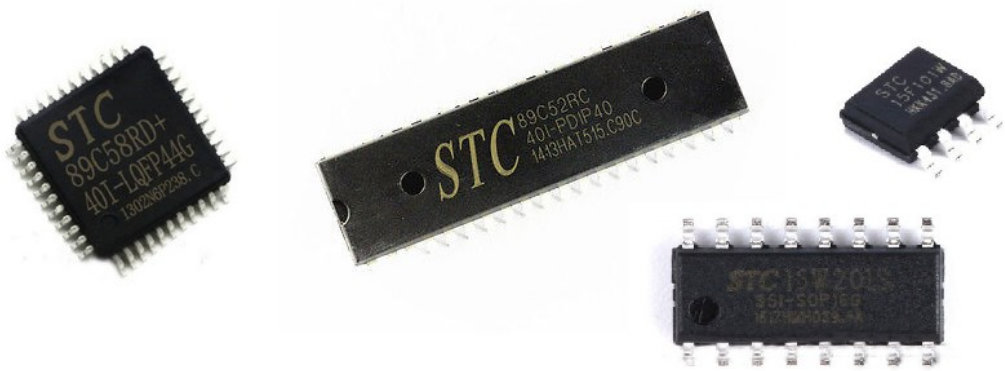
Exploring STC 8051 Microcontrollers

8051 microcontrollers are the first-generation microcontrollers that sparked the modern embedded-system era and established the basic concepts for almost all microcontrollers. In the early 1980s, 8051 microcontrollers were first introduced by Intel. Later other manufacturers like Philips (NXP), Atmel (now Microchip), Silicon Labs, Maxim, etc took the 8051 architecture and introduced their variants of 8051s. Today there are hundreds of such companies which still manufactures this old school legendary micro. of them have even added more features like ADCs, communication peripherals like SPI and I2C, etc that were not originally integrated. There are even some manufacturers who produce micros under their naming convention/branding while maintaining the basic architecture. Recently I covered an article about Nuvoton N76E003 here. It is based on such ideas. STC (not to be confused with *STMicroelectronics*) is a Chinese semiconductor manufacturer that operates in the same way as Nuvoton. STC took the model of 8051 just like other manufacturers and upgraded it to new levels by implementing some vital upgrades, enhancements and additions. It also manufactures standard 8051s which are designed to fit in place of any other 8051s from any manufacturer. At present STC has several different variants of 8051s, ranging from standard 40 pin regular DIP 8051s to tiny 8-pin variants. Some are shown below.



Documentations and Websites

STC microcontrollers are popular in China and Chinese-speaking countries. Owing to this fact, most of the documentation and even the websites are in Chinese. It is hard to get English documentations. Fortunately, we will not be needing anything else other than device datasheet documents which are luckily available both in Chinese and English.

Unlike other manufacturers who maintain one website dedicated to their products and themselves, STC maintains several websites. Most are in Chinese. This creates lot of confusion about STC. Some common STC websites are listed below:

http://www.stcmicro.com

http://www.stcmcu.com

http://www.stcisp.com

http://www.stcmcudata.com

STC 8051s vs Other 8051s

STC 8051s, as stated, offers additional hardware peripherals when compared to standard 8051s. There are some STC microcontrollers like STC89C52RC that are same as the standard ones while some others like STC8A8K64S4A12 are more robust with many advanced features. Some key differences between standard 8051s and STC micros are discussed below:

- Packages / Sizes
  STC offers microcontrollers in various DIP and SMD IC packages. Thus, instead of using a 40-pin DIP package microcontroller to solve a problem that can be solved with an 8-pin SMD low cost microcontroller, we can avoid using a big microcontroller and thereby save valuable PCB space. Most STC 8051s are, by the way, 100% pin-compatible with other 8051s. This feature makes STC micros easy and viable replacements for devices that use standard 8051s.

- Speed / Operating Frequency
  STC microcontrollers are relatively faster than common 8051s as they can operate at higher clock frequencies. For example, AT89S52 has a maximum operating frequency of 33MHz while STC89C52RC can be clocked with an 80MHz source.

- Additional Hardware Peripherals
  Some STC microcontrollers have in-built ADC, EEPROM, watchdog timer, external interrupt pins and other peripherals. Some are even equipped with higher storage capacities. These are not available in typical 8051s.

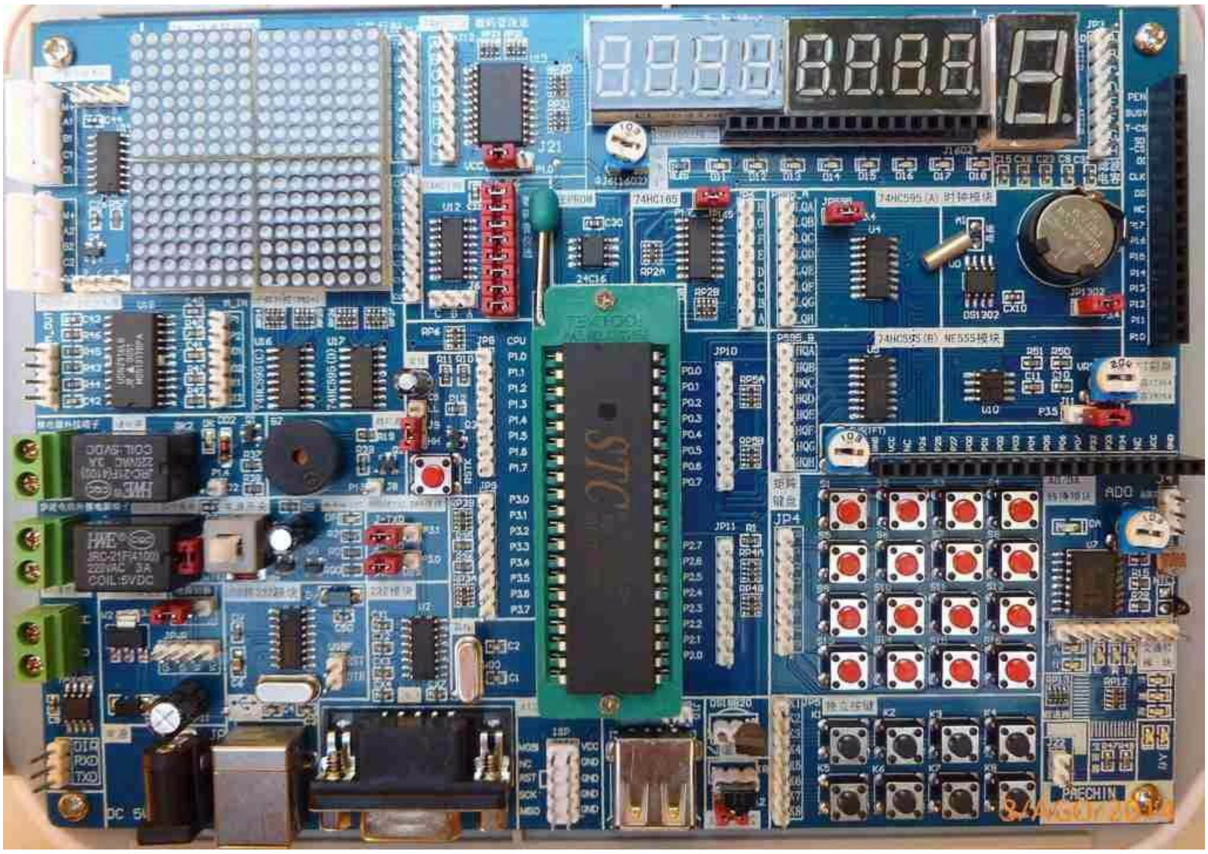
- Operating Voltage
  Most 8051 micros need 4.0 – 5.5V DC supply voltage. Some can operate with 3.3V supplies too. Same goes for STC micros. However, there are some STC micros that are designed to operate at yet lower voltage levels. STC offers low power MCUs that operate between 2.0 – 3.6V and general-purpose micros that can operate between 3.6 – 5.5V. The operating voltage ranges and low power consumption figures of STC micros make them well-suited for battery and solar-powered devices.

- Programming Interface
  Most 8051s require a parallel port programmer while some require serial port programmer or separate dedicated programmer hardware. STC micros on the other hand can be programmer with a serial port programmer and so there is no need to buy a dedicated programmer. A simple USB-TTL serial converter can be used to load codes into STC micros.

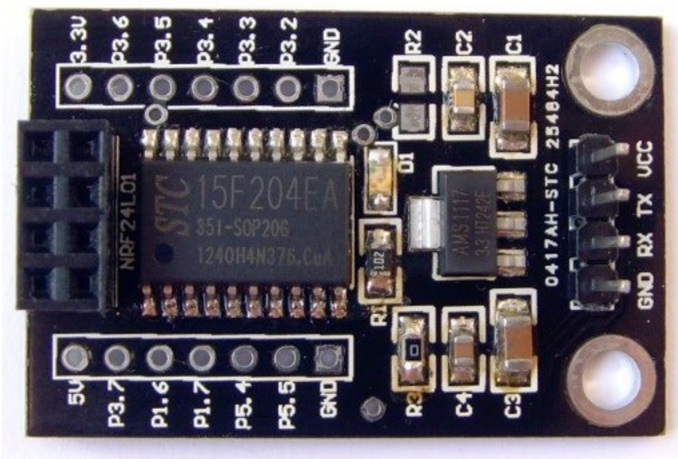
- Other Minor Differences
  Other areas of differences include added/reduced functionalities/features. In some STC micros, there additional options for GPIOs, timers, etc while in some other devices these extras are not observed. For example, in STC89C52RC, there is 13-bit timer mode for timers 0 and 1 but this feature is absent in STC15L204EA. Likewise in STC15L204EA, there are many ways for setting up GPIOs which are not present in STC89C52RC.

Hardware Tools

From AliExpress, DX, Alibaba and other similar websites/stores, you can buy any STC development board of your choice. Alternatively, you can buy common STC chips and use them with your existing development board or setup a bread-board arrangement.



One such board is shown above. These boards have lot of hardware devices like external 24 series EEPROM, I2C ADC-DAC, communication and display interfaces, etc already embedded and ready for go. Such boards are, thus easy to use and need less wiring. However, boards as such are relatively expensive and big than the one shown below:



This board is designed to bridge a serial interface between a host micro and a nRF24L01 2.4GHz wireless communication module. However, that doesn't restrict us from using the STC15F(L)204EA micro embedded in it. If you just want to give STC micros a shot with very little investment then this sort of board is all that you can ever expect.

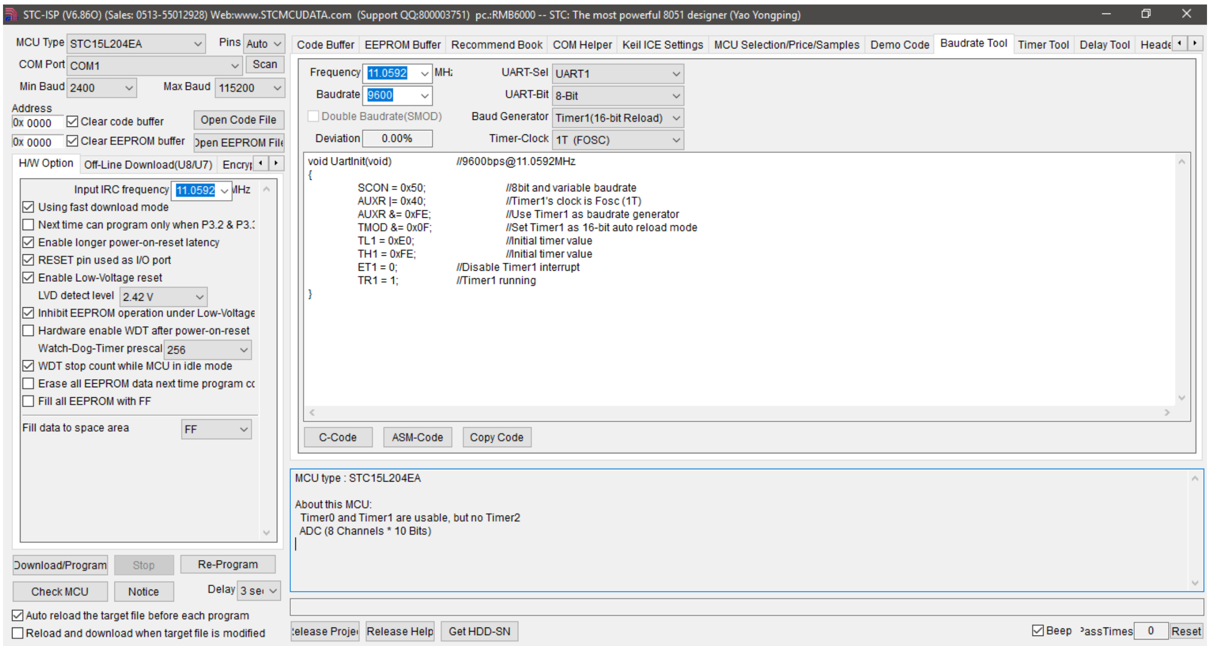
In my tutorials, I'll be using both kinds of boards but the main focus will be towards STC15L204EA or similar slightly non-standard 8051s since they are not like playing with typical 8051s.

We will also need an USB-serial converter for uploading code.

Apart from these, some regularly used hardware items like LCDs, sensors, wires, etc will be needed. These can easily be found in any starter kit and most are available in any hobbyist's collection.

Software Tools

Only two software tools will be needed. The first is Keil C51 compiler and second STC ISP tool.



STC ISP tool can be downloaded from [here](). This is one helluva tool that has many useful features. It a programmer interface, a code generator, serial port monitor, code bank and many other stuffs. It sure does make coding STC micros lot easier than you can possibly imagine.

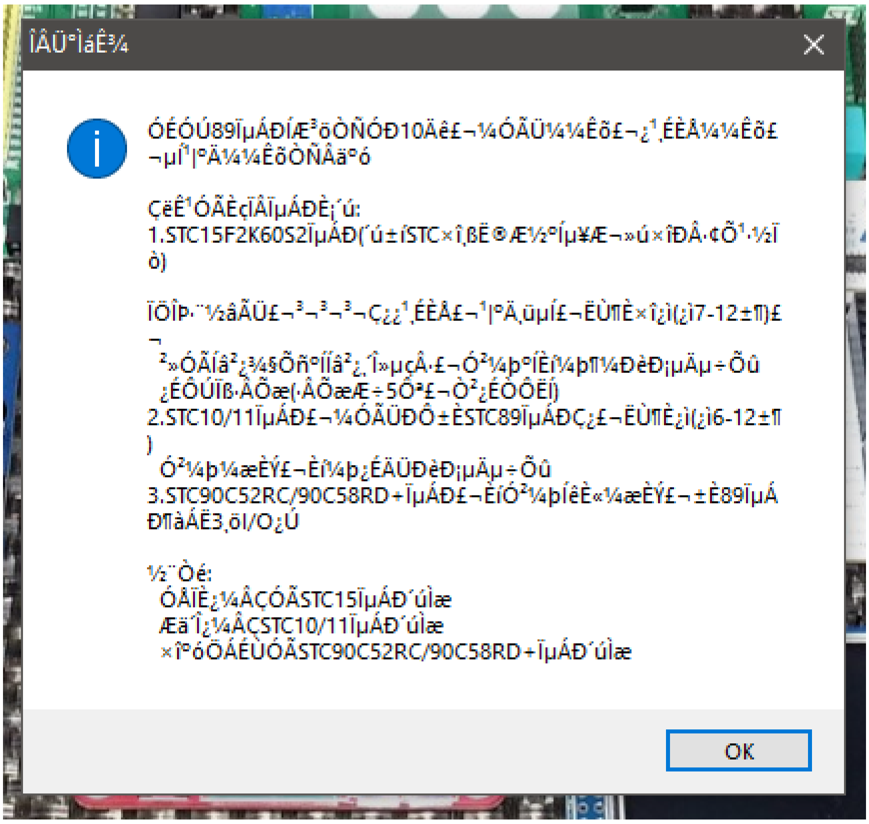
Keil C51 C compiler will be needed to code STC micros. At present, Keil is the only C compiler that can be used reliably to code STC micros. STC documentations speak of Keil mostly. If you want to use some other compiler like IAR Embedded Workbench, MikroC for 8051, etc other than Keil, you have to add the SFR definitions of your target STC micros and do other stuffs to familiarize it with the STC micro target. Alternatively, you can use models of other similar 8051 model. For example, STC89C52RC is similar to AT89S52. You can use codes for such interchangeably. However, this method won't work in cases where we have more hardware peripheral than an ordinary 8051 micro. STC15L204EA, for instance, can't be used like ordinary 8051 or like STC89/90 series micros.

Programming STC Microcontrollers

By default, STC microcontroller database is absent in Keil. It is imperative that this database is added to Keil when using it for STC micros for the very first time. Though this database is not complete in the sense that not chips are enlisted in it, it is still a must or else we will have to use unconventional coding methods by using models of similar microcontrollers of different manufacturers. Personally, I hate unconventional tactics because why use such methods when we can add the database easily. We only have to add this database once.

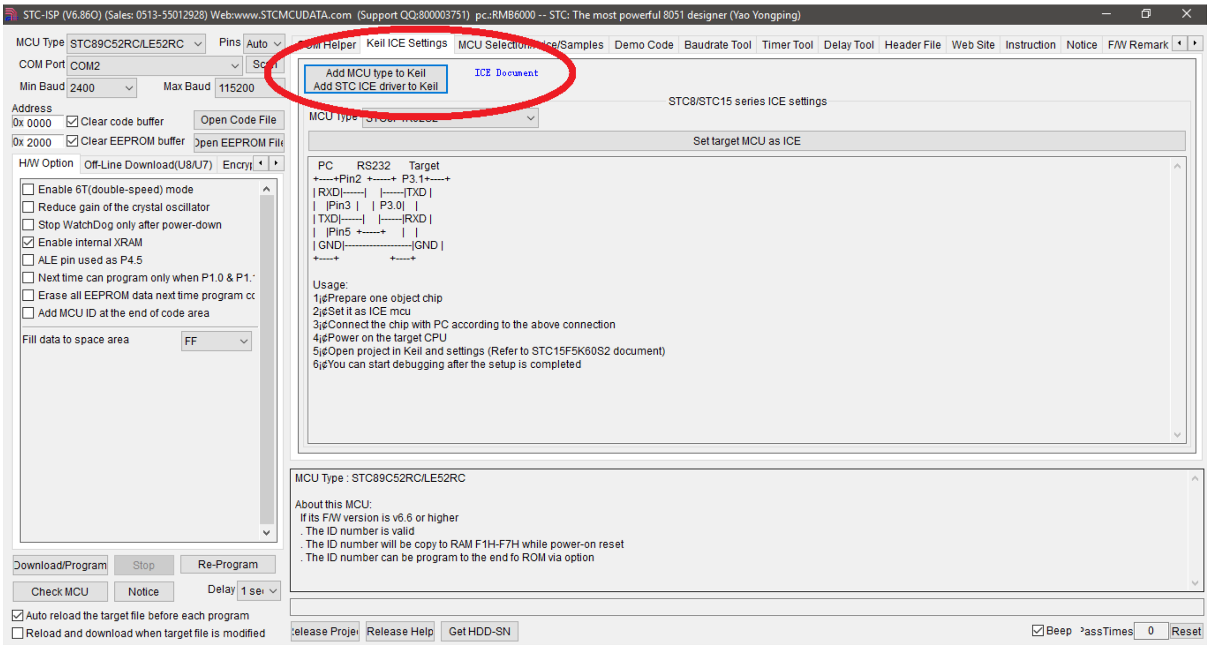
First run the STC-ISP tool.

After clicking the STC-ISP tool icon, the application starts and the following window may appear:

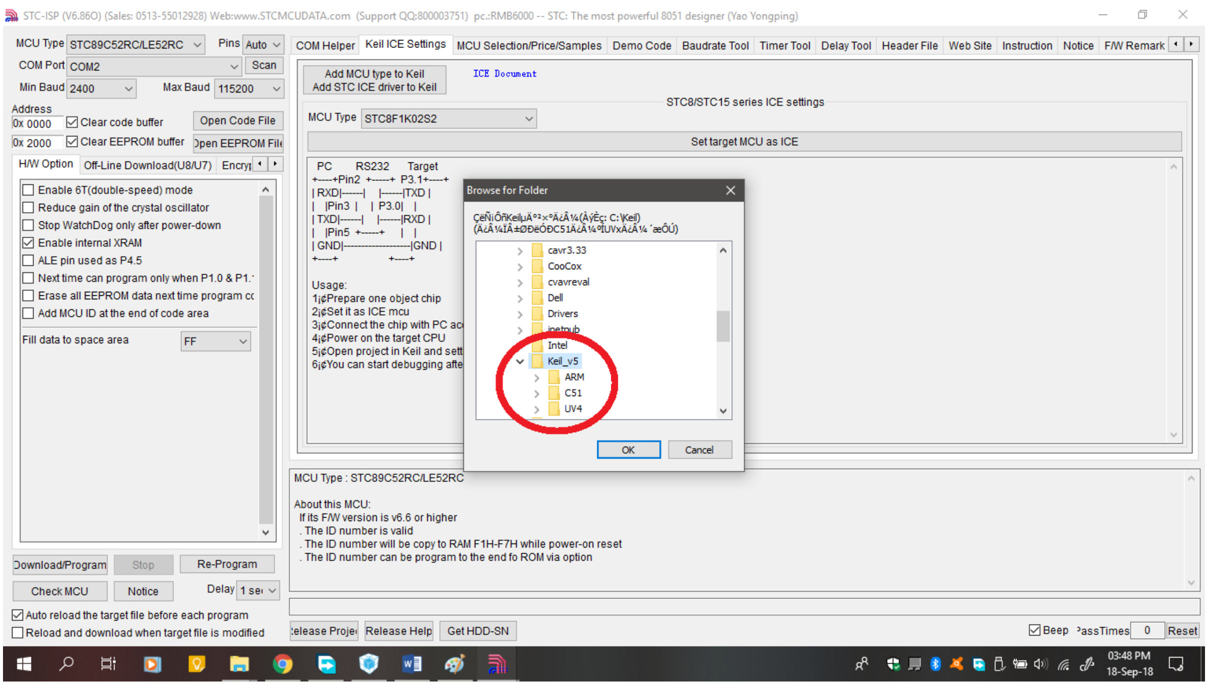


Don’t worry. It is not an error or garbage text window. It appears so if you don’t have Chinese font database installed in your PC and so just click OK to continue. Recent versions of STC-ISP don’t have this issue.

Once the application starts, navigate to ***Keil ICE Settings*** tab and locate the highlighted button as shown below:

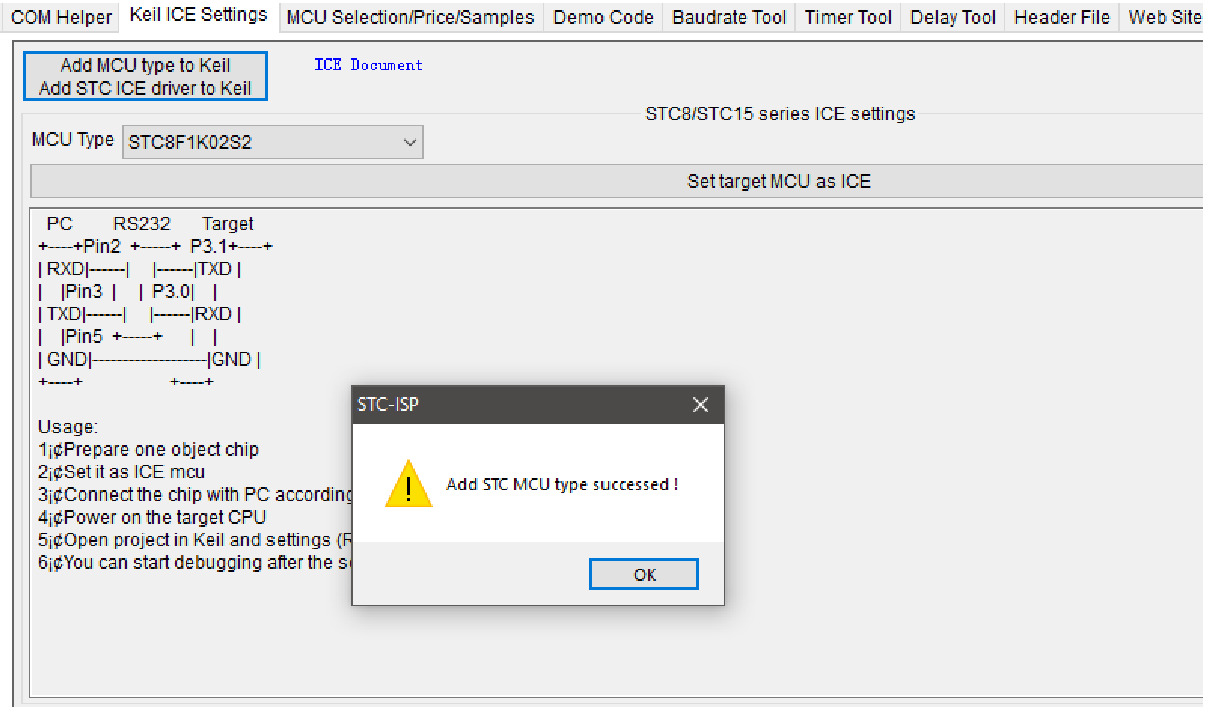


Now just navigate to Keil installation folder and hit OK as shown below:



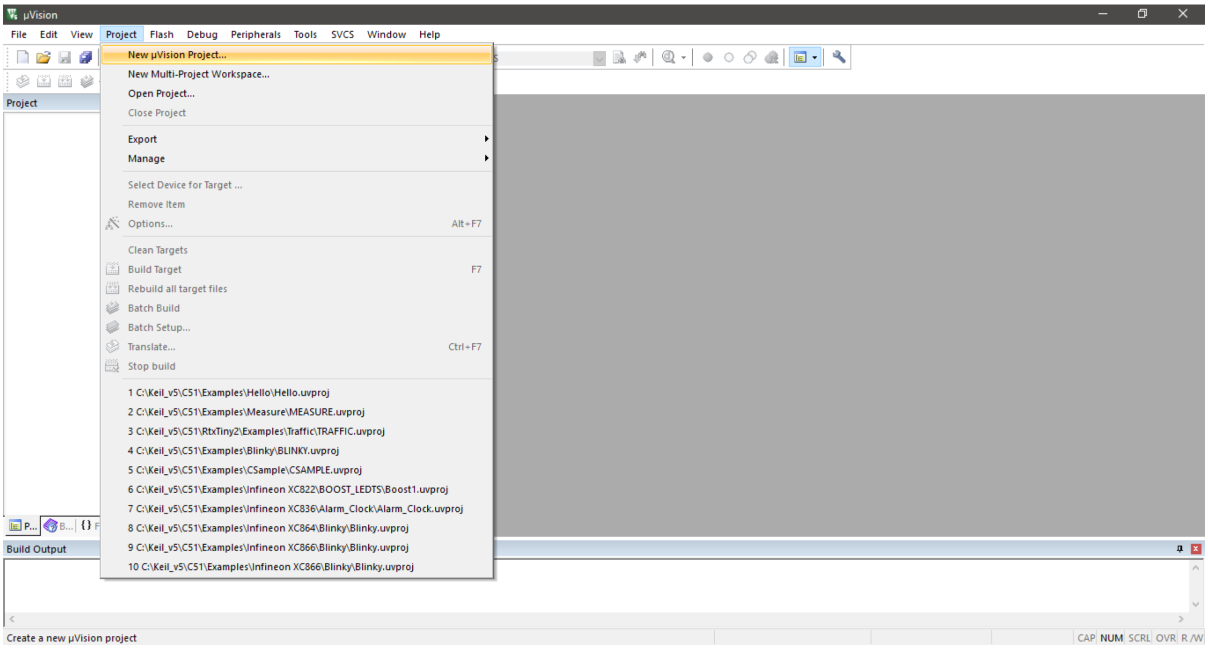
Selecting wrong folder will end up with an error and the database won't be installed.

If the database addition is a success, you'll get the following message:

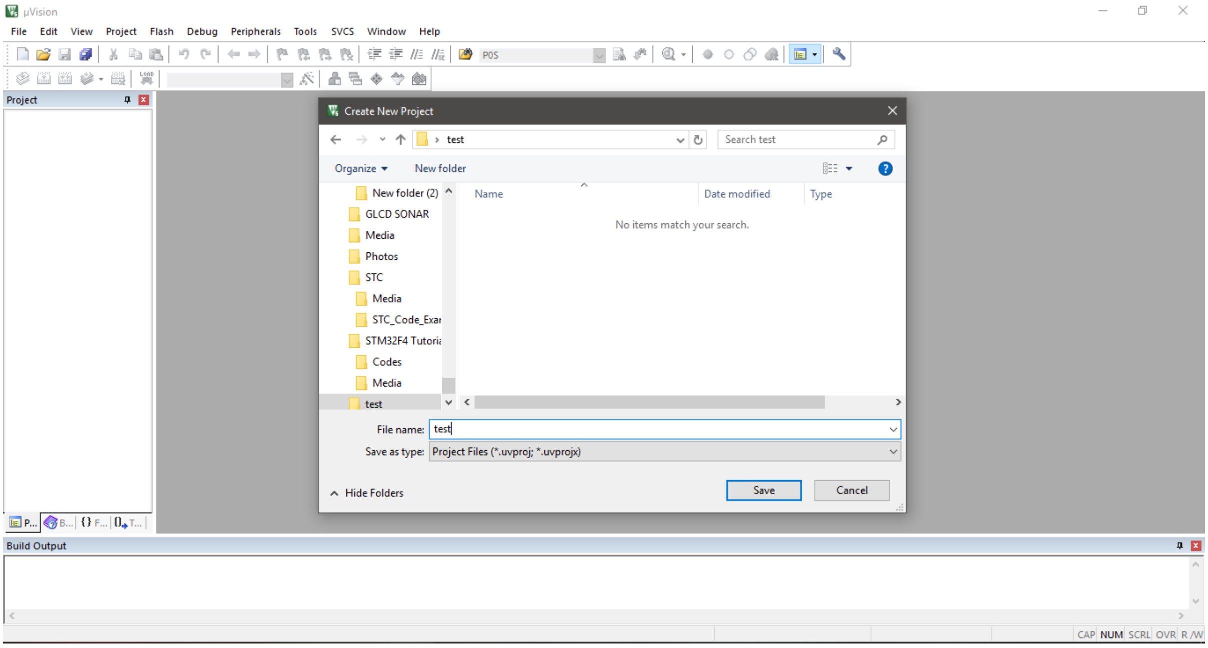


Now run Keil C51 compiler.

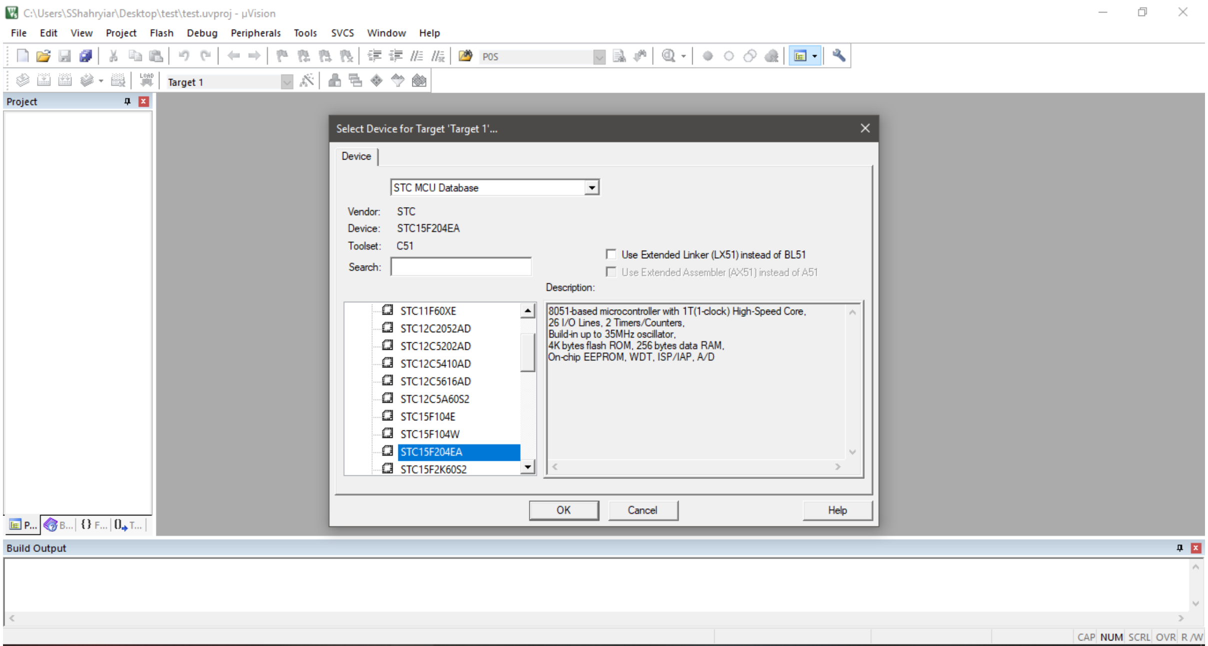
Go to ***Project >> New µVision Project…*** as shown below:



Give your project a folder and a name as shown below:

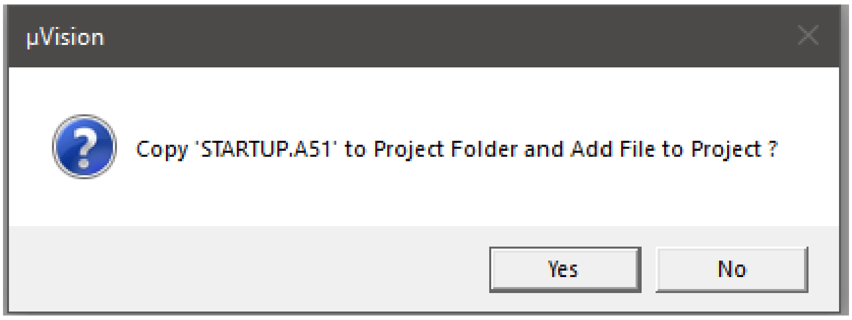


Select STC Database and appropriate chip or similar part number.

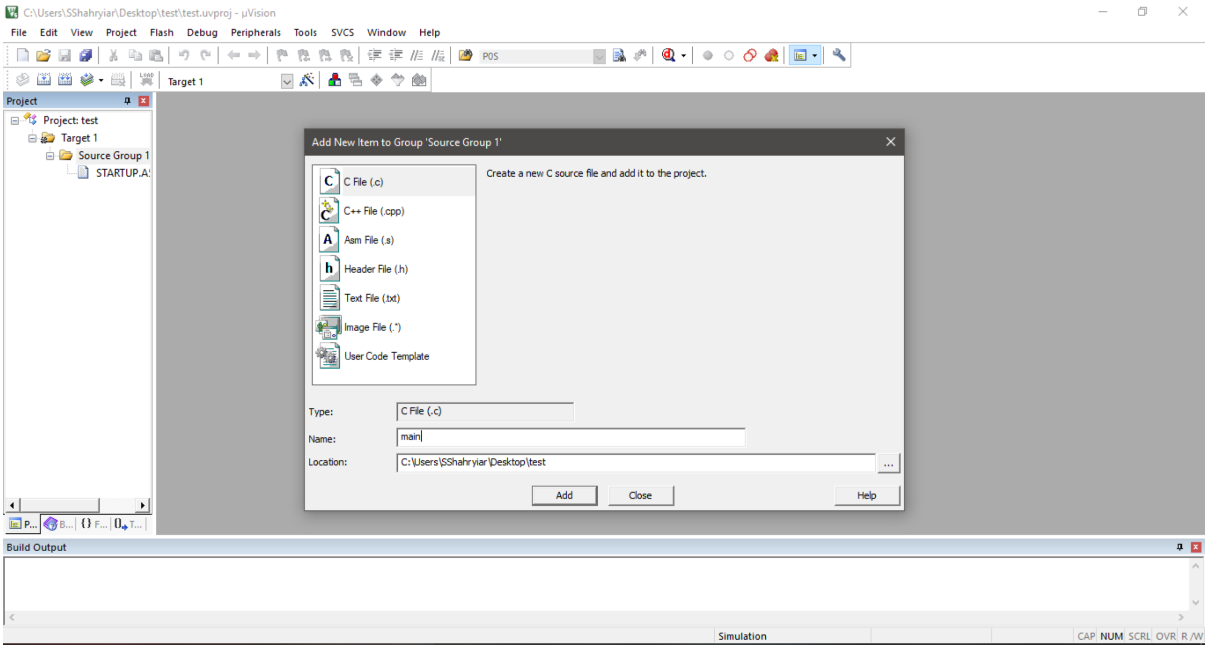


Please note that your target chip may not be in the list. For example, the L-series chips are not enlisted in the database and so they are absent in the list. STC15L204, for example, is not shown in the list. You can use STC15F204EA instead of it as they are similar stuffs. The only difference is their power consumptions. You have to use such tricks when your target chip is not listed. Make sure that the model you selected matches with the target chip or else things may not work properly.

After chip selection, add the startup assembler file to your project.



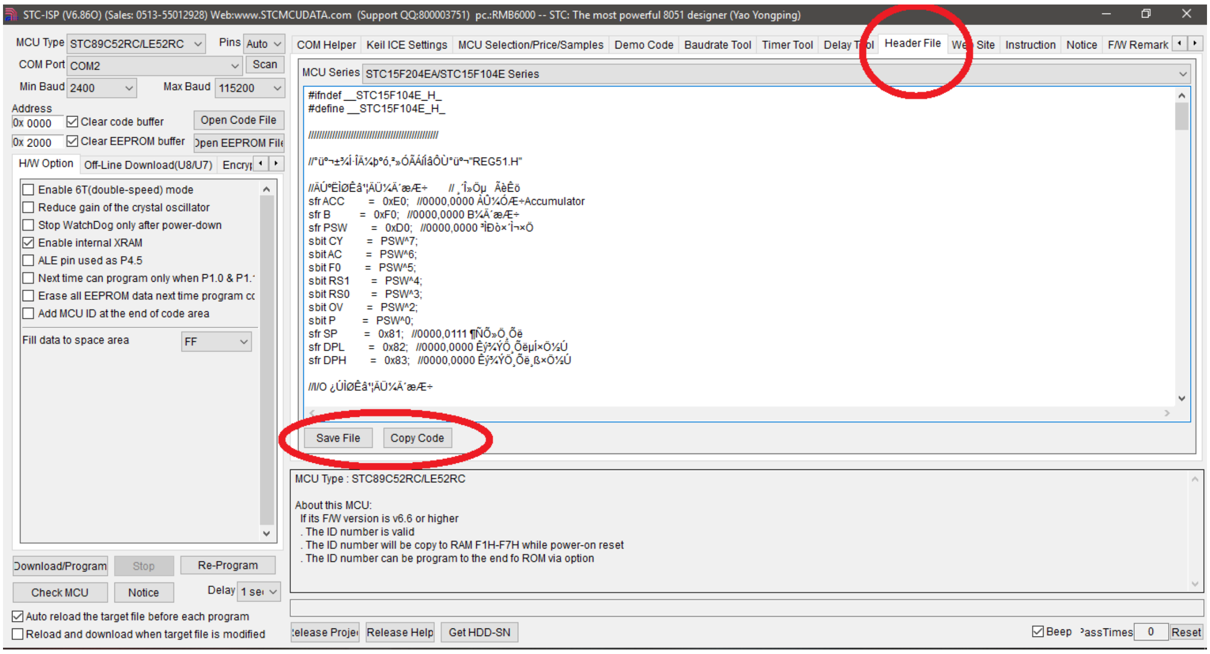
By default, Keil doesn't add/create any file and so you'll see that the project folder has no main source file. We'll have to create one main source file and add additional files if needed.



Still we are not ready to start coding. This is because we have not yet added SFR definition header file and other custom optional files.

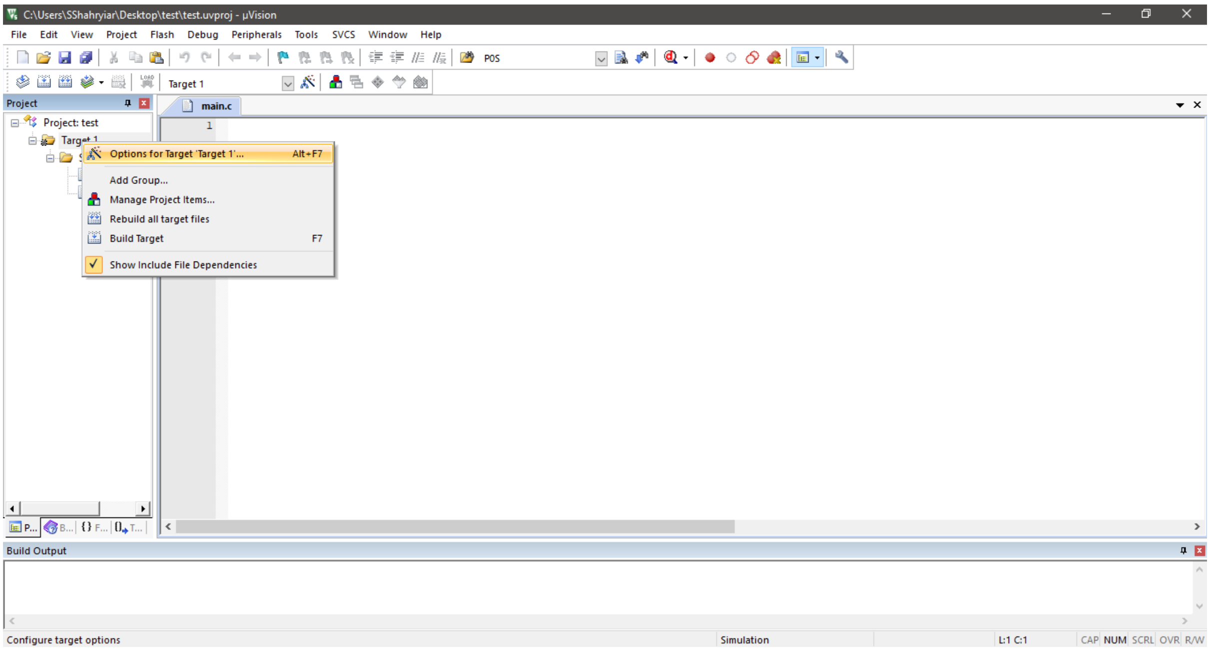
Again, we need to take the help of STC-ISP tool.

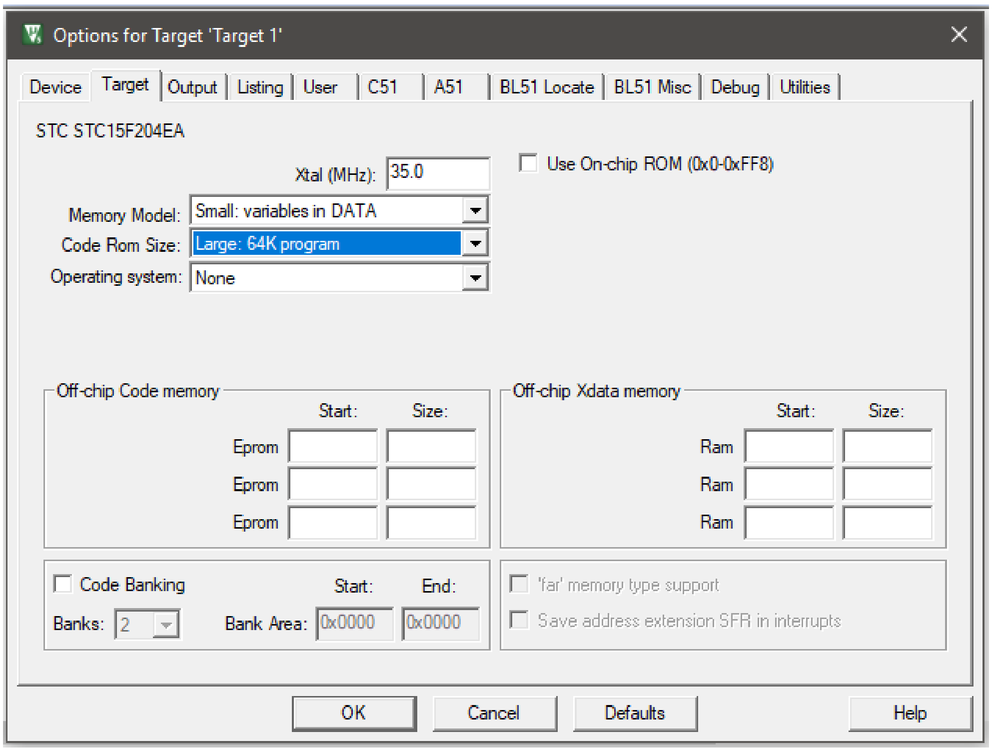
In STC-ISP tool, locate the ***Header File*** tab and select appropriate MCU series.



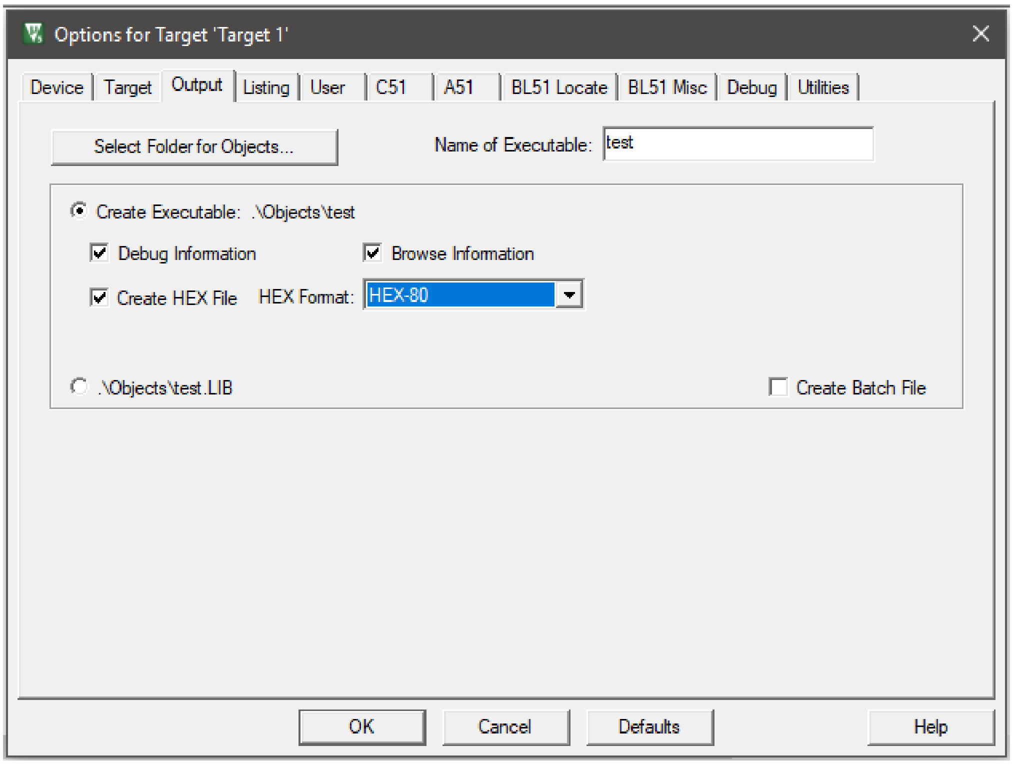
Save or copy the file to your desired location.

In Keil, go to target options by right click the folder icon and set target options as shown in the following screenshots:

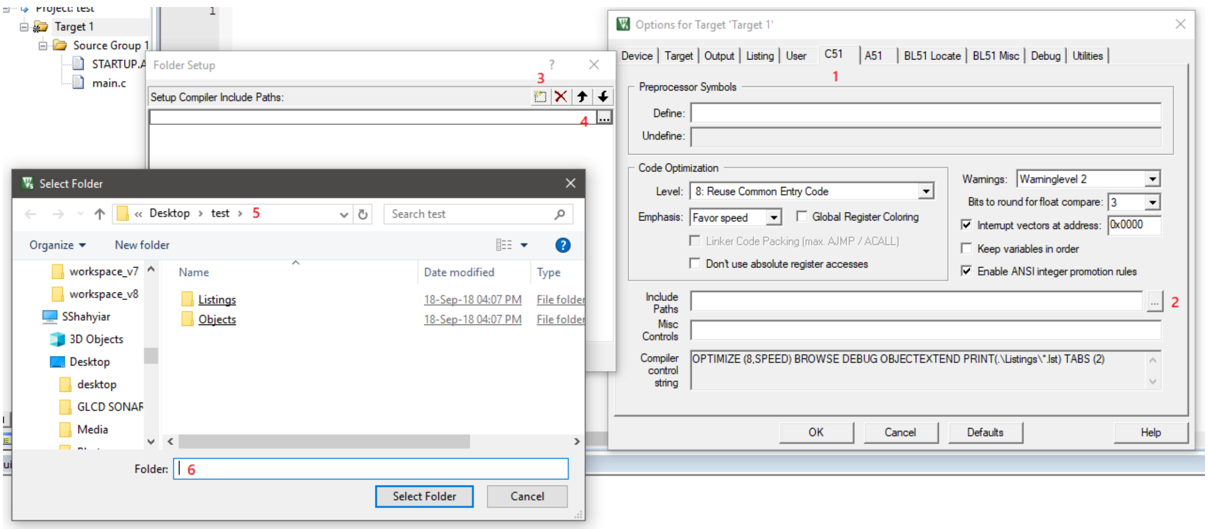




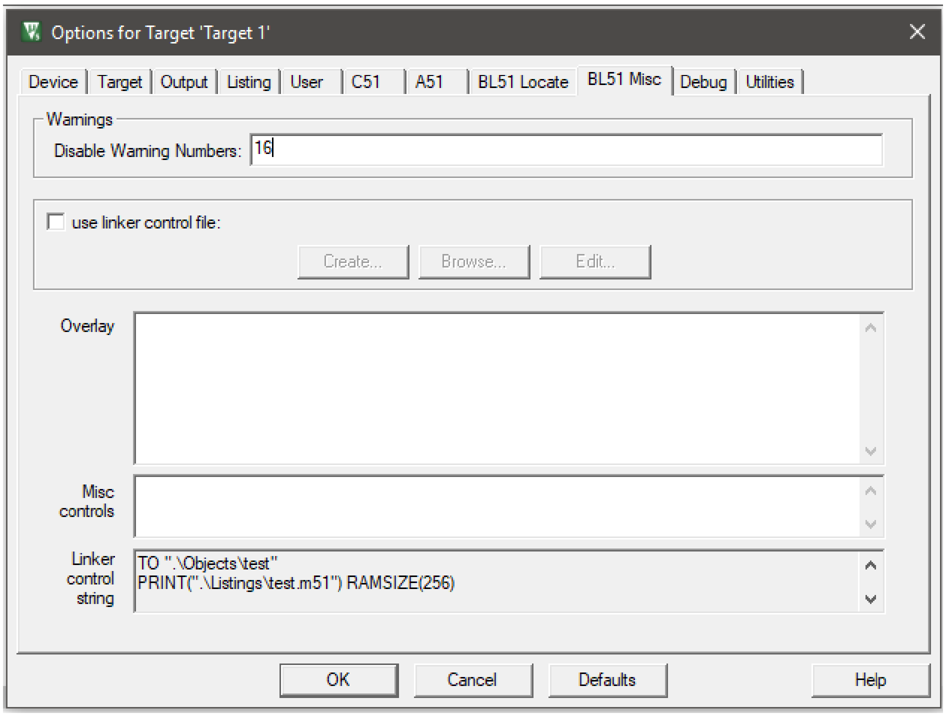
From this window select clock frequency and memory model.



Select ***Create HEX File*** from this window as this file will be uploaded in the target chip.



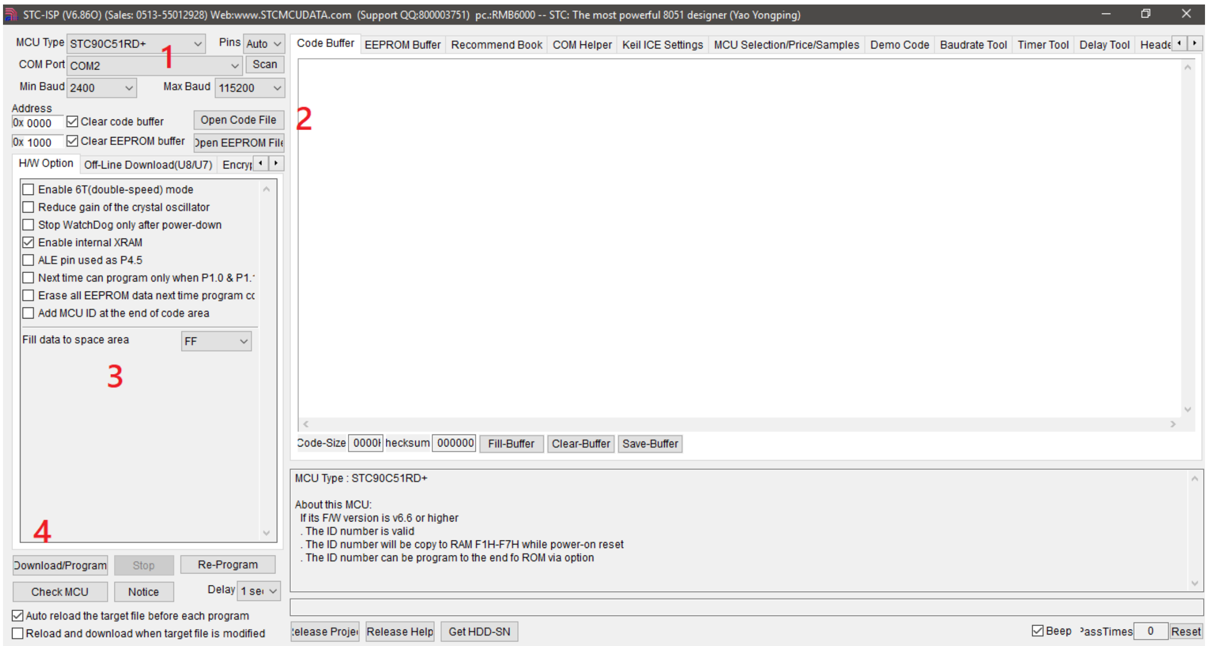
This section above is highly important because here we have to show the compiler the locations of the header or include files. Check the step numbering carefully.



