ms)*
Sync	13.5	16	10
High	2.81	3.3	2.2
Low	1.68	2	1.3

Now as per timing info and timing diagram we have to detect falling edges and time how long it takes to detect another falling edge. In this way, we can time the received pulses and use the pulse time info to decode a received signal.
In the demo, I used external interrupt channel EXTI0 and Timer 0. The system clock frequency is set to 16MHz with HIRC. Timer 0 is set in Mode 1 and its counter is rest to 0 count. EXTI0 is set to detect falling edges. Note that the timer is set but not started.

```
TIMER0_MODE1_ENABLE;
set_Timer_0(0x0000);
set_IT0;
set_EX0;
set_EA;
```

Also note that each tick of Timer 0 here is about:

$$Timer\ Tick = \frac{Timer\ Prescalar}{F_{sys}} = \frac{12}{16\ MHz} = 0.75\mu s$$

Based on this tick info, we can deduce the maximum and minimum pulse durations as like:

```
#define sync_high          22000 // 22000 × 0.75ms = 16.5ms
#define sync_low           14000 // 14000 × 0.75ms = 10.5ms
#define one_high            3600 // 3600 × 0.75ms = 2.7ms
#define one_low             2400 // 2400 × 0.75ms = 1.8ms
```

```
#define zero_high           1800 // 1800 × 0.75ms = 1.35ms
#define zero_low            1200 // 1200 × 0.75ms = 0.9ms
```

Now when an IR transmission is received, an interrupt will be triggered. Therefore, in the ISR, we immediately capture the timer's tick count and reset it. When all 33 pulses have been received this way, we temporarily halt all interrupts by disabling the global interrupt and stop the timer in order to decode the received signal. P15 is also toggled to demonstrate reception.

```
void EXTI0_ISR(void)
interrupt 0
{
  frames[bits] = get_Timer_0();
  bits++;
  set_TR0;

  if(bits >= 33)
  {
     received = 1;
     clr_EA;
     clr_TR0;
  }
  set_Timer_0(0x0000);
  P15 = ~P15;
}
```

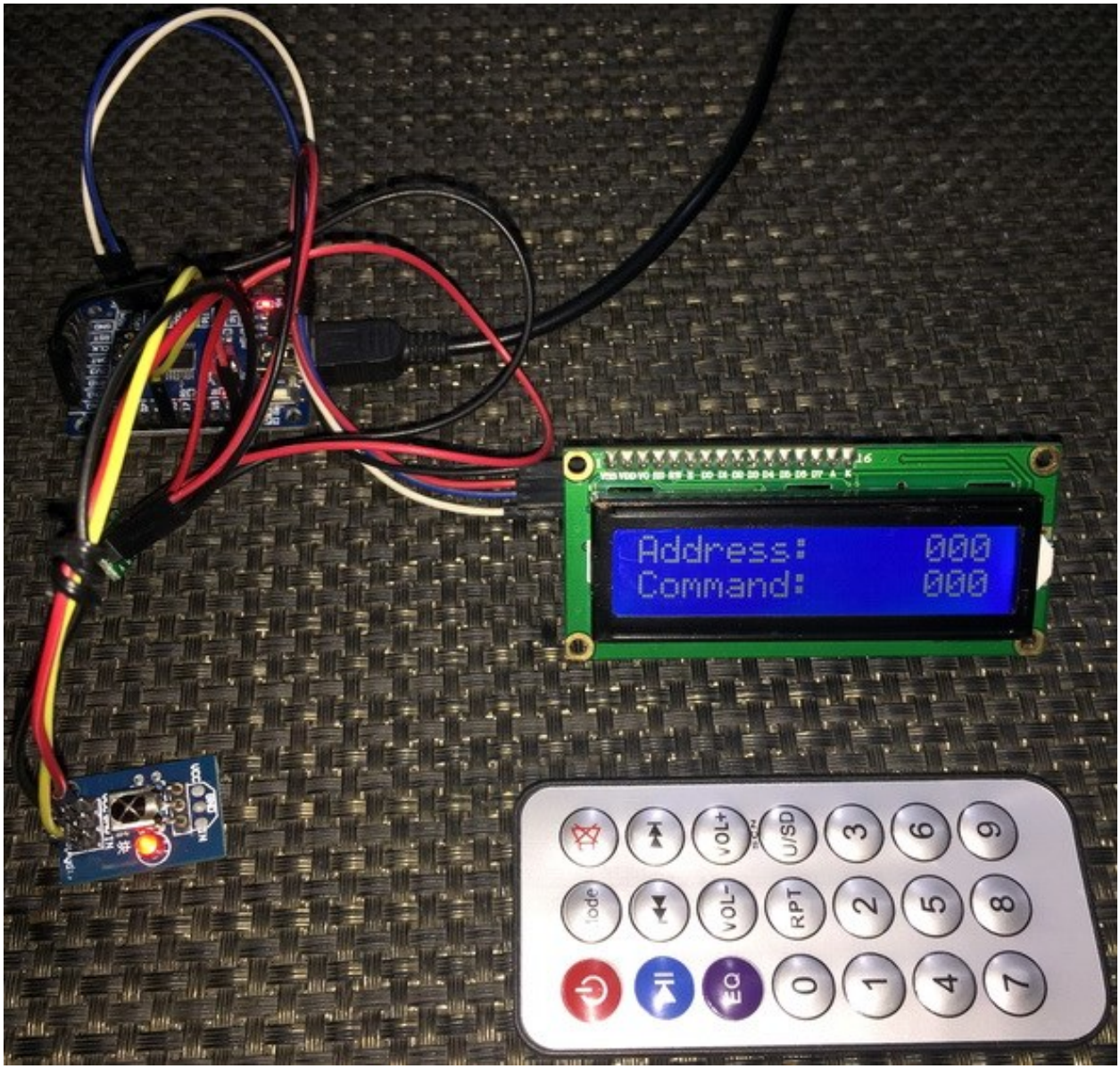
Now the signal is decoded in the ***decode*** function. This function scans a fixed length of time frames for sync, ones and zeros and return the value obtained by scanning the time info of the pulses. In this way, a received signal is decoded.

```
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;
```

```
}
```

In the main loop, the decode info are displayed on a text LCD.

Demo



Demo video: https://youtu.be/9xFYY9zVkFQ

Epilogue

After going through all these topics and exploring the N76E003 to the finest details, I have to say that just like flagship killer cell phones, this is a flagship killer 8-bit microcontroller. In terms of price vs

feature ratio, it is a winner. I'm truly impressed by its performance and hardware features. It offers something like an 8051 but with more advanced modern features. Old-school 8051 users will surely love to play with it.

Every day we talk about software piracy and hacking but little effort is done to prevent them. Students, hobbyists and low-profile business houses can't afford to use the expensive Keil/IAR compiler. They often go for pirated versions of these software and this is a very wrong path to follow. Like I said in my first post on N76E003, Nuvoton has rooms for making its devices more popular by investing on a compiler of its own or at least start doing something with free Eclipse-IDE.

All files and code examples related to Keil compiler can be downloaded from here.

All files and code examples related to IAR compiler can be downloaded from here.

Youtube playlist.

Happy coding.

Author: Shawon M. Shahryiar
https://www.youtube.com/user/sshahryiar
https://www.facebook.com/groups/microarena
https://www.facebook.com/MicroArena

06.08.2018