Lastly disable ***Warning Number 16***. Now, we are good to code.

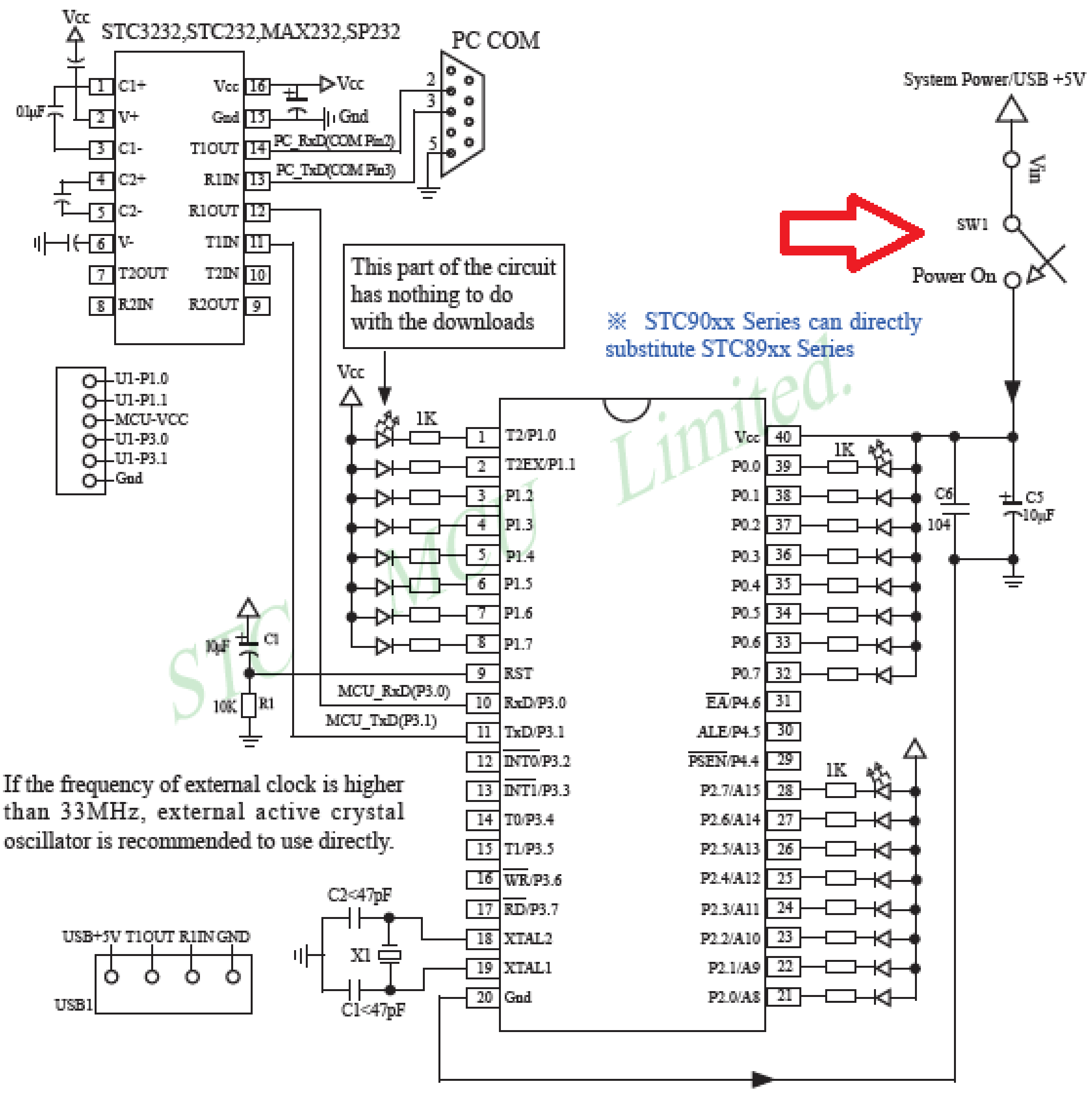
I made a small YouTube video on all the above discussed processes. If you didn't understand some part or if you are confused at some point, you can watch the video here.

Now let us discuss about uploading codes to STC micros. The most advantageous part is the fact that we don't need to invest on a dedicated programmer as with other microcontrollers as a simple USB-serial converter will do the job. However, on first go things may look confusing.



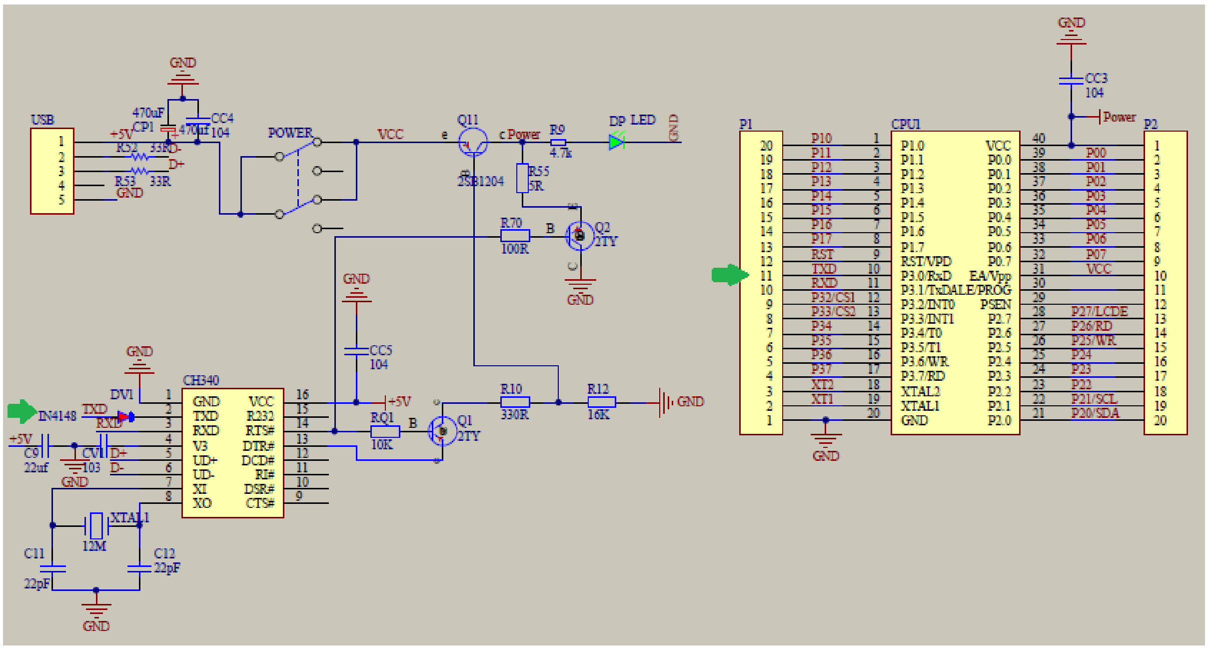
Shown above is the STC-ISP tool's screenshot with numbers. We have to follow them one-by-one and in incremental order. 1 denotes that we must select the right COM port and part number. You can use ***Windows Device Manager*** to find out which COM port is being used for uploading code. Step 2 is to select the target HEX code file. Optionally in step 3, we can set some additional internal MCU parameters. When everything has been set properly, we can hit the ***Download Program*** button shown in step 4. If the target microcontroller or board is already powered then the code won't be uploaded. This is the confusing part because it should have been the other way.

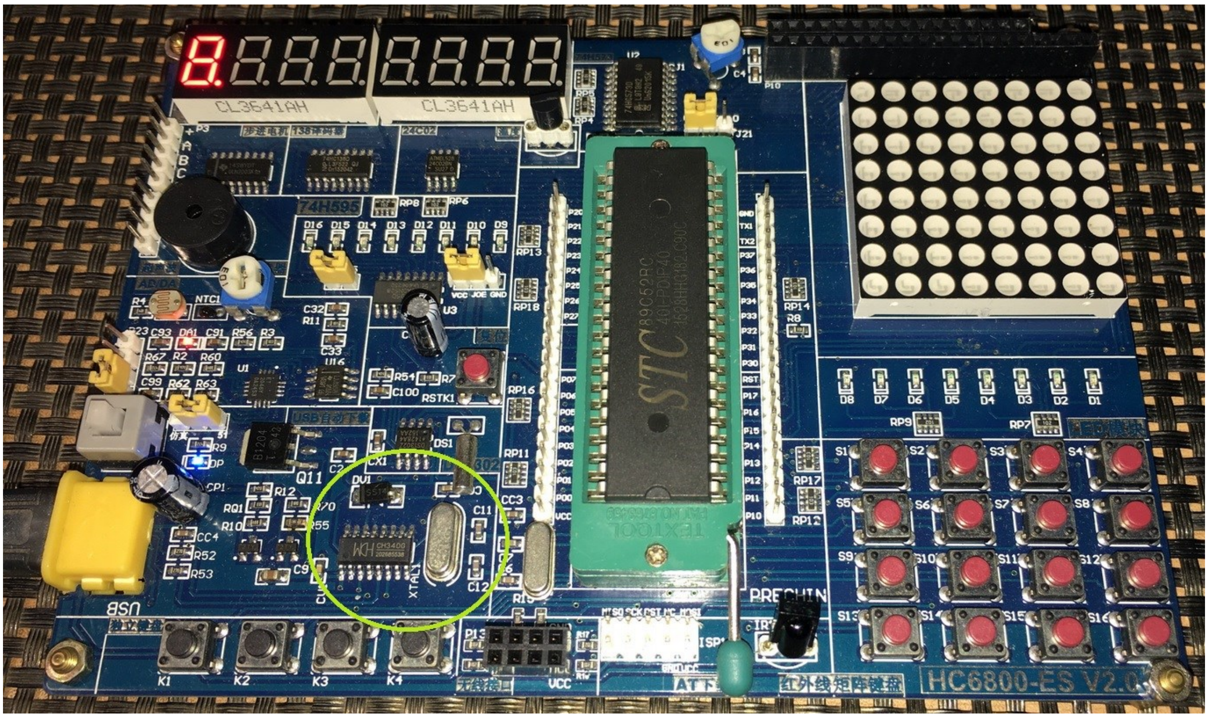
Notice the red arrow in the schematic below. The code is not uploaded by holding and releasing the reset button for some time or by pulling high or low some special pin or by some other means. We need to create a handshake between the PC and the target MCU and this is done when the MCU is powered off and then powered on, i.e., when the micro is powered up.



This is why the red arrow highlights the power switch in the schematic. Though the schematic shows a MAX232-based converter, we can use a USB-serial converter instead.

All of these steps are demoed in this video. Note that no external USB-serial converter/cable can be seen in the video as it is embedded in the board. The following schematic and photo will make this fact clearer. This is why most STC development boards come with such arrangement and without any programmer.





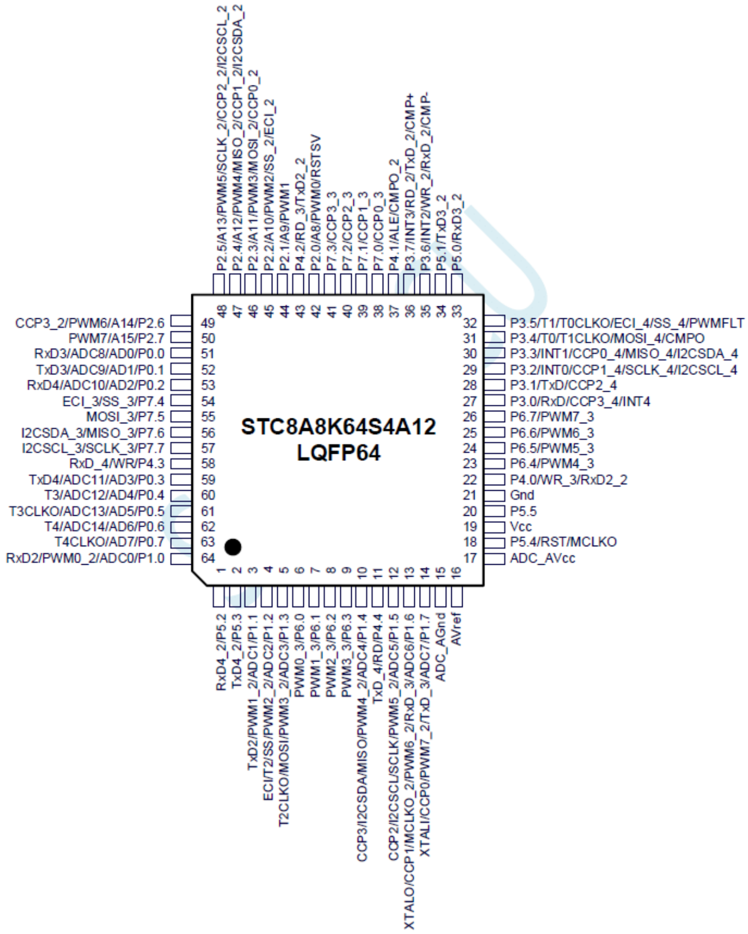
About STC8A8K64S4A12 Microcontroller and its Development Board

Many Chinese microcontroller manufacturers develop awesome and cheap general-purpose MCUs using the popular 8051 architecture. There are many reasons for that but most importantly the 8051 architecture is a very common one that has been around for quite a long time. Secondly, manufacturing MCUs with 8051 DNA allows manufacturers to focus less on developing their own proprietary core and to give more effort in adding features. Holtek, Nuvoton, STC, etc are a few manufacturers to name.

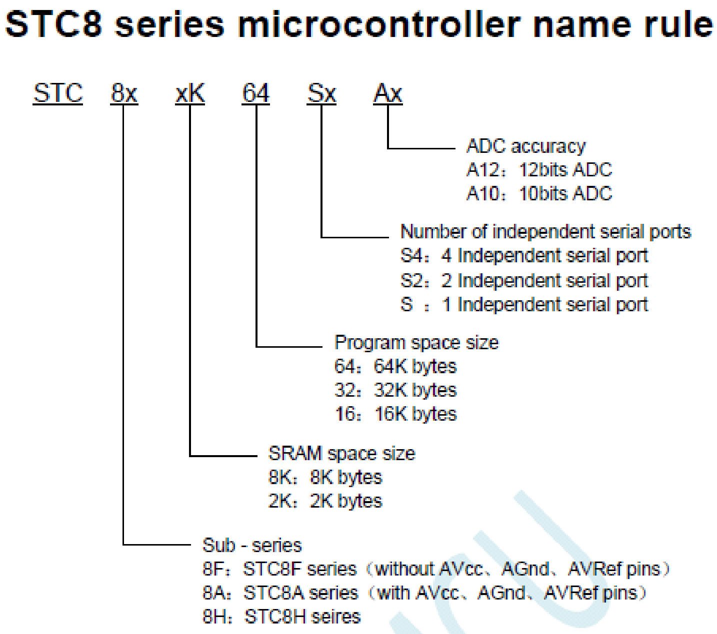
Rather than mastering a good old 8051-based microcontroller like the AT89C52 or similar, it is better to learn something new that has many similarities with that architecture. As mentioned earlier, STC has various flavour of microcontrollers based on 8051 cores. STC8A8K64S4A12 of the STC8 family is one such example. Here for this documentation, I will be using this MCU specifically. In short, it is a beast as it offers lot of additional hardware that are usually not seen in regular 8051s. Some key features are listed below. The red box highlights the STC8A8K64S4A12 micro in particular.

Microcontroller Model	Operating Voltage(V)	Flash Program Memory 100K times bytes	Large Capacity Expansion SRAM bytes	Powerful dual DPTR Increase or Decrease	EEPROM 100K times bytes	I/O maximum number	Serial ports Power-down wake-up	SPI	I²C	Timer/Counter(External Pow-down Wake-up)	16 bits advanced PWM Timers	15 bits Enhanced PWM(Dead Zone Control)	PCA/CCP/PWM(can be external interrupt)	Power-down wake-up timer	15 High speed ADC(8 PWM as 8D/A use)	Comparators(1 A/D, ext brownout detection)	Internal Low-vol Detection interrupt Pow-wk	Watchdog Reset timer	Internal Reset(optional reset threshold vol)	Internal Clock(24MHz Adjustable)	External clock output and reset	Program encrypted transmission	Set password for next update procedure	Support RS485 download	Support USB download	Online simulation
STC8A8K16S4A12	2.0-5.5	16K	8K	2	48K	59	4	Yes	Yes	5	-	8	4	Yes	12位	Yes	Yes	Yes	4lev	Yes	Yes	Yes	Yes	Yes	Yes	Yes
STC8A8K32S4A12	2.0-5.5	32K	8K	2	32K	59	4	Yes	Yes	5	-	8	4	Yes	12位	Yes	Yes	Yes	4lev	Yes	Yes	Yes	Yes	Yes	Yes	Yes
STC8A8K60S4A12	2.0-5.5	60K	8K	2	4K	59	4	Yes	Yes	5	-	8	4	Yes	12位	Yes	Yes	Yes	4lev	Yes	Yes	Yes	Yes	Yes	Yes	Yes
STC8A8K64S4A12	2.0-5.5	64K	8K	2	IAP	59	4	Yes	Yes	5	-	8	4	Yes	12位	Yes	Yes	Yes	4lev	Yes	Yes	Yes	Yes	Yes	Yes	Yes

I chose STC8A8K64S4A12 for this tutorial for all of its rich features. My favourite features include 12-bit ADC, multiple timers, reduced EMI feature and PCA module.



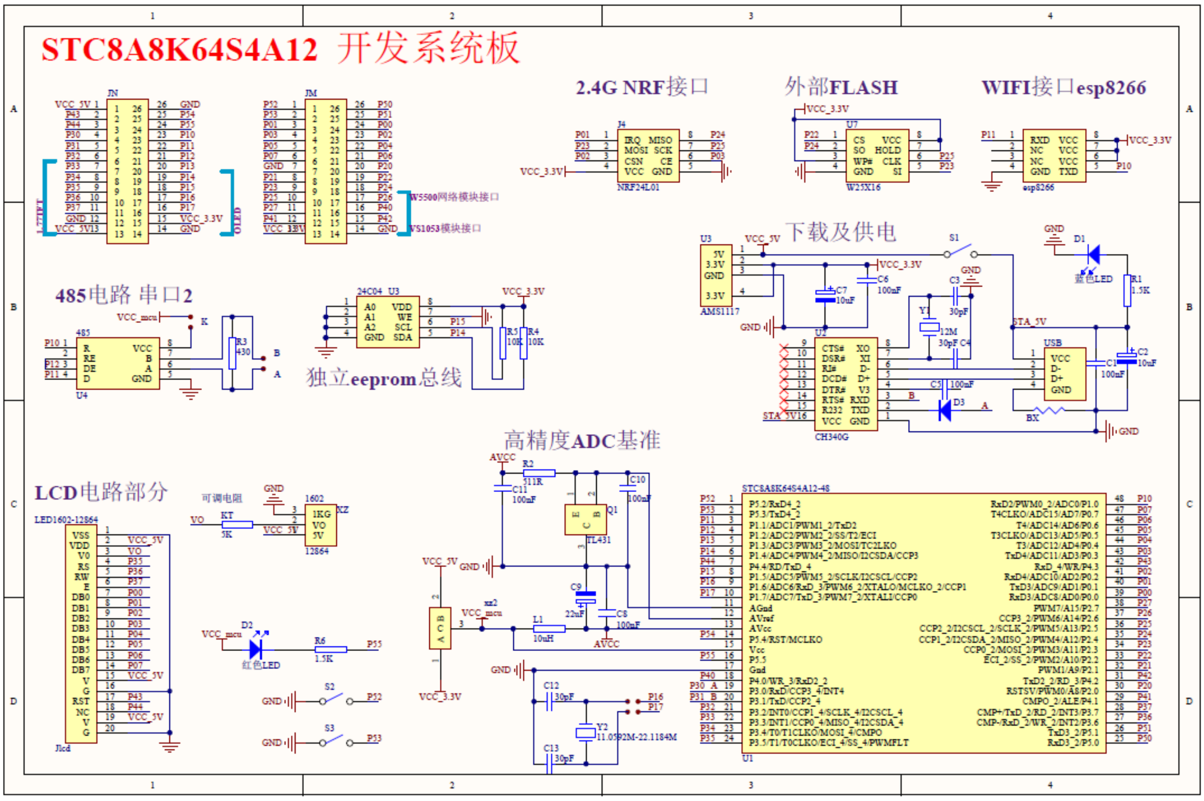
Now let's see the naming convention of STC microcontrollers. The figure below shows us what the name of a STC8 micro means:



The name STC8A8K64S4A12 is quite a mouthful and given the nomenclature info, STC8A8K64SA412 is actually a STC8 series micro with on-chip 12-bit ADC, 8kB SRAM, 64kB code space and 4 hardware serial (UART) ports. Apart from these hardware thingies, the naming convention does not reveal other cool features, for if it had been so, the device name would be even longer.

Initially I wanted to make this tutorial with the STC15F(L)204EA microcontroller but later I changed my mind because STC8A8K64S4A12 is much richer in hardware peripheral terms than STC15F(L)204EA, not to mention several similar hardware are present in both models of microcontroller. Since I planned to use STC8A8K64S4A12, I waited for the arrival of the board shown below before completing this work. Although this development board is an official board, it has been smartly designed for fast learning and rapid deployment of projects. As a matter of fact, if someone learns about this micro with this board, he/she will rule over all of STC's 8051-based line-up.

This board has the following schematic and it will be needed throughout this tutorial.



On board, we have connectors for OLED display, GLCD, LCD, TFT Display, nRF24L01 transceiver, ESP8266 Wi-Fi module, etc. We also have on board W25x16 flash, 24C04 EEPROM, RS485 communication bridge and a CH340G USB-serial converter that doubles as an on-board programmer.

My Customized BSP

To deal with the vast array of peripherals of STC8A8K64S4A12, I needed something to quickly deploy projects without going through the registers every time. Having gotten the idea of **Board Support Package** (BSP) while working with Nuvoton N76E003, it was time for me to develop something similar for STC micros here as officially STC does not have such beautiful software implementation for their microcontrollers. However, STC does provide lot of both C and assembly language examples in their reference manuals and programmer GUI unlike other manufacturers. Those examples, though helpful, do not fit in all possible scenarios and often incomplete in terms of meaning. By having a BSP, we can use our micro's hardware peripherals more efficiently with ease and in a wide variety of ways. We would no longer need to create and call functions for hardware peripherals every time. Nuvoton, TI, STMicroelectronics, Silicon Labs and many other mainstream microcontroller manufacturers are tooling various methods to reduce coding efforts and aid in rapid code development.

It is not an easy task to go through an entire reference manual, reading and trying out everything one-by-one. However, I had to do it no matter how painstaking job it was. This is because firstly, I hate setting registers repetitively every single time when I want to make a new project and secondly, I want to make work easy so that less time, resource and effort are spent. I also have a tendency to forget things quickly. This solution will be as such that it can be modified easily and ported for other STC microcontrollers as well.

I have developed my BSP for STC8A8K64S4A12 in an orderly fashion. There are header files for each hardware peripherals that contain all necessary functions and definitions. An example of STC8A8K64S4A12's watchdog timer's header file is shown below:

```
/*
    Watchdog Overflow Time = ((12 * 32768 * 2^(WDT_PS + 1)) / Sysclk)
*/
#define WDT_div_factor_2                            0x00
#define WDT_div_factor_4                            0x01
#define WDT_div_factor_8                            0x02
#define WDT_div_factor_16                           0x03
#define WDT_div_factor_32                           0x04
#define WDT_div_factor_64                           0x05
#define WDT_div_factor_128                          0x06
#define WDT_div_factor_256                          0x07

#define WDT_set_prescalar(value)                    do{ \
                                                        WDT_CONTR &= 0xF8; \
                                                        WDT_CONTR |= value; \
                                                      }while(0)

//CNT_mode
#define WDT_stop_counting_in_idle_mode              0x00
#define WDT_continue_counting_in_idle_mode          0x08

#define WDT_start                                   bit_set(WDT_CONTR, 5)

#define WDT_get_overflow_flag                       get_bit(WDT_CONTR, 7)
#define WDT_clear_overflow_flag                     bit_set(WDT_CONTR, 7)

#define WDT_reset                                   bit_set(WDT_CONTR, 4)

#define WDT_clear                                   do{ \
                                                        WDT_CONTR = 0x00; \
                                                      }while(0)

#define WDT_setup(CNT_mode, PS)                     do{ \
                                                        WDT_clear; \
                                                        WDT_CONTR |= CNT_mode; \
                                                        WDT_set_prescalar(PS); \
                                                      }while(0)
```

Now let's see what the registers and their settings look like:

Symbol	description	address	Bit address and symbol								reset value
			B7	B6	B5	B4	B3	B2	B1	B0	
WDT_CONTR	Watchdog control register	C1H	WDT_FLAG	-	EN_WDT	CLR_WDT	IDL_WDT	WDT_PS[2:0]			0x00,0000
IAP_CONTR	IAP control register	C7H	IAPEN	SWBS	SWRST	CMD_FAIL	-	IAP_WT[2:0]			0000,x000
RSTCFG	Reset configuration register	FFH	-Watchdog control register	ENLVR	-	P54RST	-	-	LVDS[1:0]		0000,0000

WDT_PS[2:0]	division factor	The overflow time of 12M in the main frequency	The overflow time of 20M in the main frequency
000	2	≈ 65.5 MS	≈ 39.3 MS
001	4	≈ 131 MS	≈ 78.6 MS
010	8	≈ 262 MS	≈ 157 MS
011	16	≈ 524 MS	≈ 315 MS
100	32	≈ 1.05 S	≈ 629 MS
101	64	≈ 2.10 S	≈ 1.26 S
110	128	≈ 4.20 S	≈ 2.52 S
111	256	≈ 8.39 S	≈ 5.03 S

$$\text{Overflow time of watchdog timer} = \frac{12 \times 32768 \times 2^{(WDT\_PS+1)}}{SYSclk}$$

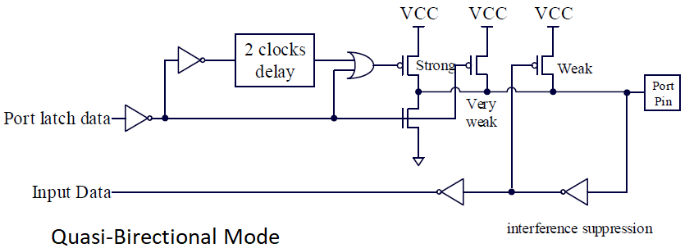
As we can see there are a whole bunch of settings that need attention while using. This is perhaps the easiest way to show what I wanted to achieve.

Please note that I have tested most of the stuffs that STC8A8K64S4A12 could offer. I tested them rigorously and with confidence I can say all of these functions have been found to be okay. Being the sole developer, I did my checks as much as possible but there could be unforeseen bugs that I may have overlooked. A typical case could be wrong function naming. This is so because during development of these BSP files I made several changes when new issues appeared. Therefore, I would like to request readers to report any issue when discovered and I would also like readers to read the reference manual of STC8 series completely if possible.

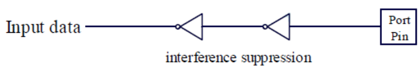
The examples presented in this document are based on my custom BSP and I believe that the journey with STC microcontroller supported by my BSP would be a joyful one.

General Purpose Input-Output (GPIO)

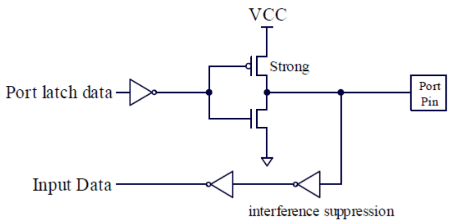
The very first thing to do with a new microcontroller is to play with its GPIOs and this is what we will begin with. STC micros are based on 8051 architecture and so it is no surprise that the GPIOs will have similarities with typical 8051s. GPIOs of STC micros are essentially same as those of Nuvoton N76E003. Those who have seen my past [Nuvoton tutorials] will find similarities.



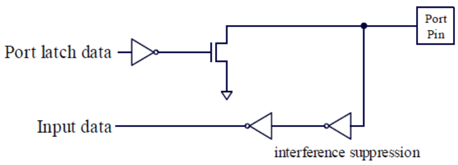
Quasi-Birectional Mode



Input Mode



Push-Pull Mode



Open-Drain Mode

I/O Type	***Description***
Quasi-Bidirectional Mode	In this mode, a GPIO behaves like the GPIO of a typical 8051, i.e., the GPIO can simultaneously act like an input and an output, hence the term “bidirectional”.
Push – Pull Mode	This mode is same as the first one but with stronger current sourcing capability and is recommended when making outputs.
Input Only Mode	Unlike other modes, this is intended specifically for dedicated high-impedance input or simply GPIO input.
Open-Drain Mode	As the name suggests, it is same as Quasi bidirectional mode but with open-drain output that can only sink current, i.e., external pull-up is required.

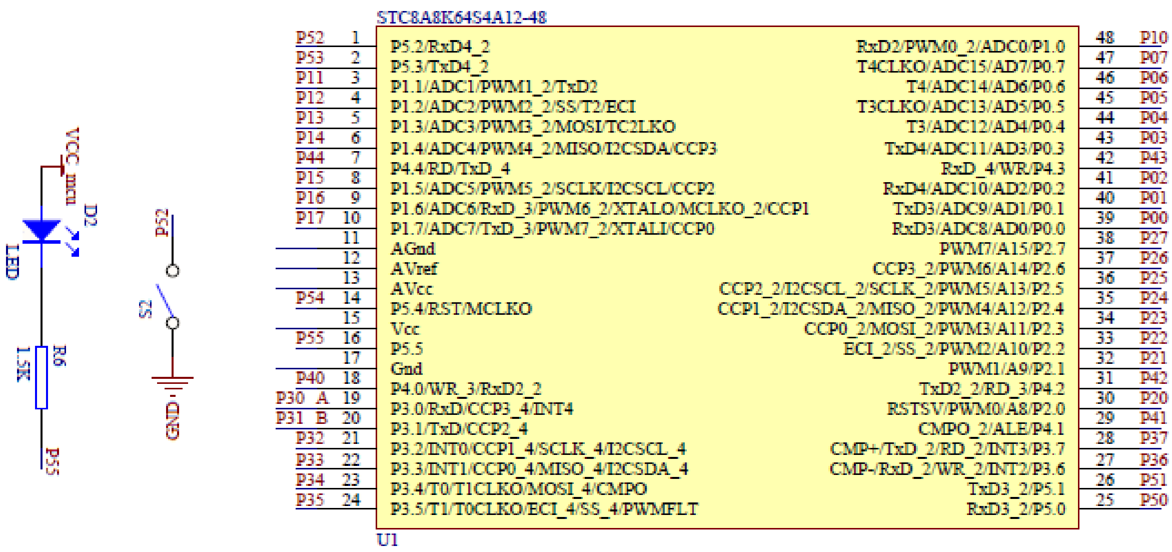
BSP

```
//P00
#define P00_quasi_bidirectional_mode    do{bit_clr(P0M1, 0); bit_clr(P0M0, 0);}while(0)
#define P00_push_pull_mode              do{P00_quasi_bidirectional_mode; bit_clr(P0M1, 0); bit_set(P0M0, 0);}while(0)
#define P00_input_mode                  do{P00_quasi_bidirectional_mode; bit_set(P0M1, 0); bit_clr(P0M0, 0);}while(0)
#define P00_open_drain_mode             do{P00_quasi_bidirectional_mode; bit_set(P0M1, 0); bit_set(P0M0, 0);}while(0)

#define P00_pull_up_enable              do{bit_set(P_SW2, 7); bit_set(P0PU, 0); bit_clr(P_SW2, 7);}while(0)
#define P00_pull_up_disable             do{bit_set(P_SW2, 7); bit_clr(P0PU, 0); bit_clr(P_SW2, 7);}while(0)

#define P00_schmitt_trigger_enable      do{bit_set(P_SW2, 7); bit_set(P0NCS, 0); bit_clr(P_SW2, 7);}while(0)
#define P00_schmitt_trigger_disable     do{bit_set(P_SW2, 7); bit_clr(P0NCS, 0); bit_clr(P_SW2, 7);}while(0)

#define P00_high                        bit_set(P0, 0)
#define P00_low                         bit_clr(P0, 0)
#define P00_toggle                      bit_tgl(P0, 0)
#define P00_get_input                   get_bit(P0, 0)
```

Code

```
#include "STC8xxx.h"
#include "BSP.h"


void setup(void);


void main(void)
{
    setup();

    while(1)
    {
        P55_toggle;

        if(P52_get_input == 0)
        {
            delay_ms(400);
        }

        delay_ms(200);
    };
}


void setup(void)
{
    P55_open_drain_mode;

    P52_input_mode;
    P52_pull_up_enable;
}
```

Schematic



Explanation

Like always this first example is a simple variable flash rate LED flasher. Onboard LED connected to P5.5 and onboard push button connected to P5.2 are used. The board's schematic shows us that P5.2's push button must have an internal pull-up to properly function because it is not tied to any external pull-up resistor and P5.5's LED must be configured as an open-drain output. These are configured so in the setup function.

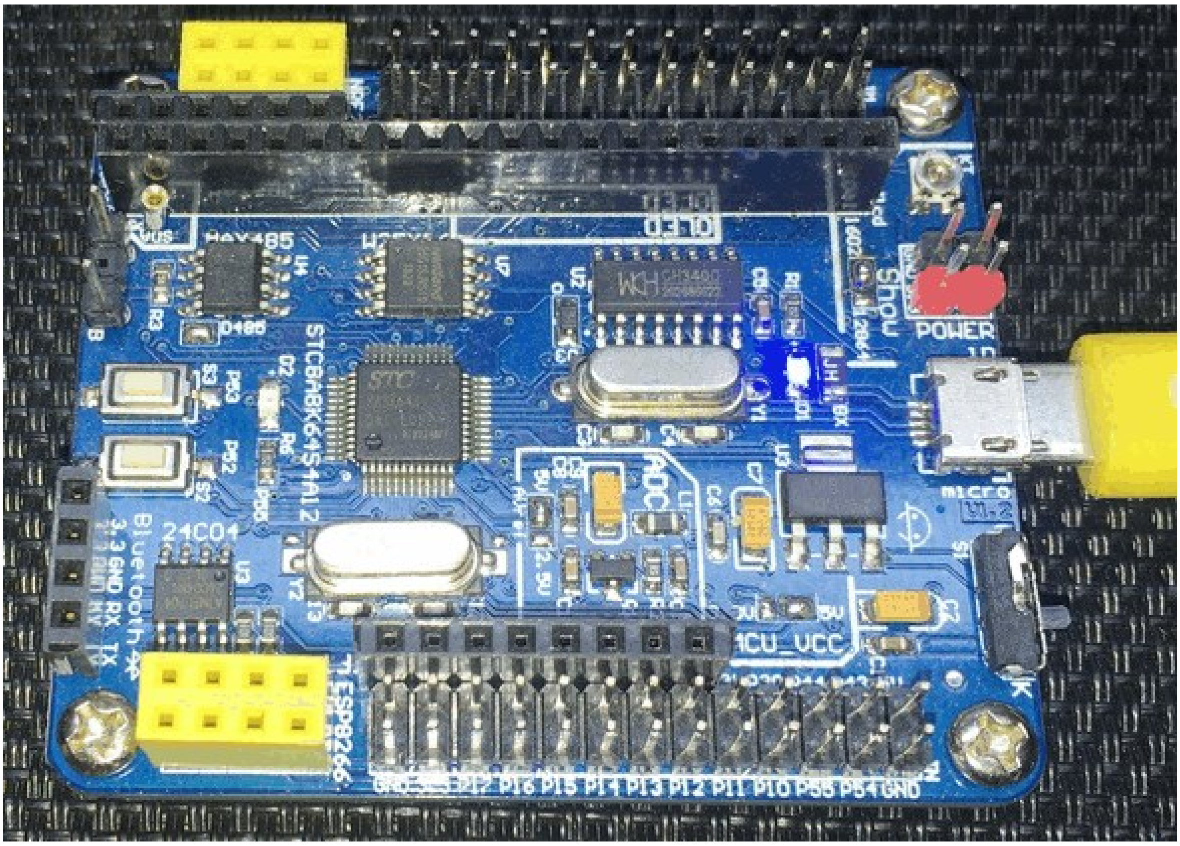
```
void setup(void)
{
    P55_open_drain_mode;
    P52_input_mode;
    P52_pull_up_enable;
}
```

In the main loop, P5.5's LED state is toggled every 200ms. If P5.2's push button is pressed, P5.2's state changes and so additional 400ms delay is added, making the total toggling delay 600ms.

```
P55_toggle;
if(P52_get_input == 0)
{
    delay_ms(400);
}
delay_ms(200);
```

Note no clock settings are applied and the micro is running at default clock frequency of 24MHz.

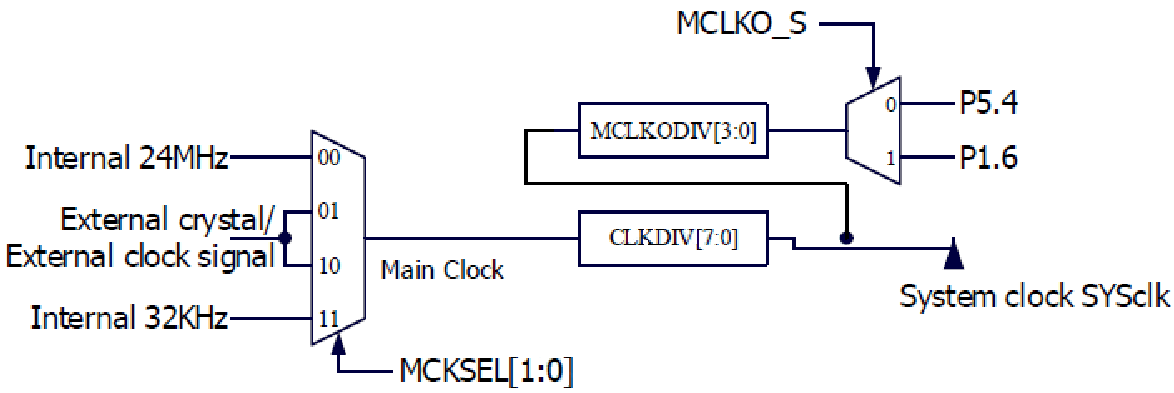
Demo



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

Clock System

STC8A8K64S4A12's clock system is very straight forward. There are three available clock sources – two of which are internal. There is a high-precision 24MHz internal oscillator and a low accuracy internal 32kHz oscillator. Apart from these internal sources, we can also use external sources like external crystals or oscillators. The clock system block diagram is shown below:



To set desired clock frequency, we have a clock divider block right after clock selection block. The final output from the clock divider is the system clock that is feed to all peripherals.

We can also get system clock output via another block of clock divider. The output can be turned off or extracted from either pin P5.4 or P1.6.

BSP

```
#define CLK_enable_IRC24_M                                  do{ \
                                                                  IRC24MCR = 0x80; \
                                                                  while((IRC24MCR & 0x01) == FALSE); \
                                                              }while(0)

#define CLK_disable_IRC24_M                                 do{ \
                                                                  bit_set(P_SW2, 7);  \
                                                                  IRC24MCR = 0x00; \
                                                                  bit_clr(P_SW2, 7);  \
                                                              }while(0)

#define CLK_enable_IRC32_k                                  do{ \
                                                                  IRC32KCR = 0x80; \
                                                                  while((IRC32KCR & 0x01) == FALSE); \
                                                              }while(0)

#define CLK_disable_IRC32_k                                 do{ \
                                                                 bit_set(P_SW2, 7);  \
                                                                 IRC32KCR = 0x00; \
                                                                 bit_clr(P_SW2, 7);  \
                                                              }while(0)

#define CLK_enable_EXT_clock_signal                         do{ \
                                                                 XOSCCR = 0x80; \
                                                                 while((XOSCCR & 0x01) == FALSE); \
                                                              }while(0)

#define CLK_disable_EXT_clock_signal                        do{ \
                                                                 bit_set(P_SW2, 7);  \
                                                                 XOSCCR = 0x00; \
                                                                 bit_clr(P_SW2, 7);  \
                                                              }while(0)

#define CLK_enable_EXT_crystal                              do{ \
                                                                 XOSCCR = 0xC0; \
                                                                 while((XOSCCR & 0x01) == FALSE); \
                                                              }while(0)
```

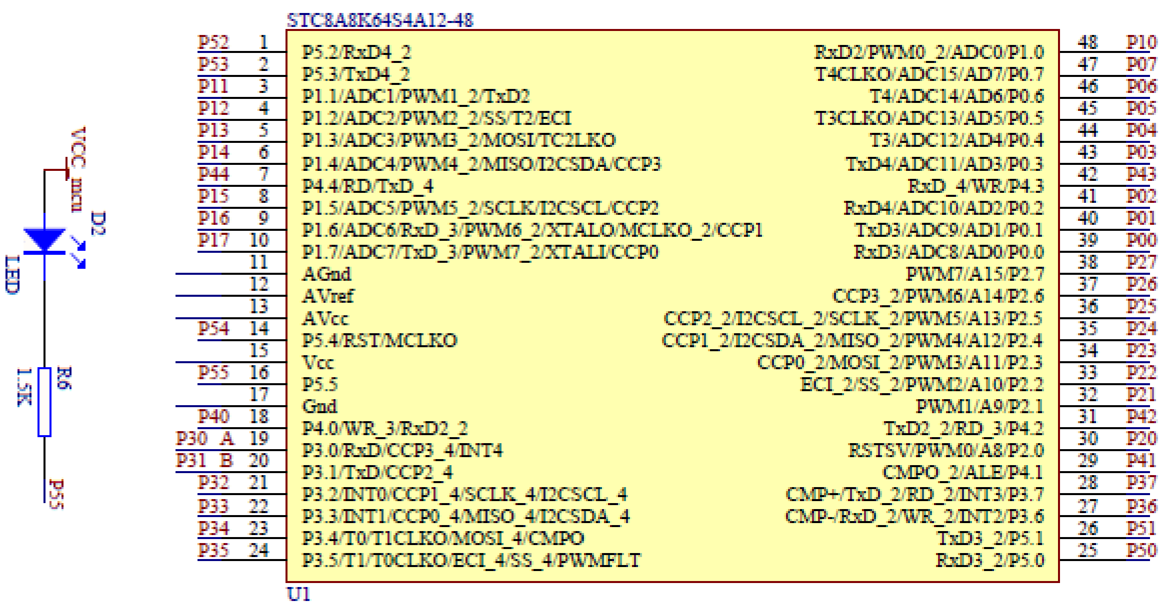
```
#define CLK_disable_EXT_crystal                                     CLK_disable_EXT_clock_signal

//sys_div
#define CLK_sys_clk_scalar(div)                                     CLKDIV = div

//src
#define IRC_24M                                                       0
#define EXT_xtal                                                      1
#define EXT_clk                                                       2
#define IRC_32k                                                       3

#define CLK_set_sys_clk_source_and_div(clk_src, div)                  do{ \
                                                                          if(clk_src == IRC_32k) \
                                                                          { \
                                                                             CLK_enable_IRC32_k; \
                                                                          } \
                                                                          else if(clk_src == EXT_clk) \
                                                                          { \
                                                                             CLK_enable_EXT_clock_signal; \
                                                                          } \
                                                                          else if(clk_src == EXT_xtal) \
                                                                          { \
                                                                             CLK_enable_EXT_crystal; \
                                                                          } \
                                                                          else \
                                                                          { \
                                                                            CLK_enable_IRC24_M; \
                                                                          } \
                                                                          CLK_sys_clk_scalar(div); \
                                                                          CKSEL = clk_src; \
                                                                      }while(0)

//mclk_div
#define MCLK_SYSCLK_no_output                                        0x00
#define MCLK_SYSCLK_div_1                                            0x10
#define MCLK_SYSCLK_div_2                                            0x30
#define MCLK_SYSCLK_div_4                                            0x50
#define MCLK_SYSCLK_div_8                                            0x70
#define MCLK_SYSCLK_div_16                                           0x90
#define MCLK_SYSCLK_div_32                                           0xB0
#define MCLK_SYSCLK_div_64                                           0xD0
#define MCLK_SYSCLK_div_128                                          0xF0

#define CLK_MCLK_scalar(div)                                         CKSEL |= div

//mclk_pin_id
#define MCLK_out_P54                                                 0
#define MCLK_out_P16                                                 1

#define CLK_set_MCLK(div, pin_id)                                    do{ \
                                                                          CLK_MCLK_scalar(div); \
                                                                          if(pin_id == MCLK_out_P16) \
                                                                          { \
                                                                             bit_set(CKSEL, 3); \
                                                                          } \
                                                                          else \
                                                                          { \
                                                                             bit_clr(CKSEL, 3); \
                                                                          } \
                                                                       }while(0)

#define CLK_set_sys_clk(sys_src, sys_div, mclk_div, mclk_pin_id)     do{ \
                                                                          bit_set(P_SW2, 7);   \
                                                                          CLK_set_sys_clk_source_and_div(sys_src, sys_div); \
                                                                          CLK_set_MCLK(mclk_div, mclk_pin_id); \
                                                                          bit_clr(P_SW2, 7);   \
                                                                       }while(0)
```

Code

```
#include "STC8xxx.h"
#include "BSP.h"


void setup(void);
void toggle(void);


void main(void)
{

    setup();

    while(1)
    {
```

```
    };
}


void setup(void)
{
    P55_open_drain_mode;

    CLK_set_sys_clk(IRC_24M, 4, MCLK_SYSCLK_div_1, MCLK_out_P54);
    toggle();

    CLK_set_sys_clk(IRC_24M, 6, MCLK_SYSCLK_div_8, MCLK_out_P54);
    toggle();

    CLK_set_sys_clk(IRC_24M, 24, MCLK_SYSCLK_div_1, MCLK_out_P54);
    toggle();

    CLK_set_sys_clk(IRC_32k, 1, MCLK_SYSCLK_div_1, MCLK_out_P54);
    toggle();
}


void toggle(void)
{
    unsigned char i = 10;

    while(i > 0)
    {
        P55_toggle;
        delay_ms(60);
        i--;
    };
}
```

Schematic



Explanation

This is yet another oversimplified example. Again, onboard LED toggling code is presented here.

Firstly, we need a LED toggling function because we want to visually observe what is happening.

```
void toggle(void)
{
    unsigned char i = 10;

    while(i > 0)
    {
        P55_toggle;
        delay_ms(60);
        i--;
    };
}
```

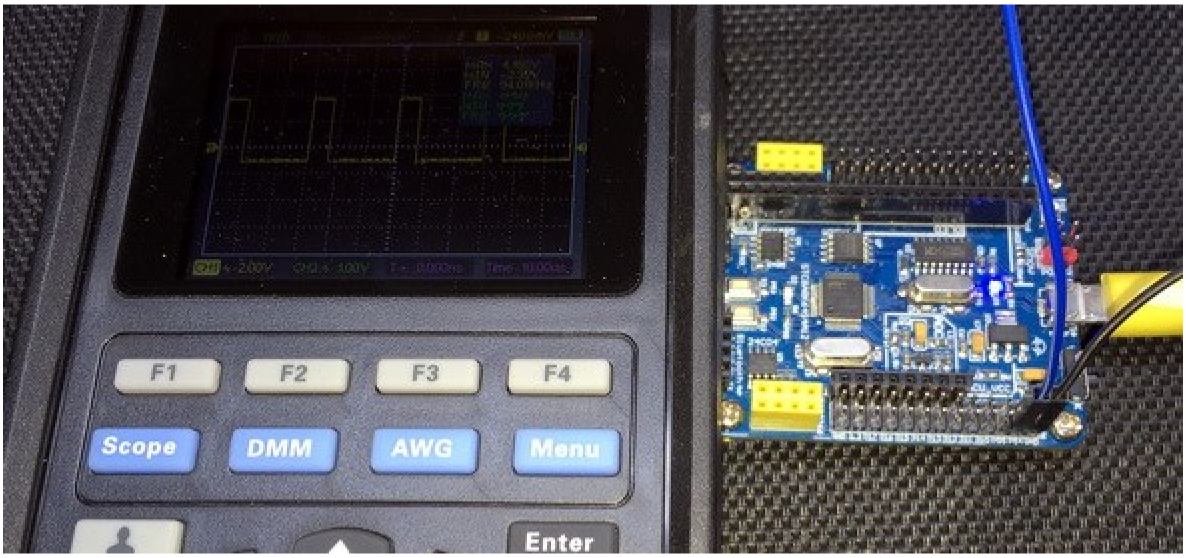
P5.5's LED is toggled 10 times. Note that although a 60ms delay is used between toggles, it will not be 60ms always as delays are dependent on system clock frequency. This will do the trick.

In the main, different clock settings are applied and the LED toggling is observed.

```
CLK_set_sys_clk(IRC_24M, 4, MCLK_SYSCLK_div_1, MCLK_out_P54);
toggle();

CLK_set_sys_clk(IRC_24M, 6, MCLK_SYSCLK_div_8, MCLK_out_P54);
toggle();

CLK_set_sys_clk(IRC_24M, 24, MCLK_SYSCLK_div_1, MCLK_out_P54);
toggle();

CLK_set_sys_clk(IRC_32k, 1, MCLK_SYSCLK_div_1, MCLK_out_P54);
toggle();
```

With different system clock speeds, the onboard LED toggling rates change.

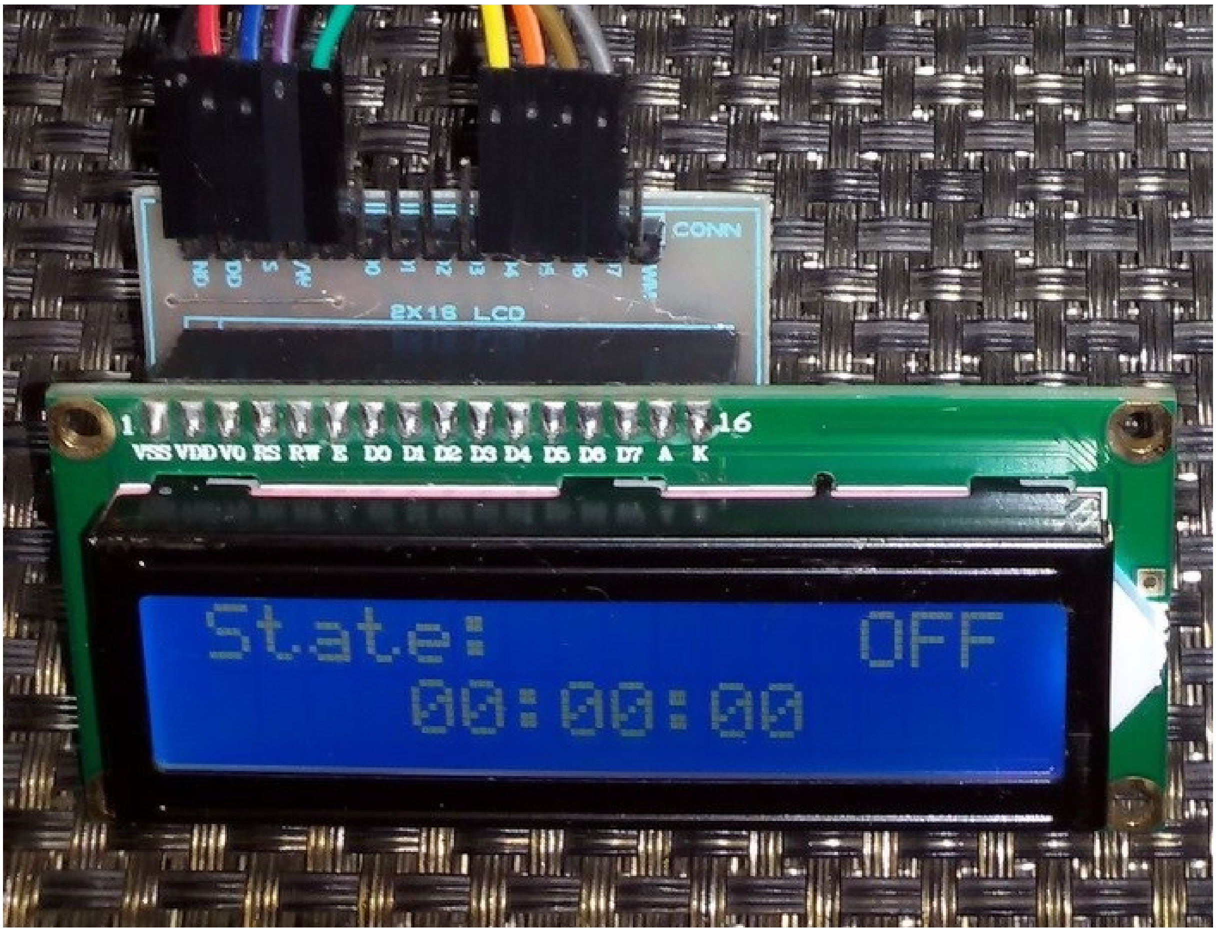
Demo



Demo video link: https://youtu.be/jNwxc3bJ1C8.

Interfacing 2x16 LCD

After having both clock system and GPIO mastered, the next important thing to do is to drive an ordinary alphanumerical LCD or simply text LCD. This is very important as text LCDs are great tools for quickly and visually displaying information.



Code

LCD.h

```
#define LCD_GPIO_init()                      do{P35_push_pull_mode; \
                                                P36_push_pull_mode; \
                                                P37_push_pull_mode; \
                                                P04_push_pull_mode; \
                                                P05_push_pull_mode; \
                                                P06_push_pull_mode; \
                                                P07_push_pull_mode; \
                                               }while(0)

#define LCD_RS_HIGH                          P35_high
#define LCD_RS_LOW                           P35_low

#define LCD_RW_HIGH                          P36_high
#define LCD_RW_LOW                           P36_low

#define LCD_EN_HIGH                          P37_high
#define LCD_EN_LOW                           P37_low

#define LCD_DB4_HIGH                         P04_high
#define LCD_DB4_LOW                          P04_low
```

```
#define LCD_DB5_HIGH                        P05_high
#define LCD_DB5_LOW                         P05_low

#define LCD_DB6_HIGH                        P06_high
#define LCD_DB6_LOW                         P06_low

#define LCD_DB7_HIGH                        P07_high
#define LCD_DB7_LOW                         P07_low

#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 "LCD.h"


void LCD_init(void)
{
    delay_ms(10);

    LCD_GPIO_init();

    LCD_RW_LOW;

    LCD_RS_LOW;
    delay_ms(10);
    toggle_EN_pin();

    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_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;
    delay_ms(2);
    LCD_EN_LOW;
    delay_ms(2);
}
```

lcd_print.h

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


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

lcd_print.c

```
#include "lcd_print.h"


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

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

    LCD_send(0x40, CMD);

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

```
    LCD_send(0x80, CMD);
}


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


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

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

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

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


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

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

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

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

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

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


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

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

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

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

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


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

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

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

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

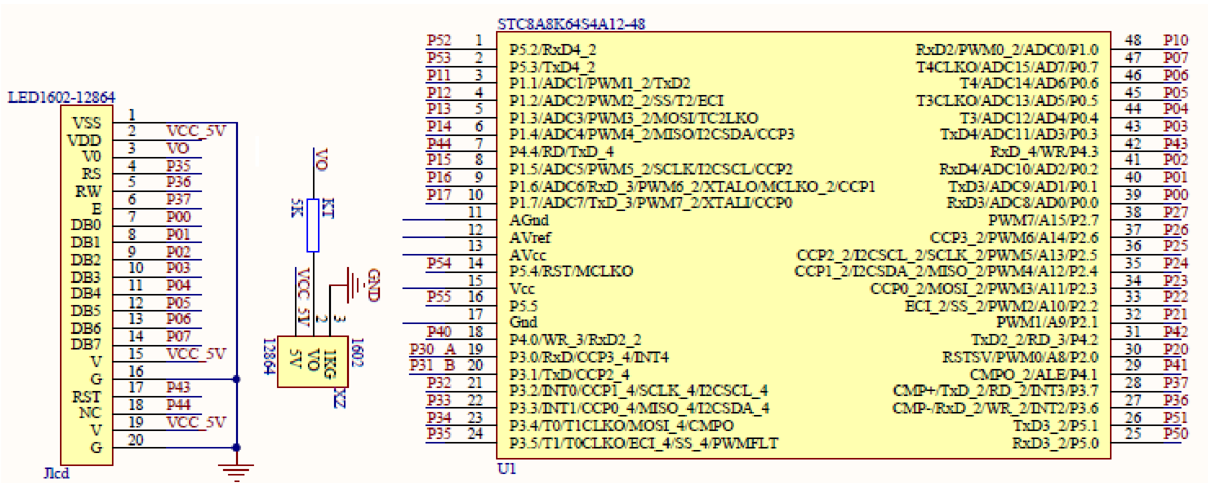
```
    if((value >= 10000) && (value < 100000))
    {
        print_D((x_pos + 6), y_pos, tmp, points);
    }
    else if((value >= 1000) && (value < 10000))
    {
        print_D((x_pos + 5), y_pos, tmp, points);
    }
    else if((value >= 100) && (value < 1000))
    {
        print_D((x_pos + 4),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

main.c

```
#include "STC8xxx.h"
#include "BSP.h"
#include "LCD.c"


void setup(void);
void show_value(unsigned char value);


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

    char txt1[] = {"MICROARENA"};
    char txt2[] = {"SShahryiar"};
    char txt3[] = {"STC8A Series"};
    char txt4[] = {"STC8A8K64S4A12"};

    setup();

    LCD_clear_home();

    LCD_goto(3, 0);
    LCD_putstr(txt1);
    LCD_goto(3, 1);
    LCD_putstr(txt2);
    delay_ms(4000);

    LCD_clear_home();

    for(s = 0; s < 12; s++)
    {
        LCD_goto((2 + s), 0);
        LCD_putchar(txt3[s]);
        delay_ms(60);
    }
    for(s = 0; s < 14; s++)
    {
        LCD_goto((1 + s), 1);
        LCD_putchar(txt4[s]);
        delay_ms(60);
    }
    delay_ms(4000);

    s = 0;
    LCD_clear_home();

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

    while(1)
```

```
    {
        show_value(s);
        s++;
        delay_ms(400);
    };
}


void setup(void)
{
    CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_div_1, MCLK_out_P54);

    LCD_init();
}


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

I have demoed this example in all of my past tutorials and so I won't be explaining it again. The only trick I would like to share is the fact that LCD datasheet documents state how to drive and initialize them. Thus, it is best to read datasheet and try to implement on own. It is very simple.

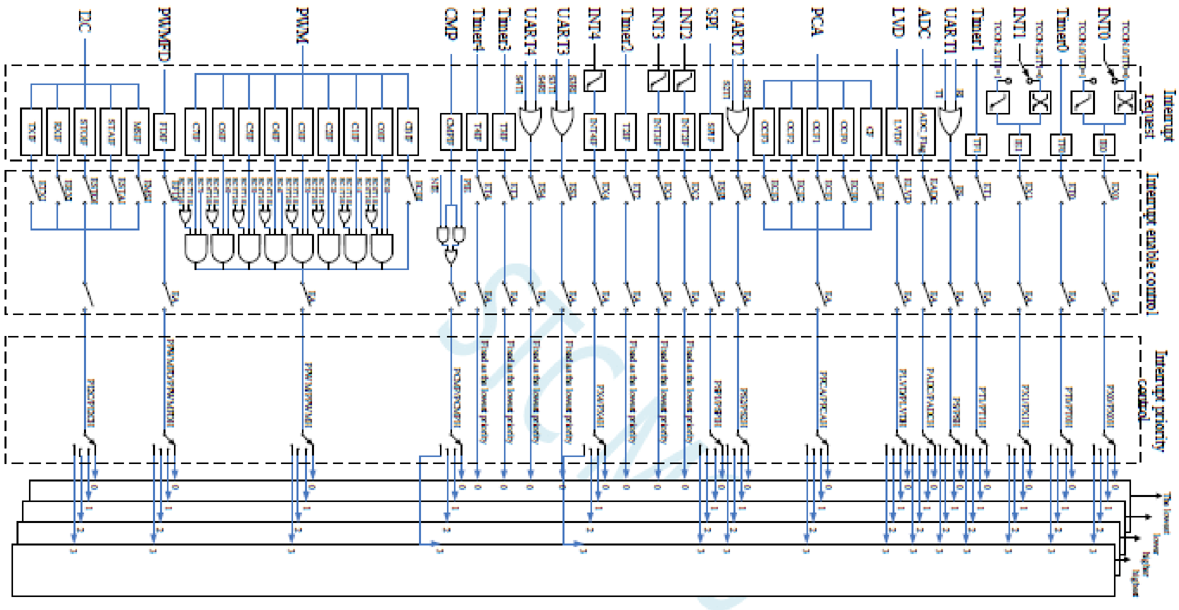
Demo

Demo video link: https://youtu.be/-fOvuPexRKM.

External Interrupt (EXTI)

STC microcontrollers have complex interrupt systems. Literally, all internal peripherals have interrupt capabilities. The block diagram of STC8A8K64S4A12’s interrupt system shows this. In this section, we would see the use of external interrupt and in later examples we would see other interrupts.

External interrupt has lot of uses in modern embedded systems. When coupled with low power consumption modes, external interrupts can be used to wake up a device and do certain task upon user’s stimulated input via a button, keypad, touchpad, etc. External interrupts are what used in most portable battery-operated devices like computer mice, keyboards, remote controllers, etc. In such devices, precious battery energy is conserved by spending most time in low power modes and waiting for user interactions. When there is a user input, external interrupts are kicked in and some tasks are done quickly before returning to dormant low power modes.

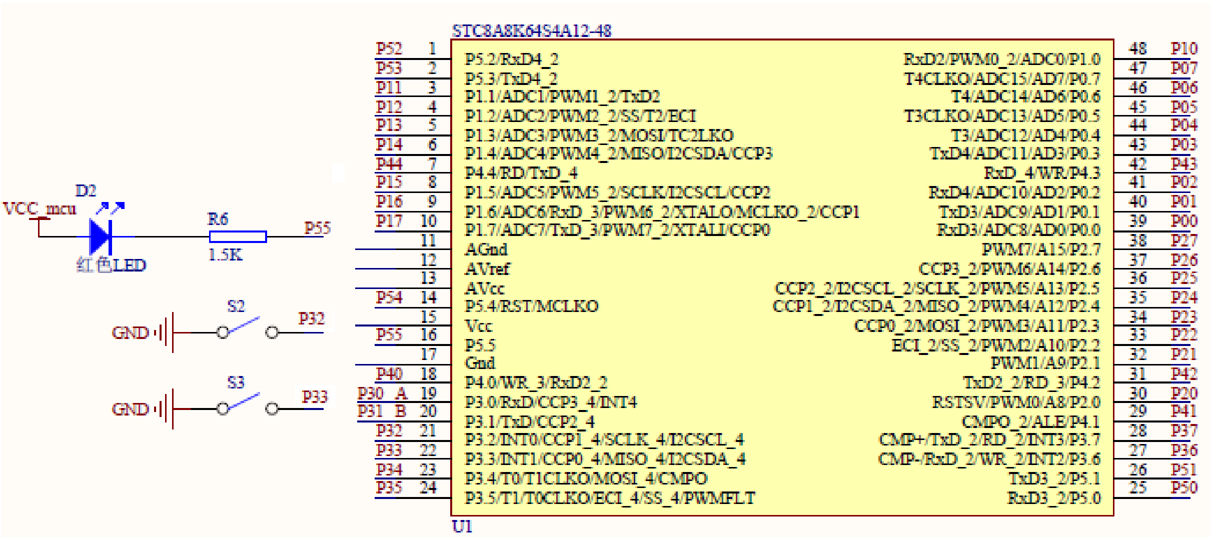


Code

```
#include "STC8xxx.h"
#include "BSP.h"


unsigned char s = 0;
unsigned int i = 0;


void setup(void);


void EXT_0_ISR(void)
interrupt 0
{
    for(s = 0; s <= 9; s++)
    {
        P55_toggle;
        for(i = 0; i < 10000; i++);
    }
}
```

```
void EXT_1_ISR(void)
interrupt 2
{
    for(s = 0; s <= 9; s++)
    {
        P55_toggle;
        for(i = 0; i < 30000; i++);
    }
}


void main(void)
{
    setup();

    while(1)
    {
        P55_low;
    };
}


void setup(void)
{
    CLK_set_sys_clk(IRC_24M, 24, MCLK_SYSCLK_no_output, MCLK_out_P54);

    P55_open_drain_mode;

    EXT_0_priority_0;
    EXT_0_falling_edge_detection_only;
    _enable_EXT_0_interrupt;

    EXT_1_priority_1;
    EXT_1_falling_edge_detection_only;
    _enable_EXT_1_interrupt;

    _enable_global_interrupt;
}
```

Schematic



Explanation

First, let us see how we can enable external interrupt. Although external interrupts are GPIO input feature, we do not have to set external interrupt pins as inputs. This is done automatically and internally.

8051 architecture allows prioritization of interrupts. This means that when there are simultaneous multiple interrupts, the interrupts are served in an orderly fashion according to the level of priority. STC8A8K64S4A12 has 22 interrupt sources and 4 levels of priority. Although it is not mandatory to set priority in most cases, some applications may need this feature. In this demo code, external interrupt 1 has higher priority than external interrupt 0 and so it would be processed first and then external interrupt 0 would be processed afterwards.

```
EXT_0_priority_0;
EXT_0_falling_edge_detection_only;
_enable_EXT_0_interrupt;

EXT_1_priority_1;
EXT_1_falling_edge_detection_only;
_enable_EXT_1_interrupt;

_enable_global_interrupt;
```

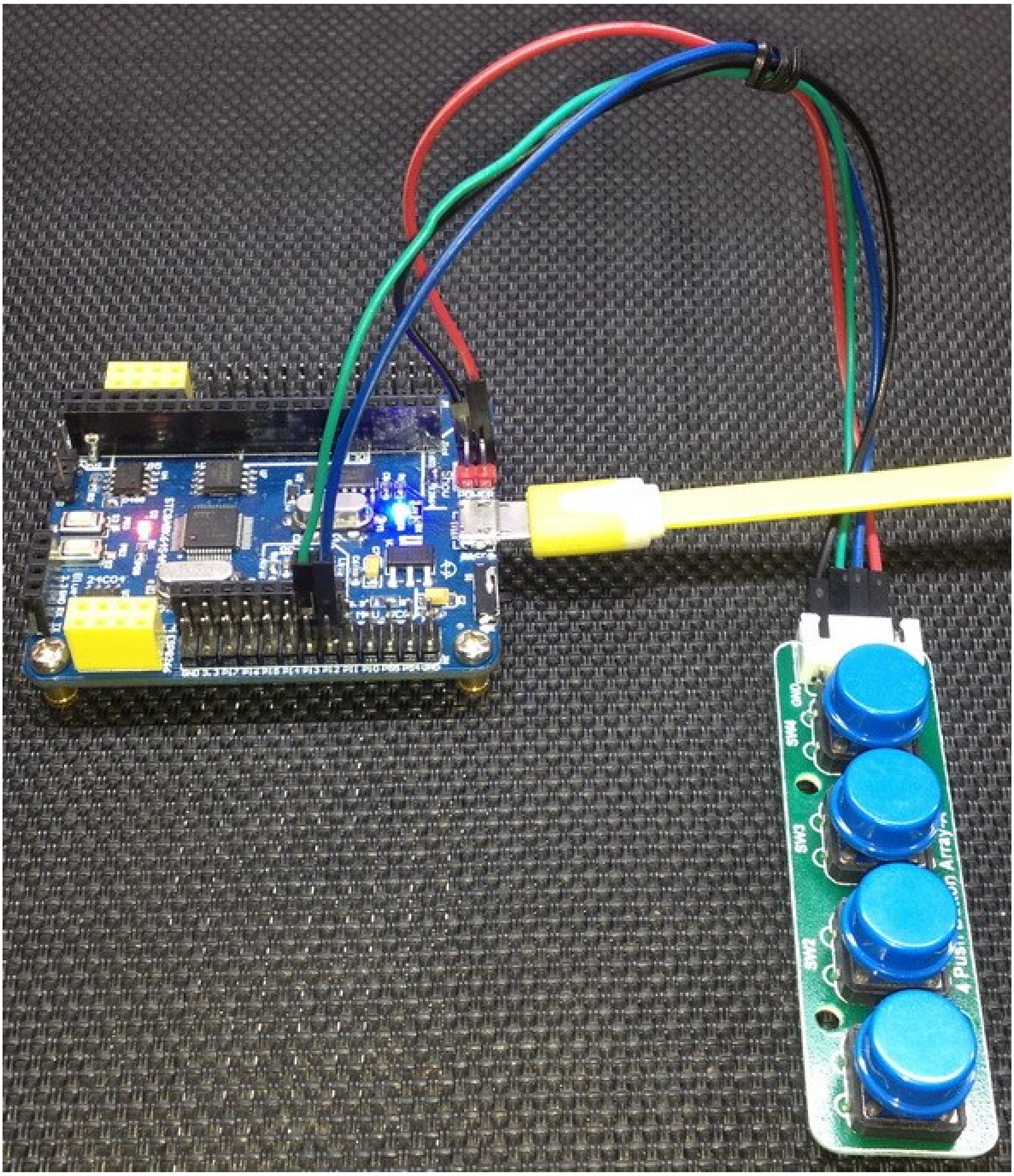
The next thing to do when setting external interrupts is to let the MCU know which edges would trigger the interrupts. Finally, the interrupts are enabled along with global interrupt flag bit.

Like other examples onboard LED is used as an indicator. Inside each interrupt service routine, this LED is toggled but the rates of toggling are different. This would differentiate the interrupts.

```
void EXT_0_ISR(void)
interrupt 0
{
    for(s = 0; s <= 9; s++)
    {
        P55_toggle;
        for(i = 0; i < 10000; i++);
    }
}


void EXT_1_ISR(void)
interrupt 2
{
    for(s = 0; s <= 9; s++)
    {
        P55_toggle;
        for(i = 0; i < 30000; i++);
    }
}
```

When external interrupt 0 is triggered, the LED toggles fast and when external interrupt 1 is triggered, the LED toggles slowly. If external interrupt 0 is triggered while external interrupt 1 is being processed, external interrupt 0's task is processed after completing external interrupt 1's service routine. It is as if the MCU remembers the low priority interrupt after completing the higher priority interrupt. If external interrupt 1 is triggered while external interrupt 0 is being processed, the tasks in external interrupt 0 is temporarily suspended and external interrupt 1 is processed. After processing external interrupt 1, the suspended tasks of external interrupt 0 are resumed from where they were left.

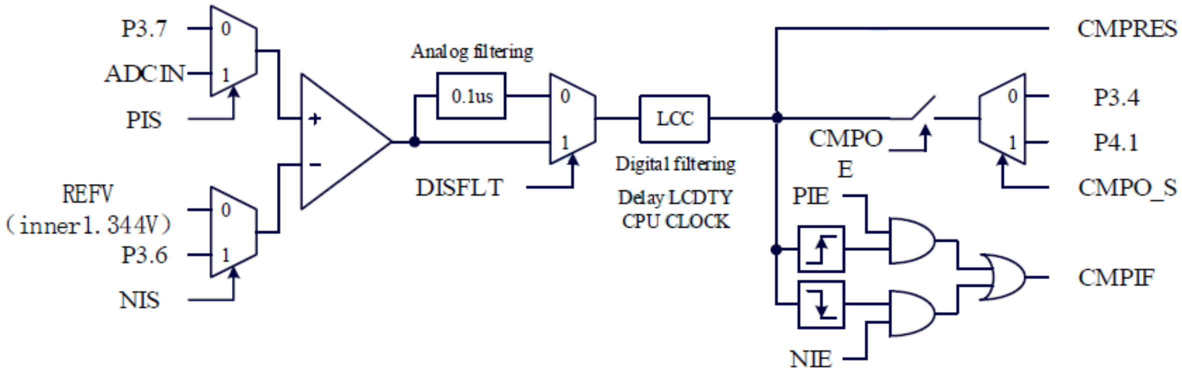
Demo



Demo video link: https://youtu.be/_dHcJr7-X0I.

Analogue Comparator (AC)

Analogue comparator is not typically found in many microcontrollers but STC8A8K64S4A12 packs one analogue comparator. Though this comparator has limited options, it is still very useful and very easy to use. It can be used for low battery/voltage detection, for comparing voltage levels, etc. The comparator can also be used to build simple switch mode power supplies and achieve feedback for various tasks.



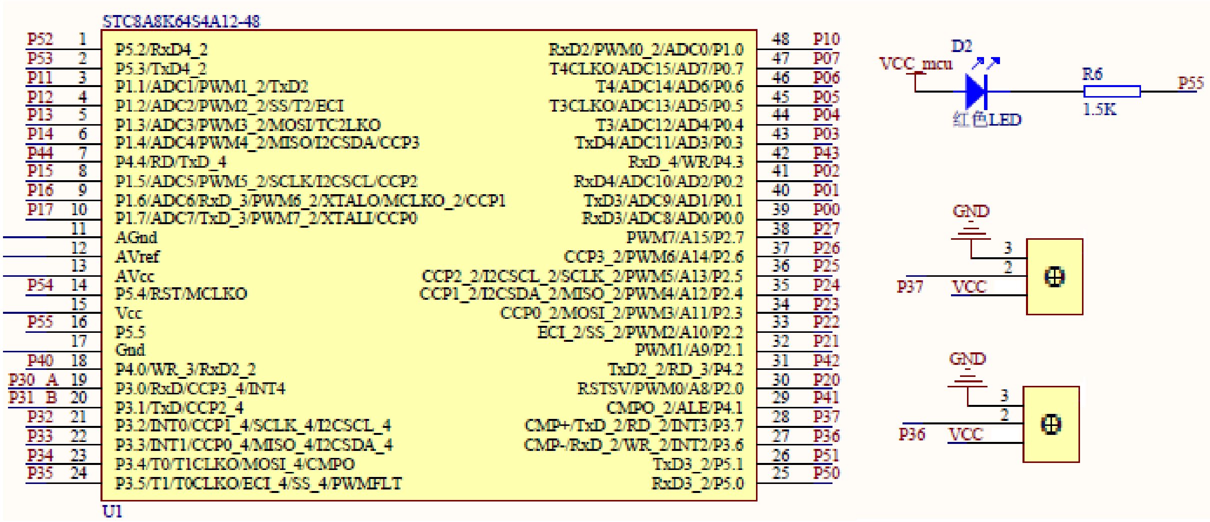
The comparator block shows that P3.6 and P3.7 GPIO pins can be used as positive and negative inputs to the comparator unit. Alternatively, we can also use internal 1.344V voltage reference source or ADC input. The comparator output can be optionally filtered both using analogue and digital techniques. Lastly, we can retrieve comparator output via GPIO pins, flag and interrupts.

Code

```
#include "STC8xxx.h"
#include "BSP.h"


void setup(void);


void main(void)
{
    setup();

    while(1)
    {
        if(CMP_get_comp_status != 0x00)
        {
            P55_low;
        }
        else
        {
            P55_high;
        }
    };
}


void setup(void)
{
    CLK_set_sys_clk(IRC_24M, \
                    2, \
                    MCLK_SYSCLK_no_output, \
                    MCLK_out_P54);

    P55_open_drain_mode;
```

```
    CMP_setup(CMP_positive_input_P37, \
              CMP_negative_input_P36, \
              CMP_output_disable, \
              CMP_result_positive_output, \
              CMP_enable_analog_filtering, \
              0x04);

    CMP_enable;
}
```

Schematic



Explanation

Analogue comparator is set up very easily by choosing the positive and negative input sources, output state, result state and filtering values. After setting all these, the comparator needs to be enabled.

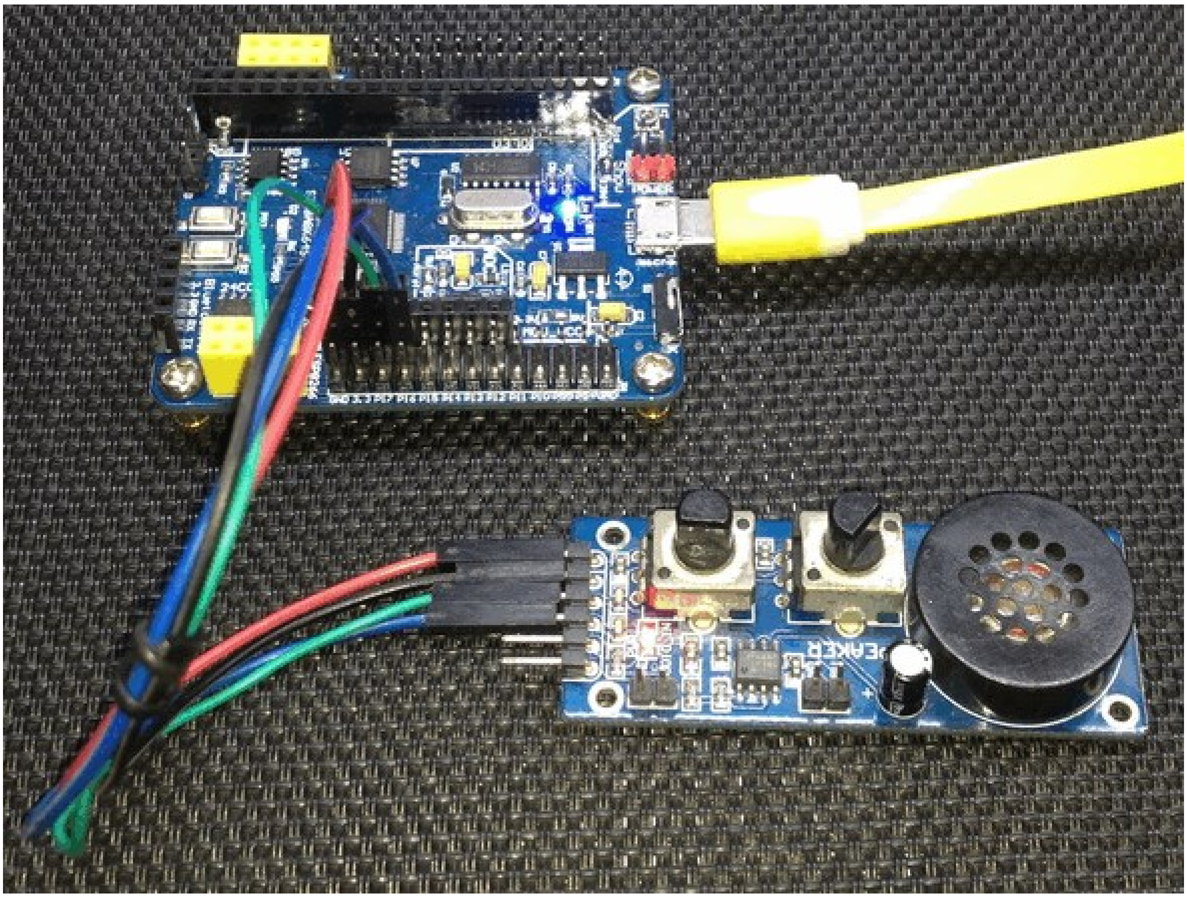
```
CMP_setup(CMP_positive_input_P37, \
          CMP_negative_input_P36, \
          CMP_output_disable, \
          CMP_result_positive_output, \
          CMP_enable_analog_filtering, \
          0x04);

CMP_enable;
```

Here, we used the GPIO pins P3.6 and P3.7 as physical inputs to the comparator unit. We disabled the physical comparator output. The comparator will give output when positive input is larger than negative input. Lastly, 0.1µs analogue filtering is used along with some digital filtering by the addition of some latency. These filtering ensure that short-lived false signals are ignored.

Inside the main loop, comparator's result flag status is continuously monitored. P5.5 onboard LED is turned on or off according to this flag's status.

```
if(CMP_get_comp_status != 0x00)
{
     P55_low;
}
else
{
     P55_high;
}
```

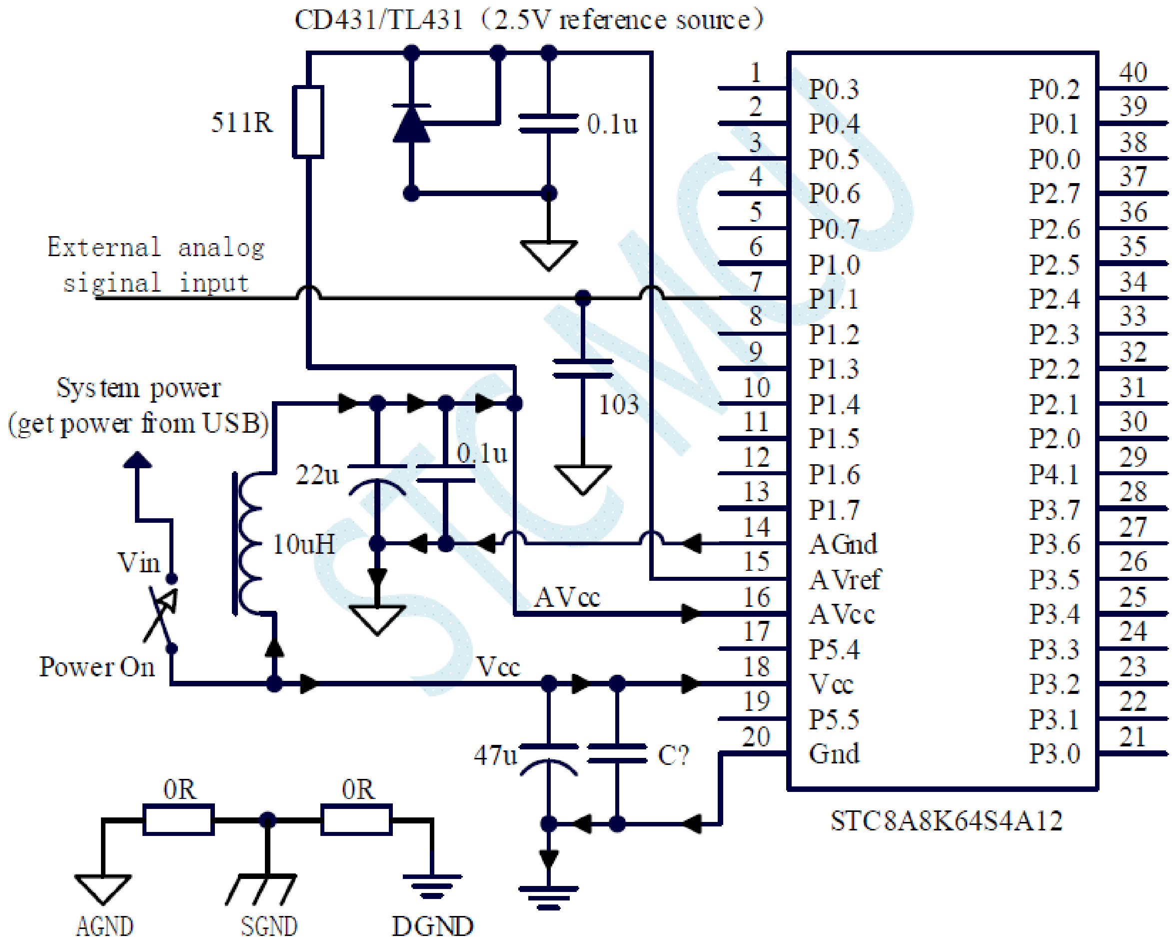
Demo



Demo video link: https://youtu.be/TuKMcBxPxf0.

Analogue-to-Digital Converter (ADC)

STC8A8K64S4A12's 12-bit analogue-to-digital converter is an attractive feature. In the market, there are many enhanced 8051-core microcontrollers but most don't have any built-in ADC while some have low resolution ones. Like the analogue comparator, the ADC of STC8A8K64S4A12 is very simple and have limited basic options. The speed of this ADC can reach up to 800ksps. However, some care needs to be taken in order to maintain accuracy and consistency in measurements. For better results it is better to use well-calibrated external voltage reference source and additional filtering. These are well documented in STC8A8K64S4A12 reference manual and a sample diagram is shown below.

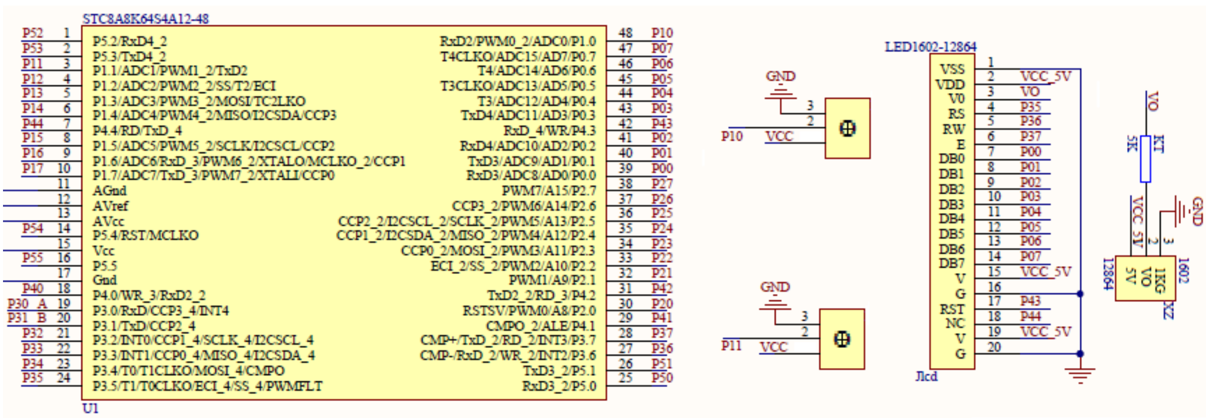


	System clock<=10MHz	System clock>10MHz
C?	104(0.1uF)	103(0.01uF)

Code

```
#include "STC8xxx.h"
#include "BSP.h"
#include "LCD.c"
#include "lcd_print.c"


void setup(void);


void main(void)
{
  unsigned int ADC_count = 0x0000;
  float voltage = 0.0;

  setup();

  LCD_goto(0, 0);
  LCD_putstr("CH0/V:");

  LCD_goto(0, 1);
  LCD_putstr("CH1/V:");

  while(1)
  {
    ADC_count = ADC_get_result(CH0);
    voltage = (((float)ADC_count * 5.0) / 4095.0);
    print_F(10, 0, voltage, 3);

    ADC_count = ADC_get_result(CH1);
    voltage = (((float)ADC_count * 5.0) / 4095.0);
    print_F(10, 1, voltage, 3);

    delay_ms(400);
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 4, MCLK_SYSCLK_no_output, MCLK_out_P54);

  P10_input_mode;
  P11_input_mode;

  ADC_enable;
  ADC_result_format_right_aligned;
  ADC_set_conversion_speed(ADC_conv_256_CLKs);

  LCD_init();
  LCD_clear_home();
}
```

Schematic



Explanation

This ADC example utilizes polling method to get voltage reading from two ADC channels associated with pins P1.0 and P1.1.

System clock setting is very important and this is so because AD conversion speed is dependent of this clock.

$$F_{ADC} = SYSclk/2/16/SPEED$$

where

SPEED[3:0]	ADC conversion time (number of CPUclocks)	SPEED[3:0]	ADC conversion time (number of CPUclocks)
0000	32	1000	288
0001	64	1001	320
0010	96	1010	352
0011	128	1011	384
0100	160	1100	416
0101	192	1101	448
0110	224	1110	480
0111	256	1111	512

```
CLK_set_sys_clk(IRC_24M, 4, MCLK_SYSCLK_no_output, MCLK_out_P54);
```

In our case, the system clock is 6MHz and so F_{ADC} is 732Hz. Though the conversion speed is not pretty impressive, it is good enough for this example as sensing voltage variations are not rapid. Nyquist criterion should always be kept in mind while sampling analogue signals.

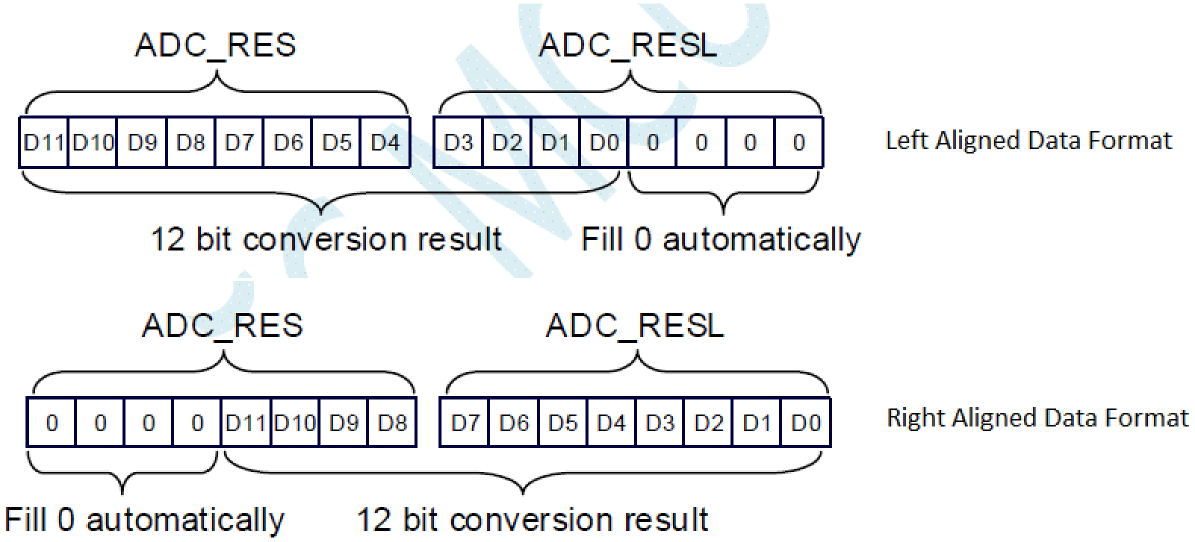
ADC Input pins need to declared as inputs.

```
P10_input_mode;
P11_input_mode;
```

ADC setup is pretty straight. All we need to do is to enable the ADC, select data output alignment and ADC clock speed prescalar.

```
ADC_enable;
ADC_result_format_right_aligned;
ADC_set_conversion_speed(ADC_conv_256_CLKs);
```

Data output can be either left-aligned or right-aligned. Right-aligned data is easy to read and that is why it my choice.



The ADC reading function reveals this fact.

```
unsigned int ADC_get_result(unsigned char channel)
{
    register unsigned int value = 0x0000;

    ADC_set_channel(channel);
    delay_ms(1);

    ADC_start_conversion;
    while(!check_ADC_flag);
    clear_ADC_flag;

    value = ((ADC_RES << 8) | ADC_RESL);

    return value;
}
```

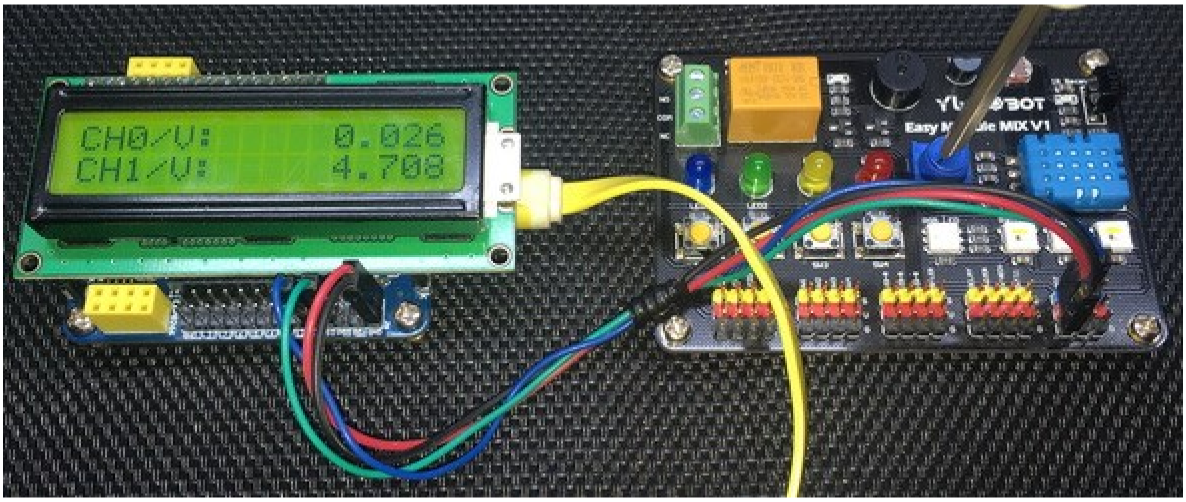
In the main, ADC channels 0 and 1 are read and their readings are converted to voltage. The voltages are shown on an LCD.

```
ADC_count = ADC_get_result(CH0);
voltage = (((float)ADC_count * 5.0) / 4095.0);
print_F(10, 0, voltage, 3);

ADC_count = ADC_get_result(CH1);
voltage = (((float)ADC_count * 5.0) / 4095.0);
print_F(10, 1, voltage, 3);

delay_ms(400);
```

ADC reading can be improved by applying several techniques like using filters, averaging technique and so on. Special attention is needed while designing PCBs in order to minimize cross-talk between channels and noise from onboard digital circuitry.

Generally, AD conversion should be fast, ADC I/O pins must be set in high impedance mode and AVCC to VCC voltage difference should not be more than 0.3V.

Demo



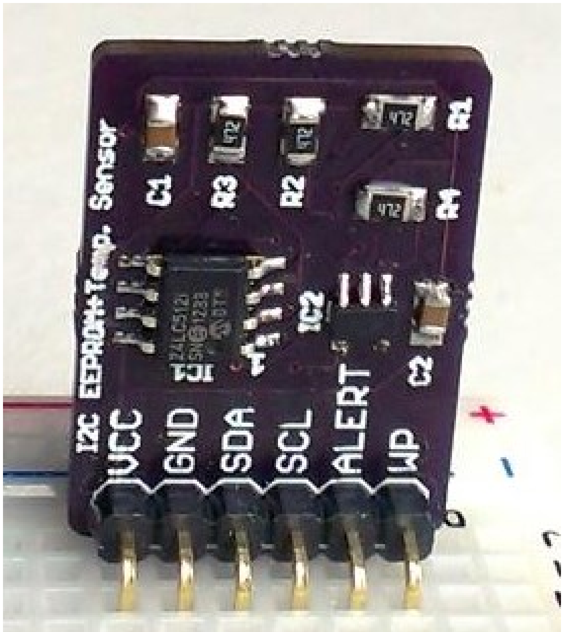
Demo video link: https://youtu.be/i2MJ1OkbhwQ.

IAP/EEPROM

Storing calibration, configuration, setting data and some preset values are required in some devices and so they need to be stored in memories that can be later modified if needed or else left alone. For such cases, we would need either EEPROM or flash memories. Like many modern microcontrollers, STC microcontrollers don't come equipped with built-in EEPROM memory but through coding we can store aforementioned data in internal flash memory through IAP/ISP technology.

Of course, there are ways to use external EEPROM and flash memories like AT24Cxx EERPOMS and W25X16 but that would require external wiring and the use of communication pins. Having flash/EEPROM memory embedded in the application chip allows us to simply avoid these and have our microcontroller ready for fast deployment.

In this section, we would see how to use internal flash to store data.

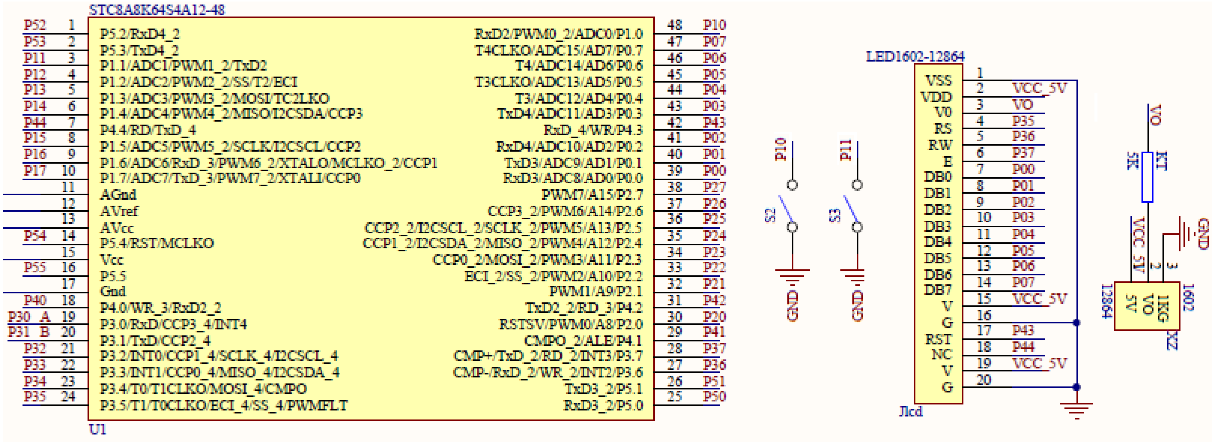


Code

```
# include "STC8xxx.h"
#include "BSP.h"
#include "LCD.c"
#include "lcd_print.c"

#define BASE_ADDRESS  0x0400

void setup(void);

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

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

  i = IAP_read(BASE_ADDRESS);
  delay_ms(10);
```

```
  print_I(11, 0, BASE_ADDRESS);
  print_C(14, 1, i);

  delay_ms(2000);

  if(i == 0)
  {
    LCD_clear_home();
    LCD_goto(0, 0);
    LCD_putstr("Performing Erase");
    LCD_goto(0, 1);
    LCD_putstr("....");
    IAP_erase(BASE_ADDRESS);
    delay_ms(1000);
  }

  LCD_clear_home();
  delay_ms(100);

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

  i = (P1 & 0x03);
  IAP_write(BASE_ADDRESS, i);
  print_I(11, 0, BASE_ADDRESS);
  print_C(14, 1, i);
  delay_ms(2000);

  LCD_clear_home();
  delay_ms(100);

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

  i = IAP_read(BASE_ADDRESS);
  delay_ms(10);
  print_I(11, 0, BASE_ADDRESS);
  print_C(14, 1, i);

  delay_ms(2000);

  while(1)
  {
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_div_1, MCLK_out_P54);

  P10_input_mode;
  P11_input_mode;

  LCD_init();
  LCD_clear_home();
}
```

Schematic



Explanation

The code in this rudimentary demo works by reading a single fixed location of internal flash, here 0x0400. An LCD display is used to show the read location and its content. If the content is 0 then an erase is performed because it is assumed to be empty. After performing erase, the same location is written. The write value is generated by reading the logic states of the lowermost bits (i.e., bit 0 and 1) of P1 port. After writing, the location is read again as to check if the data is properly saved. If the micro is reset or if there is a power-down, the saved data is retained, confirming that indeed data has been preserved in the internal flash memory.

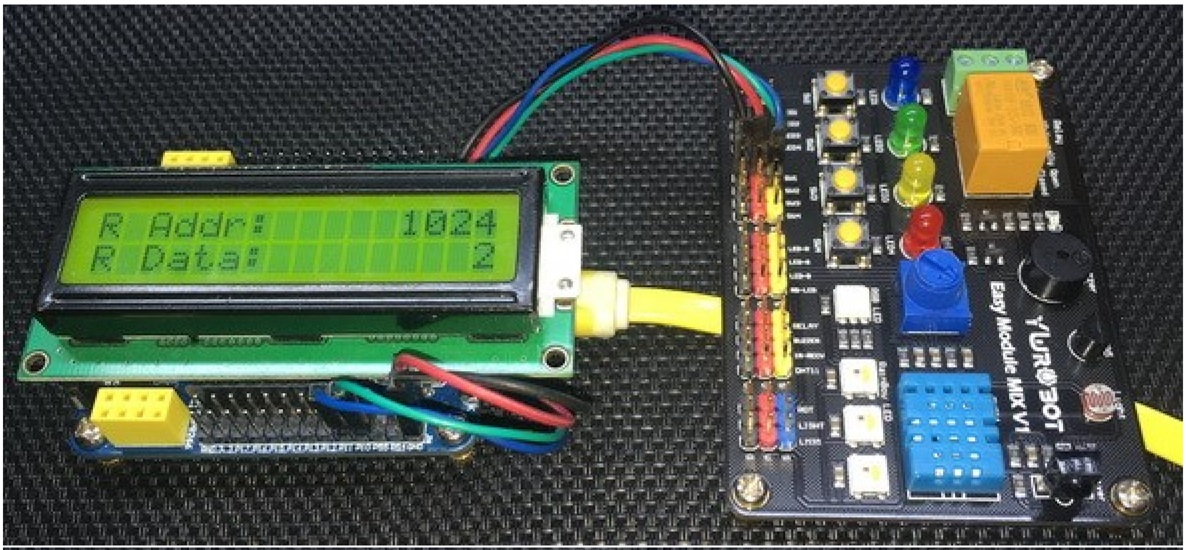
IAP functionality utilizes the following three functions. As can be seen that these functions must follow a sequence operation in order to enable access to internal flash.

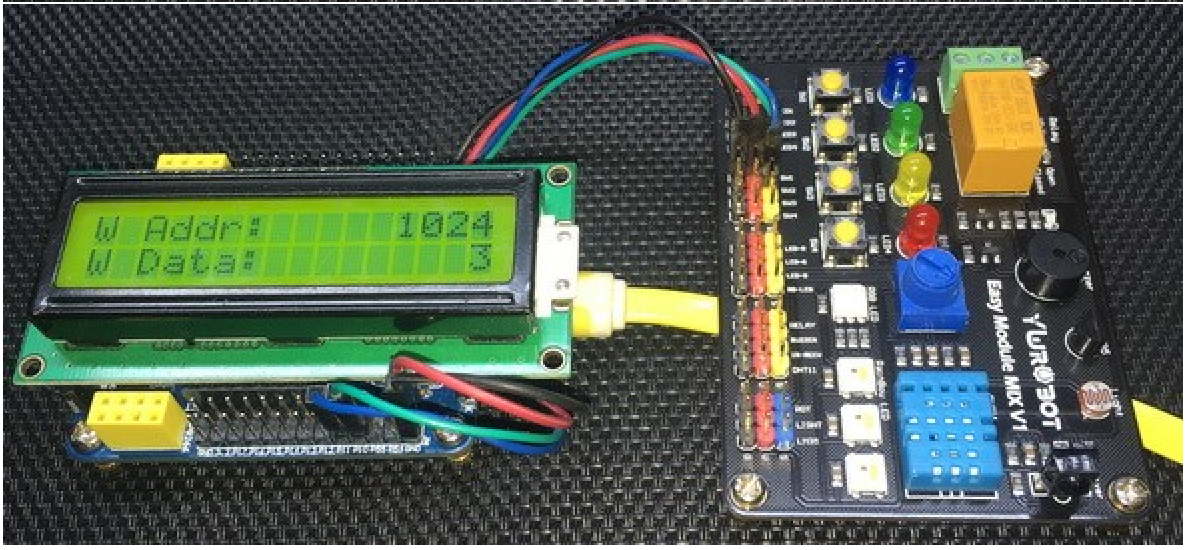
```
unsigned char IAP_read(unsigned int address)
{
  unsigned char value = 0x00;

  IAP_CONTR = IAP_WT;
  IAP_CMD = IAP_read_command;
  IAP_address(address);
  IAP_trigger;
  _nop_();
  value = IAP_DATA;
  IAP_clear;

  return value;
}

void IAP_write(unsigned int address, unsigned char value)
{
  IAP_CONTR = IAP_WT;
  IAP_CMD = IAP_write_command;
  IAP_address(address);
  IAP_DATA = value;
  IAP_trigger;
  _nop_();
  IAP_clear;
}

void IAP_erase(unsigned int address)
{
  IAP_CONTR = IAP_WT;
  IAP_CMD = IAP_erase_command;
  IAP_address(address);
  IAP_trigger;
  _nop_();
  IAP_clear;
}
```

Some key points to note while using internal flash as EEPROM:

1. The flash memory of STC micro can be divided into sectors of 512 bytes.
2. Byte write or byte erase operations are not possible as these are done at sector level. This means that to change one byte of a sector, we would have to totally erase and write that sector.
3. Basing on point 2, it is better to use two sectors for one set of data. We can, then, maintain wear-leveling and memory ring buffer.
4. We must use those locations of internal flash that won't have any part of application code. As with other microcontrollers, STC micros begin executing code from 0x0000 location of memory. Thus, it is better to use locations at the last portion of internal flash memory.
5. Flash memory has an endurance of 100,000 cycles. Thus, frequent writes/erases must be avoided. It is better to use a RAM buffer for applications that require frequent data storage. By using such a technique, flash memory is only modified before a reset or power down event. This is just like saving all your files before shutting down your PC. Most modern solid-state drives (SSD) come with a DRAM cache and this has almost similar usage.
6. IAP size can be set by using STC programmer GUI.

Demo





Demo video link: https://youtu.be/sxiDfofhr2E.

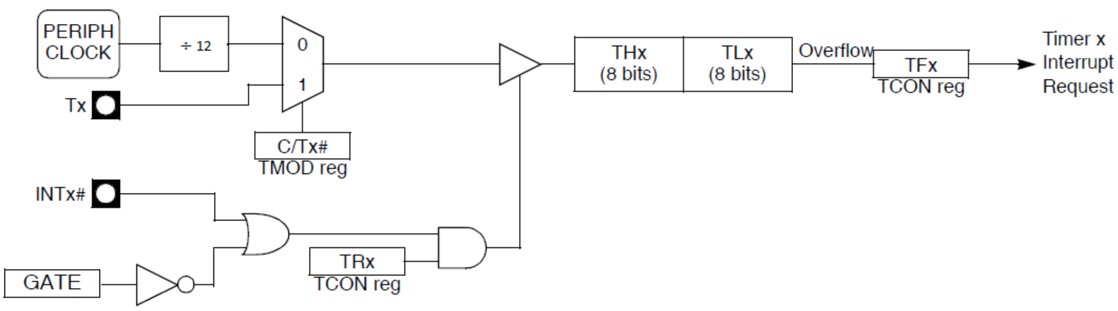
Timing-Related Hardware Overview

When it comes to timer-counters and time-related hardware, STC microcontrollers pack lot of punch. There are five 16-bit timer-counters, a Programmable Counter Array (PCA) and an enhanced PWM hardware peripheral. The timers are similar to the timers of standard 8051 microcontrollers. The rest of the hardware peripherals are bonus features that are not typically available in traditional 8051s.

Timer-Counters

Timer-Counters T0 – T4 are the five 16-bit timer-counters. As with the timers of any MCU, timers of STC microcontrollers can be employed for a number of tasks including time-base, timing events and outputs, measuring intervals, etc. Timers of 8051 architecture-based microcontrollers are usually also responsible for internal serial port (UART) hardware baud-rate generations and so is the case with STC microcontrollers. Thus, these timers are very versatile.

The typical structure of timers of traditional 8051s looks like the block diagram shown below but it is not completely the same for STC microcontrollers.



THx and ***TLx*** are cascaded counters that can be clocked using internal peripheral clock or externally via specific timer input GPIO pins. The clock that would drive the counters can be optionally scaled but unlike other microcontrollers the scaling is restricted to 1T or 12T, i.e., the driving source clock is either divided by one or divided by a factor of 12. When a timer is internally clocked, it is said to be acting like a timer and when it is clocked externally, it is said to be operating as a counter. All timers have interrupt capabilities and always count to top value from either 0 or some predefined value. Timers T0 and T1 are special timers while the rest are general purpose timers.

The timers can be operated in the following modes of operations:

Timer	16-bit Auto Reload Mode (Mode 0)	16-bit Non-auto Reload Mode (Mode 1)	8-bit Auto Reload Mode (Mode 2)	16-bit Auto Reload Mode with Interrupt (Mode 3)	Stop Mode (Mode 3)
Timer 0 (T0)	✓	✓	✓	✓	
Timer 1 (T1)	✓	✓	✓		✓
Timer 2 (T2)	✓				
Timer 3 (T3)	✓				
Timer 4 (T4)	✓				

These modes of operations have similarities with standard 8051s but there are also some minor differences.

- ***16-bit Auto Reload Mode (Mode 0)***

  In auto-reload mode, a timer's internal counter (*TLx* and *THx*) is loaded with some user-defined value at the beginning and the timer is started. It will count from that set value to the top value of 65535 (0xFFFF) and then overflow. After the overflow event, the user-defined value is automatically reloaded to the timer's counter and the process repeats. The overflow event can generate an interrupt.

- ***16-bit Non-auto Reload Mode (Mode 1)***

  Mode 1 is same as Mode 0 but the only difference between them is reloading feature. In Mode 1, a timer's internal counter (*TLx* and *THx*) needs to be reloaded manually through coding.

- ***8-bit Auto Reload Mode (Mode 2)***

  Mode 2 is similar to Mode 0. The differences are resolution and how the timer is reloaded. In this mode, *TLx* is used as counter and *THx* is used as the buffer that with reload *TLx* when overflow occurs.

- ***16-bit Auto Reload Mode with Interrupt (Mode 3)***

  This mode is same as Mode 0 but in this mode, interrupt cannot be disabled. In Mode 0, we have the option to either use timer interrupt or have it disabled.

- ***Stop Mode***

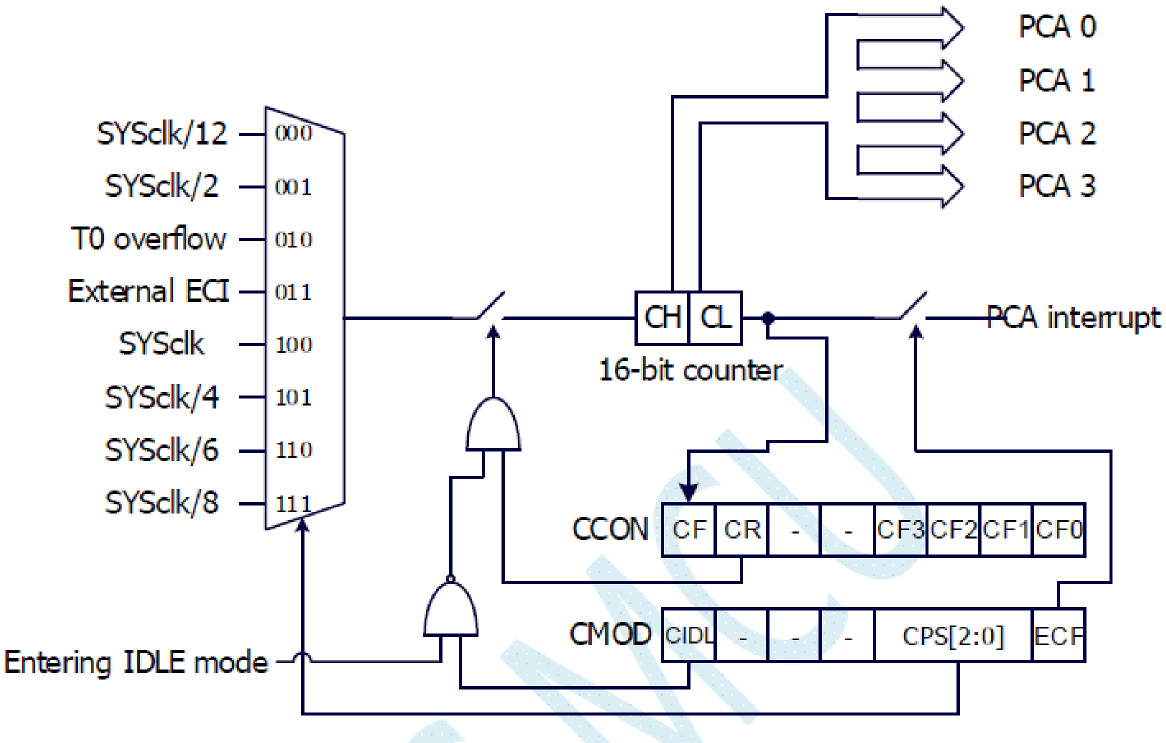
  Stop mode is only valid for Timer 1. Actually, all timers can be stopped somehow but entering this mode will disable Timer 1.

Enhanced PWM Module

The enhanced PWM module of STC8A8K64S4A12 provides 8 channel 15-bit PWM outputs. This module allows lot of advanced PWM features like complementary PWMs with dead-times, PWMs with initial level settings, etc. There are options to use this module with internal ADC and there are options to detect faults. This module is best suited for power control applications like inverters, battery chargers, switch-mode power supplies (SMPS), etc. Lastly, this module can operate independently without the need of help from other internal timer modules.

Programmable Counter Array (PCA)

STC8A8K64S4A12's programmable array counter is an interesting and rare hardware peripheral. PCA is not seen in most microcontrollers. PCA basically extends PWM capabilities of STC8A8K64S4A12 while including input capture peripherals, special outputs and robust interrupt capabilities. PCA-based PWMs are of either 6-bit, 7-bit, 8-bit and 10-bit resolution and can be considered as general-purpose PWMs. The PCA module can be considered as an additional built-in timer peripheral as it can operate independently just like the PWM module.



In STC8A8K64S4A12, there are four PCA groups – CCP0, CCP1, CCP2 and CCP3 that are dependent on a central 16-bit counter and configuration assets. CCP0, CCP1, CCP2 and CCP3 can be inputs or outputs depending on the mode of operation of PCA module itself. The counter can be clocked with a number of clock sources as shown in the block diagram.

Other Timers

Apart from all aforementioned peripherals, STC8A8K64S4A12 packs a wakeup timer and a watchdog timer. Wakeup timer can bring out a STC8A8K64S4A12 micro from low power sleep mode after fixed time intervals. Watchdog timer in STC8A8K64S4A12 is just like the watchdog timer of any other microcontroller and is mainly intended for resetting should there be an unforeseen software loop/error. These timers do not have alternative usage.

Using Timer 4 as Time-base Generator

The very basic use of a timer is to use it for time-base generation. By time-base generation, I mean that we use a timer as a free-running timer and without using any of its interrupts.

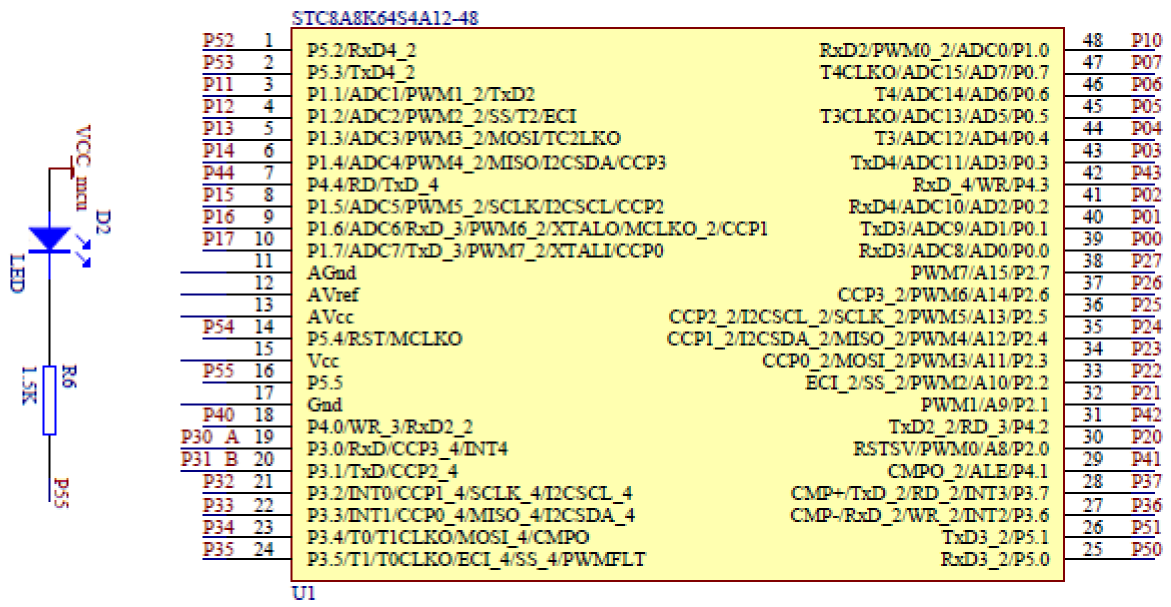
Symbol	Description	Address	Bit Address and Symbol								Value after reset
			B7	B6	B5	B4	B3	B2	B1	B0	
TCON	Timer 0 and 1 control register	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0000,0000
TMOD	Timer 0 and 1 mode	89H	GATE	C/T	M1	M0	GATE	C/T	M1	M0	0000,0000
TL0	Timer 0 low byte	8AH									0000,0000
TL1	Timer 1 low byte	8BH									0000,0000
TH0	Timer 0 high byte	8CH									0000,0000
TH1	Timer 1 high byte	8DH									0000,0000
AUXR	Auxiliary register 1	8EH	T0x12	T1x12	UART_M0x6	T2R	T2_C/T	T2x12	EXTRAM	S1ST2	0000,0001
INTCLKO	interrupt and clock output control register	8FH	-	EX4	EX3	EX2	-	T2CLKO	T1CLKO	T0CLKO	x000,x000
WKTCL	Wake-up Timer Control register low	AAH									1111,1111
WKTCH	Wake-up Timer Control register high	ABH	WKTEN								0111,1111
T4T3M	Timer4 and Timer 3 mode register	D1H	T4R	T4_C/T	T4x12	T4CLKO	T3R	T3_C/T	T3x12	T3CLKO	0000,0000
T4H	Timer 4 high byte	D2H									0000,0000
T4L	Timer 4 low byte	D3H									0000,0000
T3H	Timer 3 high byte	D4H									0000,0000
T3L	Timer 3 low byte	D5H									0000,0000
T2H	Timer 2 high byte	D6H									0000,0000
T2L	Timer 2 low byte	D7H									0000,0000

Code

```
#include "STC8xxx.h"
#include "BSP.h"


void setup(void);


void main(void)
{
  setup();

  while(1)
  {
      if(TMR4_get_counter() >= 0x9E58)
      {
        P55_high;
      }
      else
      {
        P55_low;
      }
  };
}
```

```
void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 24, MCLK_SYSCLK_no_output, MCLK_out_P54);

  P55_open_drain_mode;

  TMR4_setup(TMR4_sysclk, \
             TMR4_clk_prescalar_12T, \
             TMR4_no_clk_out);

  TMR4_load_counter_16(0x3CB0);
  TMR4_start;
}
```

Schematic



Explanation

For this demo, onboard LED is used as the demo is a simple LED blinking. The desired LED blink rate is 1.67Hz (about 600ms).

The system clock is set to 1MHz. Everything related to internal hardware timing is dependent on system clock and so its selection is very important. The system clock is set to 1MHz as the required blink rate is very small. Every system clock tick is, therefore, 1µs.

```
CLK_set_sys_clk(IRC_24M, 24, MCLK_SYSCLK_no_output, MCLK_out_P54);
```

Now let's see how the timer is configured. Firstly, timer 4 is used as it is one of the simplest timers. Obviously, this timer is working in 16-bit auto-reload mode (Mode 0) as no other mode of operation is available for it.

```
TMR4_setup(TMR4_sysclk, \
           TMR4_clk_prescalar_12T, \
           TMR4_no_clk_out);

TMR4_load_counter_16(0x3CB0);
TMR4_start;
```

As we can see the timer is fed with internal system clock (1MHz). This is further scaled down by a factor of 12, i.e., the timer is ticking at:

$$\frac{1000000\ Hz}{12} = 83333.33\text{Hz or }12\mu\text{s}$$

To achieve 600ms (1.67Hz) period, we would need:

$$\frac{600 \times 10^{-3}\ \text{s}}{12 \times 10^{-6}\ \text{s}} = 50000\ \text{timer counts or ticks}$$

All 16-bit timers start counting from a bottom value to a fixed top value of 65535 (0xFFFF) counts. A timer's internal counter overflows after exceeding the top value and then rolls over or repeats counting from its bottom value. The bottom value is 0 (0x0000) unless some other value is assigned in the code. Thus, unlike the top value, the bottom value can be changed. In this example, the bottom value is 15536 (0x3CB0). This, in effect, help us achieve 50000 counts.

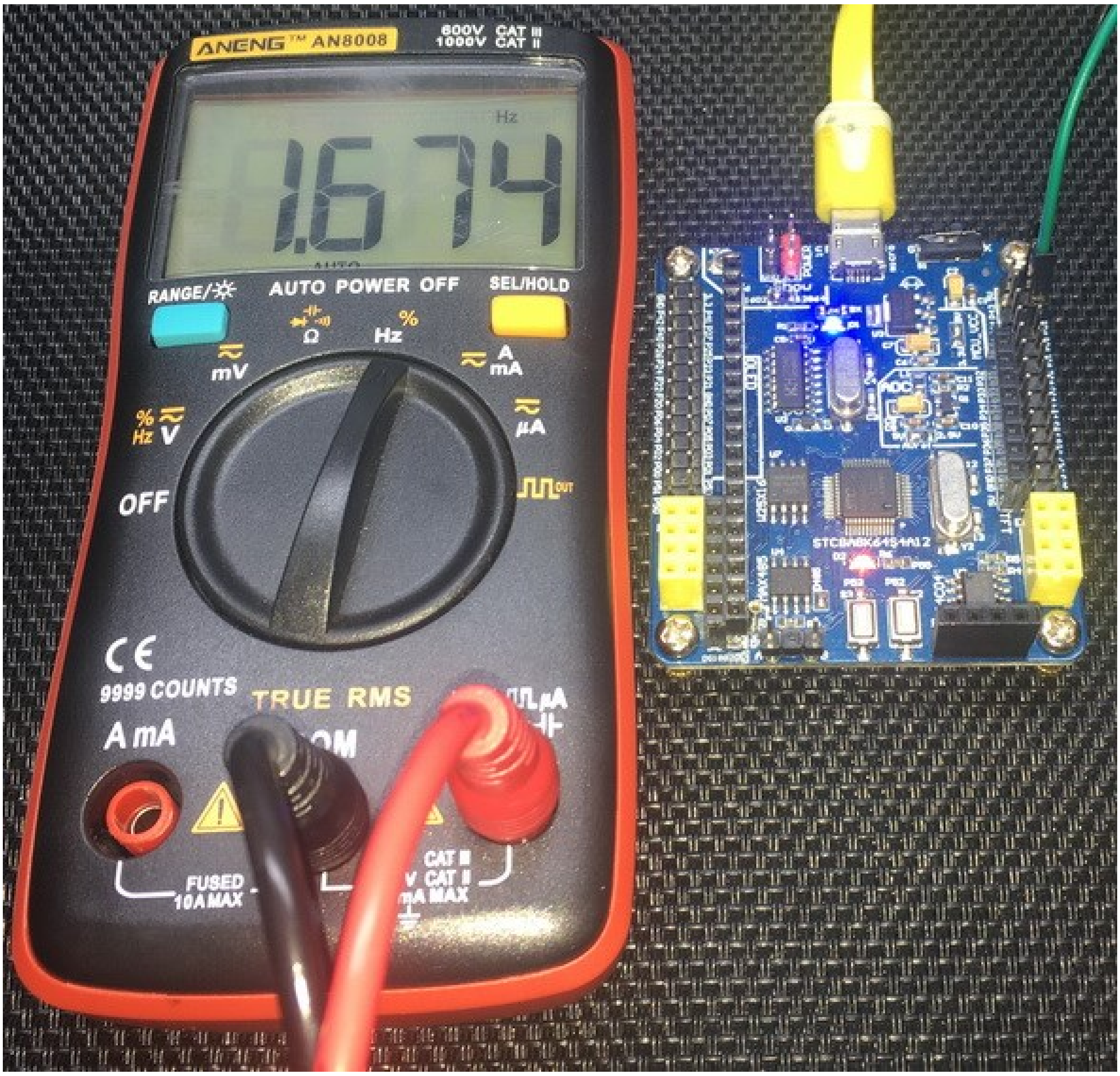
$$65536 - 15536 = 50000\ \text{counts}$$

Now, we want to equally divide the on and off times of the LED and so the 50000 counts are divided into two halves, each 25000 counts individually. Thus, in the main loop, the timer's counter count is polled.

```
if(TMR4_get_counter() >= 0x9E58)
{
  P55_high;
}
else
{
  P55_low;
}
```

If the count is greater than or equal to 40536 (0x9E58), P55 is held high or else it is held low. Thus, from 15536 count to 40535 count (40535 – 15536 = 24999), the pin's state is low and from 40536 count to 65535 count (65535 – 40536 = 24999), the pin state is high.

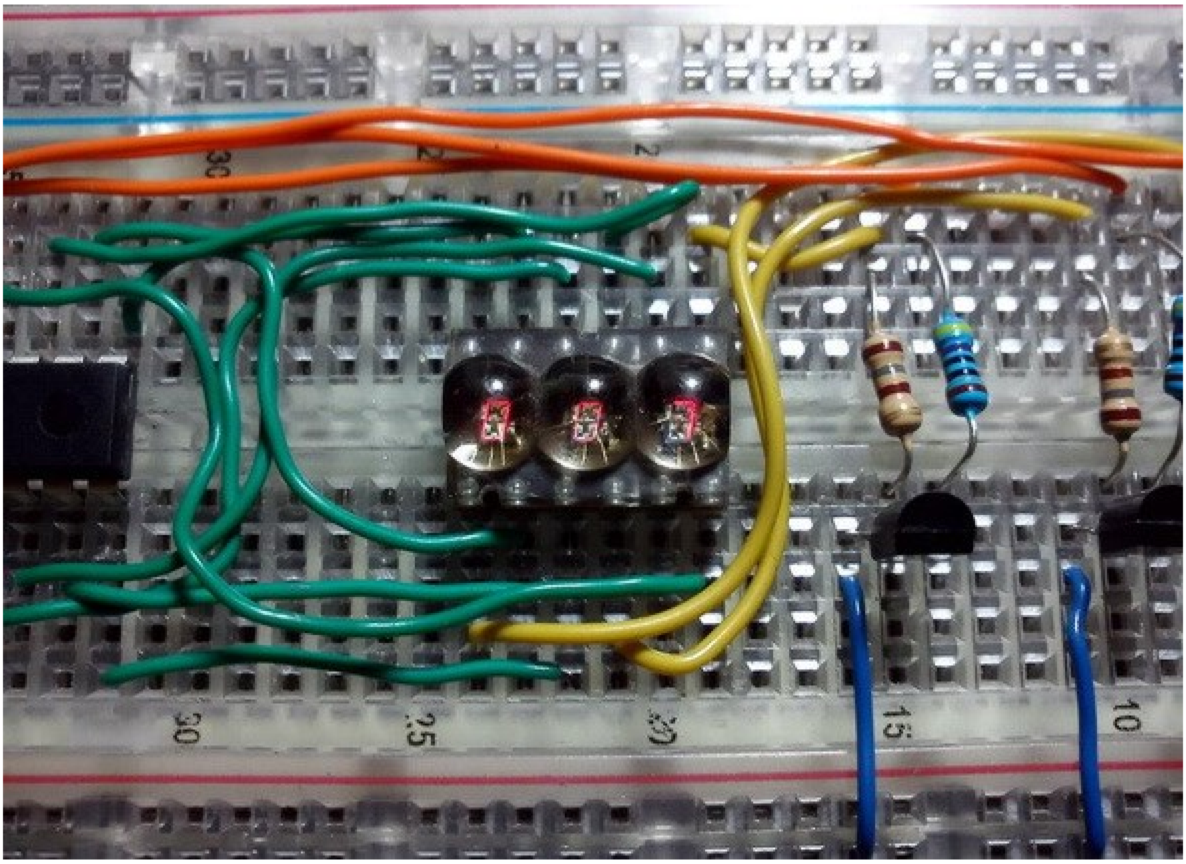
Demo



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

Timer 1 as a Counter

Apart from timing events, the other role of timers is to count pulses. Pulse counting has number of applications like motor speed control, object counting, frequency measurements, encoder counting, etc.

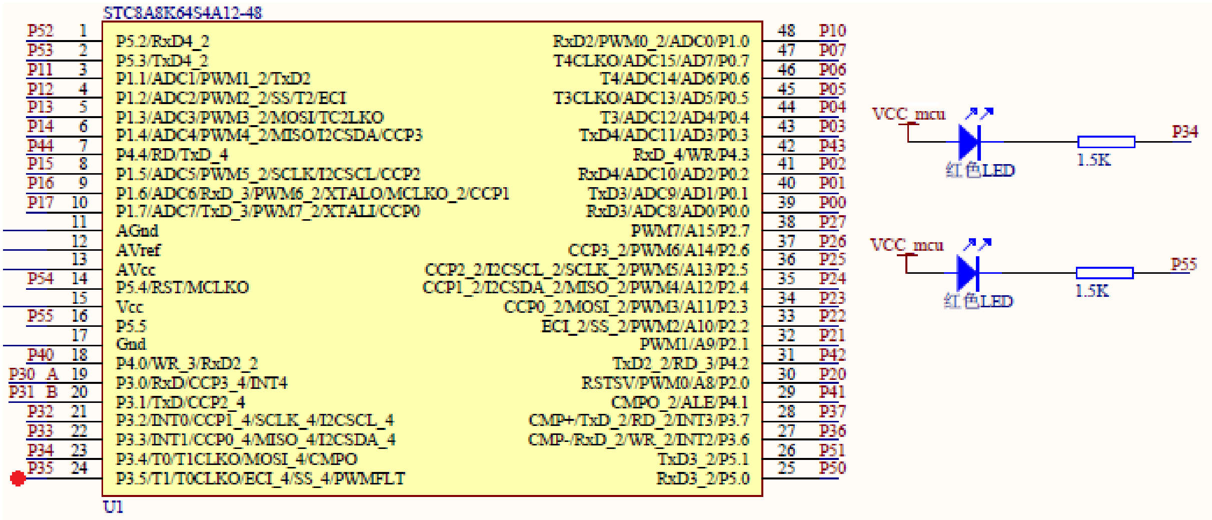


Code

```
#include "STC8xxx.h"
#include "BSP.h"


void setup(void);


void main(void)
{
  setup();

  while(1)
  {
    if(TMR1_get_counter() >= 32768)
    {
      P55_high;
    }
    else
    {
      P55_low;
    }
  };
}
```

```
void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  P34_push_pull_mode;

  P35_input_mode;

  P55_open_drain_mode;

  TMR1_setup(TMR1_16_bit_non_auto_reload, \
             TMR1_ext_T1_P35_clk, \
             TMR1_clk_prescalar_12T, \
             TMR1_int_gate, \
             TMR1_P34_clk_out);

  TMR1_load_counter_8(0x00, 0x00);

  TMR1_start;
}
```

Schematic



Explanation

This example is same as the previous timer example, the only difference is the clock source of the timer involved here. Timer 1 is used and it is externally clocked by using a signal generator. In this example, the blinking rate of the onboard LED changes with the clock speed of the timer.

First let's see how the timer is set up.

```
TMR1_setup(TMR1_16_bit_non_auto_reload, \
           TMR1_ext_T1_P35_clk, \
           TMR1_clk_prescalar_12T, \
           TMR1_int_gate, \
           TMR1_P34_clk_out);

TMR1_load_counter_8(0x00, 0x00);

TMR1_start;
```

Here, we can, firstly, see that the timer is set in non-auto reload mode. This means that timer needs periodic reloads after overflow or else it will restart counting from 0. In our case, this won't matter because we want the timer to count from 0 after overflow. Secondly, we can see that the timer is externally clocked and this is achieved by feeding clock pulses to P3.5 (T1) GPIO pin. Since the timer is externally clocked, it doesn't matter what system clock settings are. The next thing to note is the timer's clock prescalar. This is needed because we want to visually see the outcome with LEDs. If the LEDs blink too fast, we won't not be able to realize what is going on. Lastly, internal gating is used because we don't the timer to depend on other hardware events and we want to get timer output. Timer 1 will toggle the logic state of P3.4 (T1CLKO) GPIO pin when it overflows. Both GPIO pins P3.4 and P3.5 need to configured.

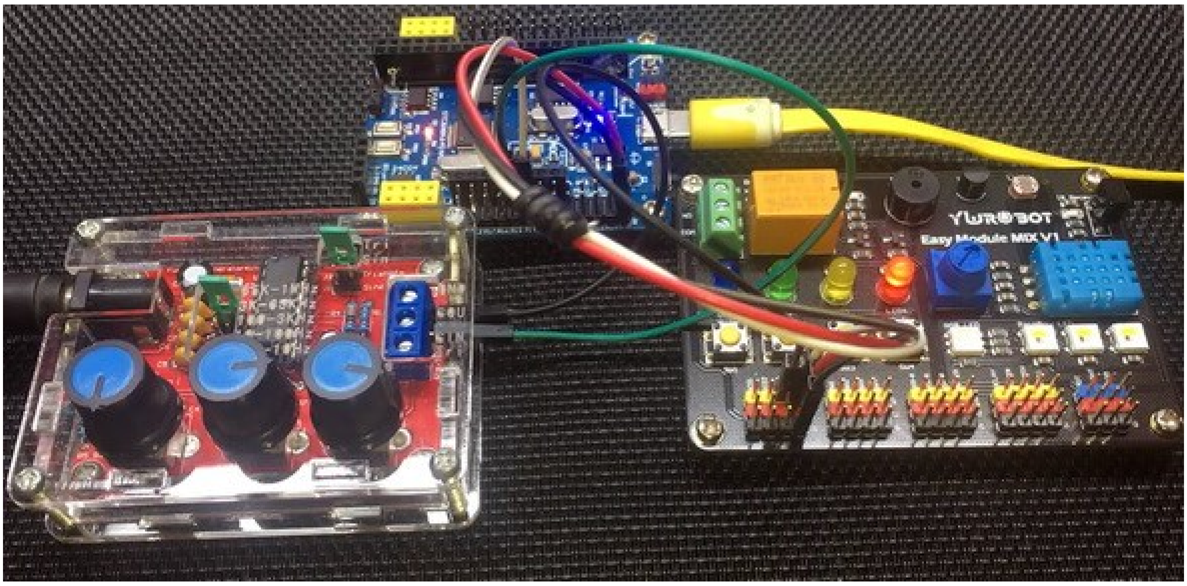
```
P34_push_pull_mode;

P35_input_mode;
```

State of P5.5 LED changes according to the value of timer's internal counter. For one half of the 16-bit count the LED is held low and for the other half it is held high.

```
if(TMR1_get_counter() >= 32768)
{
  P55_high;
}

else
{
  P55_low;
}
```

Thus, in the main, this is only job done. Rate of blinking depends on external clock speed. The higher the clock frequency, the higher is the blinking rate.

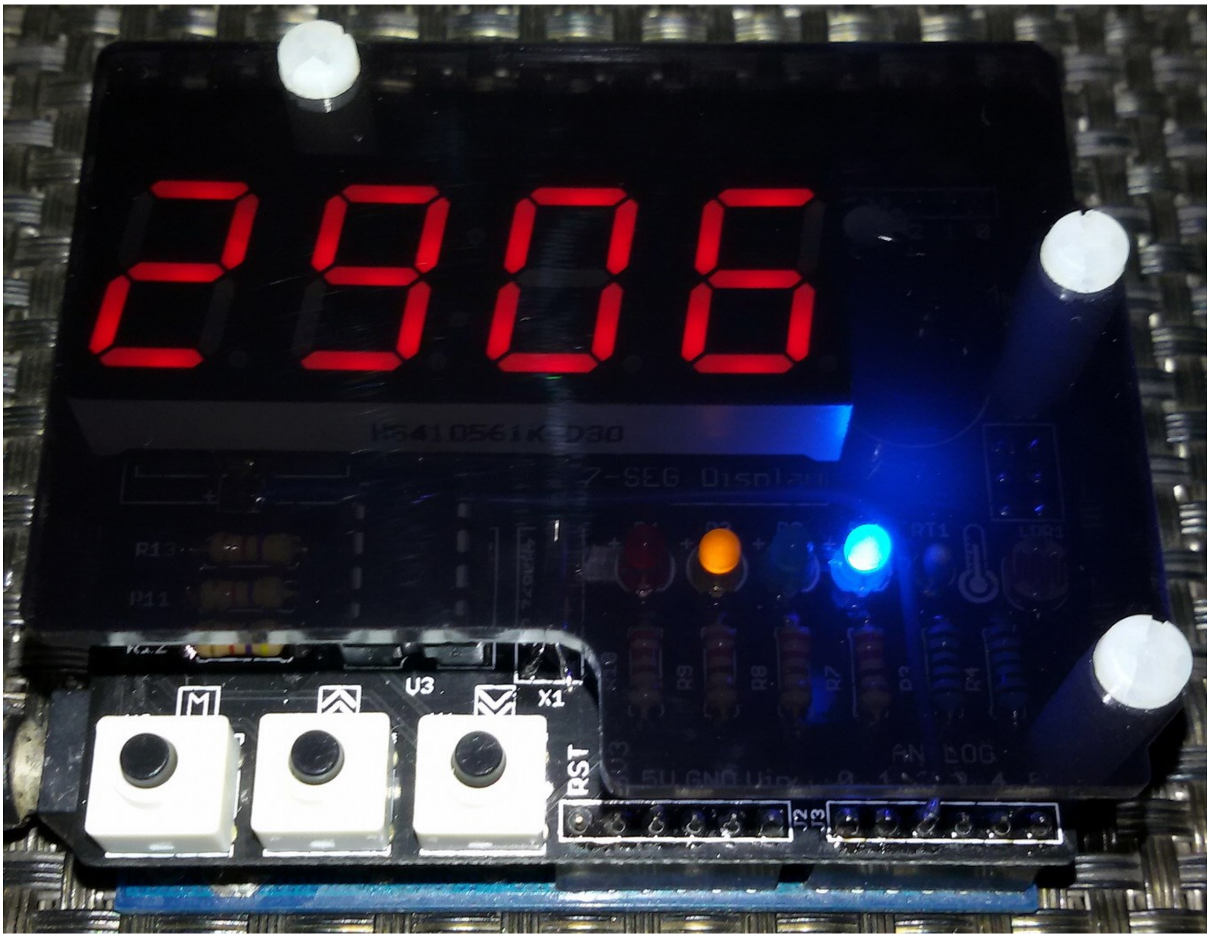
Demo



Demo video link: https://youtu.be/GIA7gT3RxY4.

Using Timer 2 to Multiplex Seven Segment Displays

Rather than polling, a better way to use timers is to use their interrupts. Timer overflow interrupt is particularly useful for timing event accurately. Timer overflow interrupt will periodically interrupt main tasks and can perform certain things inside the interrupt and then get back to main tasks. In this way, several tasks will apparently seem to occur concurrently. Multiplexing seven segment displays is one such example. These displays need periodic scanning, multiplexing and updating. If these tasks are done swiftly and in short duration, human eyes can be tricked. If a CPU is occupied doing these tasks alone then it would not be able to do other tasks. If the tasks of seven segment displays are put inside a timer interrupt, the CPU can do other tasks in other interrupts and main code as seven segment displays need only periodic attention and not all code processing is spent on the displays alone.



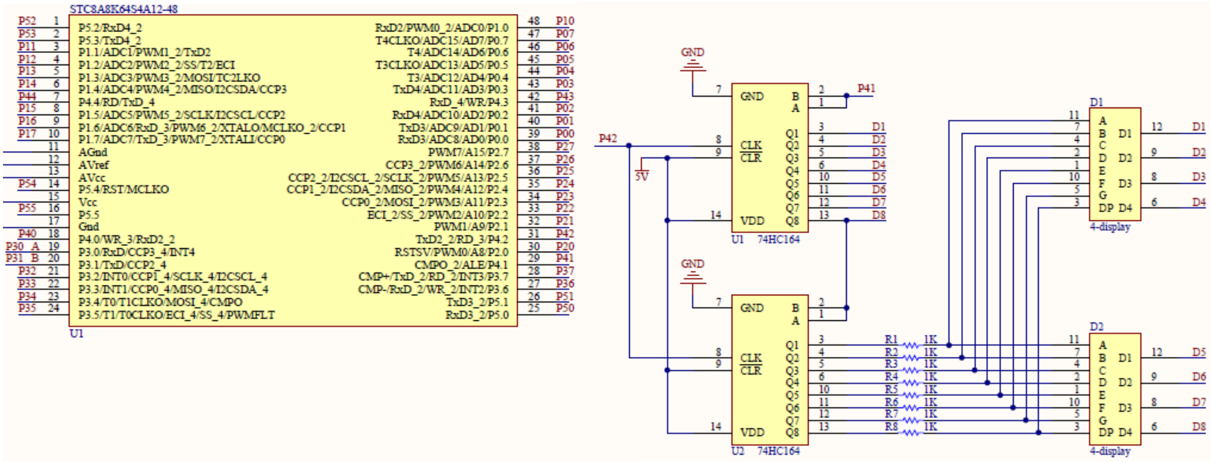
Code

```
#include "STC8xxx.h"
#include "BSP.h"


#define DAT_pin_HIGH                    P41_high
#define DAT_pin_LOW                     P41_low

#define CLK_pin_HIGH                    P42_high
#define CLK_pin_LOW                     P42_low

#define refresh_time_in_us              315
```

```
#define max_tmr_cnt                     0xFFFF
#define tmr_val                         (max_tmr_cnt - refresh_time_in_us)

const unsigned char pos[0x08] =

{
    0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80
};

const unsigned char num[0x0A] =
{
    0x03, 0x9F, 0x25, 0x0D, 0x99, 0x49, 0x41, 0x1F, 0x01, 0x09
};

unsigned char i = 0x00;

signed int value_1 = 0;
signed int value_2 = 0;

void setup(void);
void Write_74HC164(unsigned char val, unsigned char seg);

void ADC_ISR(void)
interrupt 5
{
  value_1 = ((ADC_RES << 8) | ADC_RESL);
  value_2 = ((value_1 * 5000.0) / 4096.0);
  clear_ADC_flag;
}

void TMR_2_ISR(void)
interrupt 12
{
    switch(i)
    {
        case 0:
        {
            Write_74HC164(num[(value_1 / 1000)], pos[3]);
            break;
        }

        case 1:
        {
            Write_74HC164((num[((value_1 % 1000) / 100)]), pos[2]);
            break;
        }

        case 2:
        {
            Write_74HC164(num[((value_1 % 100) / 10)], pos[1]);
            break;
        }

        case 3:
        {
            Write_74HC164(num[(value_1 % 10)], pos[0]);
            break;
        }

        case 4:
        {
            Write_74HC164(num[(value_2 / 1000)] & 0xFE, pos[7]);
            break;
        }

        case 5:
        {
            Write_74HC164((num[((value_2 % 1000) / 100)]), pos[6]);
            break;
        }

        case 6:
        {
            Write_74HC164(num[((value_2 % 100) / 10)], pos[5]);
            break;
        }

        case 7:
        {
```

```
            Write_74HC164(num[(value_2 % 10)], pos[4]);
            break;
        }
    }

    i++;

    if(i >= 8)
    {
        i = 0;
    }
    clear_TMR_2_overflow_flag;
}

void main(void)
{
  setup();

    while(1)
    {
        ADC_set_channel(CH15);
        delay_ms(10);
        ADC_start_conversion;
        delay_ms(400);
    };
}

void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  P41_push_pull_mode;
  P42_push_pull_mode;

  ADC_enable;
  ADC_result_format_right_aligned;
  ADC_set_conversion_speed(ADC_conv_128_CLKs);
  _enable_ADC_interrupt;

  TMR2_setup(TMR2_sysclk, TMR2_clk_prescalar_12T);
  TMR2_load_counter_16(tmr_val);
  TMR2_start;
  _enable_TMR_2_interrupt;
  _enable_global_interrupt;
}
void Write_74HC164(unsigned char val, unsigned char seg)
{
    unsigned char s = 0x10;
    unsigned int temp = 0x0000;

    temp = (unsigned int)seg;
    temp <<= 8;
    temp |= (unsigned int)val;

    while(s > 0)
    {
        if((temp & 0x0001) != 0x0000)
        {
            DAT_pin_HIGH;
        }
        else
        {
            DAT_pin_LOW;
        }

        CLK_pin_HIGH;
        temp >>= 1;

        s--;
        CLK_pin_LOW;
    }
}
```

Schematic



Explanation

For demonstrating timer interrupt and seven segment displays, I used a simple dual 4-digit seven segment displays. It is made with a set of 74HC164 Serial-In-Parallel-Out (SPIO) shift register. In order to use it in real-time, its displays need to be scanned and updated at a fast rate without affecting other tasks.

This demo is a very important one as here multiple interrupts are employed and literally there is nothing inside the main loop. Things are done in such a way as if everything is parallelly done and in real-time.

Firstly, the system clock and GPIOs are set. The system clock is set to 12 MHz. P4.1 and P4.2 are GPIOs that are used for driving the display. These drive data and clock lines of the display respectively.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

P41_push_pull_mode;
P42_push_pull_mode;
```

ADC setup is as simple as the past ADC example. The conversion rate is set to about 3 kHz and the ADC interrupt is enabled. Note no external ADC pin is used as the goal here is to read the internal reference voltage source, i.e., ADC Channel 15.

```
ADC_enable;
ADC_result_format_right_aligned;
ADC_set_conversion_speed(ADC_conv_128_CLKs);
_enable_ADC_interrupt;
```

ADC interrupt is triggered when a conversion is completed. Inside this interrupt all we have to do is to read the ADC and clear the AD conversion completion flag.

```
void ADC_ISR(void)
interrupt 5
{
  value_1 = ((ADC_RES << 8) | ADC_RESL);
  value_2 = ((value_1 * 5000.0) / 4096.0);
  clear_ADC_flag;
}
```

Timer 2 is used in this example and so let us now see how it is configured. There are eight seven segment displays and we would want them to be multiplexed quickly in order to make them appear to update simultaneously. We cannot afford to stay long for each display. Thus, this task should be completed within 2 – 3ms time. Due to this reason, the timer is set to interrupt every 315µs. The total time for all displays is therefore about 2.52ms (8 × 0.315ms).

```
#define refresh_time_in_us              315
#define max_tmr_cnt                     0xFFFF
#define tmr_val                         (max_tmr_cnt - refresh_time_in_us)

....

TMR2_setup(TMR2_sysclk, TMR2_clk_prescalar_12T);
TMR2_load_counter_16(tmr_val);
TMR2_start;
_enable_TMR_2_interrupt;
```

Timer 2 is a 16-bit timer with auto-reload feature. In this demo, it is clocked internally with the 12 MHz system clock. This clock is further scaled down by a factor of 12 and so the timer's ticks have a frequency of 1 MHz, i.e., the timer ticks are 1µs apart.

$$Timer\ Frequency = \frac{System\ Clock}{Prescalar} = \frac{12\ MHz}{12} = 1\ MHz$$

$$Timer\ Tick = \frac{1}{Timer\ Frequency} = \frac{1}{1\ MHz} = 1\mu s$$

We want the timer to interrupt every 315µs and so the timer is loaded with a value of 65220 since all 16-bit timers count up to 65535.

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

$$= 65535 - \left(\frac{315\mu s}{1\mu s}\right) = 65520$$

Lastly, the timer is started with its interrupt enabled.

Inside the timer interrupt the displays are updated by writing to the 74HC164 chips and clearing the timer overflow interrupt flag.

```
void TMR_2_ISR(void)
interrupt 12
{
    switch(i)
    {
        case 0:
        {
            Write_74HC164(num[(value_1 / 1000)], pos[3]);
            break;
        }

        case 1:
        {
            Write_74HC164((num[((value_1 % 1000) / 100)]), pos[2]);
            break;
        }

        case 2:
        {
            Write_74HC164(num[((value_1 % 100) / 10)], pos[1]);
            break;
        }

        case 3:
        {
            Write_74HC164(num[(value_1 % 10)], pos[0]);
            break;
        }

        case 4:
        {
            Write_74HC164(num[(value_2 / 1000)] & 0xFE, pos[7]);
            break;
        }

        case 5:
        {
            Write_74HC164((num[((value_2 % 1000) / 100)]), pos[6]);
            break;
        }

        case 6:
        {
            Write_74HC164(num[((value_2 % 100) / 10)], pos[5]);
            break;
        }

        case 7:
        {
            Write_74HC164(num[(value_2 % 10)], pos[4]);
            break;
        }
    }

    i++;

    if(i >= 8)
    {
        i = 0;
    }

    clear_TMR_2_overflow_flag;
}
```

The function below is the function for writing the 74HC164 shift-register ICs. Since the shift-registers are cascaded and we need them to work simultaneously, it is better to bit-bang seven segment data as 16-bit chunks – each word consisting of segment position and segment value data.

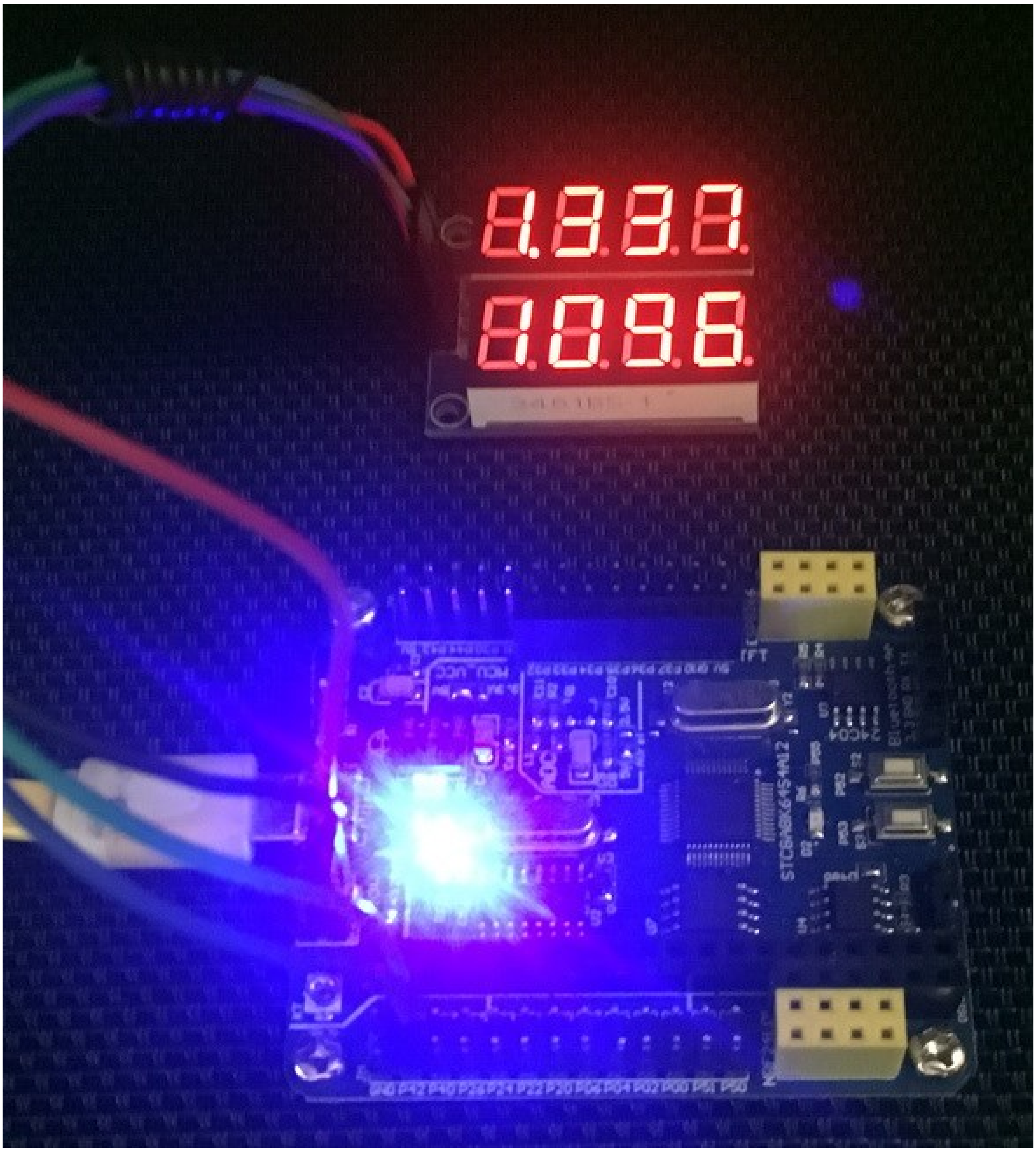
```
void Write_74HC164(unsigned char val, unsigned char seg)
{
    unsigned char s = 0x10;
    unsigned int temp = 0x0000;

    temp = (unsigned int)seg;
    temp <<= 8;
    temp |= (unsigned int)val;

    while(s > 0)
    {
        if((temp & 0x0001) != 0x0000)
        {
            DAT_pin_HIGH;
        }
        else
        {
            DAT_pin_LOW;
        }

        CLK_pin_HIGH;

        temp >>= 1;
        s--;

        CLK_pin_LOW;
    }
}
```

Note interrupt priorities are not changed and are left in their default conditions.

In the main, the channel to be read is selected and AD conversion is triggered. This operation is cycled every 400ms. Channel 15 is read here. Channel 15 is the internal reference voltage source. The internal reference voltage source has a typical value of 1.344V. The seven segment displays show ADC counts and voltage value.

```
ADC_set_channel(CH15);
delay_ms(10);
ADC_start_conversion;
delay_ms(400);
```

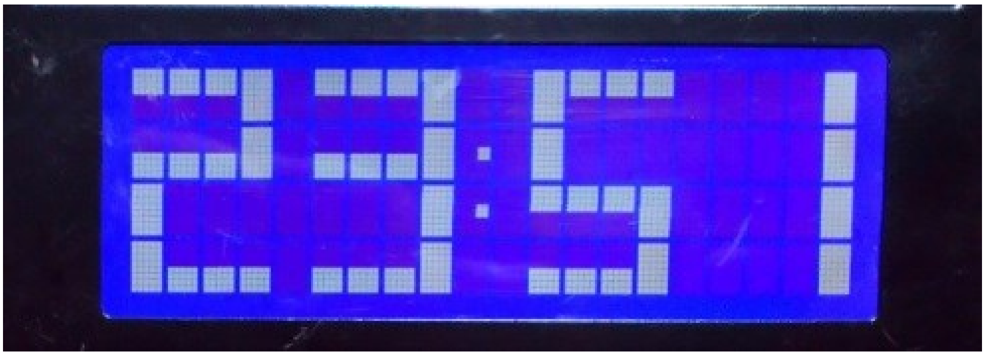
Demo



Demo video link: https://youtu.be/gFKg9aYkMrM.

Timer 3 as Real Time Clock & Calendar (RTCC)

Since timers can time event precisely, we can use timers for time keeping too. Although there is no dedicated real-time clock-calendar (RTCC) hardware in STC8A8K64S4A12, we can make one by using a timer and some coding.



Code

```
#include "STC8xxx.h"
#include "BSP.h"
#include "LCD.c"


#define required_time              1000
#define max_tmr_cnt                0xFFFF
#define tmr_val                    (max_tmr_cnt - required_time)

#define PM                         0
#define AM                         1

#define ETR                        1
#define INC                        2
#define DEC                        3
#define ESC                        4
#define NON                        0

#define ETR_button                 !P10_get_input
#define INC_button                 !P11_get_input
#define DEC_button                 !P12_get_input
#define ESC_button                 !P13_get_input


unsigned char dow = 1;
unsigned char s = 10;
unsigned char hr = 10;
unsigned char min = 10;
unsigned char date = 1;
unsigned char month = 1;
unsigned char toggle = 0;
unsigned char leap_year = 0;
unsigned char ampm = AM;

unsigned int ms = 0;
unsigned int year = 2000;


void setup(void);
void LCD_print_value(unsigned char x_pos, unsigned char y_pos, unsigned char value);
void LCD_print_int_value(unsigned char x_pos, unsigned char y_pos, unsigned int value);
void display_time(void);
void AM_PM_disp(unsigned char state);
void dow_disp(unsigned char dow);
unsigned char get_buttons(void);
unsigned char set_value(unsigned char x_pos, unsigned char y_pos, signed char value, unsigned char max, unsig
ned char min, unsigned char type);
```

```
void setup_RTC(void);


void TMR_3_ISR(void)
interrupt 19
{
  ms++;
  if(ms > 999)
  {
    s++;
    ms = 0;
    toggle ^= 0x01;
    P54_toggle;
  }

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

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

      if(hr > 12)
      {
          hr = 1;
      }

      if((hr == 12) && (min == 0) && (s == 0))
      {
        ampm ^= 1;

        if(ampm == AM)
        {
          date++;
          dow++;
        }

        if(dow > 7)
        {
          dow = 1;
        }

        if((month == 1) || (month == 3) || (month == 5) || (month == 7) || (month == 8) || (month == 10) || (
month == 12))
        {
          if(date > 31)
          {
            date = 1;
            month++;
          }
        }

        else if((month == 4) || (month == 6) || (month == 9) || (month == 11))
        {
          if(date > 30)
          {
            date = 1;
            month++;
          }
        }

        else
        {
          if((year % 4) == 0)
          {
            if((year % 100) == 0)
            {
              if((year % 400) == 0)
              {
                leap_year = 1;
              }

              else
              {
                leap_year = 0;
              }
```

```
            }

            else
            {
              leap_year = 1;
            }
          }

          else
          {
              leap_year = 0;
          }

          if((leap_year) && (date > 29))
          {
             date = 1;
             month++;
          }

          else if((!leap_year) && (date > 28))
          {
             date = 1;
             month++;
          }
        }

        if(month > 12)
        {
          month = 1;
          year++;
        }
      }
    }
  }

  clear_TMR_3_overflow_flag;
}


void main(void)
{
  setup();

  while(1)
  {
    if(get_buttons() == ESC)
    {
        while(get_buttons() == ESC);
        setup_RTC();
    }
    else
    {
        display_time();
    }
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  P10_input_mode;
  P11_input_mode;
  P12_input_mode;
  P13_input_mode;

  P54_push_pull_mode;

  LCD_init();
  LCD_clear_home();

  TMR3_setup(TMR3_sysclk, TMR3_clk_prescalar_12T, TMR3_no_clk_out);
  TMR3_load_counter_16(tmr_val);
  TMR3_start;
  _enable_TMR_3_interrupt;
  _enable_global_interrupt;
}
```

```
void LCD_print_value(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 LCD_print_int_value(unsigned char x_pos, unsigned char y_pos, unsigned int value)
{
  unsigned char temp = 0x00;

  temp = (value / 100);
  LCD_print_value(x_pos, y_pos, temp);

  temp = (value % 100);
  LCD_print_value((x_pos + 2), y_pos, temp);

}


void display_time(void)
{
  LCD_print_value(2, 0, hr);
  LCD_print_value(5, 0, min);
  LCD_print_value(8, 0, s);

  switch(toggle)
  {
    case 1:
    {
      LCD_goto(4, 0);
      LCD_putchar(':');
      LCD_goto(7, 0);
      LCD_putchar(':');
      break;
    }
    default:
    {
      LCD_goto(4, 0);
      LCD_putchar(' ');
      LCD_goto(7, 0);
      LCD_putchar(' ');
      break;
    }
  }

  AM_PM_disp(ampm);
  dow_disp(dow);

  LCD_print_value(6, 1, date);

  LCD_goto(8, 1);
  LCD_putchar('.');
  LCD_print_value(9, 1, month);

  LCD_goto(11, 1);
  LCD_putchar('.');
  LCD_print_int_value(12, 1, year);
}


void dow_disp(unsigned char dow)
{
  LCD_goto(0, 1);

  switch(dow)
  {
    case 1:
    {
      LCD_putstr("MON");
      break;
    }
    case 2:
    {
```

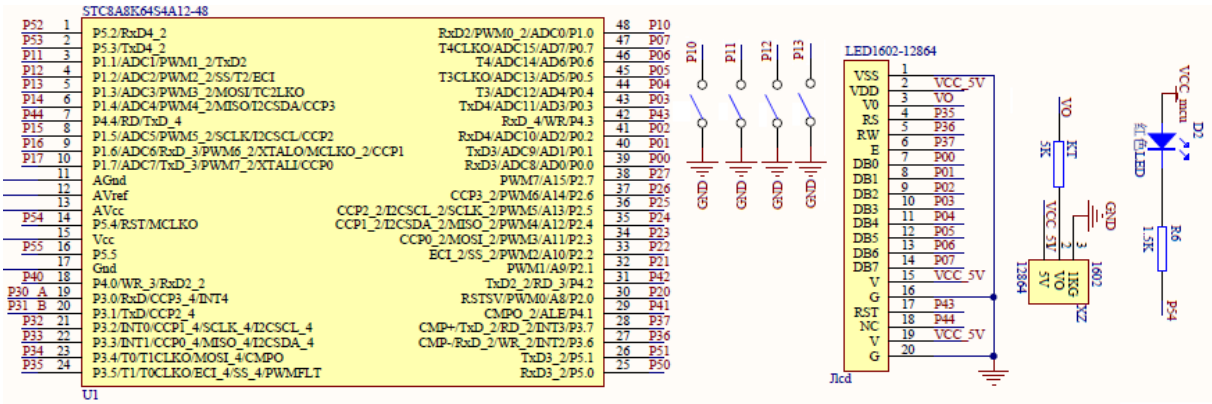
```
      LCD_putstr("TUE");
      break;
    }
    case 3:
    {
      LCD_putstr("WED");
      break;
    }
    case 4:
    {
      LCD_putstr("THR");
      break;
    }
    case 5:
    {
      LCD_putstr("FRI");
      break;
    }
    case 6:
    {
      LCD_putstr("SAT");
      break;
    }
    case 7:
    {
      LCD_putstr("SUN");
      break;
    }
    default:
    {
      LCD_putstr("   ");
      break;
    }
  }
}


void AM_PM_disp(unsigned char state)
{
  LCD_goto(12, 0);

  switch(state)
  {
    case AM:
    {
      LCD_putstr("AM");
      break;
    }
    default:
    {
      LCD_putstr("PM");
      break;
    }
  }
}


unsigned char get_buttons(void)
{
  if(ETR_button)
  {
    return ETR;
  }

  else if(INC_button)
  {
    return INC;
  }

  else if(DEC_button)
  {
    return DEC;
  }

  else if(ESC_button)
  {
    return ESC;
  }
```

```
  else
  {
    return NON;
  }
}


unsigned char set_value(unsigned char x_pos, unsigned char y_pos, signed char value, unsigned char max, unsig
ned char min, unsigned char type)
{
  unsigned char tgl = 0;

  while(1)
  {
    tgl ^= 0x01;
    delay_ms(90);

    if(get_buttons() == INC)
    {
        value++;
    }

    if(value > max)
    {
        value = min;
    }

    if(get_buttons() == DEC)
    {
        value--;
    }

    if(value < min)
    {
        value = max;
    }

    switch(type)
    {
      case 1:
      {
        switch(tgl)
        {
          case 1:
          {
            LCD_print_value(x_pos, y_pos, value);
            break;
          }

          default:
          {
            LCD_goto(x_pos, y_pos);
            LCD_putstr("  ");
            break;
          }
        }

        break;
      }

      case 2:
      {
        switch(tgl)
        {
          case 1:
          {
            AM_PM_disp(value);
            break;
          }

          default:
          {
            LCD_goto(12, 0);
            LCD_putstr("  ");
            break;
          }
        }
```

```
        break;
      }

      default:
      {
        switch(tgl)
        {
          case 1:
          {
            dow_disp(value);
            break;
          }
          default:
          {
            dow_disp(0);
            break;
          }
        }
        break;
      }
    }

    if((get_buttons() == ETR) && (tgl == 1))
    {
       return value;
    }
  };
}


void setup_RTC(void)
{
  unsigned int yr1 = 0;
  unsigned int yr2 = 0;

  TMR3_stop;
  _disable_TMR_3_interrupt;

  yr1 = (year / 100);
  yr2 = (year % 100);

  hr = set_value(2, 0, hr, 12, 1, 1);
  delay_ms(200);
  min = set_value(5, 0, min, 59, 0, 1);
  delay_ms(200);
  s = set_value(8, 0, s, 59, 0, 1);
  delay_ms(200);
  ampm = set_value(12, 0, ampm, 1, 0, 2);
  delay_ms(200);
  dow = set_value(0, 1, dow, 7, 1, 0);
  delay_ms(200);
  date = set_value(6, 1, date, 31, 1, 1);
  delay_ms(200);
  month = set_value(9, 1, month, 12, 1, 1);
  delay_ms(200);
  yr1 = set_value(12, 1, yr1, 99, 0, 1);
  delay_ms(200);
  yr2 = set_value(14, 1, yr2, 99, 0, 1);
  delay_ms(200);

  year = ((yr1 * 100) + yr2);

  if((month == 1) || (month == 3) || (month == 5) || (month == 7) || (month == 8) || (month == 10) || (month
== 12))
  {
    if(date > 31)
    {
        date = 1;
    }
  }

  else if((month == 4) || (month == 6) || (month == 9) || (month == 11))
  {
    if(date > 30)
    {
        date = 1;
    }
  }
```

```
  else
  {
    if((year % 4) == 0)
    {
      if((year % 100) == 0)
      {
          if((year % 400) == 0)
          {
              leap_year = 1;
          }

          else
          {
              leap_year = 0;
          }
      }

      else
      {
          leap_year = 1;
      }
    }

    else
    {
        leap_year = 0;
    }

    if((leap_year) && (date > 29))
    {
         date = 1;
    }

    else if((!leap_year) && (date > 28))
    {
         date = 1;
    }
  }

  ms = 0;
  TMR3_setup(TMR3_sysclk, TMR3_clk_prescalar_12T, TMR3_no_clk_out);
  TMR3_load_counter_16(tmr_val);
  TMR3_start;
  _enable_TMR_3_interrupt;
}
```

Schematic



Explanation

For making a RTCC, we would need a display and a set of buttons for displaying and setting time respectively. The system clock in this example is set to 12MHz.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

P10_input_mode;
P11_input_mode;
P12_input_mode;
P13_input_mode;

P54_push_pull_mode;

LCD_init();
LCD_clear_home();
```

Timer 3 is used in example. It is driven with the 12MHz system clock and this clock is precaled by 12. Thus, the timer is ticking at 1MHz speed.

```
TMR3_setup(TMR3_sysclk, TMR3_clk_prescalar_12T, TMR3_no_clk_out);
TMR3_load_counter_16(tmr_val);
TMR3_start;
_enable_TMR_3_interrupt;
_enable_global_interrupt;
```

$$Timer\ Frequency = \frac{System\ Clock}{Prescalar} = \frac{12\ MHz}{12} = 1\ MHz$$

$$Timer\ Tick = \frac{1}{Timer\ Frequency} = \frac{1}{1\ MHz} = 1\mu s$$

We want the timer to interrupt every 1000µs (1ms) and so the timer is loaded with a value of 64535. Timer 3 is same as Timer2 and so the same concepts are used.

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

$$= 65535 - \left(\frac{1000\mu s}{1\mu s}\right) = 64535$$

Lastly, the timer is started with its interrupt enabled.

In order to ensure accuracy of timing the following should be maintained as much as possible:

- Clock source must be as precise as possible. I recommend using a temperature-compensated crystal oscillator (TCXO).
- Precalar should be as less as possible. This is not possible here with STC8A8K64S4A12 though due to hardware limitations.
- Accuracy can be increased by synchronizing time with GPS at periodic interval. This can be achieved by adding extra codes and synchronizing with a GPS receiver or NTP server.

Since the timer is now ticking at 0.001ms rate and 1 second equals 1000ms, a variable called ***"ms"*** is incremented at every timer overflow interrupt. When this get larger than 999, one second is registered. In similar way, minute, hour, day, month and year are increased. The process is just like a software timer implementation. At the end of the interrupt, timer overflow flag is cleared.

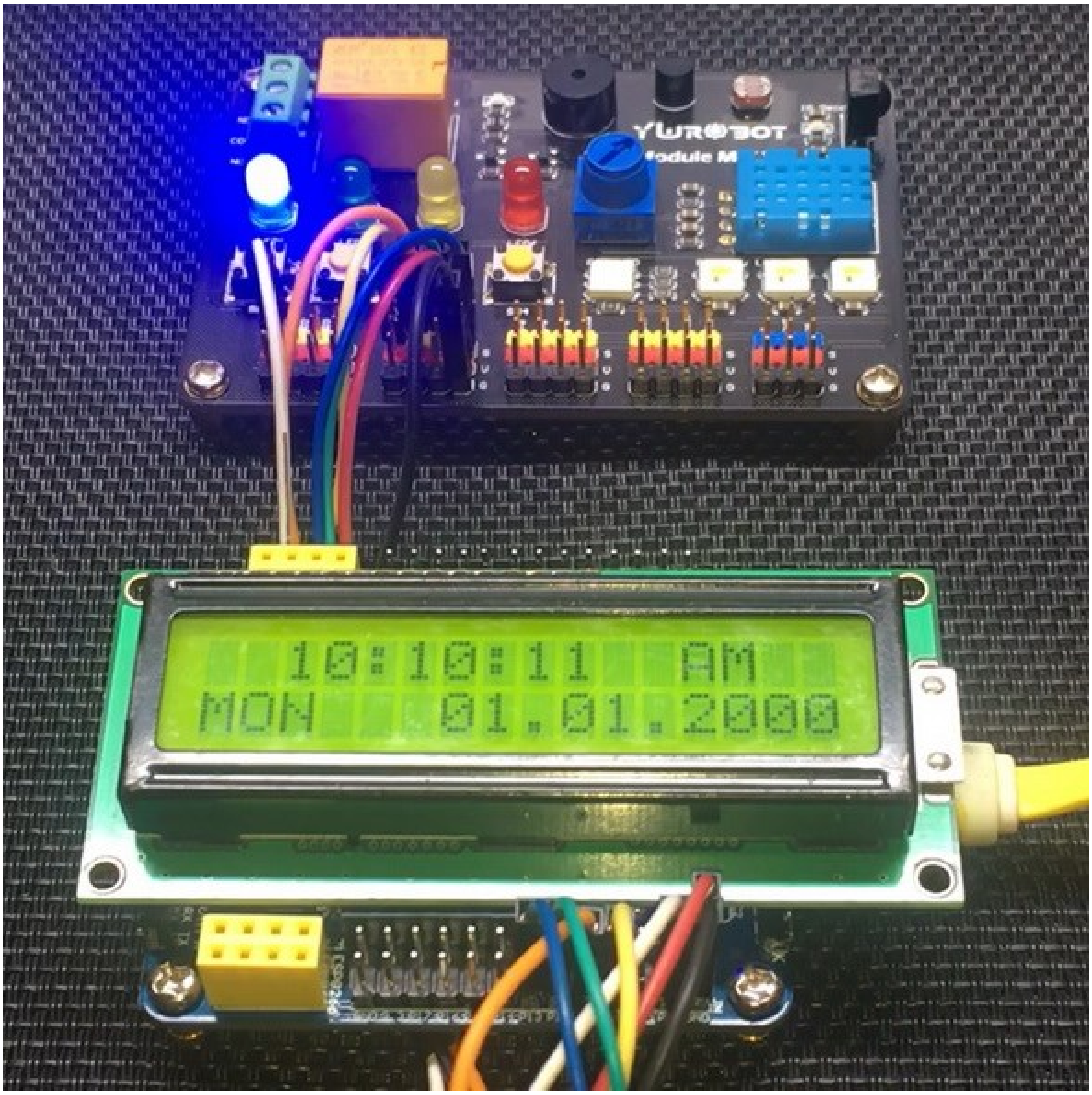
```
void TMR_3_ISR(void)
interrupt 19
{
  ms++;
  if(ms > 999)
  {
    s++;
    ms = 0;
    toggle ^= 0x01;
    P54_toggle;
  }

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

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

      if(hr > 12)
      {
          hr = 1;
      }

      if((hr == 12) && (min == 0) && (s == 0))
      {
        ampm ^= 1;

        if(ampm == AM)
        {
          date++;
          dow++;
        }

        if(dow > 7)
        {
          dow = 1;
        }

        if((month == 1) || (month == 3) || (month == 5) || (month == 7) || (month == 8) || (month == 10) \\
            || (month == 12))
        {
          if(date > 31)
          {
            date = 1;
            month++;
          }
        }

        else if((month == 4) || (month == 6) || (month == 9) || (month == 11))
        {
          if(date > 30)
          {
            date = 1;
            month++;
          }
        }

        else
        {
          if((year % 4) == 0)
          {
            if((year % 100) == 0)
            {
              if((year % 400) == 0)
              {
                leap_year = 1;
              }
```

```
            else
            {
              leap_year = 0;
            }
          }

          else
          {
            leap_year = 1;
          }
        }

        else
        {
            leap_year = 0;
        }

        if((leap_year) && (date > 29))
        {
           date = 1;
           month++;
        }

        else if((!leap_year) && (date > 28))
        {
           date = 1;
           month++;
        }
      }

      if(month > 12)
      {
        month = 1;
        year++;
      }
    }
  }
}

clear_TMR_3_overflow_flag;
}
```

In the main, either we can set time or read it.

```
if(get_buttons() == ESC)
{
    while(get_buttons() == ESC);
    setup_RTC();
}
else
{
    display_time();
}
```

Demo

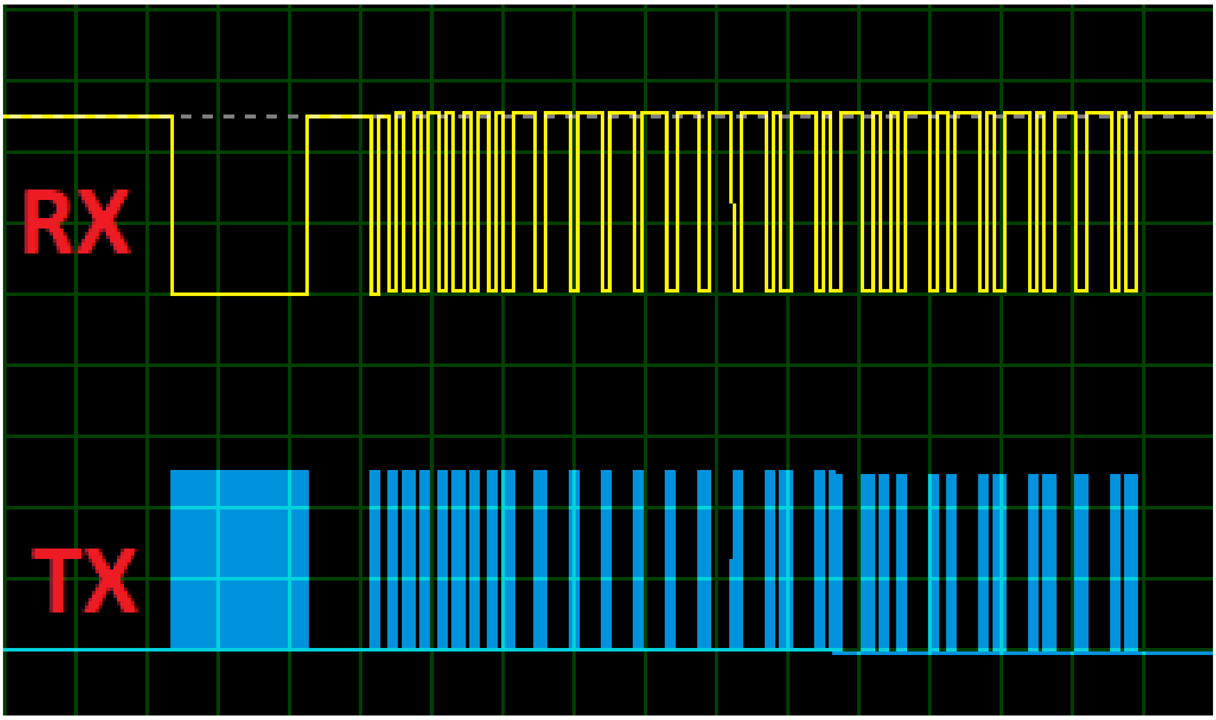


Demo video link: https://youtu.be/DwMn9KGtnKs.

NEC IR Remote Decoding with Timer 0 and External Interrupt

In many projects, we need to control devices remotely, for example a remote-controlled appliance and so it becomes necessary to incorporate some form of remote controller. Infrared (IR) remote controllers are cheap and are widely used as remote controllers in many devices.

Data to be sent from a remote IR transmitter to a receiver is transmitted by modulating it with a carrier wave having frequency between 30kHz to 40kHz. Modulation technique ensures safety from data corruption over long-distance transmissions. An IR receiver at the receiving end picks up and demodulates the sent out modulated data and outputs a stream of pulses. These pulses can vary in terms of pulse widths/timing/position/phase and these variable properties carry the data. Pulse properties are defined by a set of rules called protocol. Decoding the pulses as per protocol help in retrieving information. In most micros, there is no dedicated hardware for decoding IR remote protocols. A micro that needs to decode IR remote data also does not know when it will be receiving an IR burst. Thus, a combination of external interrupt and timer is needed for decoding IR data.



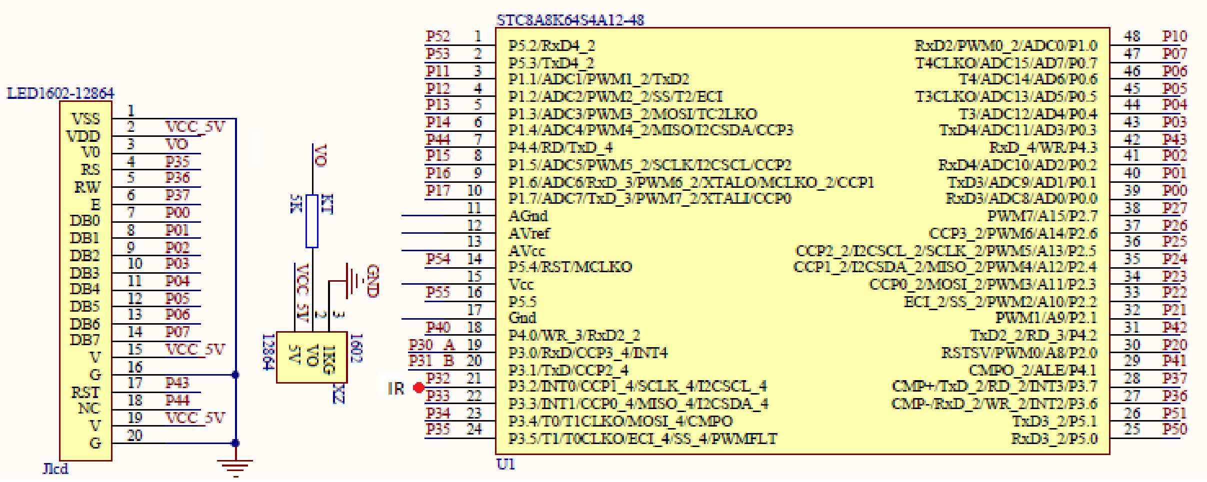
In this segment, we will see how we can use an external interrupt and a timer to easily decode a NEC IR remote. This same method can be applied to any IR remote controller. At this stage, I recommend studying NEC IR protocol from [here](). SB-Project's website is an excellent page for IR remote-related info.

Code

```
#include "STC8xxx.h"
#include "BSP.h"
#include "LCD.c"
#include "lcd_print.c"


#define sync_high     5850    //approx 1.3 * 4500us
#define sync_low      3150    //approx 0.7 * 4500us
#define one_high      2200    //approx 1.3 * (2250 - 562.5)us
#define one_low       1180    //approx 0.7 * (2250 - 562.5)us
#define zero_high     732     //approx 1.3 * (1125 - 562.5)us
#define zero_low      394     //approx 0.7 * (1125 - 562.5)us


unsigned char bits = 0x00;
unsigned char received = 0x00;

unsigned int frames[33];


void setup(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 EXT_0_ISR(void)
interrupt 0
{
  frames[bits] = TMR0_get_counter();
  bits++;
  TMR0_start;

  if(bits >= 33)
  {
    received = 1;
    _disable_global_interrupt;
    TMR0_stop;
  }

  TMR0_load_counter_16(0x0000);
}


void main(void)
{
  unsigned char addr = 0x00;
  unsigned char cmd = 0x00;

  setup();

  LCD_goto(1, 0);
  LCD_putstr("NEC IR Decoder");
  LCD_goto(0, 1);
  LCD_putstr("Adr      Cmd");

  while(1)
  {
     if(received != FALSE)
     {
       decode_NEC(&addr, &cmd);

       print_C(3, 1, addr);
       print_C(12, 1, cmd);

       delay_ms(100);

       erase_frames();
       _enable_global_interrupt;
     }
  };
}
```

```
void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_div_1, MCLK_SYSCLK_no_output);

  erase_frames();

  TMR0_setup(TMR0_16_bit_non_auto_reload, \
             TMR0_sysclk, \
             TMR0_clk_prescalar_12T, \
             TMR0_ext_gate, \
             TMR0_no_clk_out);

  EXT_0_priority_0;
  EXT_0_falling_edge_detection_only;
  _enable_EXT_0_interrupt;
  _enable_global_interrupt;

  LCD_init();
  LCD_clear_home();
}


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

  TMR0_load_counter_16(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);
}
```

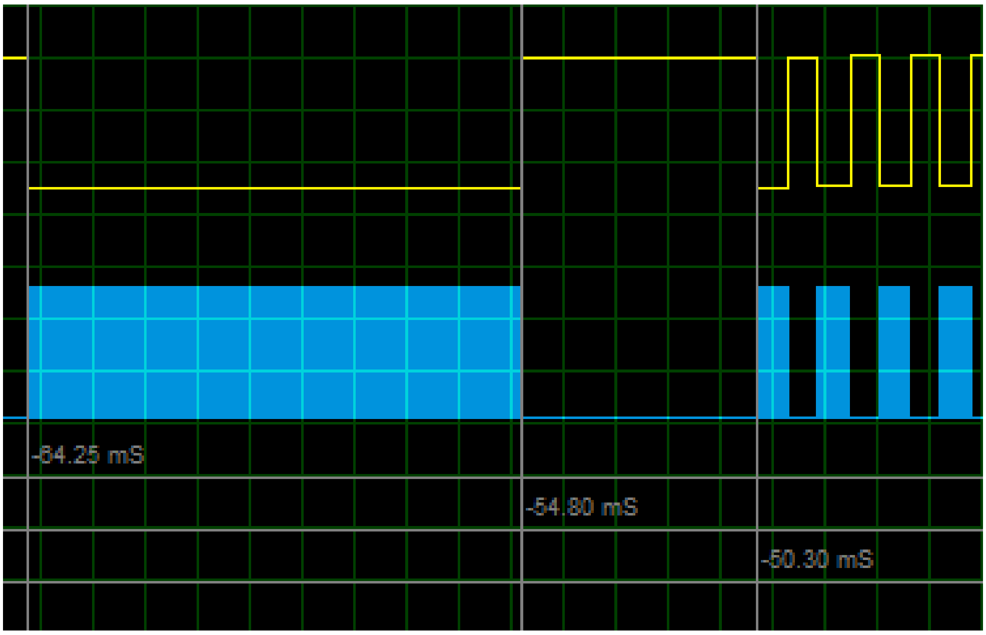
Schematic



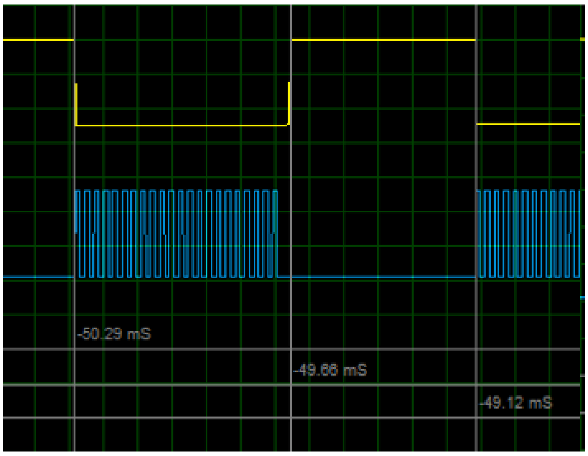
Explanation

A NEC transmission sends out a total of 33 pulses. The first one is a sync pulse and the rest 32 contain address and command info. These 32 address and command bits can be divided into 2 groups. The first 16 bits is one group that contain address and inverted address while the second group of 16-bits contain command and inverted command. The non-inverted signals can be used against the inverted ones for checking data integrity.

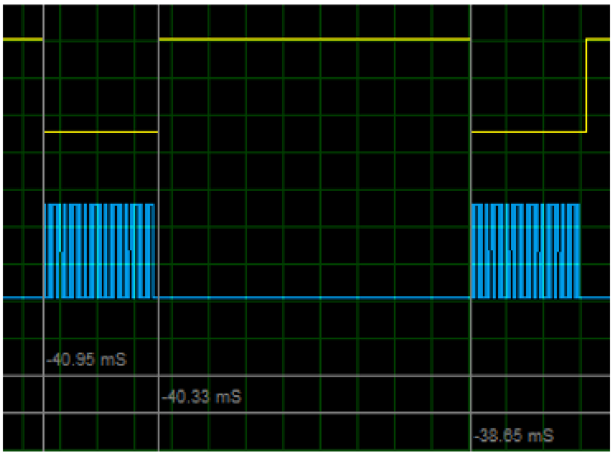
We already know that IR data is received as a steam of pulses. Pulses represent sync bit or ones and zeros. In case of NEC IR protocol, the pulses have variable widths. Sync bit is characterized by a pulse of 9ms high and 4.5ms low – the total pulse time is 13.5ms. Check the timing diagram shown below. Please note that in this timing diagram the blue pulses are from the transmitter and the yellow ones are those received by the receiver. Clearly the pulse streams are inverted. This inversion is done at the IR receiver side.



Logic one is represented by a pulse of 560µs high time and 2.25ms of low time – the total pulse time is 2.81ms. Likewise, logic zero is represented by a pulse of 560µs high time and 1.12ms of low time – the total pulse time is 1.68ms.



Logic 0



Logic 1

These timings can vary up to 30% of their ideal values due to a number of factors like medium, temperature, reflections, etc.

Now let's look at the coding. Firstly, the system clock is set to 12MHz.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_div_1, MCLK_SYSCLK_no_output);
```

External interrupt is set up to detect falling edges. This is a crucial part in detecting IR pulses because using this interrupt allows us to do other tasks and leave the MCU free for other tasks.

```
EXT_0_priority_0;
EXT_0_falling_edge_detection_only;
_enable_EXT_0_interrupt;
_enable_global_interrupt;
```

Unlike past timer examples, we will need an advanced timer here and so Timer 0 is used. This timer is clocked with the 12MHz system clock that has been divided by a prescalar of 12. Thus, the timer is operating at 1MHz speed, i.e., 1 timer tick = 1µs. We don't want any output from the timer and so this feature is disabled. The only different thing here that sets this setup apart from the other timer examples is the *"gate"* feature. *Gate* tells Timer 0 when and when not to run. There are two possible cases with gating - either the timer runs just like other timers when instructed to run via software or it runs when it is coded to run *and* INT0 pin is in logic high state. The latter feature is required in this example.

```
TMR0_setup(TMR0_16_bit_non_auto_reload, \
           TMR0_sysclk, \
           TMR0_clk_prescalar_12T, \
           TMR0_ext_gate, \
           TMR0_no_clk_out);
```

Note timer interrupt is not needed.

As soon as there is an external interrupt, the timer immediately stops because of the falling edge that triggered the interrupt and the subsequent low level at INT0 pin. Timer's count is captured and stored in an array.

```
void EXT_0_ISR(void)
interrupt 0
{
  frames[bits] = TMR0_get_counter();
  bits++;
  TMR0_start;

  if(bits >= 33)
1`          {
    received = 1;
    _disable_global_interrupt;
    TMR0_stop;
  }

  TMR0_load_counter_16(0x0000);
}
```

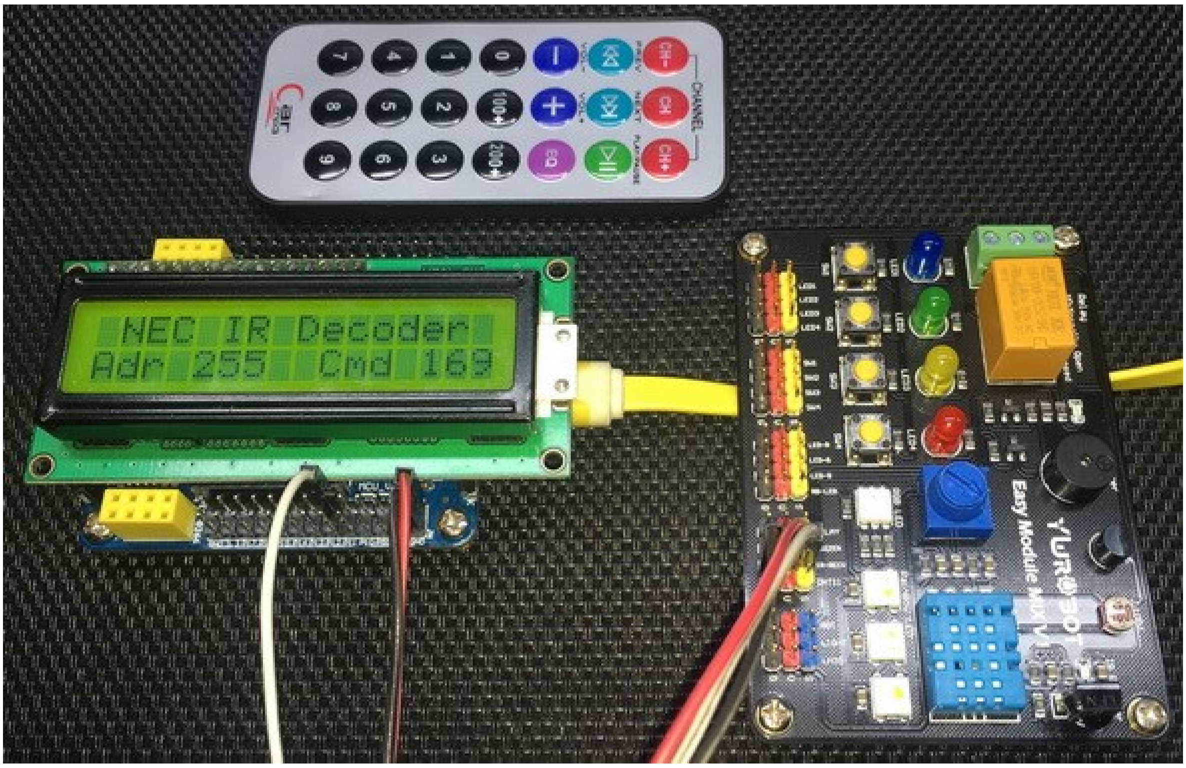
The captured counts are compared against maximum and minimum interval limits because as stated, pulse widths may slightly vary from their ideal figures. From captured counts, we can sort pulses and decode pulses. This decoded information can be used to deduce command and address.

```
#define sync_high      5850    //approx 1.3 * 4500us
#define sync_low       3150    //approx 0.7 * 4500us
#define one_high       2200    //approx 1.3 * (2250 - 562.5)us
#define one_low        1180    //approx 0.7 * (2250 - 562.5)us
#define zero_high      732     //approx 1.3 * (1125 - 562.5)us
#define zero_low       394     //approx 0.7 * (1125 - 562.5)us

. . . .

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, address and command data are displayed on an LCD after receiving and decoding a NEC frame. After that the microcontroller is readied to receive new NEC frame.

```
if(received != FALSE)
{
  decode_NEC(&addr, &cmd);

  print_C(3, 1, addr);
  print_C(12, 1, cmd);
  delay_ms(100);
  erase_frames();
  _enable_global_interrupt;
}
```

Demo



Demo video link: https://youtu.be/9hWbi4iQIjo.

Using Single Channel Enhanced PWM to Drive a Servo Motor

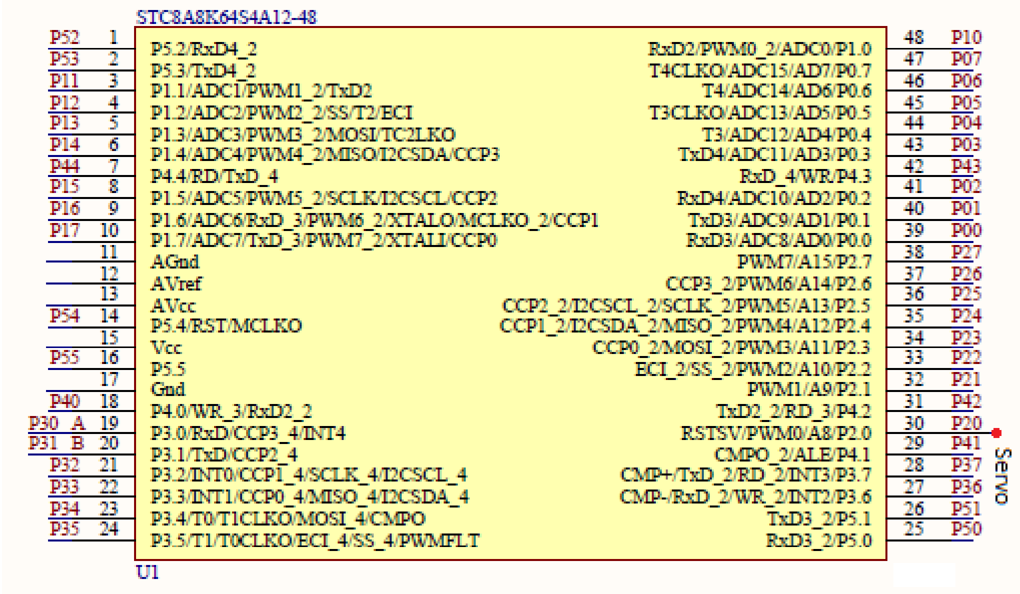
Servo motors have varieties of applications both in industrial automation and robotics arena. They are simple and can be digitally controlled with only one wire and over long distances. However, this one wire digital control mechanism is somewhat unique and sometimes stressful because trains of low duty cycle pulses are needed to be sent constantly in order to lock a particular angular position.

To resolve this issue, we can use GPIOs coupled with software delays or timers or PWMs module to control one or several servo motors. One key advantage of using PWM modules over GPIOs is the accuracy in timing since PWM modules use timer-counters for timing instead of software delays. In this section, we will see how we can use a PWM module to drive an ordinary servo motor. STC8A8K64S4A12's enhanced PWM module is used as the PWM generator here.

Code

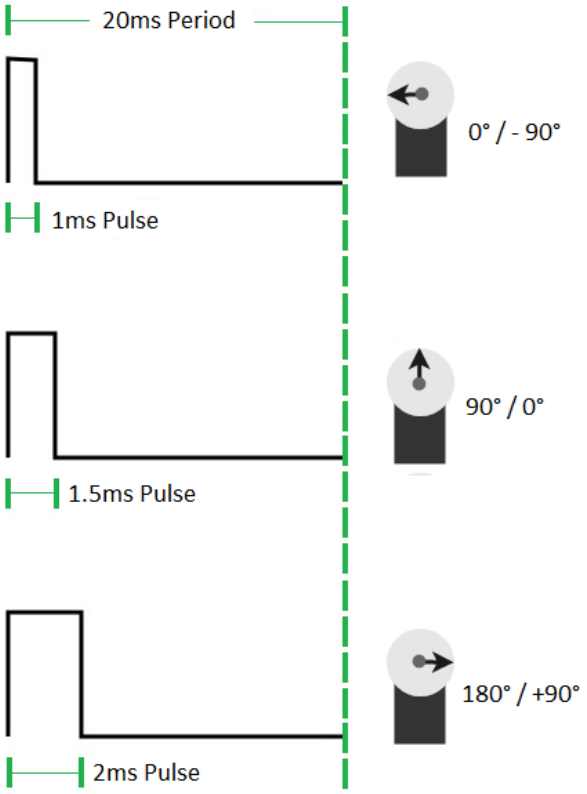
```
#include "STC8xxx.h"
#include "BSP.h"


#define servo_min_duty      800
#define servo_max_duty     2200
#define step_change           5


void setup(void);
void PWM_idle(void);


void main(void)
{
  unsigned int i = 0x0000;

  setup();

  while(1)
  {
    for(i = servo_min_duty; i < servo_max_duty; i += step_change)
    {
      PWM_set_PWM0_T1(i);
      delay_ms(4);
    }
    for(i = servo_max_duty; i > servo_min_duty; i -= step_change)
    {
      PWM_set_PWM0_T1(i);
      delay_ms(4);
    }
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_12);
  PWM_set_counter(20000);

  PWM0_setup(PWM_pin_is_PWM_output, \
             PWM_init_lvl_low, \
             PWM_0_pin_P20, \
             PWM_level_normal);

  PWM_set_PWM0_T1(1000);
  PWM_set_PWM0_T2(0);

  PWM_start_counter;
}
```

Schematic



Explanation

Servo motors typically have timing diagrams as shown below:



Each pulse has a period of 20ms but the duty cycle/pulse high time is varied from 5 - 10% to rotate from one direction to the other. This clearly demonstrates why timing is very important for servo motors.

The demo program here is coded to run at 12MHz.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);
```

PWM0 hardware is configured to use P2.0 pin as servo control pin. The PWM hardware clock is set to 1MHz by using system clock as clock source and dividing it with a prescalar of 12. This is, however, not the PWM output frequency because the PWM counter is loaded with count value of 20000. The resultant PWM output frequency is 50Hz as shown in the following equation:

$$PWM\ Frequency = \frac{PWM\ Input\ Clock\ Frequency}{(PWM\ Counter\ Value\ \times\ PWM\ Clock\ Prescalar)} = \frac{12\ MHz}{(20000\ \times\ 12)} = 50\ Hz$$

and

$$PWM\ Period = \frac{1}{PWM\ Frequency} = \frac{1}{50Hz} = 20ms\ =\ 20000\mu s$$

and

$$PWM\ Duty\ Cycle = \frac{|PWM\ T2\ Value\ -\ PWM\ T1\ Value|}{PWM\ Counter\ Value} \times 100\% = \frac{|0 - 1000|}{20000} \times 100\% = 5\%$$

The following codes does all mentioned so far. However, PWM setup is not just about timing only. PWM hardware also need an output. We have to mention which PWM we would be using and what would be its level and polarity.

```
PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_12);

PWM_set_counter(20000);

PWM0_setup(PWM_pin_is_PWM_output, \
           PWM_init_lvl_low, \
           PWM_0_pin_P20, \
           PWM_level_normal);

PWM_set_PWM0_T1(1000);
PWM_set_PWM0_T2(0);

PWM_start_counter;
```

After setting all these, we just have to start the PWM hardware by starting the PWM counter.

Inside the main loop, the duty cycle of the PWM is slightly varied in steps. This results in smooth servo motion. Note that the maximum and minimum duty cycles are not 2000 and 1000 respectively as they ideally should have been. This is so because these values are typical ones and not always practical. Thus, the maximum and minimum duty cycles are set to 2200 and 800 respectively.

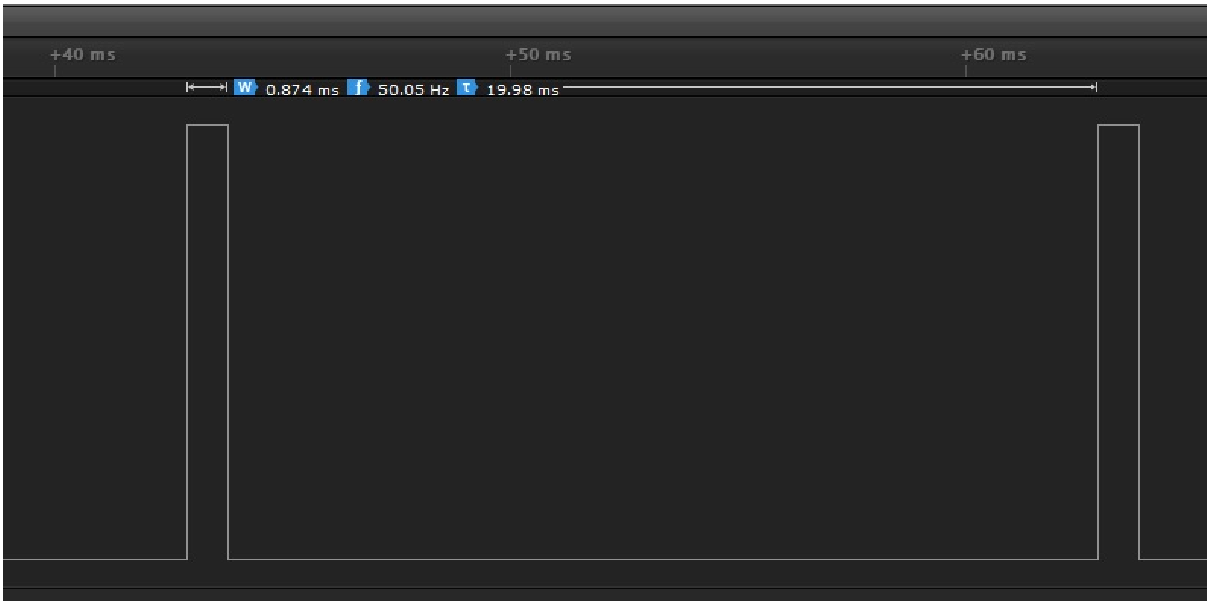
$$PWM\ Duty\ Cycle = \frac{|PWM\ T2\ Value\ -\ PWM\ T1\ Value|}{PWM\ Counter\ Value} \times 100\% = \frac{|0 - 2200|}{20000} \times 100\% = 11\%$$

$$PWM\ Duty\ Cycle = \frac{|PWM\ T2\ Value\ -\ PWM\ T1\ Value|}{PWM\ Counter\ Value} \times 100\% = \frac{|0 - 800|}{20000} \times 100\% = 4\%$$

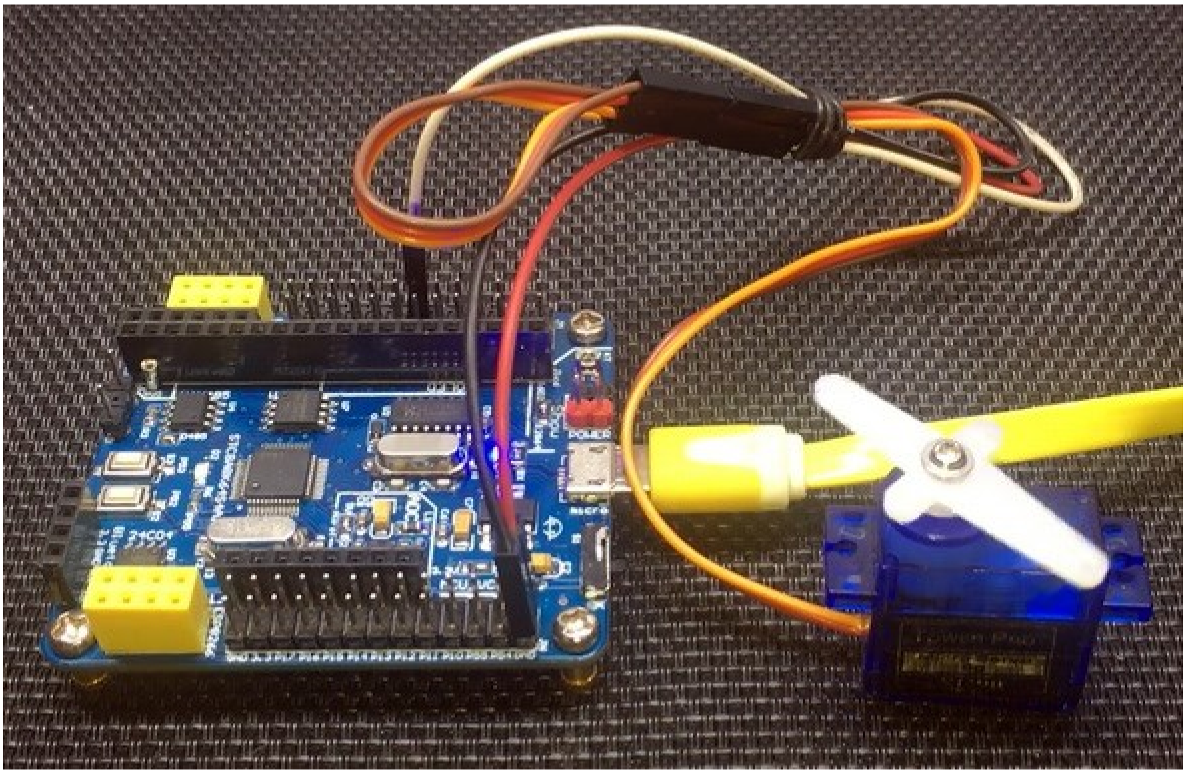
The 1% variation is needed because of non-ideality.

```
#define servo_min_duty     800
#define servo_max_duty    2200
#define step_change          5

....

for(i = servo_min_duty; i < servo_max_duty; i += step_change)
{
  PWM_set_PWM0_T1(i);
  delay_ms(4);
}

for(i = servo_max_duty; i > servo_min_duty; i -= step_change)
{
  PWM_set_PWM0_T1(i);
  delay_ms(4);
}
```

The logic analyser data shown below verifies that whatever stated so far has been achieved.



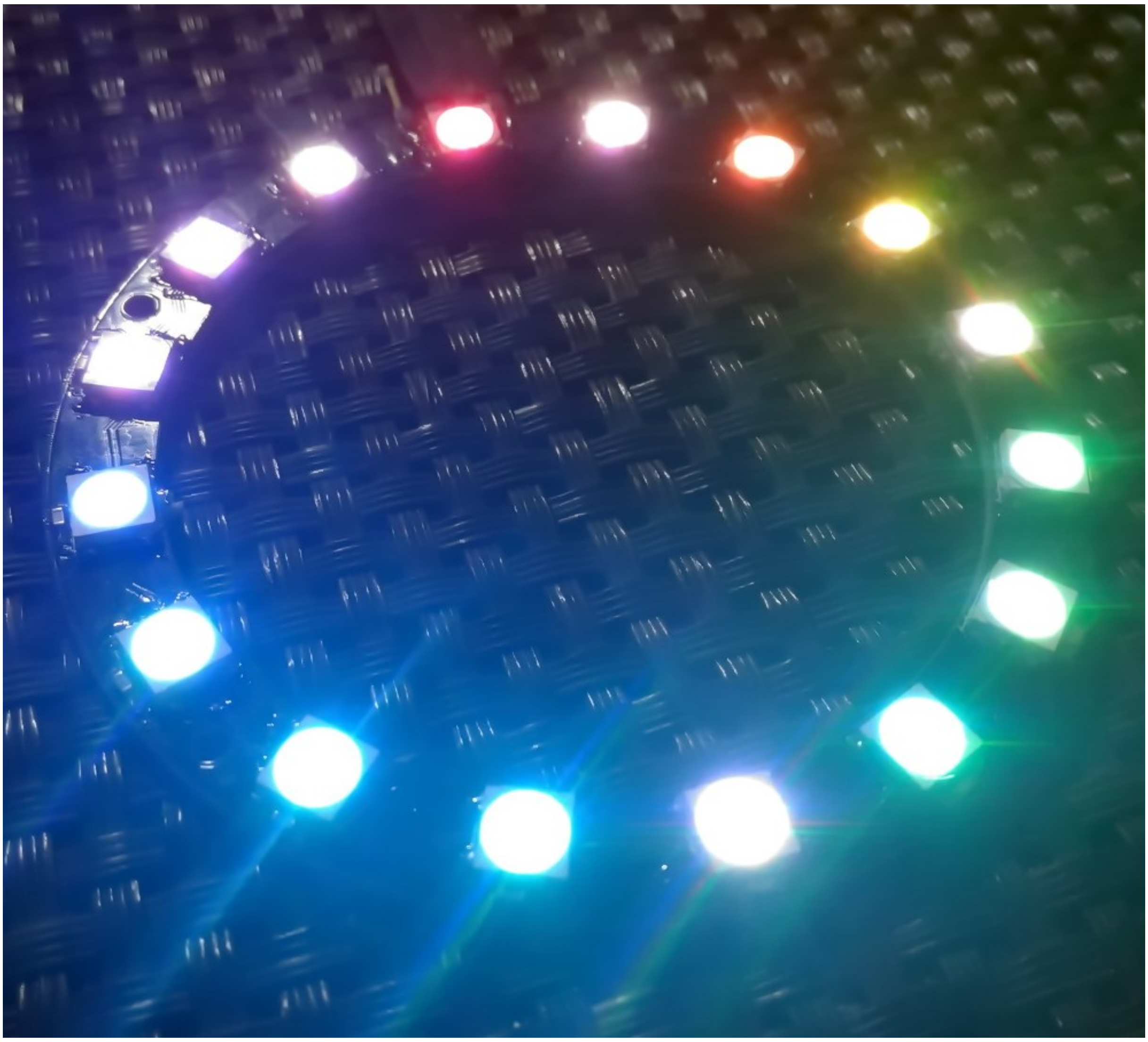
Demo



Demo video link: https://youtu.be/TdHbkxH7ZWs.

Using Multi Channel Enhanced PWM to Drive an RGB LED

Since we now have seen how to use single channel enhanced PWM, we must also see how to use multichannel PWM. Multi-channel PWM has many applications. For example, multi-channel PWM is used in SMPSs, inverters, motor controllers, LED drivers, etc.



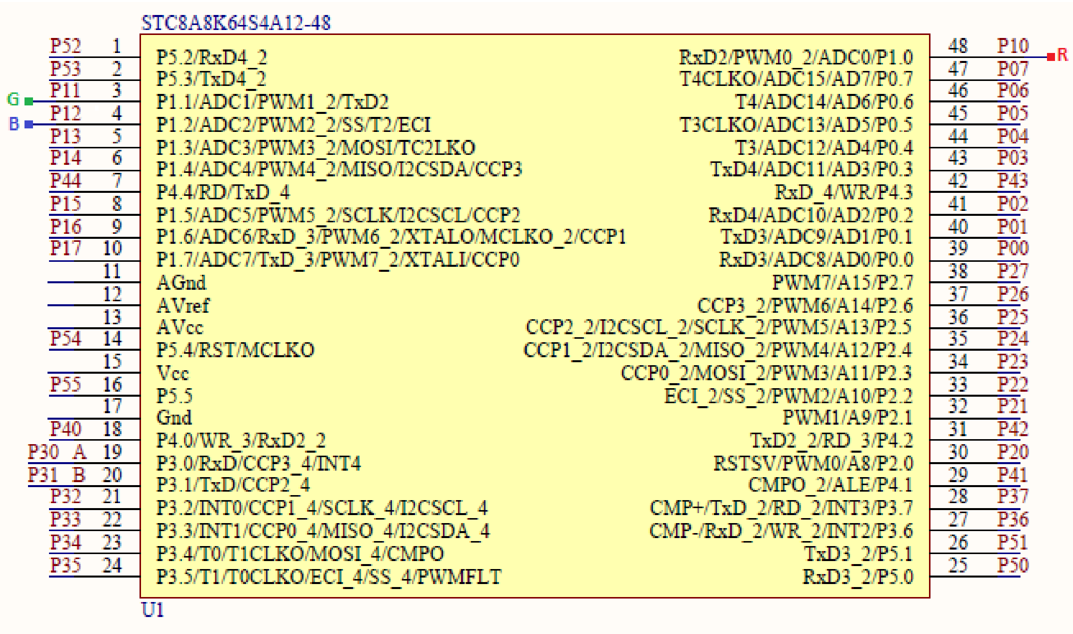
Code

```
#include "STC8xxx.h"
#include "BSP.h"


#define steps 33


void setup(void);
void PWM_idle(void);


void main(void)
{
```

```
  signed char i = 0x00;
  signed char j = 0x00;

  const unsigned int duty1_value[steps] = {0, 950, 1892, 2817, 3717, 4582, 5407, 6182, 6901, 7558, 8146,
                                            8660, 9096, 9450, 9718, 9898, 9988, 9995, 9897, 9717, 9449, 9095,
                                            8658, 8144, 7555, 6898, 6179, 5403, 4579, 3713, 2813, 1888 ,946};

  const unsigned int duty2_value[steps] = {8658, 8144, 7555, 6898, 6179, 5403, 4579, 3713, 2813, 1888 ,946,
                                            0, 950, 1892, 2817, 3717, 4582, 5407, 6182, 6901, 7558, 8146,
                                            8660, 9096, 9450, 9718, 9898, 9988, 9995, 9897, 9717, 9449, 9095};

  const unsigned int duty3_value[steps] = {8660, 9096, 9450, 9718, 9898, 9988, 9995, 9897, 9717, 9449, 9095,
                                            8658, 8144, 7555, 6898, 6179, 5403, 4579, 3713, 2813, 1888 ,946,
                                            0, 950, 1892, 2817, 3717, 4582, 5407, 6182, 6901, 7558, 8146};

  setup();

  while(1)
  {
    for(j = 0; j < 6; j++)
    {
      for(i = 0; i < steps; i++)
      {
          PWM_set_PWM0_T2(duty1_value[i]);
          PWM_set_PWM1_T2(duty2_value[i]);
          PWM_set_PWM2_T2(duty3_value[i]);
          delay_ms(60);
      }
    }

    PWM_idle();

    for(j = 0; j < 6; j++)
    {
      for(i = 0; i < steps; i++)
      {
          PWM_set_PWM0_T2(duty1_value[i]);
          PWM_set_PWM1_T2(duty1_value[i]);
          PWM_set_PWM2_T2(duty1_value[i]);
          delay_ms(60);
      }
    }

    PWM_idle();
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_2);

  PWM_set_counter(10000);

  PWM0_setup(PWM_pin_is_PWM_output, \
             PWM_init_lvl_low, \
             PWM_0_pin_P10, \
             PWM_level_normal);

  PWM1_setup(PWM_pin_is_PWM_output, \
             PWM_init_lvl_low, \
             PWM_1_pin_P11, \
             PWM_level_normal);

  PWM2_setup(PWM_pin_is_PWM_output, \
             PWM_init_lvl_low, \
             PWM_2_pin_P12, \
             PWM_level_normal);

  PWM_set_PWM0_T1(0);
  PWM_set_PWM0_T2(0);

  PWM_set_PWM1_T1(0);
  PWM_set_PWM1_T2(0);

  PWM_set_PWM2_T1(0);
  PWM_set_PWM2_T2(0);
```

```
    PWM_start_counter;
}


void PWM_idle(void)
{
    PWM0_hold_level(PWM_HLD_L_low);
    PWM1_hold_level(PWM_HLD_L_low);
    PWM2_hold_level(PWM_HLD_L_low);
    delay_ms(100);
    PWM0_hold_level(PWM_level_normal);
    PWM1_hold_level(PWM_level_normal);
    PWM2_hold_level(PWM_level_normal);
    delay_ms(100);
}
```

Schematic



Explanation

In this demo, a RGB LED is lit with multi-channel PWM. This results in smooth color shifting of the LED. The concepts are same as that of single channel PWM example.

The system clock is set to 12MHz.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);
```

The PWM frequency is set to 600 Hz.

$$PWM\ Frequency = \frac{PWM\ Input\ Clock\ Frequency}{(PWM\ Counter\ Value\ \times\ PWM\ Clock\ Prescalar)} = \frac{12\ MHz}{(10000\ \times\ 2)} =\ 600\ Hz$$

Since we are using a RGB LED, we need three PWM channels and so their GPIOs are also set. Lastly, the initial duty cycle is set 0 and the PWM counter is started.

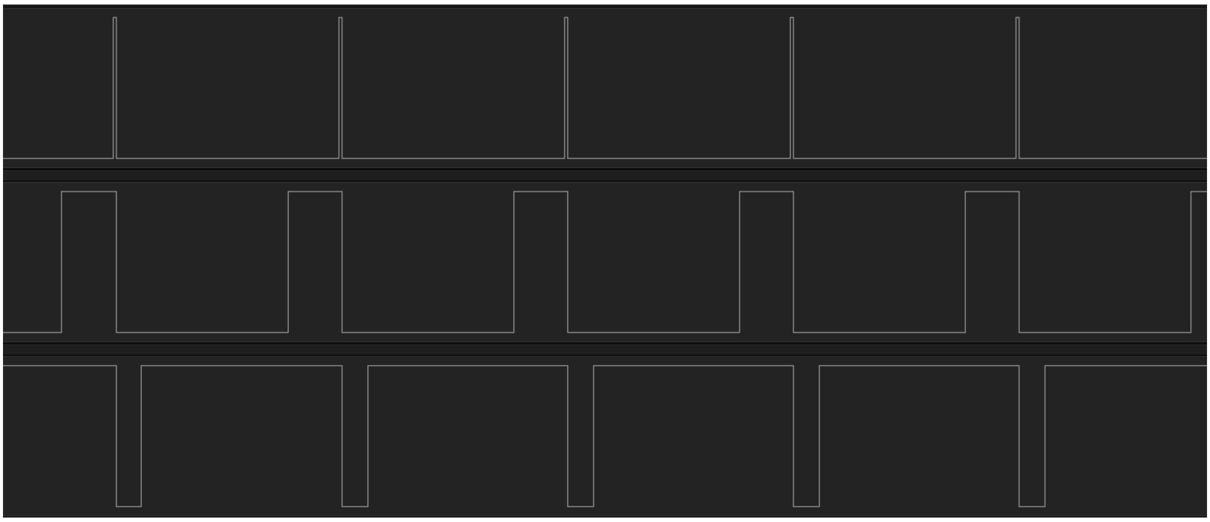
```
PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_2);

PWM_set_counter(10000);

PWM0_setup(PWM_pin_is_PWM_output, \
           PWM_init_lvl_low, \
           PWM_0_pin_P10, \
           PWM_level_normal);

PWM1_setup(PWM_pin_is_PWM_output, \
           PWM_init_lvl_low, \
           PWM_1_pin_P11, \
           PWM_level_normal);

PWM2_setup(PWM_pin_is_PWM_output, \
           PWM_init_lvl_low, \
           PWM_2_pin_P12, \
           PWM_level_normal);

PWM_set_PWM0_T1(0);
PWM_set_PWM0_T2(0);
PWM_set_PWM1_T1(0);
PWM_set_PWM1_T2(0);
PWM_set_PWM2_T1(0);
PWM_set_PWM2_T2(0);

PWM_start_counter;
```

To achieve smooth fading the duty cycle of each LED is mapped using a sine table and the duty cycles are varied according to the formula as have already seen.

$$PWM\ Duty\ Cycle = \frac{|PWM\ T2\ Value\ -\ PWM\ T1\ Value|}{PWM\ Counter\ Value} \times 100\%$$

However, there are phase shifts in all of these tables, meaning that the three PWM channels do not have the same duty cycle at the same time.

```
const unsigned int duty1_value[steps] = {0, 950, 1892, 2817, 3717, 4582, 5407, 6182, 6901, 7558, 8146,
                                         8660, 9096, 9450, 9718, 9898, 9988, 9995, 9897, 9717, 9449, 9095,
                                         8658, 8144, 7555, 6898, 6179, 5403, 4579, 3713, 2813, 1888 ,946};

const unsigned int duty2_value[steps] = {8658, 8144, 7555, 6898, 6179, 5403, 4579, 3713, 2813, 1888 ,946,
                                         0, 950, 1892, 2817, 3717, 4582, 5407, 6182, 6901, 7558, 8146,
                                         8660, 9096, 9450, 9718, 9898, 9988, 9995, 9897, 9717, 9449, 9095};

const unsigned int duty3_value[steps] = {8660, 9096, 9450, 9718, 9898, 9988, 9995, 9897, 9717, 9449, 9095,
                                         8658, 8144, 7555, 6898, 6179, 5403, 4579, 3713, 2813, 1888 ,946,
                                         0, 950, 1892, 2817, 3717, 4582, 5407, 6182, 6901, 7558, 8146};
```



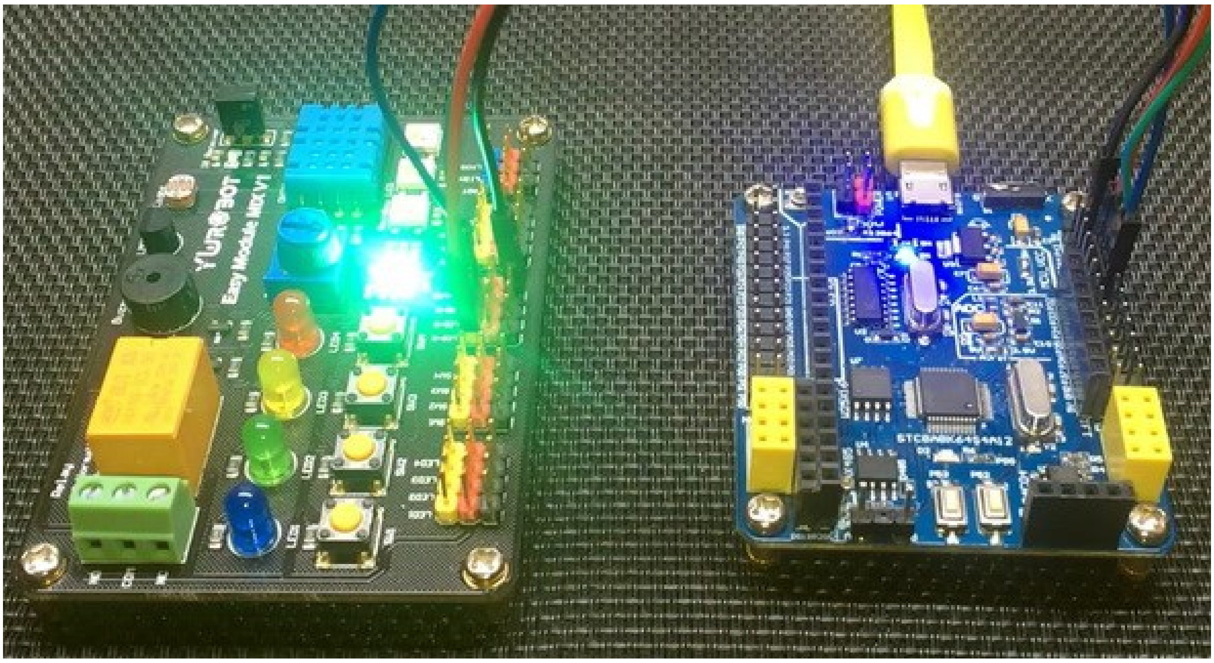
In the main, the PWM channels are driven as per their respective sine tables.

```
for(j = 0; j < 6; j++)
{
  for(i = 0; i < steps; i++)
  {
      PWM_set_PWM0_T2(duty1_value[i]);
      PWM_set_PWM1_T2(duty2_value[i]);
      PWM_set_PWM2_T2(duty3_value[i]);
      delay_ms(60);
  }
}

PWM_idle();

for(j = 0; j < 6; j++)
{
  for(i = 0; i < steps; i++)
  {
     PWM_set_PWM0_T2(duty1_value[i]);
     PWM_set_PWM1_T2(duty1_value[i]);
     PWM_set_PWM2_T2(duty1_value[i]);
     delay_ms(60);
  }
}

PWM_idle();
```

Note there is new function in this example unlike the last one. The ***PWM_idle*** function ensures that PWM channels are reset once it is called.

```
void PWM_idle(void)
{
  PWM0_hold_level(PWM_HLD_L_low);
  PWM1_hold_level(PWM_HLD_L_low);
  PWM2_hold_level(PWM_HLD_L_low);
  delay_ms(100);
  PWM0_hold_level(PWM_level_normal);
  PWM1_hold_level(PWM_level_normal);
  PWM2_hold_level(PWM_level_normal);
  delay_ms(100);
}
```

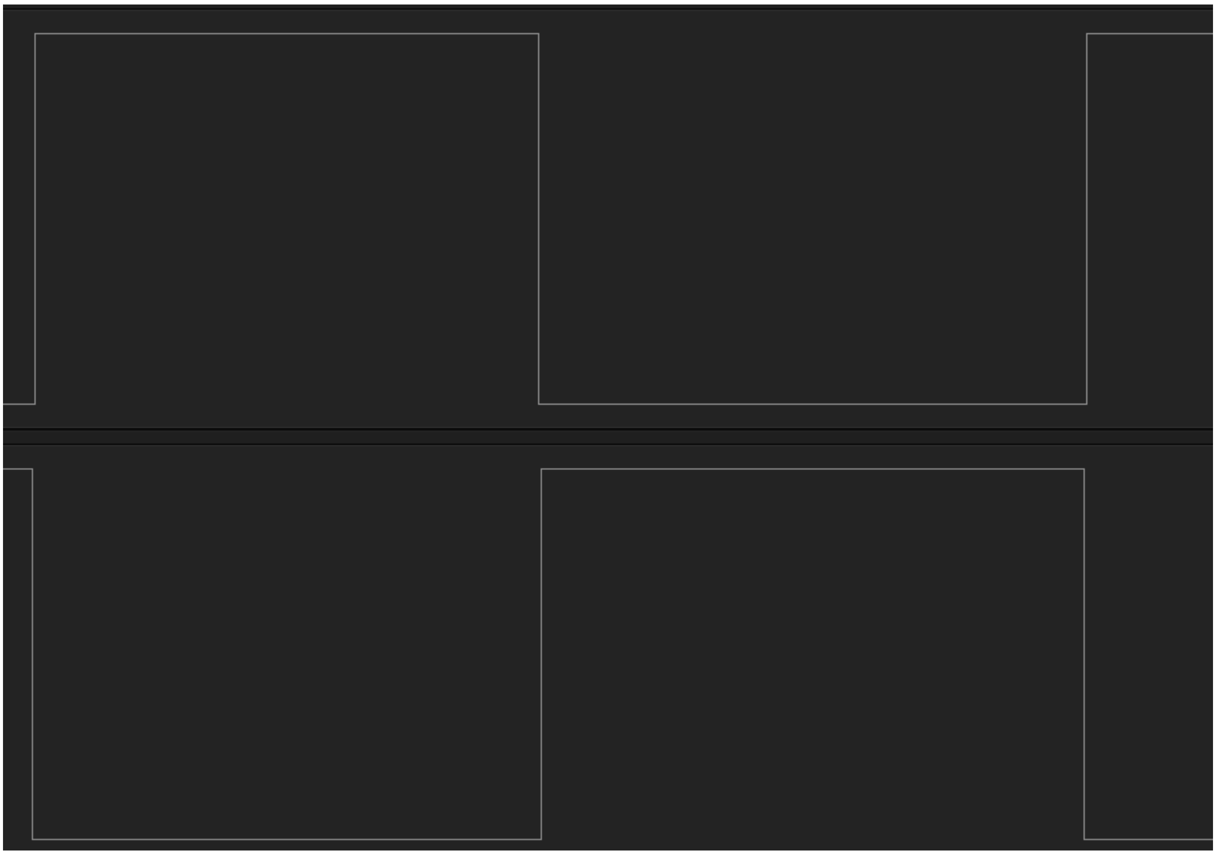
Demo



Demo video link: https://youtu.be/4bA-ISSlyrQ.

Controlling a DC Motor with Complementary PWM

Complementary PWM is needed in many power system applications like three phase motor control, sine wave inverters, BLDC motor control, etc. STC8A8K64S4A12 fortunately adds features to generate complementary PWMs for the applications aforementioned.

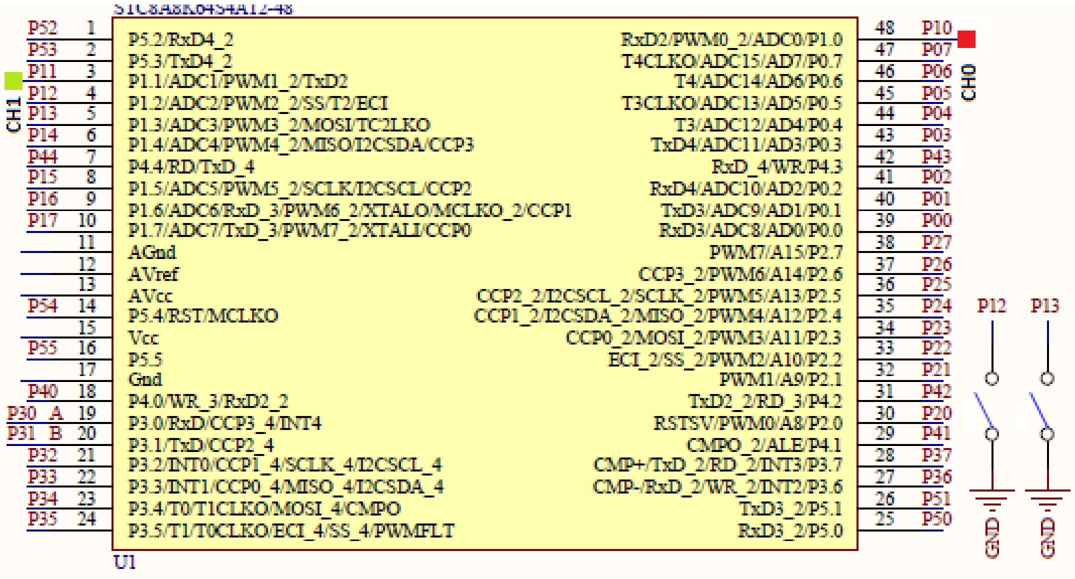


Code

```
#include "STC8xxx.h"
#include "BSP.h"


#define dead_time_cnt               1
#define pwm_max_cnt                 400

#define PB_1                        !P12_get_input
#define PB_2                        !P13_get_input


void setup(void);
void set_PWM_duty(signed int value);


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

  setup();

  while(1)
  {
      if(PB_1)
      {
          duty += 10;
```

```
          delay_ms(100);

          if(duty >= pwm_max_cnt)
          {
            duty = pwm_max_cnt;
          }
      }
      if(PB_2)
      {
          duty -= 10;
          delay_ms(100);

          if(duty <= 0)
          {
            duty = 0;
          }
      }
      if(PB_1 || PB_2)
      {
        set_PWM_duty(duty);
      }
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  P12_input_mode;
  P13_input_mode;

  PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_1);

  PWM_set_counter(pwm_max_cnt);

  PWM0_setup(PWM_pin_is_PWM_output, \
             PWM_init_lvl_low, \
             PWM_0_pin_P10, \
             PWM_level_normal);

  PWM1_setup(PWM_pin_is_PWM_output, \
             PWM_init_lvl_low, \
             PWM_1_pin_P11, \
             PWM_level_normal);

  PWM_start_counter;
}


void set_PWM_duty(signed int value)
{
  PWM_set_PWM0_T1(value);
  PWM_set_PWM0_T2(0);

  PWM_set_PWM1_T1(pwm_max_cnt - dead_time_cnt);
  PWM_set_PWM1_T2(value + dead_time_cnt);
}
```

Schematic



Explanation

Since complementary PWM is a requirement for proper motor drive, the demo here is a rudimentary DC motor drive example with STC's complementary PWM and a L293 motor driver. The motor's speed is governed by PWM duty cycle alternation. Pressing buttons assigned with pins P1.2 and P1.3 changes motor speed.

The system clock is set to 12MHz.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);
```

The same clock is used by the enhanced PWM module. Most of the stuffs in the setup are like the previous PWM examples.

```
#define dead_time_cnt                1
#define pwm_max_cnt                  400

....

PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_1);

PWM_set_counter(pwm_max_cnt);

PWM0_setup(PWM_pin_is_PWM_output, \
           PWM_init_lvl_low, \
           PWM_0_pin_P10, \
           PWM_level_normal);

PWM1_setup(PWM_pin_is_PWM_output, \
           PWM_init_lvl_low, \
           PWM_1_pin_P11, \
           PWM_level_normal);

PWM_start_counter;
```

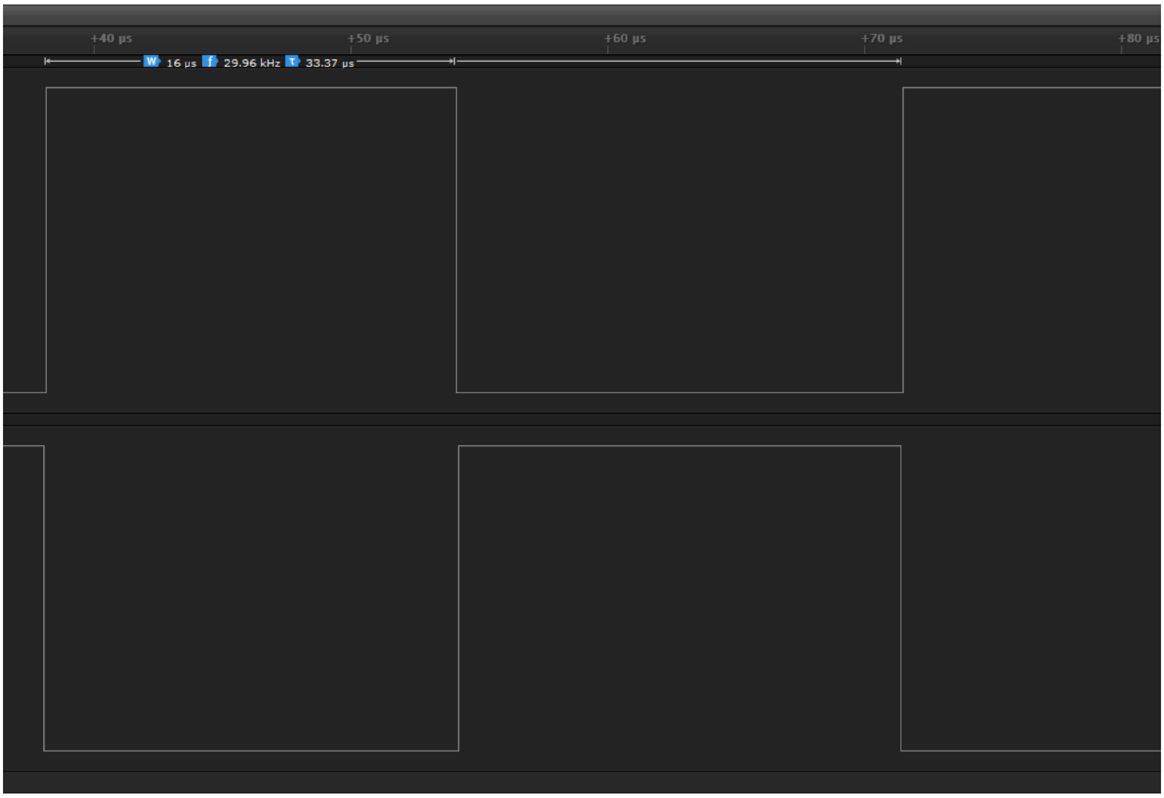
The math behind PWM generation is same as the ones we have already seen:

$$PWM\ Frequency = \frac{PWM\ Input\ Clock\ Frequency}{(PWM\ Counter\ Value\ \times\ PWM\ Clock\ Prescalar)} = \frac{12\ MHz}{(400\ \times\ 1)} = 30\ kHz$$

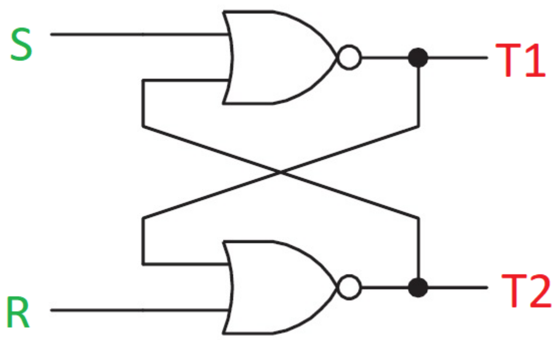
and

$$PWM\ Period = \frac{1}{PWM\ Frequency} = \frac{1}{30\ kHz} = 33.33\ \mu s$$

The logic analyser data proves that the above figures are correct:

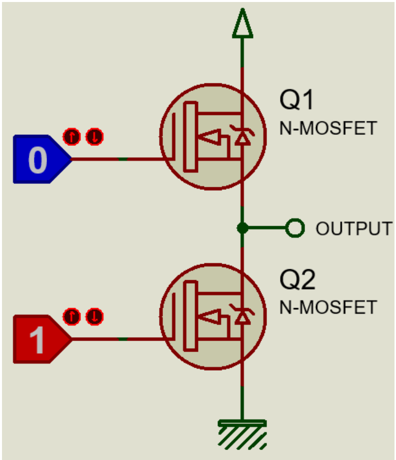


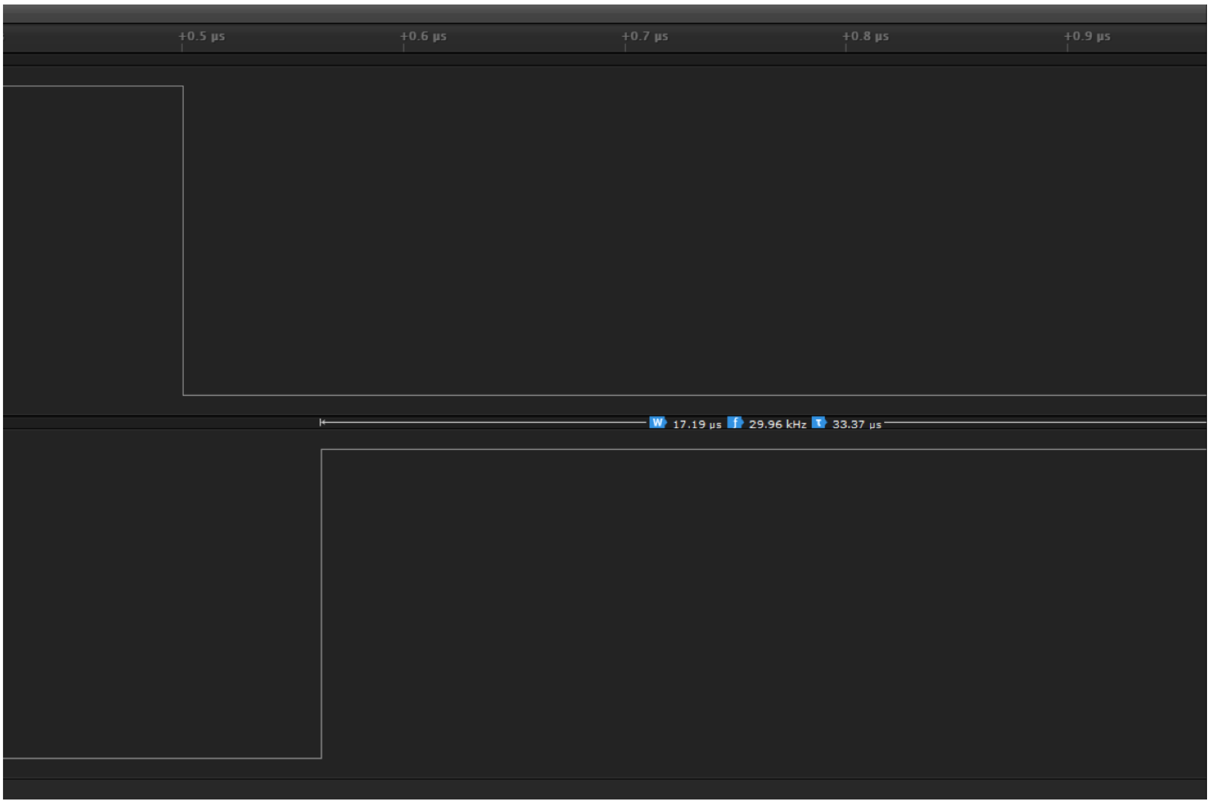
Now imagine a SR flip-flop with outputs ***T1*** and ***T2***. So far, we have seen that varying T1 and T2 alter PWM duty cycle. Thus, ***S*** and ***R*** can be imagined as logic transition counts. When the PWM counter reaches up to these values, high-to-low or low-to-high transition occurs in the PWM waveform.



To generate complementary PWM, we would need two PWM channels and so in this example PWM channels 0 and 1 are selected. The word “*complementary*” suggests that two things are opposite of the other and so in such PWM technique, there are two PWMs that have opposite logic polarities. This is the simplistic presentation of complementary PWM.

In ideal terms, if one PWM channel is running with 40% duty cycle, the other should be running at 60% (-40%) duty cycle. However, doing so practically would lead to some issues because when one PWM is going from high-to-low transition, the other is doing just the opposite and during these transition times at some point both PWMs are at same logic level. If these PWMs are fed to external devices like transistors or MOSFETs as shown below then quite possibly during the transition times both MOSFETs would be momentarily turn on, leading to temporary short-circuit and unnecessary overheating. The temporal short circuit may even cause the external devices to get damaged or cause momentary power shortages. To avoid this phenomenon, we have to apply dead-time technique to ensure that the transitions occur separately with some minute delay.





Now let's see how the PWM duty cycle is ensured while maintaining the needs so far discussed. The following code snippet is responsible for maintaining PWM duty cycles in complementary format. It may look confusing unlike the previous examples.

```
#define dead_time_cnt                   1
#define pwm_max_cnt                     400

....

void set_PWM_duty(signed int value)
{
  PWM_set_PWM0_T1(value);
  PWM_set_PWM0_T2(0);

  PWM_set_PWM1_T1(pwm_max_cnt - dead_time_cnt);
  PWM_set_PWM1_T2(value + dead_time_cnt);
}
```

The confusion will vanish after carefully looking at the math below. Imagine that we want to set the duty cycle to 75%.

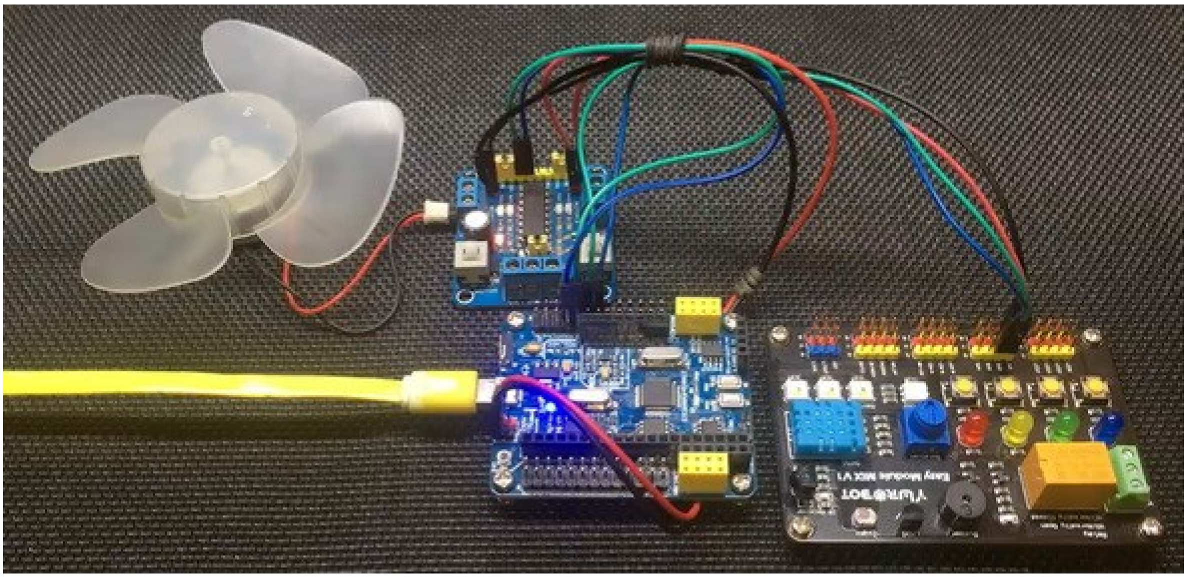
$$PWM0\ Duty = \frac{|PWM0\ T2 - PWM0\ T1|}{PWM\ Counter\ Value} \times 100\% = \frac{|300 - 0|}{400} \times 100\% = 75\%$$

and

$$PWM1\ Duty = \frac{|PWM1\ T2 - PWM1\ T1|}{PWM\ Counter\ Value} \times 100\% = \frac{|(400 - 1) - (300 + 1)|}{400} \times 100\% = 24.5\%$$

As the math shows due to the application of dead-time, PWM1's duty cycle is slightly less than the ideal 25% mark. In this example, 1 count of dead time is equivalent to 0.25% duty cycle and since there are two such counts the total duty cycle is reduced by 0.5%.

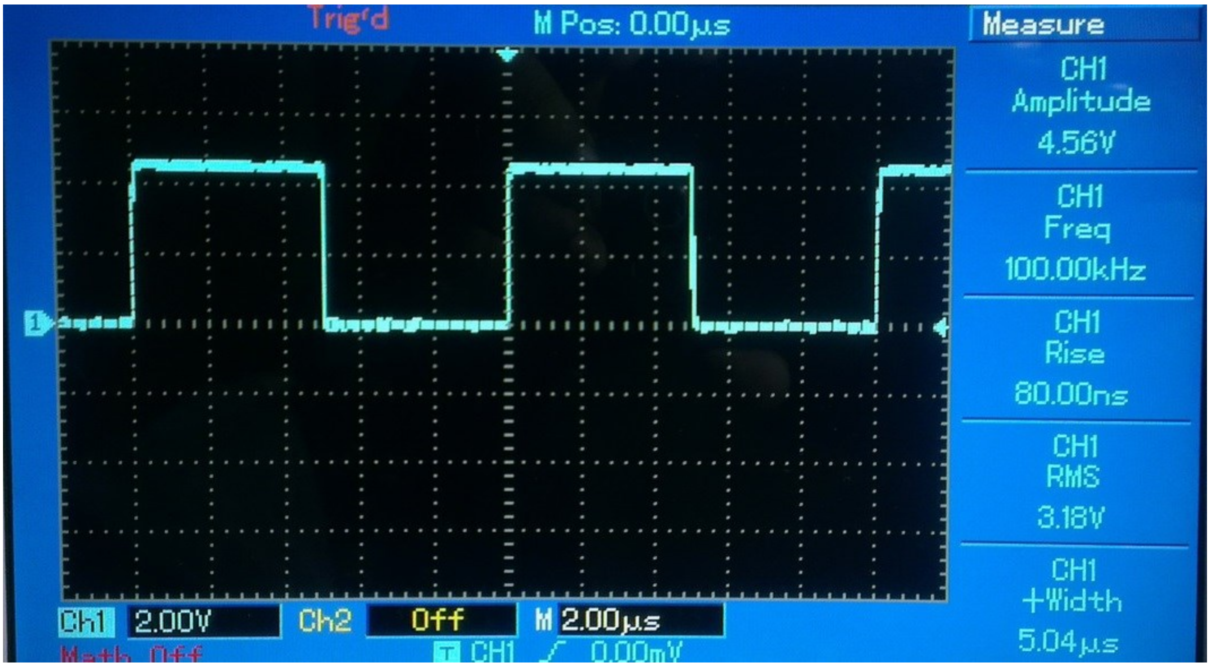
Demo



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

Using PCA to Measure Frequency

Frequency measurement is often a challenging task as it requires good understanding of timers and input capture modules. Frequency measurement becomes very important while designing power electronics devices like generator controllers, inverters, power line measurement meters, etc. In STC8A8K64S4A12, there is no dedicated input capture hardware module but there is a PCA module that can be used to replenish this absence of a dedicated input capture peripheral.



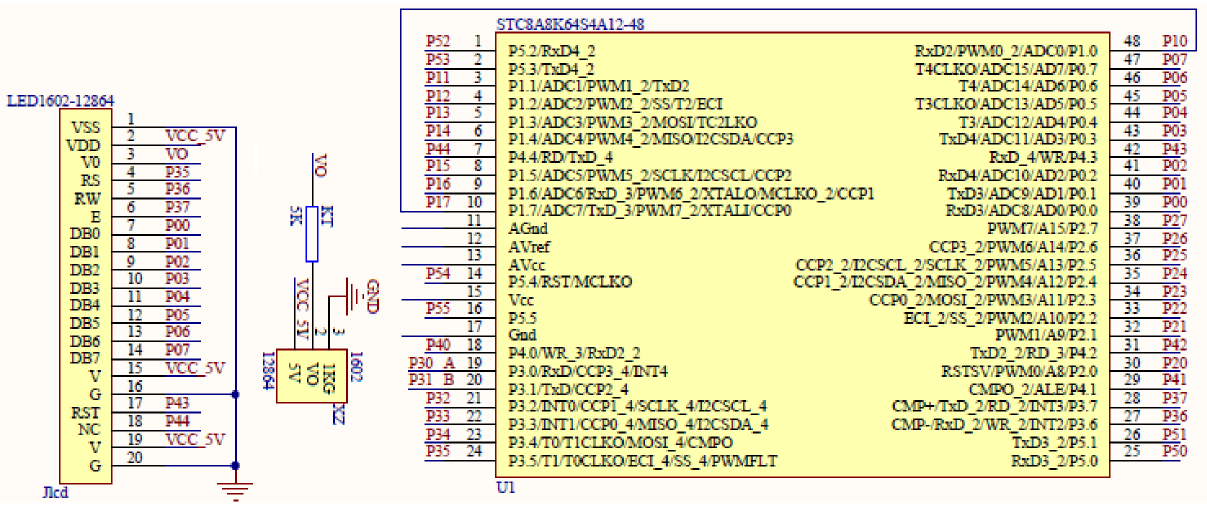
Code

```
#include "STC8xxx.h"
#include "BSP.h"
#include "LCD.c"
#include "lcd_print.c"


unsigned int first_edge = 0x0000;
unsigned int second_edge = 0x0000;

unsigned long clks = 0x00000000;
unsigned long ov_cnt = 0x00000000;


void setup(void);


void PCA_ISR(void)
interrupt 7
{
    if(check_PCA_Counter_overflow_flag)
    {
      ov_cnt++;;
      clear_PCA_Counter_overflow_flag;
    }

    if(check_PCA_0_flag)
    {
      second_edge = PCA_get_CCAP0();
      clks = ((65536 * ov_cnt) + second_edge - first_edge);
      ov_cnt = 0;
```

```
      first_edge = second_edge;
      second_edge = 0;
      clear_PCA_0_flag;
    }
}


void main(void)
{
  float f = 0.0;

  unsigned char s = 0;

  setup();

  LCD_goto(0, 0);
  LCD_putstr("Period/ms:");

  LCD_goto(0, 1);
  LCD_putstr("Freq./kHz:");

  while(1)
  {
    PWM_stop_counter;

    switch(s)
    {
      case 1:
      {
        PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_1);
        break;
      }

      case 2:
      {
        PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_4);
        break;
      }

      case 3:
      {
        PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_3);
        break;
      }

      default:
      {
        PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_2);
        break;
      }
    }

    PWM_start_counter;
    delay_ms(1000);

    f = (12000.0 / ((float)clks));
    print_I(11, 0, clks);
    print_F(11, 1, f, 1);

    delay_ms(1000);

    s++;
    if(s > 3)
    {
      s = 0;
    }
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  PWM0_setup(PWM_pin_is_PWM_output, \
               PWM_init_lvl_low, \
               PWM_0_pin_P10, \
               PWM_level_normal);
```

```
    PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_2);
    PWM_set_counter(500);

    PWM_set_PWM0_T1(0);
    PWM_set_PWM0_T2(200);

    PWM_start_counter;

    PCA_pin_option(0x00);

    PCA_setup(PCA_continue_counting_in_idle_mode, PCA_clk_sys_clk_div_1);
    PCA_load_counter(0x0000);

    PCA_0_mode(PCA_16_bit_falling_edge_capture);

    _enable_PCA_0_interrupt;
    _enable_PCA_counter_interrupt;
    _enable_global_interrupt;

    PCA_start_counter;

    LCD_init();
    LCD_clear_home();
}
```

Schematic



Explanation

Measurement of frequency can easily be done by timing the time difference between two successive rising or falling edge captures. Input capture hardware along with a timer would scan and sample incoming waveform. Thus, input capture hardware must have at least twice the sampling frequency than the frequency of the waveform coming to the capture input. This is simply the Nyquist criterion.

$$F_S \geq (2 \times F)$$

To demonstrate frequency measurement with PCA, I have used two hardware. Firstly, I used a PWM generator to generate waveform of different frequencies and secondly, a PCA module to capture the PWM waveform.

To begin with, note that the micro is running at 12MHz.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);
```

The PWM peripheral is setup with P1.0 pin as PWM channel 0. The initial frequency of the PWM is 12kHz and the duty cycle is about 60%.

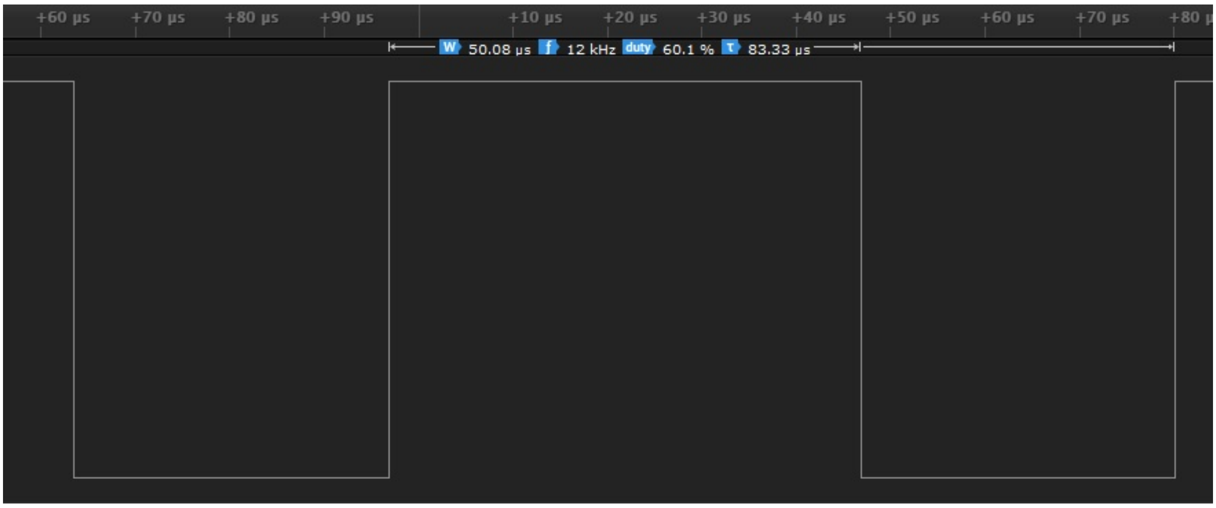
```
PWM0_setup(PWM_pin_is_PWM_output, \
           PWM_init_lvl_low, \
           PWM_0_pin_P10, \
           PWM_level_normal);

PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_2);
PWM_set_counter(500);

PWM_set_PWM0_T1(0);
PWM_set_PWM0_T2(300);

PWM_start_counter;
```

The following equations show us how we fabricated this frequency and duty cycle:

$$PWM\ Frequency = \frac{PWM\ Input\ Clock\ Frequency}{(PWM\ Counter\ Value\ \times\ PWM\ Clock\ Prescalar)} = \frac{12\ MHz}{(500\ \times\ 2)} = 12\ kHz$$

and

$$PWM\ Duty\ Cycle = \frac{|PWM\ T2\ Value\ -\ PWM\ T1\ Value|}{PWM\ Counter\ Value} \times 100\% = \frac{|300\ -\ 0|}{500} \times 100\% = 60\%$$



In the main, the PWM frequency if changed four times and these frequencies are 24kHz, 12kHz, 8kHz and 6kHz. The change is done about every two seconds.

Now let us see how we have to setup the PCA's capture part. P1.7 pin (PCA0) is setup as the capture pin that would look for falling edges. PCA hardware clock is running at full system clock speed of 12MHz. The Nyquist criterion is fulfilled because PWM out frequency is not more than 24kHz.

```
PCA_pin_option(0x00);

PCA_setup(PCA_continue_counting_in_idle_mode, PCA_clk_sys_clk_div_1);
PCA_load_counter(0x0000);

PCA_0_mode(PCA_16_bit_falling_edge_capture);

_enable_PCA_0_interrupt;
_enable_PCA_counter_interrupt;
_enable_global_interrupt;

PCA_start_counter;
```

PCA interrupt will be triggered when a falling edge is detected or when there is PCA counter overflows. Both of these events will be needed. Though under a same interrupt subroutine, these events are differentiated by two separate flags. The first will be needed to make sure that we took account of any PCA overflow in between two successive captures and the second will be needed to snap PCA counter count at the moment of falling-edge detection.

```
void PCA_ISR(void)
interrupt 7
{
    if(check_PCA_Counter_overflow_flag)
    {
      ov_cnt++;;
      clear_PCA_Counter_overflow_flag;
    }

    if(check_PCA_0_flag)
    {
      second_edge = PCA_get_CCAP0();
      clks = ((65536 * ov_cnt) + second_edge - first_edge);
      ov_cnt = 0;
      first_edge = second_edge;
      second_edge = 0;
      clear_PCA_0_flag;
    }
}
```

When PCA interrupt occurs, two flags are checked. Firstly, the PCA counter overflow is checked and secondly, PCA0 interrupt flag is checked. If PCA counter overflow occurs then it is saved in a variable. This will mark that an overflow event occurred during a snap.

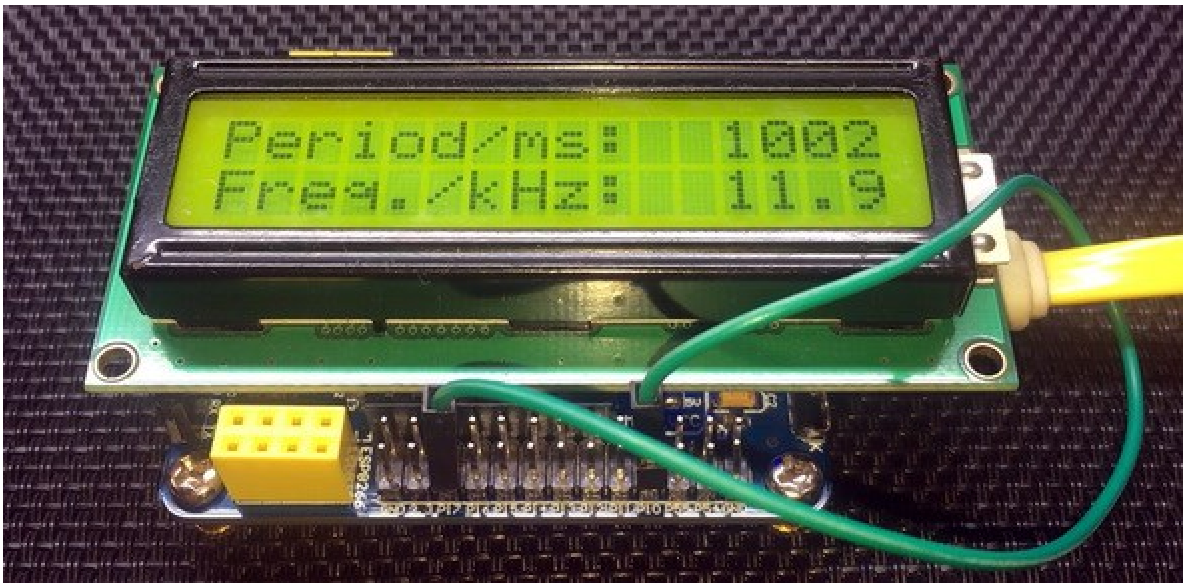
Since frequency measurement is done by measuring the PCA counter's count difference between two consecutive falling edges, we need to assume that the first capture is basically the second capture instead of the first falling edge capture. When the second capture is recorded, it is measured against the first one. In this measurement, we also have to take note if any PCA overflow occurred during the captures. Note that no timer has been used so far and that is because the PCA module is doubling as a timer here.

In the main, the PWM frequencies are changed periodically and the PWM frequencies are measured. The measured frequency is displayed on an LCD.

```
PWM_stop_counter;

switch(s)
{
  case 1:
  {
    PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_1);
    break;
  }

  case 2:
  {
    PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_4);
    break;
  }

  case 3:
  {
    PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_3);
    break;
  }

  default:
  {
    PWM_clk_set(PWM_clk_sys_PS, PWM_clk_ps_sys_clk_div_2);
    break;
  }
}

PWM_start_counter;
delay_ms(1000);

f = (12000.0 / ((float)clks));
print_I(11, 0, clks);
print_F(11, 1, f, 1);

delay_ms(1000);

s++;
if(s > 3)
{
  s = 0;
}
```

Variable ***clks*** represents the count difference between capture snaps and so dividing PCA clock frequency with this count gives capture waveform frequency in hertz. I have coded 12000 instead of 12000000 as clock frequency because I want measurements to be in kHz instead of Hz.

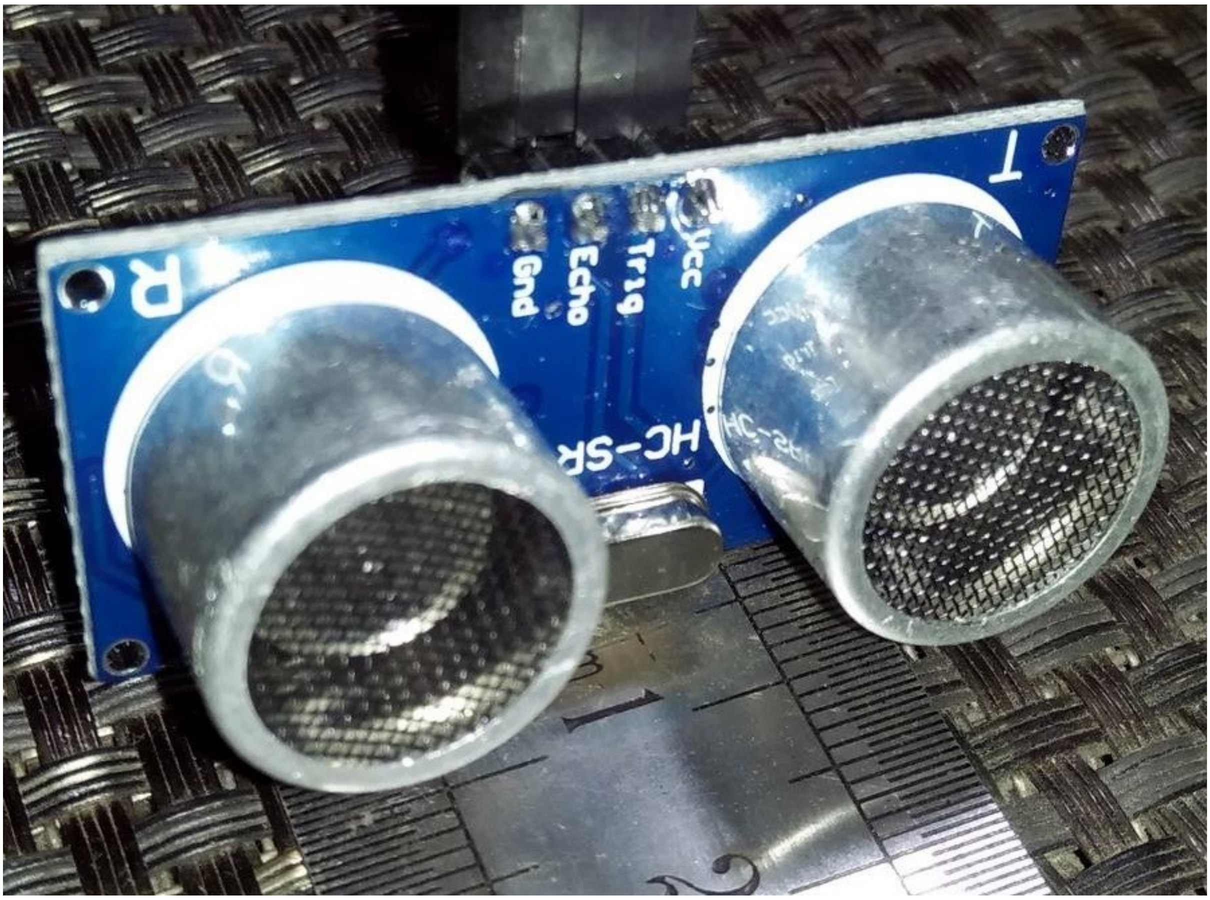
Demo



Demo video link: https://youtu.be/2uYhZw7FXSE.

Using PCA to Measure Pulse Width from HCSR-04 SONAR Module

Apart from frequency measurement, we can apply the same input capture technique to measure pulse widths or PWM duty cycles. Pulse width measurement requires capturing PCA counts of two successive opposite edges unlike two successive same edges. In this segment, we will see how we can measure distance measured by a HCSR-04 SONAR sensor by measuring pulse width output from it.

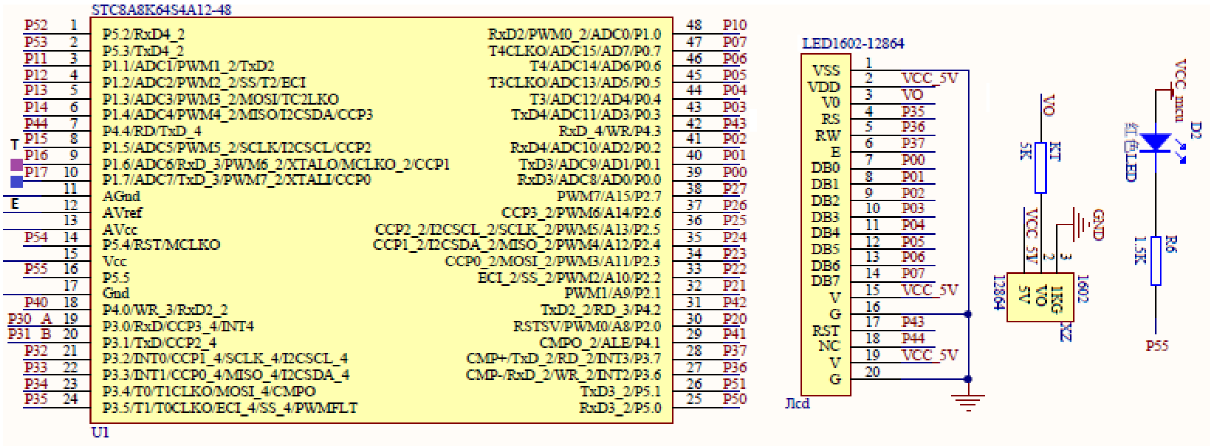


Code

```
#include "STC8xxx.h"
#include "BSP.h"
#include "LCD.c"
#include "lcd_print.c"


unsigned char state = 0x00;
unsigned int pulse_width = 0x0000;


void setup(void);


void PCA_ISR(void)
interrupt 7
{
  if(check_PCA_0_flag)
  {
    state ^= 1;
    clear_PCA_0_flag;
  }
```

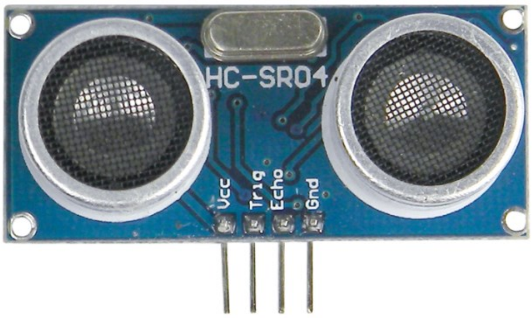
```
  switch(state)
  {
    case 1:
    {
      PCA_load_counter(0x0000);
      break;
    }

    default:
    {
      pulse_width = PCA_get_CCAP0();
      break;
    }
  }
}


void main(void)
{
  float range = 0.0;

  setup();

  LCD_goto(0, 0);
  LCD_putstr("Range/cm:");
  LCD_goto(0, 1);
  LCD_putstr("Pulse/us:");

  while(1)
  {
    P55_low;
    delay_ms(10);

    P16_high;
    delay_us(10);
    P16_low;
    P55_high;

    range = ((((float)pulse_width) / 58.0));
    print_F(10, 0, range, 1);
    print_I(10, 1, pulse_width);

    delay_ms(490);
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  P16_push_pull_mode;
  P55_open_drain_mode;

  PCA_pin_option(0x00);

  PCA_setup(PCA_continue_counting_in_idle_mode, PCA_clk_sys_clk_div_12);
  PCA_load_counter(0x0000);

  PCA_0_mode(PCA_16_bit_both_edge_capture);

  _enable_PCA_0_interrupt;
  _enable_global_interrupt;

  PCA_start_counter;

  LCD_init();
  LCD_clear_home();
}
```

Schematic

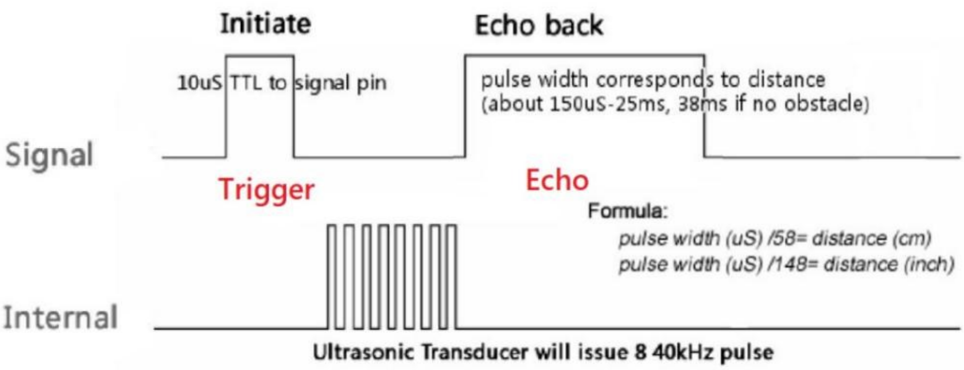


Explanation

HC-SR04 SONAR sensor works by sending a stream of ultrasonic pulses and then timing how long it took for an echo to bounce back from a nearby object. Based on timing, the sensor gives a pulse of variable width. The pulse width is a representation of distance or object range.



There are two pins apart from the power supply pins and these are needed to communicate with the sensor. When the trigger pin of the sensor is pulled high for about 10µs, the sensor acknowledges this short duration pulse as a command from its host micro to measure and return range data. A pulse output from the echo pin is the representation of distance.



Now let's see how the code is working in this example. Firstly, the system clock is set to 12MHz.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);
```

Pins P1.6 and P1.7 are tied to trigger and echo pins of HC-SR04 sensor respectively.

The PCA hardware is setup by setting its clock to 1MHz. Thus, each of PCA counter's count is equal to 1µs tick. Default PCA pin configuration is used. The PCA counter is set to 0 count and 16-bit both edge capture mode is selected. Interrupt method is used and so PCA and global interrupts are enabled before starting the PCA counter.

```
PCA_pin_option(0x00);

PCA_setup(PCA_continue_counting_in_idle_mode, PCA_clk_sys_clk_div_12);
PCA_load_counter(0x0000);

PCA_0_mode(PCA_16_bit_both_edge_capture);

_enable_PCA_0_interrupt;
_enable_global_interrupt;

PCA_start_counter;
```

The PCA hardware is now ready to detect edge transitions. When transitions occur, PCA interrupt is triggered. During a low-to-high transition or rising edge, the PCA counter is reset to 0 count and left to continue counting. Upon detecting a high-to-low transition or falling edge, the PCA counter is read.

```
void PCA_ISR(void)
interrupt 7
{
  if(check_PCA_0_flag)
  {
    state ^= 1;
    clear_PCA_0_flag;
  }

  switch(state)
  {
    case 1:
    {
      PCA_load_counter(0x0000);
      break;
    }

    default:
    {
      pulse_width = PCA_get_CCAP0();
      break;
    }
  }
}
```

$$Pulse\ Width = Falling\ Edge\ Capture\ Count - Rising\ Edge\ Capture\ Count$$

Since the PCA counter is set to 0 during rising edge, the equation simplifies to

$$Pulse\ Width = Falling\ Edge\ Capture\ Count - 0$$

or simply

$$Pulse\ Width = Falling\ Edge\ Capture\ Count$$

Since the PCA counter is counting with 1µs resolution, the falling edge capture count is the pulse width period in microseconds.

$$Pulse\ Width\ in\ \mu s = Falling\ Edge\ Capture\ Count\ \times 1\mu s$$

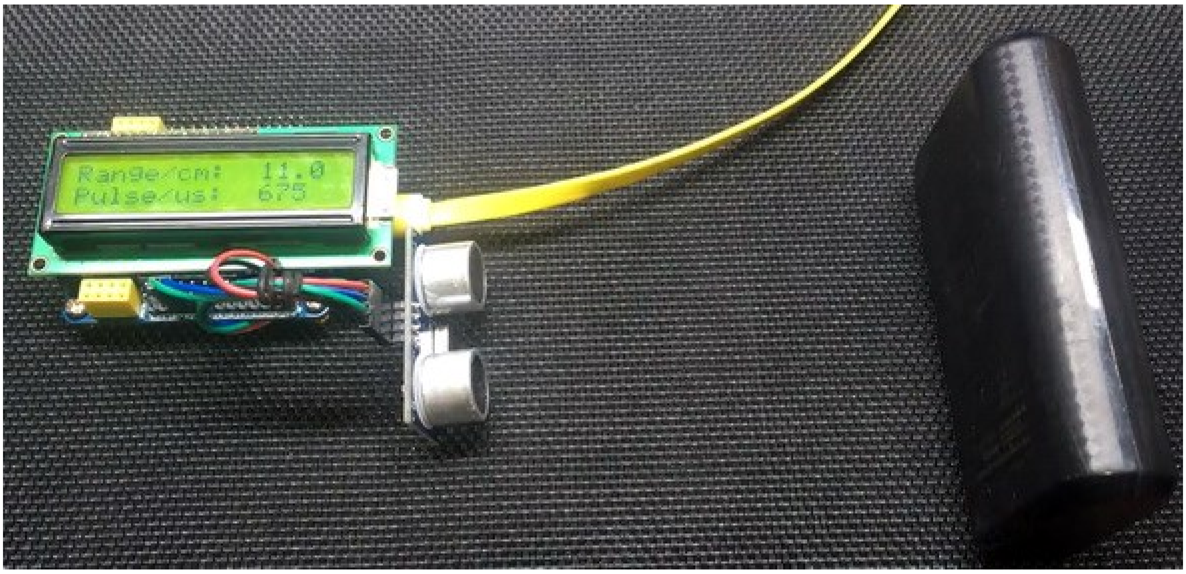
In the main loop, the SONAR sensor is triggered and the result from the PCA interrupt is processed. Target object distance detected by the sensor is a function of pulse width measured in microseconds. Thus, as per datasheet of the sensor, pulse width divided by 58 is equal to distance in centimeters.

$$Object\ Range\ in\ cm = \frac{Pulse\ Width\ in\ \mu s}{58}$$

The range and the captured pulse width are displayed in an LCD after performing the aforementioned computations.

```
P55_low;
delay_ms(10);

P16_high;
delay_us(10);
P16_low;
P55_high;

range = ((((float)pulse_width) / 58.0));
print_F(10, 0, range, 1);
print_I(10, 1, pulse_width);

delay_ms(490);
```

Demo

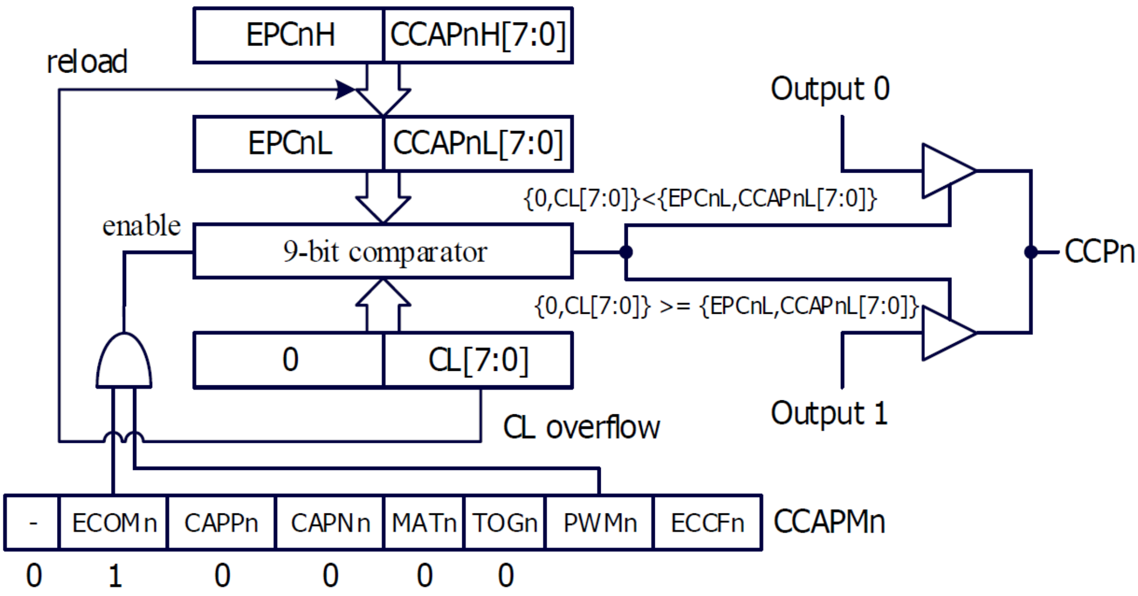


Demo video link: https://youtu.be/h8NIdGpSZLI.

Using PCA to Generate PWM

Unlike enhanced PWM module, PCA module can used to generate PWMs of different resolutions. However, PCA-generated PWMs are like general purpose PWMs and do not have additional functionalities. These PWMs can be used for simple tasks like driving LEDs, servo motors, simple motor controls, simple DC-DC converters, etc.

The following block diagram shows PCA module in 8-bit PWM mode. The 9-bit internal comparator compares the values *CL* and *CCAPnL + EPCnL.* When *CL* is equal or greater than *CCAPnL + EPCnL*, the PWM output is set high and when *CL* is less than *CCAPnL + EPCnL*, the PWM output is set low. Thus, *CCAPnL + EPCnL* are responsible for the duty cycle of PWM because *CL* keeps counting with each tick of PCA clock. When *CL* overflows, *CCAPnL + EPCnL* are reloaded automatically with the contents of *CCAPnH + EPCnH.* Updating *CCAPnH + EPCnH*, results in the alternation of PWM duty cycle. The same concept is true for PCA PWMs of other resolutions. Similar strategies are applied for PCA PWMs of other resolutions.

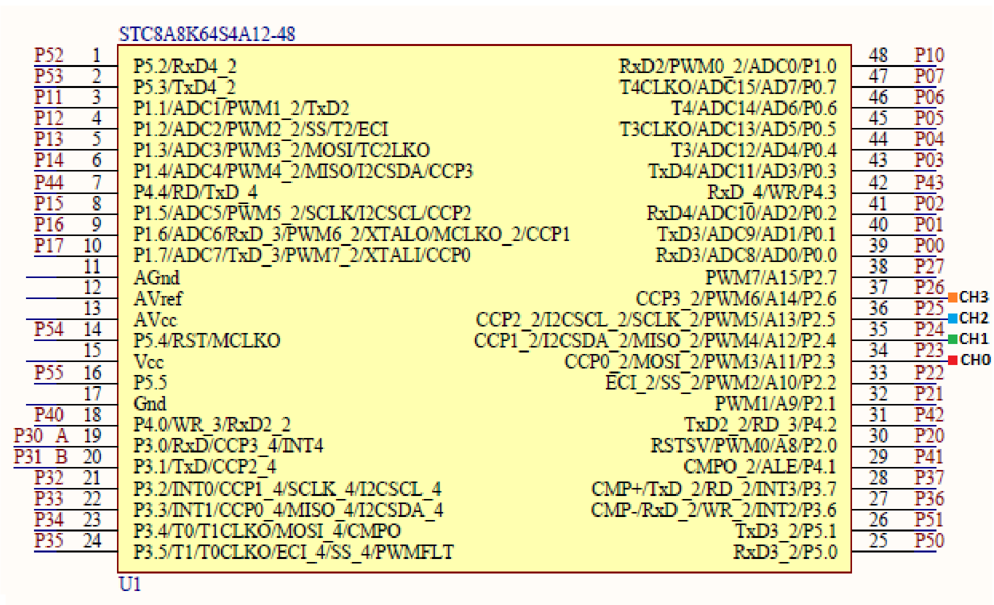


Code

```
#include "STC8xxx.h"
#include "BSP.h"


void setup(void);
void PWM_idle(void);


void main(void)
{
  signed int j = 0x0000;

  setup();

  while(1)
  {
    for(j = 0; j < 256; j++)
    {
      PCA_0_8_bit_PWM_reload_value(255 - j);
```

```
      PCA_1_8_bit_PWM_reload_value(j);
      PCA_2_8_bit_PWM_reload_value(255 - j);
      PCA_3_8_bit_PWM_reload_value(j);
      delay_ms(10);
    }

    for(j = 255; j > -1; j--)
    {
      PCA_0_8_bit_PWM_reload_value(255 - j);
      PCA_1_8_bit_PWM_reload_value(j);
      PCA_2_8_bit_PWM_reload_value(255 - j);
      PCA_3_8_bit_PWM_reload_value(j);
      delay_ms(10);
    }
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  PCA_pin_option(0x10);

  PCA_setup(PCA_continue_counting_in_idle_mode, PCA_clk_sys_clk_div_1);

  PCA_n_PWM(0, PCA_PWM_without_interrupt, PCA_8_bit_PWM);
  PCA_n_PWM(1, PCA_PWM_without_interrupt, PCA_8_bit_PWM);
  PCA_n_PWM(2, PCA_PWM_without_interrupt, PCA_8_bit_PWM);
  PCA_n_PWM(3, PCA_PWM_without_interrupt, PCA_8_bit_PWM);

  PCA_0_8_bit_PWM_compare_value(0);
  PCA_0_8_bit_PWM_reload_value(0);

  PCA_1_8_bit_PWM_compare_value(0);
  PCA_1_8_bit_PWM_reload_value(0);

  PCA_2_8_bit_PWM_compare_value(0);
  PCA_2_8_bit_PWM_reload_value(0);

  PCA_3_8_bit_PWM_compare_value(0);
  PCA_3_8_bit_PWM_reload_value(0);

  PCA_load_counter(0);

  PCA_start_counter;
}
```

Schematic



Explanation

In this example, four PCA PWM channels are simply used to drive LEDs of different colors.

The system clock is set to 12MHz.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);
```

Pins P2.3 to P2.6 are used for PWM outputs. Each of these channels are set to 8-bit PWM mode and their compare and reload values are set to zero. Thus, initially all PWM channels are at same logic state. The PCA counter is set to run at system clock speed. All PWM channels will be running at same clock frequency as all are sharing the same PCA counter.

```
PCA_pin_option(0x10);

/*
ECI         CCP0        CCP1        CCP2        CCP3        Hex         Option
P1.2        P1.7        P1.6        P1.5        P1.4        0x00        option 1
P2.2        P2.3        P2.4        P2.5        P2.6        0x10        option 2
P7.4        P7.0        P7.1        P7.2        P7.3        0x20        option 3
P3.5        P3.3        P3.2        P3.1        P3.0        0x30        option 4
*/

PCA_setup(PCA_continue_counting_in_idle_mode, PCA_clk_sys_clk_div_1);

PCA_n_PWM(0, PCA_PWM_without_interrupt, PCA_8_bit_PWM);
PCA_n_PWM(1, PCA_PWM_without_interrupt, PCA_8_bit_PWM);
PCA_n_PWM(2, PCA_PWM_without_interrupt, PCA_8_bit_PWM);
PCA_n_PWM(3, PCA_PWM_without_interrupt, PCA_8_bit_PWM);

PCA_0_8_bit_PWM_compare_value(0);
PCA_0_8_bit_PWM_reload_value(0);

PCA_1_8_bit_PWM_compare_value(0);
PCA_1_8_bit_PWM_reload_value(0);

PCA_2_8_bit_PWM_compare_value(0);
PCA_2_8_bit_PWM_reload_value(0);

PCA_3_8_bit_PWM_compare_value(0);
PCA_3_8_bit_PWM_reload_value(0);

PCA_load_counter(0);

PCA_start_counter;
```

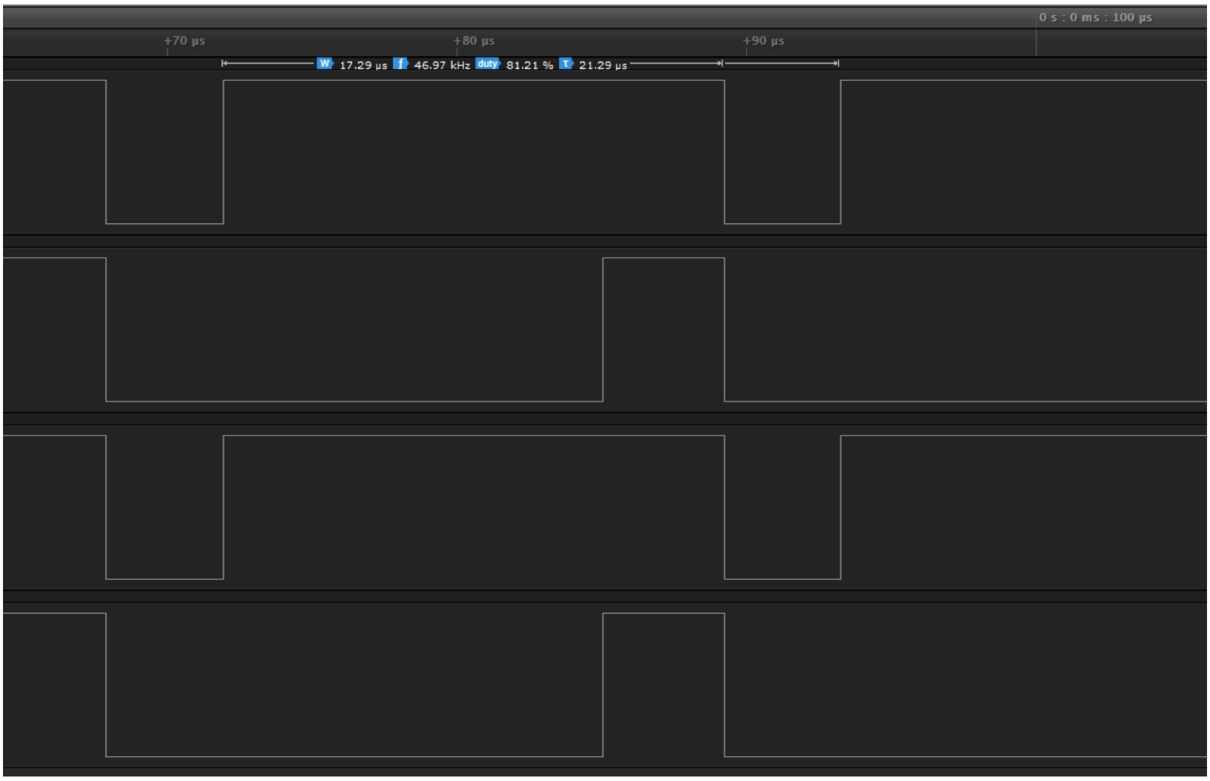
The PWM frequency of PCA channels is calculated to be as follows.

$$PCA\ PWM\ Frequency = \frac{System\ Clock}{(PCA\ Prescalar\ \times\ 2^{PWM\ Resolution})}$$

$$PCA\ PWM\ Frequency = \frac{12\ MHz}{(1\ \times\ 2^{8})} = 46.88\ kHz$$

Thus, the period of the PWM is 21.33µs.

The logic analyser data confirms that the frequency is indeed around 46 kHz.



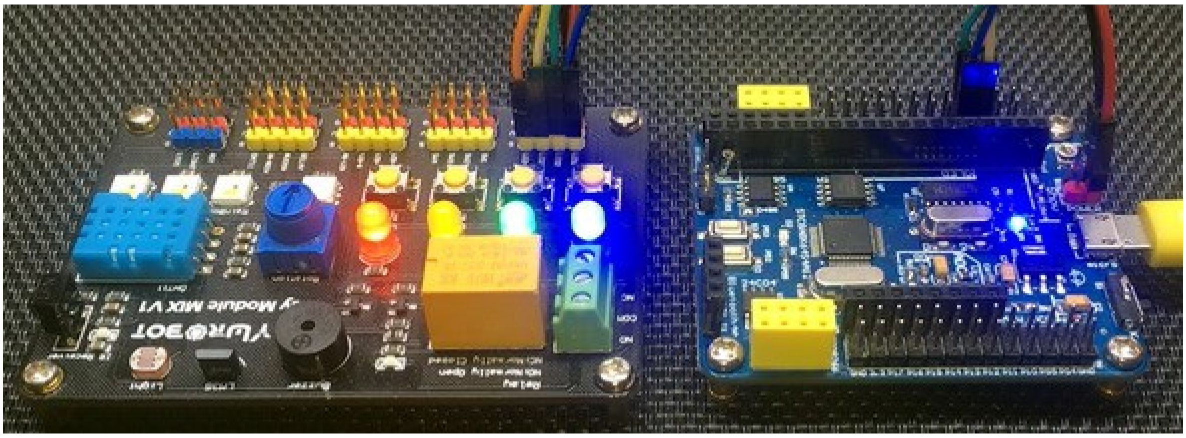
The duty cycles of the PWMs are calculated using the following formula.

$$PCA\ PWM\ Duty\ Cycle = \frac{Reload\ Value}{(2^{PWM\ Resolution} - 1)}$$

Inside the main loop, each of the channels are updated every 10ms by updating their reload values.

```
for(j = 0; j < 256; j++)
{
  PCA_0_8_bit_PWM_reload_value(255 - j);
  PCA_1_8_bit_PWM_reload_value(j);
  PCA_2_8_bit_PWM_reload_value(255 - j);
  PCA_3_8_bit_PWM_reload_value(j);
  delay_ms(10);
}

for(j = 255; j > -1; j--)
{
  PCA_0_8_bit_PWM_reload_value(255 - j);
  PCA_1_8_bit_PWM_reload_value(j);
  PCA_2_8_bit_PWM_reload_value(255 - j);
  PCA_3_8_bit_PWM_reload_value(j);
  delay_ms(10);
}
```

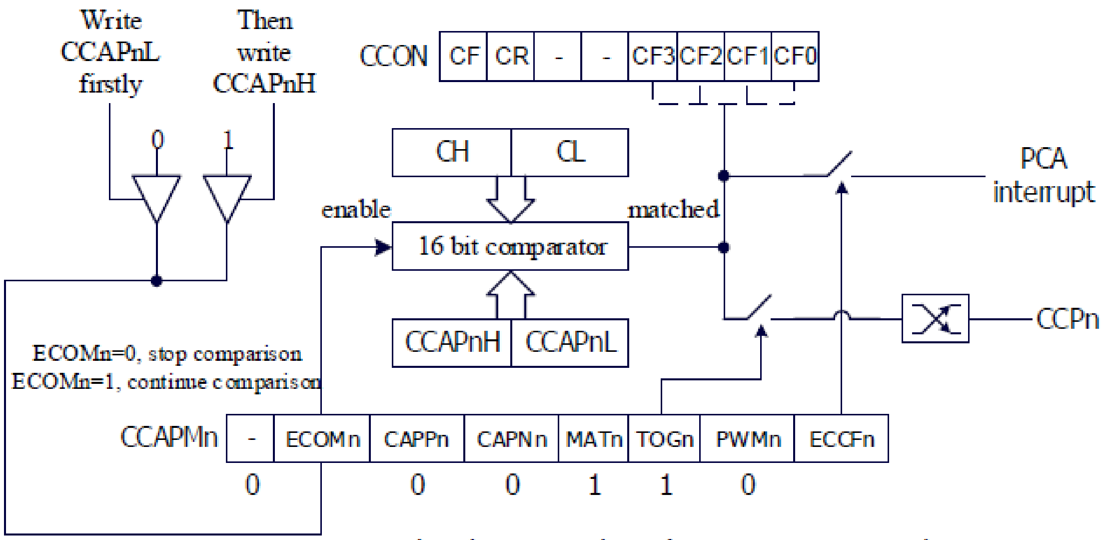
Demo



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

Using PCA as a High-Speed Pulse Generator

In many cases, we need to generate high-frequency carrier waveform or pulse trains. For example, this is needed when we need to make an infrared (IR) remote transmitter. This job can easily be done by using a timer or software delays. However, in STC micros, we have PCA hardware to accomplish the same job much more easily. This lets us use general purpose timers for other tasks.



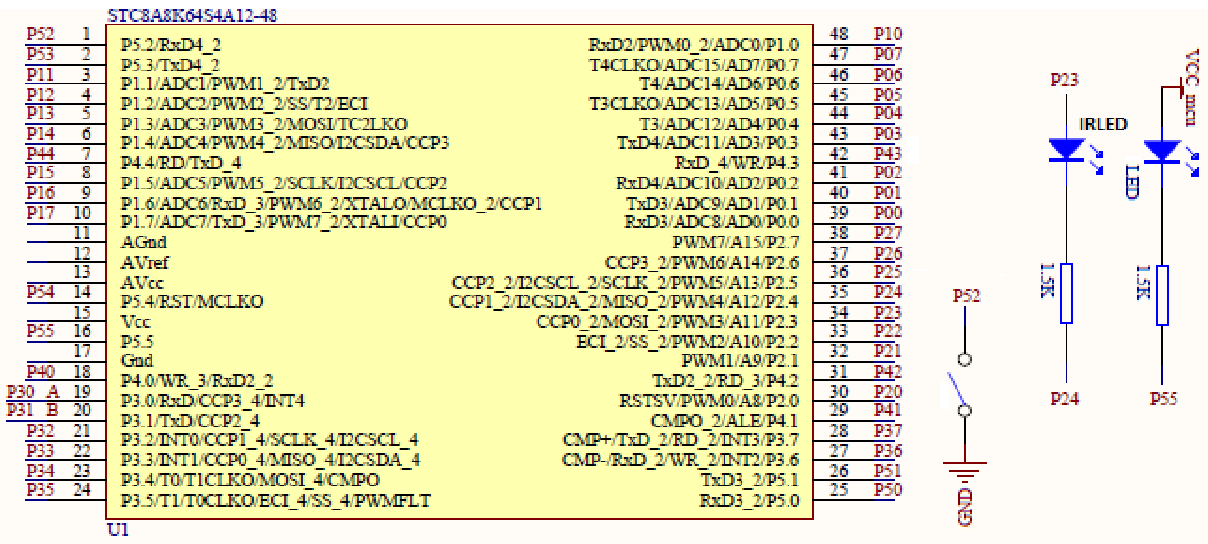
PCA high speed pulse output mode

Code

```
#include "STC8xxx.h"
#include "BSP.h"


#define freq_change         (12000000L / 2 / 38400)


unsigned int value = freq_change;


void setup(void);


void PCA_ISR(void)
interrupt 7
{
  clear_PCA_0_flag;
  PCA0_load_value(value);
  value += freq_change;
}


void main(void)
{

  setup();

  while(1)
  {
    if(P52_get_input == FALSE)
    {
      P24_high;
      P55_low;
```

```
    }

    else
    {
      P24_low;
      P55_high;
    }
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  P01_push_pull_mode;
  P55_open_drain_mode;

  P52_input_mode;
  P52_pull_up_enable;

  PCA_pin_option(0x10);

  PCA_setup(PCA_continue_counting_in_idle_mode, PCA_clk_sys_clk_div_1);

  PCA_0_mode(PCA_16_bit_high_speed_pulse_output);

  PCA_load_counter(0);
  PCA0_load_value(value);

  _enable_PCA_0_interrupt;
  _enable_global_interrupt;

  PCA_start_counter;
}
```

Schematic



Explanation

In this example, PCA high-speed pulse mode is used to make an IR remote transmitter.

Since PCA is a time-dependent hardware peripheral, we need to be sure of the system clock settings. As with most examples, the system clock is set to 12MHz.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);
```

Four GPIO pins are required for this example and are set right after selecting the system clock. A button is connected with pin P5.2. Pressing the button would change the logic state of both the onboard LED (P5.5) and pin P2.4. Lastly, pin P2.3 is set as the PCA CCP0 pin. This pin is the high-speed pulse output pin. P2.3 and P2.4 are connected to an IR LED.

Note that P2.4 is not set like other GPIOs because it is already set by PCA hardware.

```
P01_push_pull_mode;
P55_open_drain_mode;

P52_input_mode;
P52_pull_up_enable;

PCA_pin_option(0x10);
```

Now let's see how the PCA hardware is set. Firstly, the PCA clock and counting mode are set. PCA clock is set to run at 12MHz, i.e., at system clock speed. Next, the PCA channel to be used and its mode of operation is set, here – 16-bit high speed pulse output mode. The PCA counter is set 0 count and PCA0 channel is loaded with a count value.

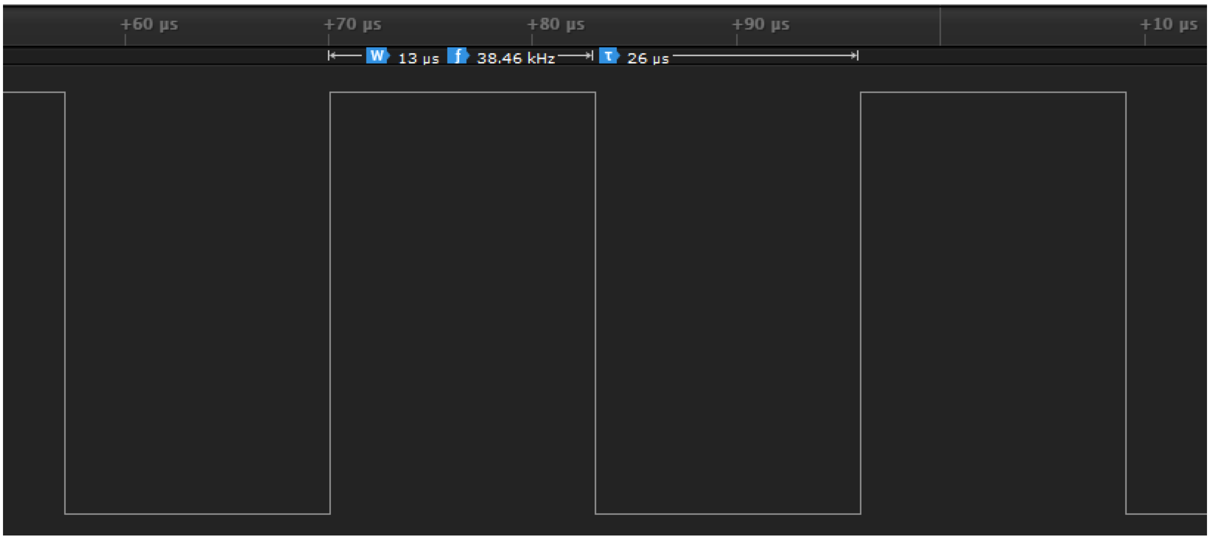
```
PCA_setup(PCA_continue_counting_in_idle_mode, PCA_clk_sys_clk_div_1);

PCA_0_mode(PCA_16_bit_high_speed_pulse_output);

PCA_load_counter(0);
PCA0_load_value(value);

_enable_PCA_0_interrupt;
_enable_global_interrupt;

PCA_start_counter;
```

The loaded value is the count value required to generate the desired frequency of 38.4kHz (26µs period).

$$PCA\ Load\ Value = \frac{System\ Clock}{(PCA\ Prescalar\ \times 2\ \times Target\ Frequency)}$$

$$PCA\ Load\ Value = \frac{12\ MHz}{(1\ \times 2\ \times 38.4\ kHz)} = 156\ (rounded)$$

Finally, the PCA0 and global interrupts are enabled before starting the PCA counter.



Note that PCA counter keeps counting and is not reloaded anywhere in the whole code. It is left on its own and it will rollover after 65535 counts. When the PCA counter counts to 156 counts, PCA0 interrupt is triggered. Note that 156 counts equal 13µs because the PCA clock is running at 12MHz. On each interrupt, PCA0's channel polarity is altered. This means 13µs is the pulse width of high-speed pulse output waveform.

$$Pulse\ Width = \frac{156}{12MHz} = 13\mu s$$

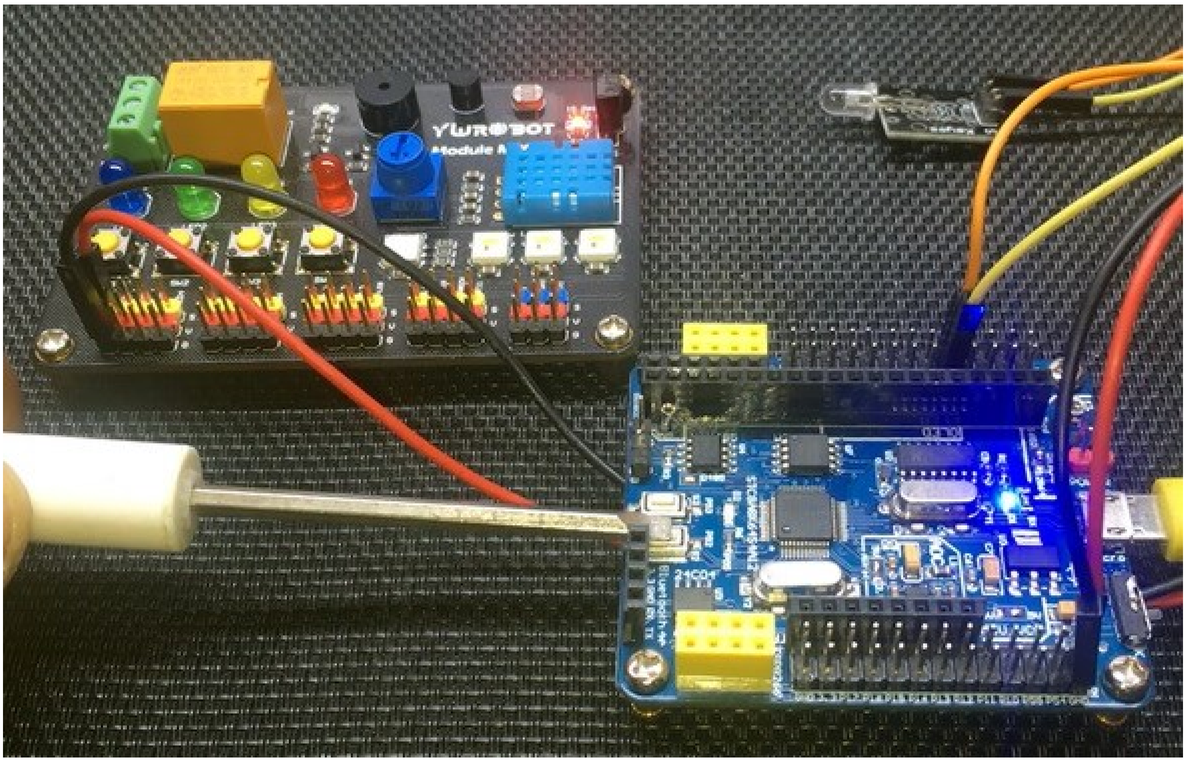
PCA interrupt is processed by clearing PCA0 interrupt flag and by loading PCA0 with a new count value (previous count + load value, i.e., 156 + 156 = 312) because after that new count (312) PCA0 interrupt will be triggered again and the process repeats over and over again.

```
void PCA_ISR(void)
interrupt 7
{
  clear_PCA_0_flag;
  PCA0_load_value(value);
  value += freq_change;
}
```

Inside the main loop, the push button pin is polled. If the button is pressed, the IR LED is enabled by setting P2.4 high or else it is disabled. Since P2.3 is the PCA 0 output, there is always 38.4 kHz square pulses present in that pin. Only when P2.4 is set is high, the IR LED lights up. It is just like a simple AND logic.

```
if(P52_get_input == FALSE)
{
  P24_high;
  P55_low;
}

else
{
  P24_low;
  P55_high;
}
```

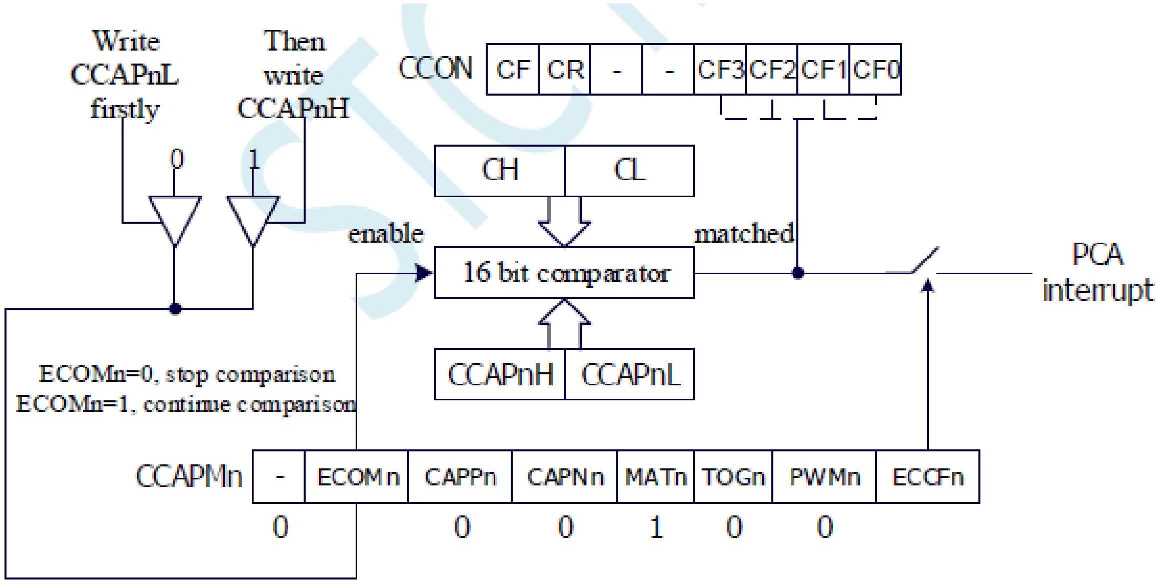
Demo



Demo video link: https://youtu.be/V9xbGAijxjQ.

Using PCA as a Software Timer

We have already seen how the PCA module doubles as timer in the capture examples. However, unlike actual hardware timers, we would have to use some software tricks and keep certain things in mind.

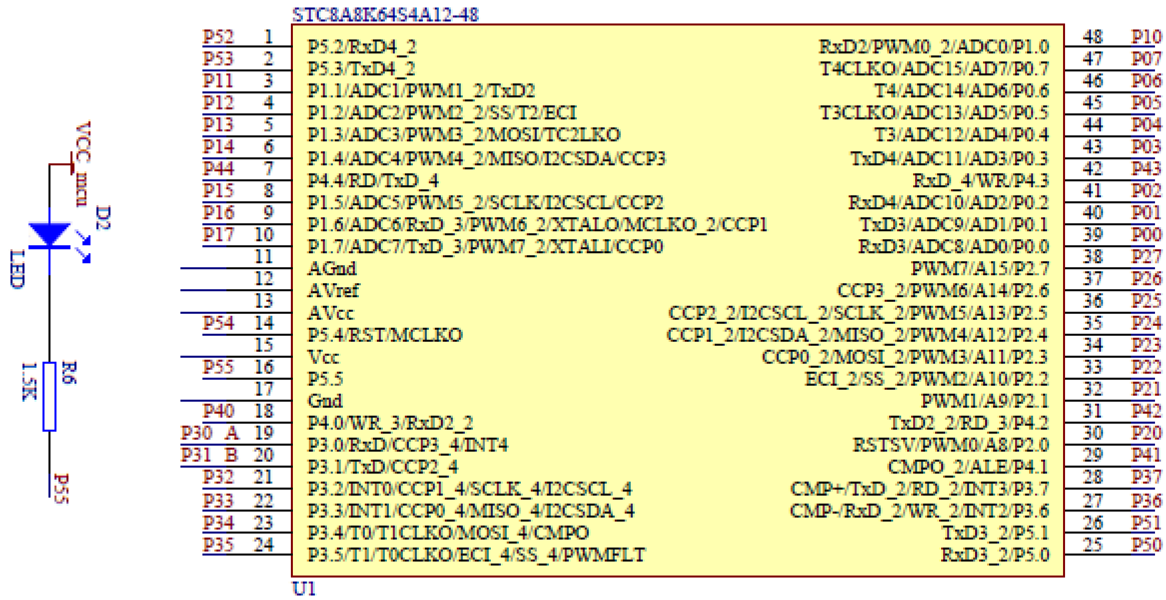


Code

```
#include "STC8xxx.h"
#include "BSP.h"


#define T_Load          (1000000L / 12 / 2)


unsigned int value = T_Load;


void setup(void);


void PCA_ISR(void)
interrupt 7
{
  clear_PCA_0_flag;
  PCA0_load_value(value);
  P55_toggle;
  value += T_Load;
}


void main(void)
{

  setup();

  while(1)
  {
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 24, MCLK_SYSCLK_no_output, MCLK_out_P54);
```

```
  P55_open_drain_mode;

  PCA_pin_option(0x10);

  PCA_setup(PCA_continue_counting_in_idle_mode, PCA_clk_sys_clk_div_12);

  PCA_0_mode(PCA_16_bit_software_timer);

  PCA_load_counter(0);
  PCA0_load_value(T_Load);

  _enable_PCA_0_interrupt;
  _enable_global_interrupt;

  PCA_start_counter;
}
```

Schematic



Explanation

This example uses the same concepts as in the high-speed pulse output example and yes, again it is a simple onboard LED blinking example. Between the two examples almost everything is same.

The system clock is set to 1MHz.

```
CLK_set_sys_clk(IRC_24M, 24, MCLK_SYSCLK_no_output, MCLK_out_P54);
```

PCA is setup with just as in the past example except for the clock prescalar and mode of operation. In this example, the PCA clock is set to 83.3kHz and software timer mode is selected. Note that since no PCA input-output channel is needed in this example, it doesn't matter which pin arrangement is initialized.

```
PCA_pin_option(0x10);

PCA_setup(PCA_continue_counting_in_idle_mode, PCA_clk_sys_clk_div_12);

PCA_0_mode(PCA_16_bit_software_timer);

PCA_load_counter(0);
```

```
PCA0_load_value(T_Load);

_enable_PCA_0_interrupt;
_enable_global_interrupt;

PCA_start_counter;
```

The desired LED blinking frequency is 1Hz and so the PCA0 is loaded with the following count value.

$$PCA\ Load\ Value = \frac{1\ MHz}{(12 \times 2 \times 1\ Hz\ )} = 41666\ (rounded)$$

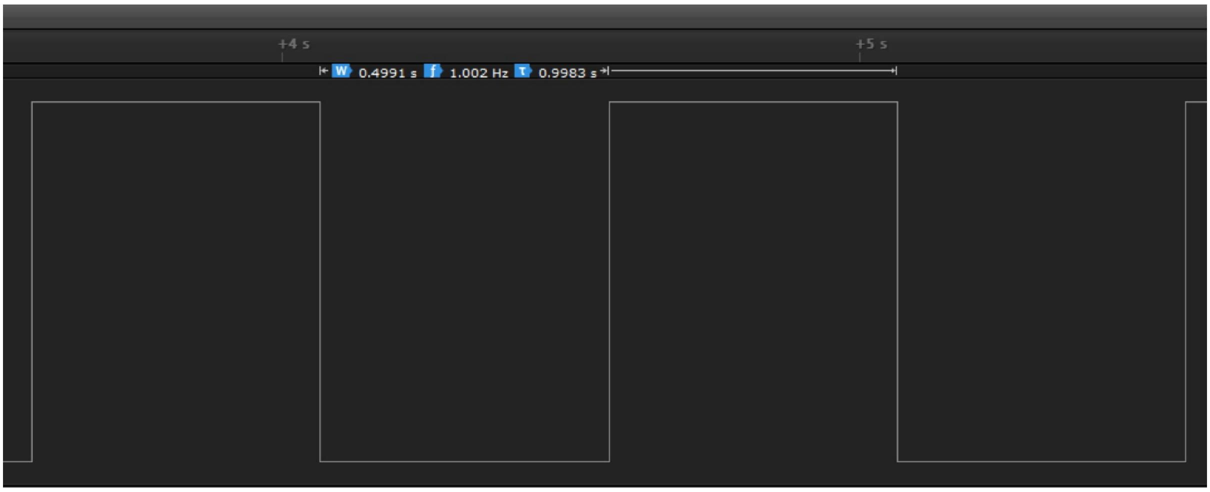
The PCA clock frequency is as follows:

$$PCA\ Clock\ Frequency = \frac{1\ MHz}{12} = 83.33\ kHz\ or\ approx.12\ \mu s\ period$$

Thus, PCA interrupt will be triggered at every half second as per following formula:

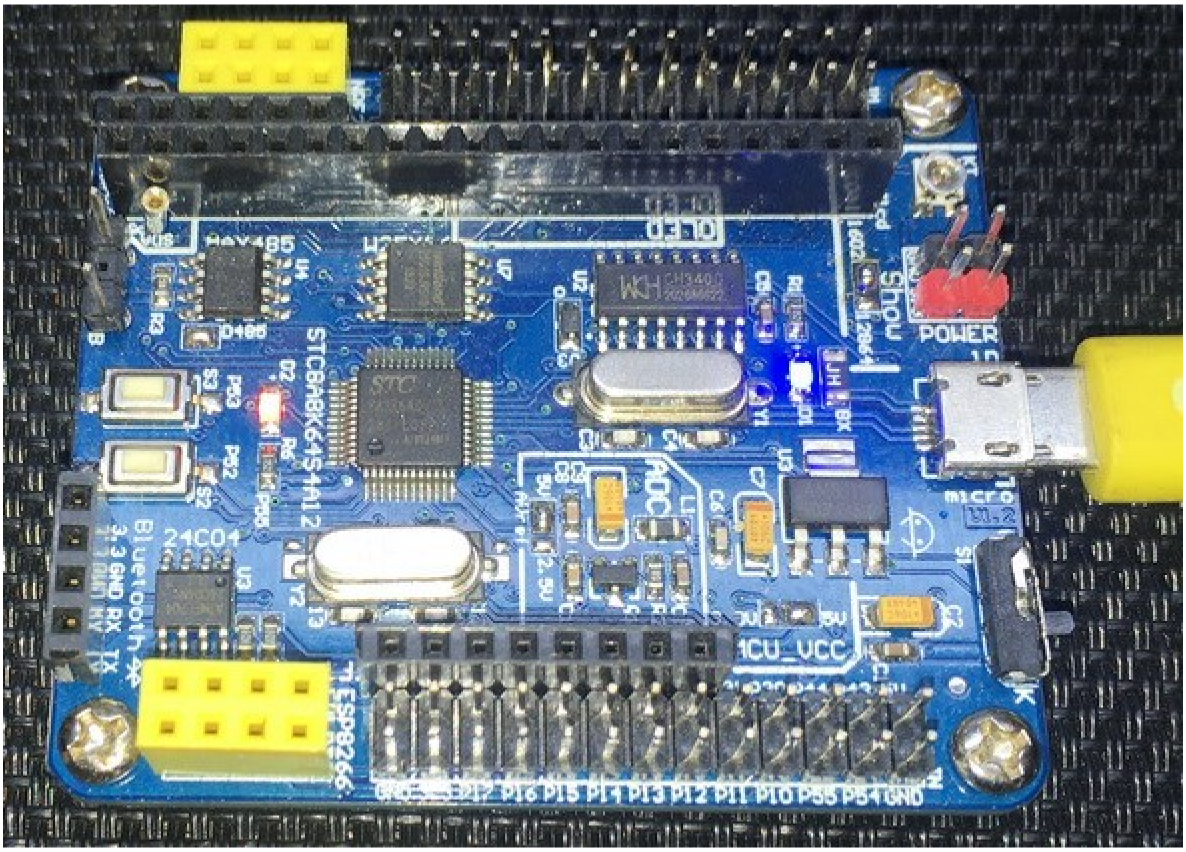
$$PCA\ Interrupt\ Interval = \ 41666\ \times 12\ \mu s = 499.99\ ms$$

Since we are toggling the onboard LED, every half second, it will change its state and so the toggling frequency is 1 Hz.



Note there is no operation in the main loop.

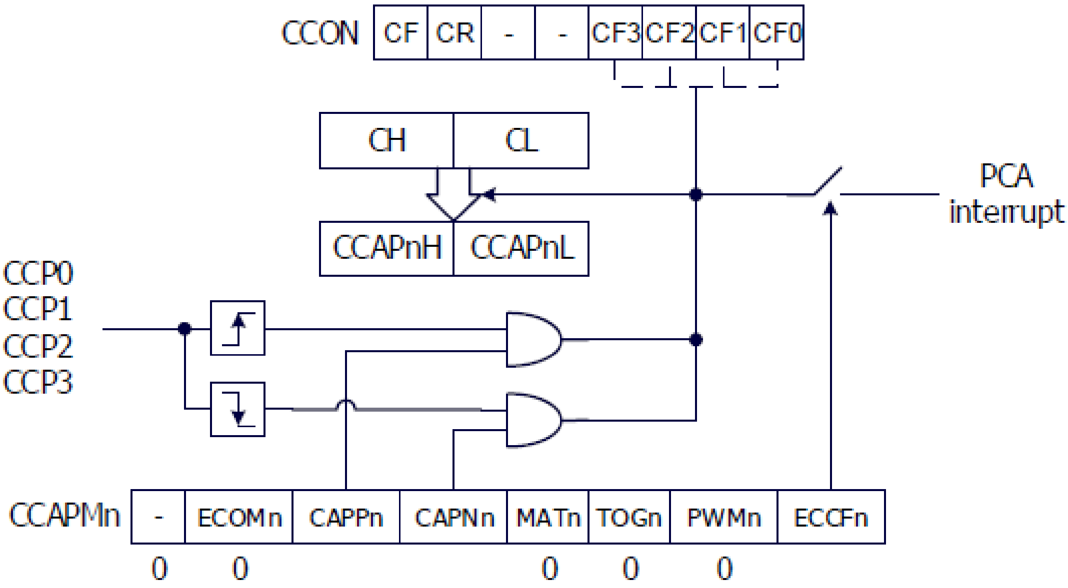
Demo



Demo video link: https://youtu.be/QuXtghib1KQ.

Using PCA to Extend External Interrupt Capability

An alternative way of using the hardware PCA module is to use it to extend external interrupt capability of our STC micro.

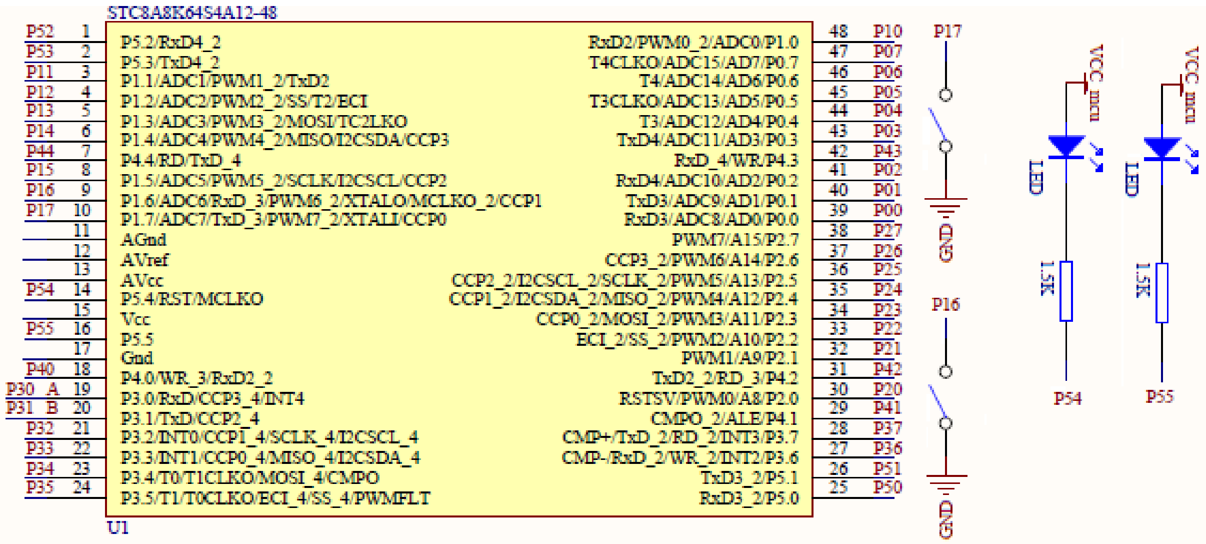


Code

```
#include "STC8xxx.h"
#include "BSP.h"


void setup(void);


void PCA_ISR(void)
interrupt 7
{
  if(check_PCA_0_flag)
  {
    clear_PCA_0_flag;
    P54_low;
  }
  if(check_PCA_1_flag)
  {
    clear_PCA_1_flag;
    P54_high;
  }
}


void main(void)
{

  setup();

  while(1)
  {
    P55_toggle;
    delay_ms(400);
  };
}
```

```
void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  P54_push_pull_mode;
  P55_open_drain_mode;

  PCA_pin_option(0x00);

  PCA_setup(PCA_continue_counting_in_idle_mode, PCA_clk_sys_clk_div_1);

  PCA_0_mode(PCA_16_bit_falling_edge_capture);
  PCA_1_mode(PCA_16_bit_falling_edge_capture);

  PCA_load_counter(0);

  _enable_PCA_0_interrupt;
  _enable_PCA_1_interrupt;
  _enable_global_interrupt;

  PCA_start_counter;
}
```

Schematic



Explanation

Perhaps this is the one and only PCA module example that doesn't require precise selection of timing parameters and so we will directly go to PCA settings. In this example, a LED connected to pin P5.4 is either turned on or off using two buttons. These buttons are attached to default PCA channel 0 and 1 pins.

```
PCA_pin_option(0x00);

PCA_setup(PCA_continue_counting_in_idle_mode, PCA_clk_sys_clk_div_1);

PCA_0_mode(PCA_16_bit_falling_edge_capture);
PCA_1_mode(PCA_16_bit_falling_edge_capture);

PCA_load_counter(0);

_enable_PCA_0_interrupt;
_enable_PCA_1_interrupt;
_enable_global_interrupt;

PCA_start_counter;
```

Both PCA channel input pins are set to capture falling edges. It is not mandatory to use falling edge capture. In this case however, I used so because the buttons have external pullup resistors and so pressing them would lead to a high-to-low pin transition or falling edge.

The whole PCA setup seem to look like PCA capture mode setup but the exception is the fact that we won't be needing the capture counts this time.

The state of the P5.4 LED is altered inside PCA interrupt subroutine and everything will appear as if the things are being done with external interrupts. Note that though this external interrupt technique works just like ordinary external interrupts, there is no way of setting interrupt priorities amongst PCA channels themselves.

```
void PCA_ISR(void)
interrupt 7
{
  if(check_PCA_0_flag)
  {
    clear_PCA_0_flag;
    P54_low;
  }
  if(check_PCA_1_flag)
  {
    clear_PCA_1_flag;
    P54_high;
  }
}
```

The only task that is done in the main loop is a simple onboard LED blinking. This is not a must but this is only needed to show that PCA operations are done without blocking anything of the main loop.

```
P55_toggle;
delay_ms(400);
```

Demo

Demo video link: https://youtu.be/w9Ys7HBVvPQ.

Watchdog Timer (WDT)

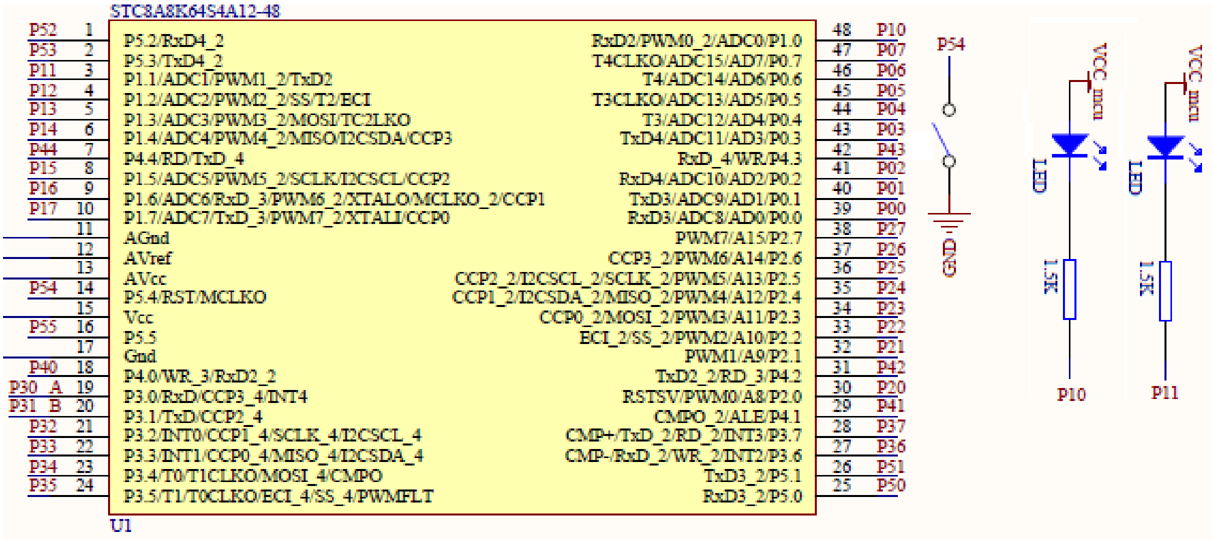
As with any other microcontroller, reset has many sources and watchdog timer is one of them. STC8A8K64S4A12's watchdog timer is a typical independent watchdog timer that will reset the MCU core in the event of an unforeseen software failure. The watchdog timer of STC8A8K64S4A12 has no alternative use. In some other microcontroller families like TI MSP430s, watchdog timer can be used like an ordinary timer.

Symbol	description	address	Bit address and symbol								reset value
			B7	B6	B5	B4	B3	B2	B1	B0	
WDT_CONTR	Watchdog control register	C1H	WDT_FLAG	-	EN_WDT	CLR_WDT	IDL_WDT	WDT_PS[2:0]			0x00,0000
IAP_CONTR	IAP control register	C7H	IAPEN	SWBS	SWRST	CMD_FAIL	-	IAP_WT[2:0]			0000,x000
RSTCFG	Reset configuration register	FFH	-Watchdog control register	ENLVR	-	P54RST	-	-	LVDS[1:0]		0000,0000

Code

```
#include "STC8xxx.h"
#include "BSP.h"


void setup(void);


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

    setup();

    while(1)
    {
        P11_toggle;

        if(P52_get_input == FALSE)
        {
            for(i = 0; i <= 9; i++)
            {
              P10_toggle;
              delay_ms(100);
            }

            while(1);
        }

        delay_ms(200);
        WDT_reset;
    };
}


void setup(void)
{
    CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

    P52_input_mode;
    P52_pull_up_enable;

    P10_push_pull_mode;
    P11_push_pull_mode;

    WDT_setup(WDT_continue_counting_in_idle_mode, WDT_div_factor_32);
}
```

Schematic



Explanation

For demoing WDT functionality, a simple LED blinking code is used. Two LEDs are used – one with pin P1.0 and the other with P1.1. P5.2 pin is used for a push button and the system clock is set at 12MHz.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

P52_input_mode;
P52_pull_up_enable;

P10_push_pull_mode;
P11_push_pull_mode;
```

STC8A8K64S4A12's WDT is dependent on system clock and this is an old concept. I say it is primitive concept because many microcontrollers nowadays use a totally independent clock for WDT. Having an independent clock for WDT reduces the chance of WDT getting affected by issues with main or system clock.

WDT is setup according to the following formula:

$$\text{Overflow time of watchdog timer} = \frac{12 \times 32768 \times 2^{(\text{WDT\_PS}+1)}}{\text{SYSclk}}$$

From the formula, we can clearly see that WDT overflow timing is dependent on WDT prescalar and system clock. The following table shows some typical values.

WDT_PS[2:0]	division factor	The overflow time of 12M in the main frequency	The overflow time of 20M in the main frequency
000	2	≈ 65.5 MS	≈ 39.3 MS
001	4	≈ 131 MS	≈ 78.6 MS
010	8	≈ 262 MS	≈ 157 MS
011	16	≈ 524 MS	≈ 315 MS
100	32	≈ 1.05 S	≈ 629 MS
101	64	≈ 2.10 S	≈ 1.26 S
110	128	≈ 4.20 S	≈ 2.52 S
111	256	≈ 8.39 S	≈ 5.03 S

In this code, a prescalar value of 32 is set with 12MHz system clock. Thus, the WDT will reset after about one second if not refreshed or reset.

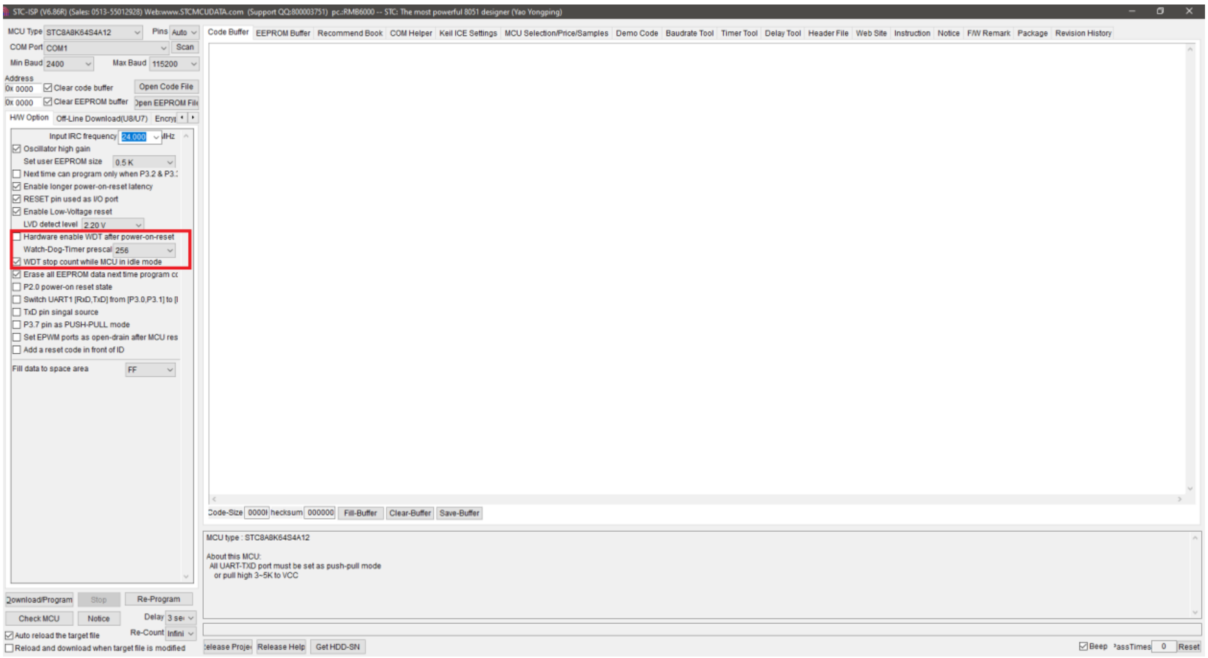
```
WDT_setup(WDT_continue_counting_in_idle_mode, WDT_div_factor_32);
```

In the main loop, a LED connected to pin P1.1 is toggled every 200ms while P1.0 LED is kept turned on. The WDT is kept resetting continuously under this condition and it does not overflow.

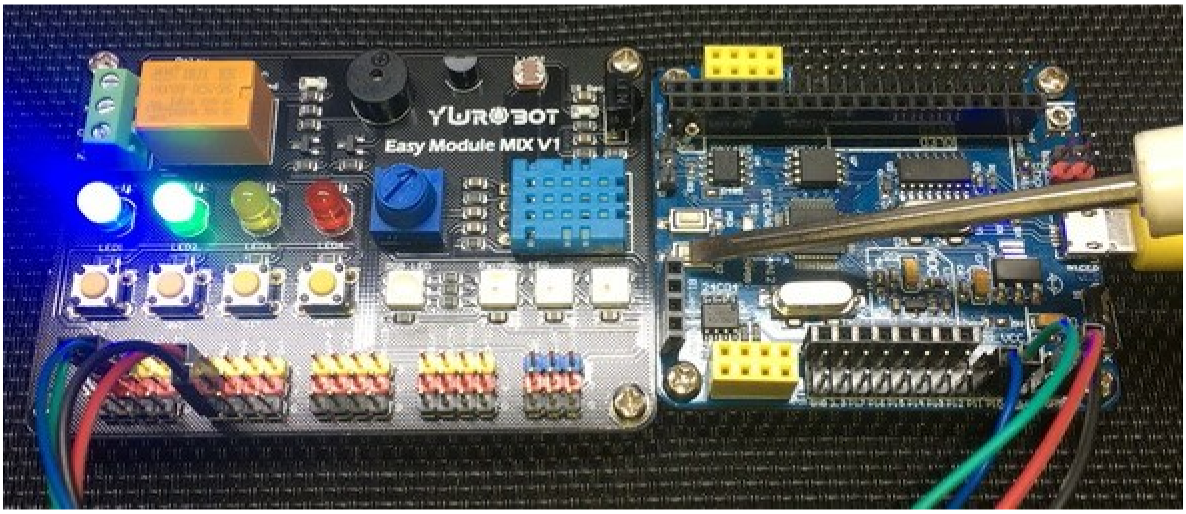
```
P11_toggle;

if(P52_get_input == FALSE)
{
    for(i = 0; i <= 9; i++)
    {
        P10_toggle;
        delay_ms(100);
    }

    while(1);
}

delay_ms(200);
WDT_reset;
```

If, however, P5.2 push button is pressed, the code enters a new loop which simulates an undesired condition. In this condition the state of P1.1 LED seems to get stuck. P1.0 LED flashes briefly, suggesting the code entered undesired loop and WDT is not being reset. Thus, after one second the CPU resets and P1.1 LED again starts to toggle. This suggests that the WDT has triggered a reset due to its overflow.

Lastly before signing off from this topic, please note the following setting in programmer GUI. These can also be set in the GUI too apart from coding.



Demo



Demo video link: https://youtu.be/0EyRSlsJx-g.

Wakeup Timer and Low Power Mode

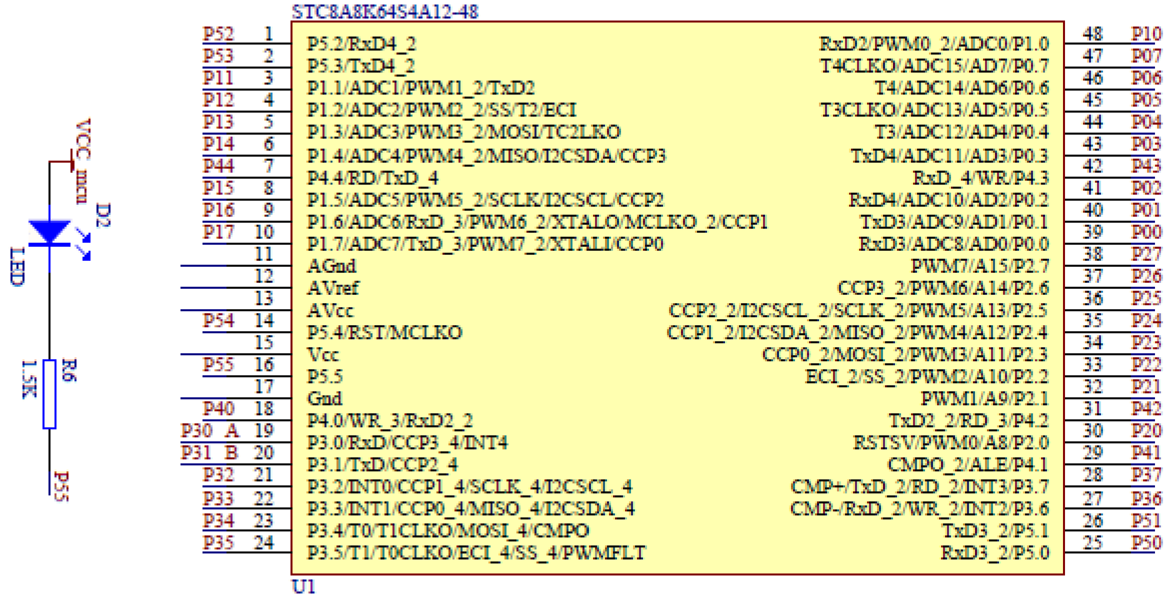
Wakeup timer is an interesting feature that is not seen in most microcontrollers. It allows us to automatically wakeup a microcontroller from sleep or low power mode after a fixed interval. It is especially useful in cases where we need to take periodic short measurements and then keep our device in low power state to conserve power. A good example is a refrigerator temperature controller. The temperature controller would periodically check interior temperature and decide when to operate the compressor for cooling. Since the compressor would work for short times, the control system of the refrigerator need not to be kept active all the times. Periodic short measurements are all that are needed. This, in turn, would help in energy saving. This tactic becomes even more vital if refrigerator of the discussion is battery-operated.

Symbol	Address	B7	B6	B5	B4	B3	B2	B1	B0
WKTCL	AAH								
WKTCH	ABH	WKTEN							

Code

```
#include "STC8xxx.h"
#include "BSP.h"


void setup(void);


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

    while(1)
    {
      WKT_disable;

      for(i = 0; i <= 15; i++)
      {
        P55_toggle;
        delay_ms(100);
      }

      WKT_enable;
      Go_to_Power_Down_State;
    };
}


void setup(void)
{
    CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

    P55_open_drain_mode;

    WKT_set_interval(9000);
}
```

Schematic



Explanation

In this example, two features of STC8A8k64S4A12 have been shown. Firstly, low power mode is demoed and secondly, the wakeup timer.

First, let's see how the wakeup timer is set. Wakeup timer is clocked with internal 32kHz oscillator. This keeps it independent from the system clock. However, the internal 32KHz oscillator is not very accurate and so there could be some deviations. To set the wakeup timer, all we need to do is to load its internal 15-bit counter with some value so that we get our desired wakeup time. The wakeup time is calculated according to the following formula:

$$\frac{1000}{32} \times 16 \times (1 + counter_{value}) \approx Wakeup\ Time\ in\ \mu s$$

or simply:

$$0.5 \times (1 + counter_{value}) \approx Wakeup\ Time\ in\ ms$$

In the code, the counter value is set to 9000. This would roughly give a wake-up time of 4.5s once the core goes to low-power state.

```
WKT_set_interval(9000);
```

Note that although the counter is loaded the wake-up time is not started. This is so because we want to wake the micro after it enters low-power mode.

In the main loop, the wake-up timer is disabled. The onboard LED is flashed a few times to indicate that the code in the main loop is running. After flashing the LED, the wake-up timer is started and low-power mode is entered.

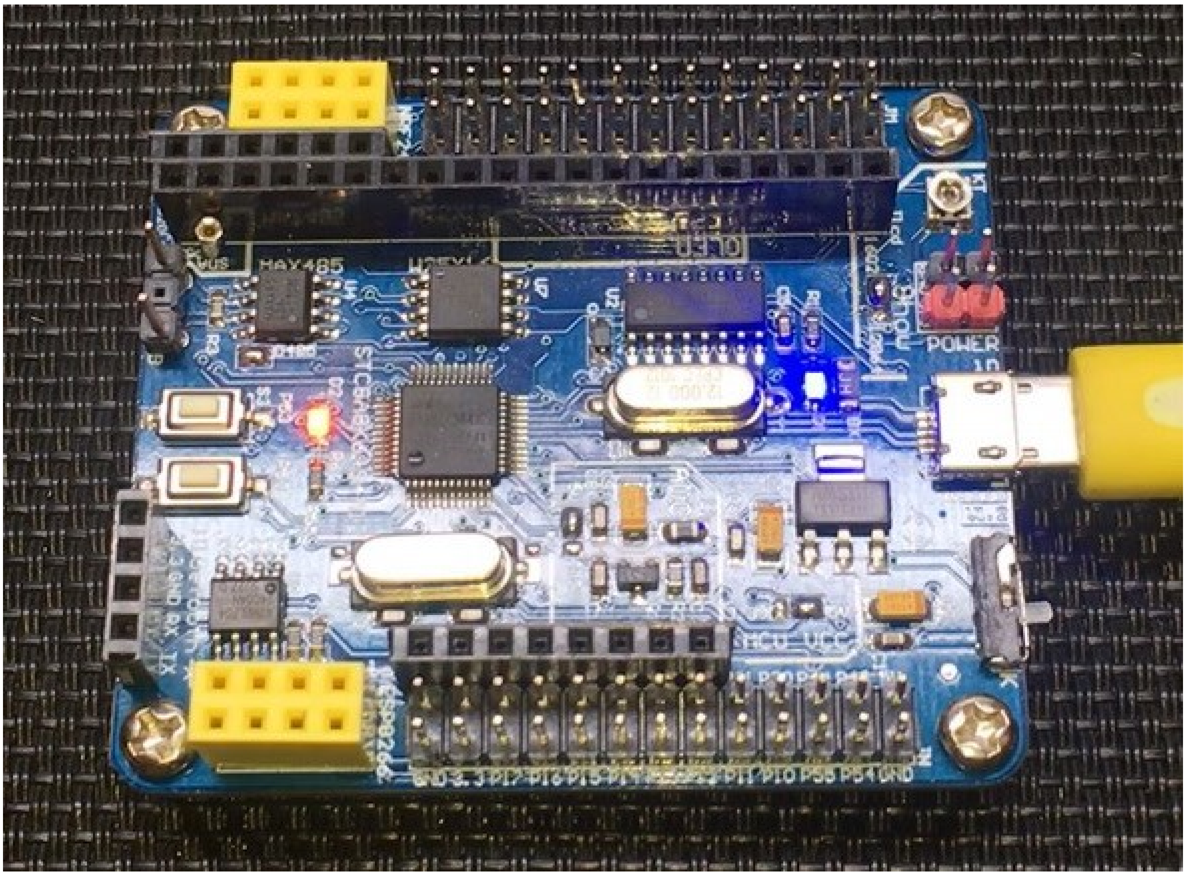
```
WKT_disable;

for(i = 0; i <= 15; i++)
{
   P55_toggle;
   delay_ms(100);
}

WKT_enable;
Go_to_Power_Down_State;
```

Entering low power mode is easy. Just add the following command:

```
Go_to_Power_Down_State;
```

After entering low-power mode, the LED seems to freeze and there seems no activity at all for some time. After about 4.5s, the LED repeats flashing just like before. This time the LED flashing not only indicates main loop code execution but it also indicates that the wake timer ran and woke up the CPU core from low-power sleep mode.

Demo



Demo video link: https://youtu.be/slPWU7c_idM.

Communication Hardware Overview

STC8A8K64S4A12 packs all kinds of common communication hardware peripheral that most modern microcontrollers can offer. The following are available in STC8A8K64S4A12's arsenal:

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	3	115.2kbps	1.2km	Several

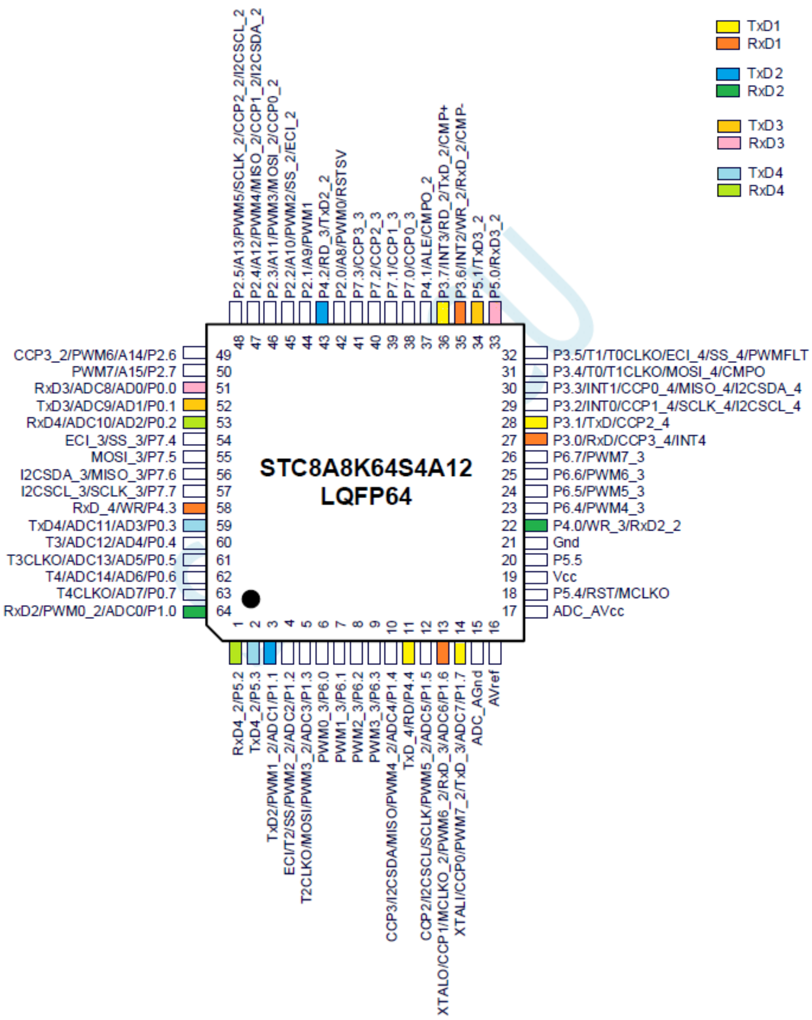
As always, we also have options for software-based communications. Using some coding, we can implement software-based SPI, I2C and other forms of communications. One wire communication is a good example. One wire communication is fully implemented in software. Using a combination of hardware and software, we can also implement methods to decode IR communication.

Though software methods will rarely be needed, these methods allow us to understand the working principle of different communication methods and in this way build confidence for coding devices that do not follow standard I2C, SPI or UART methods. The downside of software communications is their relative processing speed when compared with their hardware counterparts because software-based communications are made virtually in codes rather than in physical hardware.

I won't be discussing much about the software techniques because in all of my past tutorials, I have been discussing about them briefly and most of these stuffs are mere confirmation of implementation and repetition.

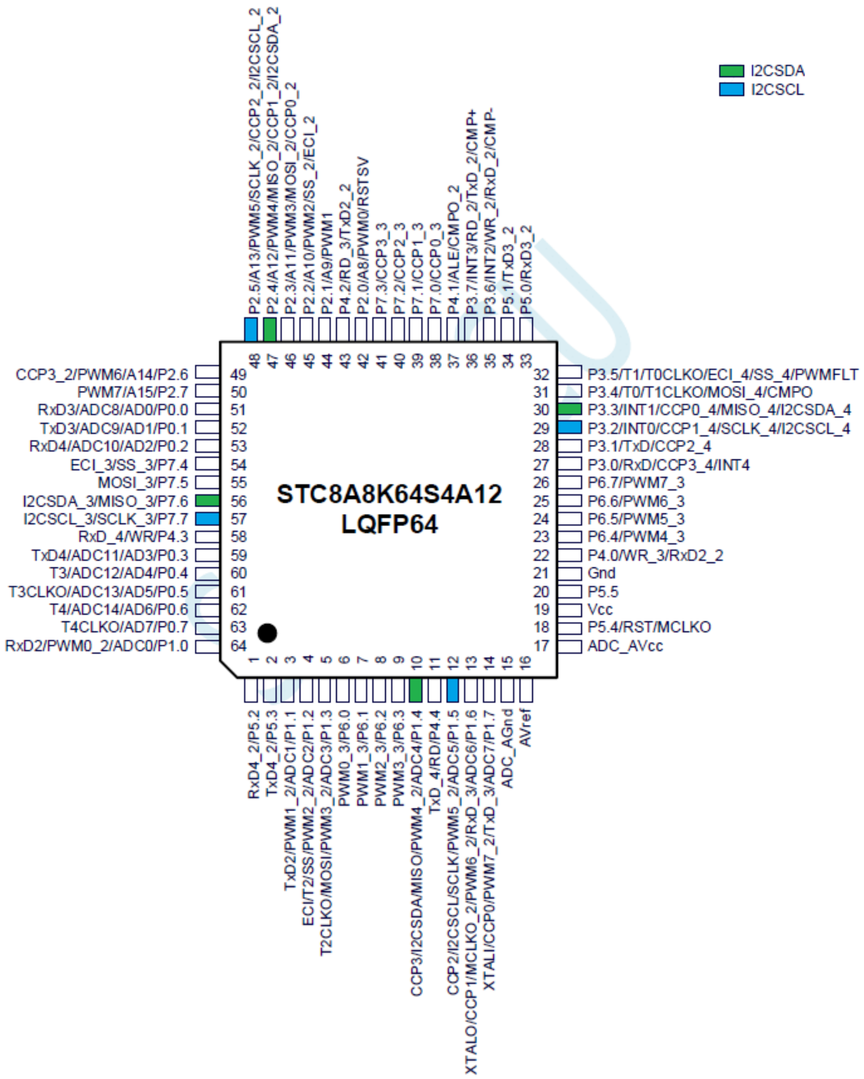
UART offers long-distance serial communication and can also be employed for simple SPI-like serial and industry-standard RS-485 communications. STC8A8K64S4A12 has four UART hardware with independent interrupts. UART1 is the most advanced UART of all. All UART hardware peripheral need a timer for baud rate generation. Timer 2 is common to all of these UARTs and so it is wise to leave it for UARTs. All of these UART have multiple alternative pins associated with them. These pin mapping is summarized below:

Comm.	Block	No of GPIOs Used	Associated Timer	Default GPIO Pin Pair	Pair	Alternative GPIO Pin Pair
UART	UART 1	2	Timer 1	TXD = P3.1	Pair 2	TXD = P3.7
						RXD = P3.6
					Pair 3	TXD = P1.7
			Timer 2	RXD = P3.0		RXD = P1.6
					Pair 4	TXD = P4.4
						RXD = P4.3
	UART2	2	Timer 2	TXD = P1.1	Pair 2	TXD = 4.2
				RXD = 1.0		RXD = 4.0
	UART3	2	Timer 2	TXD = P0.1	Pair 2	TXD = 5.1
			Timer 3	RXD = P0.0		RXD = 5.0
	UART4	2	Timer 2	TXD = P0.3	Pair 2	TXD = 5.3
			Timer 4	RXD = P0.2		RXD = 5.2



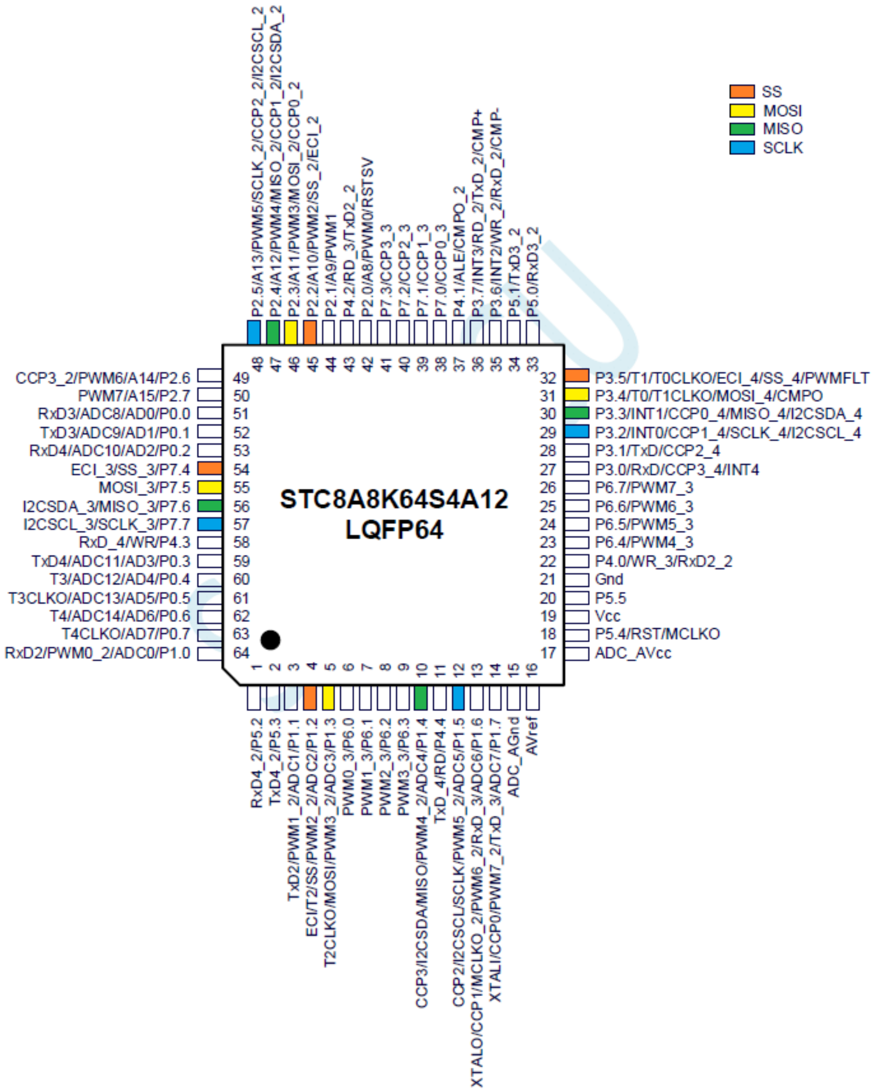
I2C hardware available in STC8A8K64S4A12 is very simple but it offers lot of flexibilities. The I2C block can be used in both I2C master or slave roles. Just like UART it has lot of alternative pin pair options as the table below shows.

Comm.	***Block***	***No of GPIOs Used***	***Default GPIO Pin Pair***	***Pair***	***Alternative GPIO Pin Pair***
I2C	Only one I2C block present	2	SDA = P1.4	Pair 2	SDA = P2.4
					SCL = P2.5
				Pair 3	SDA = P7.6
			SCL = P1.5		SCL = P7.7
				Pair 4	SDA = P3.3
					SCL = P3.2



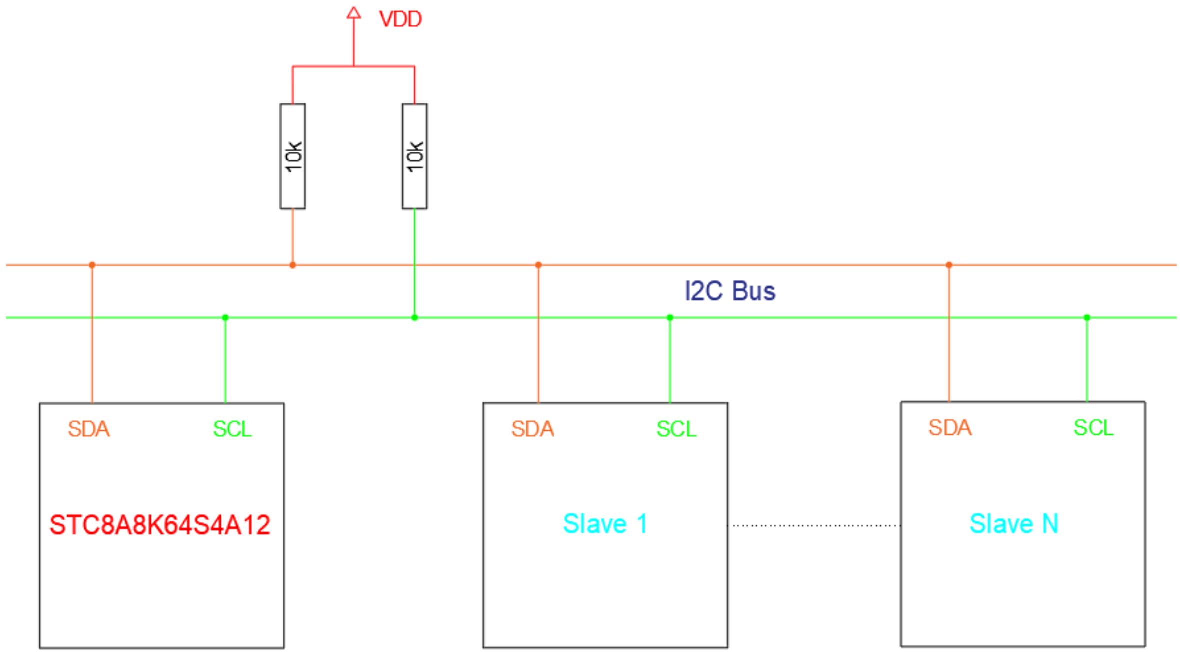
SPI hardware is also fully implemented in STC8A8K64S4A12. Using the SPI hardware is very simple and like other communication peripherals, we have options to use various alternative GPIO pin groups. Both SPI and I2C blocks are fully independent blocks unlike UARTs as UART blocks are dependent on internal hardware timers.

Comm.	*Block*	*No of GPIOs Used*	*GPIO Pin Groups*	*SS*	*MOSI*	*MISO*	*SCLK*
SPI	Only one SPI block present	3 or 4 Depending on MOSI – MISO usage	Default	P1.2	P1.3	P1.4	P1.5
			Alternate Pair 1	P2.2	P2.3	P2.4	P2.5
			Alternate Pair 2	P7.4	P7.5	P7.6	P7.7
			Alternate Pair 3	P3.5	P3.4	P3.3	P3.2



I2C

Inter Integrated Circuit (I2C) was pioneered by Philips (now NXP) about three decades ago. I2C, just like SPI, is meant for short-distance synchronous onboard communications. I2C is very simple to use as only two wires are needed for communication and this is why often I2C is alternatively called two-wire communication (TWI). In an I2C communication bus, there can be one master or host device and one or several slave devices. The maximum number of devices that can coexist in an I2C bus at a time is 127. Usually, the communication master or host is a microcontroller that is responsible for initiating communications while slave devices can be anything from another microcontroller to sensors, memory devices, digital peripherals, etc. Only master and slave data transactions are possible because only a master can request data from or write to a slave device. Slave-slave communication is not possible. In an I2C bus, only one master-slave pair can communicate at a time. The rest of the devices stay idle during this time.



To know more about I2C communication, visit 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

Code

I2C.h

```
/*
SCL         SDA         Hex             Option
P1.5        P1.4        0x00            option 1
P2.5        P2.4        0x10            option 2
P7.7        P7.6        0x20            option 3
P3.2        P3.3        0x30            option 4
*/

#define I2C_pin_option(value)                              do{P_SW2 |= value;}while(0)

//Timeout
#define I2C_timeout                                         1000


//state
#define I2C_disable                                          0x00
#define I2C_enable                                           0x80

//mode
#define I2C_slave_mode                                       0x00
#define I2C_master_mode                                      0x40


#define I2C_setup(state, mode, clk)                          do{ \
                                                                 bit_set(P_SW2, 7); \
                                                                 I2CCFG = (state | mode | (clk & 0x3F)); \
                                                                 bit_clr(P_SW2, 7); \
                                                               }while(0)

void I2C_wait(void)
{
  unsigned int t = I2C_timeout;

  while((check_I2C_master_flag == FALSE) && (t > 0))
  {
    t--;
    delay_ms(1);
  };

  clear_I2C_master_flag;
}


void I2C_start(void)
{
    bit_set(P_SW2, 7);
    I2CMSCR = 0x01;
    I2C_wait();
    bit_clr(P_SW2, 7);
}


void I2C_stop(void)
{
    bit_set(P_SW2, 7);
    I2CMSCR = 0x06;
    I2C_wait();
    bit_clr(P_SW2, 7);
}


void I2C_write(unsigned char value)
{
    bit_set(P_SW2, 7);
    I2CTXD = value;
    I2CMSCR = 0x02;
    I2C_wait();
    I2CMSCR = 0x03;
    I2C_wait();
    bit_clr(P_SW2, 7);
```

```
}


unsigned char I2C_read(unsigned char ACK_state)
{
    unsigned char value = 0x00;

    bit_set(P_SW2, 7);
    I2CMSCR = 0x04;
    I2C_wait();
    value = I2CRXD;
    I2CMSST = ~ACK_state;
    I2CMSCR = 0x05;
    I2C_wait();
    bit_clr(P_SW2, 7);

    return value;
}
```

DHT12.h

```
#define I2C_W                                   0x00
#define I2C_R                                   0x01

#define no_of_bytes_to_read                    0x05

#define DHT12_I2C_address                      0xB8

#define no_error                               0x00
#define CRC_error                              0x01


void DHT12_init(void);
unsigned char DHT12_CRC(unsigned char *array_values);
unsigned char DHT12_read_byte(unsigned char address);
unsigned char DHT12_get_data(float *DHT12_RH, float *DHT12_T);
```

DHT12.c

```
#include "DHT12.h"


void DHT12_init(void)
{
    I2C_pin_option(0x00);
    I2C_setup(I2C_enable, I2C_master_mode, 0xFF);
    delay_ms(100);
}


unsigned char DHT12_CRC(unsigned char *array_values)
{
    signed char i = 0x03;
    unsigned char crc_result = 0x00;

    while(i > -1)
    {
        crc_result += array_values[i];
        i--;
    }

    return crc_result;
}



unsigned char DHT12_read_byte(unsigned char address)
{
    unsigned char value = 0x00;

    I2C_start();
    I2C_write(DHT12_I2C_address);
    I2C_write(address);
```

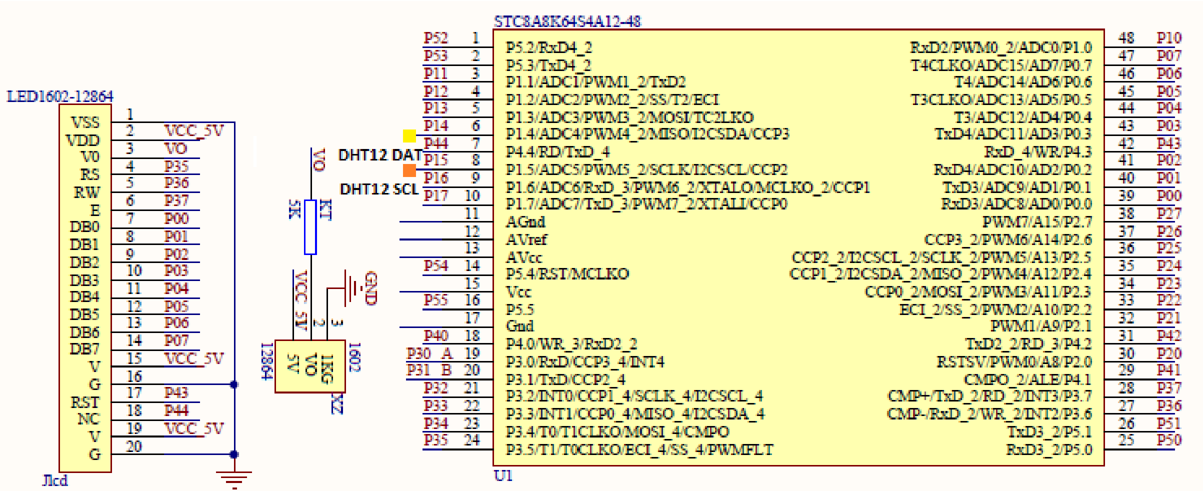
```
    I2C_start();
    I2C_write(DHT12_I2C_address | I2C_R);
    value = I2C_read(0);
    I2C_stop();

    return value;
}



unsigned char DHT12_get_data(float *DHT12_RH, float *DHT12_T)
{
    signed char i = no_of_bytes_to_read;

    unsigned char values[0x05] = {0x00, 0x00, 0x00, 0x00, 0x00};

    while(i > 0x00)
    {
        values[(no_of_bytes_to_read - i)] = DHT12_read_byte((no_of_bytes_to_read - i));
        i--;
    };

    if(values[0x04] == DHT12_CRC(values))
    {
        *DHT12_RH = (((float)values[0x00]) + (((float)values[0x01]) * 0.1));
        *DHT12_T = (((float)values[0x02]) + (((float)values[0x03]) * 0.1));

        return no_error;
    }

    else
    {
        return CRC_error;
    }
}
```

main.c

```
#include "STC8xxx.h"
#include "BSP.h"
#include "LCD.c"
#include "lcd_print.c"
#include "DHT12.c"


void setup(void);


void main(void)
{
  unsigned char state = 0x00;

    float T = 0.0;
    float RH = 0.0;

    setup();

    LCD_goto(0, 0);
    LCD_putstr("R.H / %:");
    LCD_goto(0, 1);
    LCD_putstr("Temp/ C:");
    print_symbol(5, 1, 0);

    while(1)
    {
        state = DHT12_get_data(&RH, &T);

            switch(state)
            {
                case no_error:
                {
                    print_F(11, 0, RH, 1);
          print_F(11, 1, T, 1);
                    break;
                }
```

```
                default:
                {
                    LCD_goto(12, 0);
                    LCD_putstr("--.-");
                    LCD_goto(12, 1);
                    LCD_putstr("--.-");
                }
            }

            P55_toggle;
            delay_ms(400);
    };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  P55_open_drain_mode;

  LCD_init();
  LCD_clear_home();
  load_custom_symbol();

  DHT12_init();
}
```

Schematic



Explanation

I2C.h header file describes how the I2C hardware will work. Firstly, I2C hardware pin pairs can be any of the following combinations:

```
/*
SCL         SDA         Hex             Option
P1.5        P1.4        0x00            option 1
P2.5        P2.4        0x10            option 2
P7.7        P7.6        0x20            option 3
P3.2        P3.3        0x30            option 4
*/

#define I2C_pin_option(value)                         do{P_SW2 |= value;}while(0)
```

Secondly, we have the following I2C library functions that control I2C-related registers. Mainly, we would need these functions in order to properly use the I2C hardware. These the commonly used ones. Details of register handling can be avoided in this way.

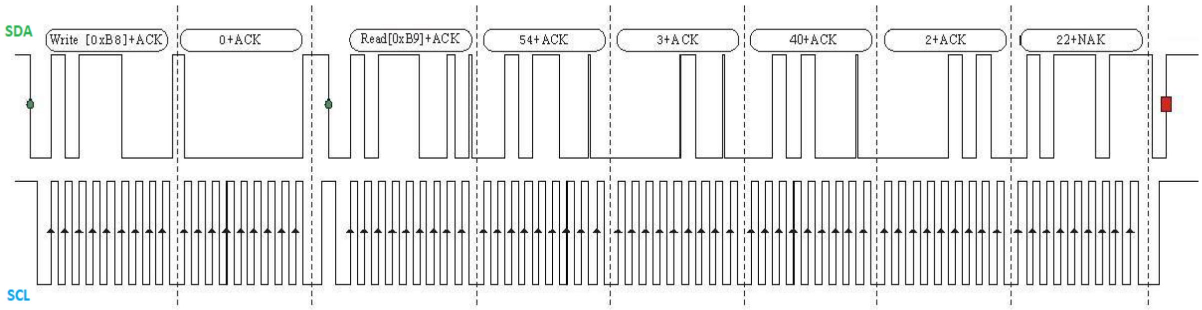
```
void I2C_start(void)
void I2C_stop(void)
void I2C_write(unsigned char value)
unsigned char I2C_read(unsigned char ACK_state)
```

I have avoided interrupt-based I2C communication because I2C slave devices generally do not send out data without request from master device and master device only initiates data transaction when needed. There is no need to make things unnecessarily complicated.

Before using I2C hardware, we have to set it up and ***I2C_setup*** function exactly does that. It dictates mode of operation and I2C bus clock speed.

```
I2C_setup(I2C_enable, I2C_master_mode, 0xFF);
```

For demoing I2C communication, Aosong's DHT12 relative humidity-temperature sensor is used. This device is pretty simple to use.



The timing diagram shown above depicts that the entire sequence of data transfer is a typical I2C read. However, to simplify the task of reading the sensor, I coded the read function to read the I2C bus in byte format rather than in word format. The reading process is a usual I2C read.

```
unsigned char DHT12_read_byte(unsigned char address)
{
    unsigned char value = 0x00;

    I2C_start();
    I2C_write(DHT12_I2C_address);
    I2C_write(address);

    I2C_start();
    I2C_write(DHT12_I2C_address | I2C_R);
    value = I2C_read(0);
    I2C_stop();

    return value;
}
```

Since we have to take care of Cyclic Redundancy Check (CRC) in order to ensure data integrity several byte reads are done and the read data are stored in an array. The array consists of two bytes of relative humidity data followed by two bytes of temperature data and a CRC byte - five bytes in total. After getting all the values from the sensor, CRC check is performed. If there is no CRC error, the temperature and relative humidity data are processed or else CRC error is flagged.

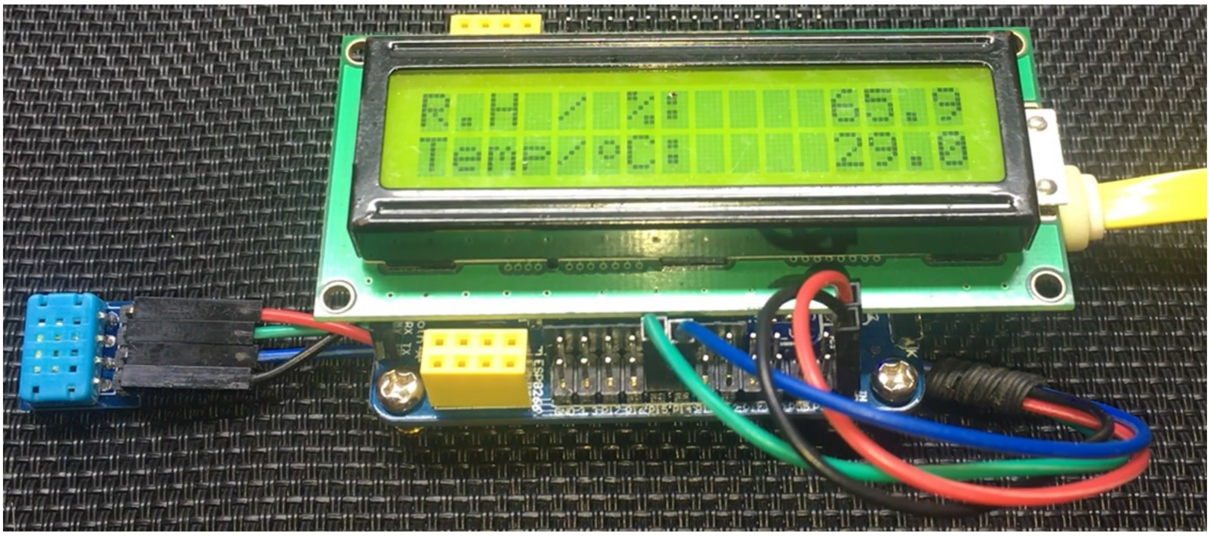
```
unsigned char DHT12_get_data(float *DHT12_RH, float *DHT12_T)
{
    signed char i = no_of_bytes_to_read;
    unsigned char values[0x05] = {0x00, 0x00, 0x00, 0x00, 0x00};

    while(i > 0x00)
    {
        values[(no_of_bytes_to_read - i)] = DHT12_read_byte((no_of_bytes_to_read - i));
        i--;
    };

    if(values[0x04] == DHT12_CRC(values))
    {
        *DHT12_RH = (((float)values[0x00]) + (((float)values[0x01]) * 0.1));
        *DHT12_T = (((float)values[0x02]) + (((float)values[0x03]) * 0.1));

        return no_error;
    }
    else
    {
        return CRC_error;
    }
}
```

DH12's CRC check is straight-forward. The temperature and humidity bytes are summed up and checked against the sent-out CRC value. If the values are same then there is no error in sent data.

```
unsigned char DHT12_CRC(unsigned char *array_values)
{
    signed char i = 0x03;
    unsigned char crc_result = 0x00;

    while(i > -1)
    {
        crc_result += array_values[i];
        i--;
    }

    return crc_result;
}
```

In the main, the sensor is read and the temperature-relative humidity data are displayed on an LCD display.

Demo



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

SPI

Serial Peripheral Interface (SPI) communication is another short-distance synchronous communication method like I2C. Unlike I2C, SPI requires 3 or more connections but the bus speed is significantly higher than that of I2C's. Except chip selection pin, all SPI devices in a SPI bus can share the same set of communication pins. Typical full-duplex SPI bus requires four basic I/O pins. Their naming suggests their individual purpose.

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

Since SPI bus is very fast, it is widely used for interfacing displays, flash memories, etc. that needs fast responsiveness.

SPI is can visualized as a shift register that shifts data in and out one-by-one with clock pulse transitions. Like I2C bus, there can be one master device in a SPI bus and virtually unlimited amount of slave devices only separated by slave selection pins. Master sends commands to slave(s) by generating clock signals and selecting one slave device at a time. Selected slave responds to commands sent by the master.

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

SPI.h

```
/*
SS          MOSI        MISO        SCLK    Hex         Option
P1.2        P1.3        P1.4        P1.5    0x00        option 1
P2.2        P2.3        P2.4        P2.5    0x04        option 2
P7.4        P7.5        P7.6        P7.7    0x08        option 3
P3.5        P3.4        P3.3        P3.2    0x0C        option 4
*/

#define SPI_clear                                                           do{\
                                                                                SPCTL = 0x00; \
                                                                                SPSTAT = 0xC0; \
                                                                                SPDAT = 0x00;}while(0)


#define SPI_timeout                                                         300


#define SPI_pin_option(value)                                               do{P_SW1 |= value;}while(0)


#define SPI_pins(value)

#define SPI_disable                                                         bit_clr(SPCTL, 6)
#define SPI_enable                                                          bit_set(SPCTL, 6)

//clk
#define SPI_clk_sysclk_div_4                                                0x00
#define SPI_clk_sysclk_div_8                                                0x01
#define SPI_clk_sysclk_div_16                                               0x02
#define SPI_clk_sysclk_div_32                                               0x03

//MS
#define SPI_slave                                                           0x00
#define SPI_master                                                          0x10

//data_orientation
#define SPI_MSB_first                                                       0x00
#define SPI_LSB_first                                                       0x20

//CPHA
#define SPI_CPHA_leading                                                    0x00
#define SPI_CPHA_trailing                                                   0x04

//CPOL
#define SPI_CLK_CPOL_idle_low                                               0x00
#define SPI_CLK_CPOL_idle_high                                              0x08

//ss
#define SPI_SS_not_ignored                                                  0x00
#define SPI_SS_ignored                                                      0x80



#define SPI_setup(clk, MS, data_orientation, CPHA, CPOL, ss)                do{ \
                                                                               SPI_clear; \
                                                                               SPCTL |= (clk | MS \
                                                                                        | data_orientation \
                                                                                        | CPHA \
                                                                                        | CPOL \
                                                                                        | ss); \
                                                                              }while(0)


unsigned char SPI_transfer(unsigned char write_value)
{
  unsigned char read_value = 0x00;
  unsigned int timeout = SPI_timeout;
```

```
  SPDAT = write_value;
  while((!check_SPI_flag) && (timeout > 0))
  {
    timeout--;
    delay_ms(1);
  };

  clear_SPI_flag;
  clear_SPI_write_collision_flag;

  read_value = SPDAT;

  return read_value;
}
```

DS3234.h

```
#define read_cmd                      0x00
#define write_cmd                     0x80

#define second_reg                    0x00
#define minute_reg                    0x01
#define hour_reg                      0x02
#define day_reg                       0x03
#define date_reg                      0x04
#define month_reg                     0x05
#define year_reg                      0x06
#define alarm1sec_reg                 0x07
#define alarm1min_reg                 0x08
#define alarm1hr_reg                  0x09
#define alarm1date_reg                0x0A
#define alarm2min_reg                 0x0B
#define alarm2hr_reg                  0x0C
#define alarm2date_reg                0x0D
#define control_reg                   0x0E
#define status_reg                    0x0F
#define ageoffset_reg                 0x10
#define tempMSB_reg                   0x11
#define tempLSB_reg                   0x12
#define tempdisable_reg               0x13
#define sramaddr_reg                  0x18
#define sramdata_reg                  0x19

#define am                             0
#define pm                             1
#define hr24                           0
#define hr12                           1

#define DS3234_SS_HIGH                 P12_high
#define DS3234_SS_LOW                  P12_low


unsigned char s = 10;
unsigned char min = 10;
unsigned char hr = 10;
unsigned char ampm = pm;
unsigned char dy = 1;
unsigned char dt = 1;
unsigned char mt = 1;
unsigned char yr = 1;
unsigned char hr_format = hr12;


unsigned char bcd_to_decimal(unsigned char d);
unsigned char decimal_to_bcd(unsigned char d);
void DS3234_init(void);
unsigned char DS3234_read(unsigned char addr);
void DS3234_write(unsigned char addr, unsigned char value);
float DS3234_get_temp(void);
void DS3234_write_SRAM(unsigned char addr, unsigned char value);
unsigned char DS3234_read_SRAM(unsigned char addr);
void DS3234_get_time(unsigned short hour_format);
void DS3234_get_date(void);
void DS3234_set_time(unsigned char hSet, unsigned char mSet, unsigned char sSet, unsigned char am_pm_state, u
nsigned char hour_format);
```

```
void DS3234_set_date(unsigned char daySet, unsigned char dateSet, unsigned char monthSet, unsigned char yearS
et);
```

DS3234.c

```
#include "DS3234.h"


unsigned char bcd_to_decimal(unsigned char d)
{
  return ((d & 0x0F) + (((d & 0xF0) >> 4) * 10));
}


unsigned char decimal_to_bcd(unsigned char d)
{
  return (((d / 10) << 4) & 0xF0) | ((d % 10) & 0x0F);
}


void DS3234_init(void)
{
  P12_push_pull_mode;
  DS3234_SS_HIGH;

  SPI_pin_option(0x00);

  SPI_setup(SPI_clk_sysclk_div_16, \
            SPI_master, \
            SPI_MSB_first, \
            SPI_CPHA_trailing, \
            SPI_CLK_CPOL_idle_low, \
            SPI_SS_ignored);

  SPI_enable;

  DS3234_write(control_reg, 0x20);
  DS3234_write(status_reg, 0x48);
  DS3234_set_time(hr, min, s, ampm, hr_format);
  DS3234_set_date(dy, dt, mt, yr);
}


unsigned char DS3234_read(unsigned char addr)
{
  unsigned char value = 0;

  DS3234_SS_LOW;
  SPI_transfer(addr | read_cmd);
  value = SPI_transfer(0x00);
  DS3234_SS_HIGH;

  return value;
}


void DS3234_write(unsigned char addr, unsigned char value)
{
  unsigned long temp = 0;

  DS3234_SS_LOW;
  temp = (addr | write_cmd);
  SPI_transfer(temp);
  SPI_transfer(value);
  DS3234_SS_HIGH;
}


float DS3234_get_temp(void)
{
  float t = 0;
  signed char highByte = 0;
  unsigned char lowByte = 0;

  highByte = DS3234_read(tempMSB_reg);
```

```
  lowByte = DS3234_read(tempLSB_reg);

  lowByte >>= 6;
  t = (lowByte & 0x03);
  t *= 0.25;
  t += highByte;

  return t;
}


void DS3234_write_SRAM(unsigned char addr, unsigned char value)
{
  DS3234_write(sramaddr_reg, addr);
  DS3234_write(sramdata_reg, value);
}


unsigned char DS3234_read_SRAM(unsigned char addr)
{
  unsigned char value = 0;

  DS3234_SS_LOW;
  SPI_transfer(sramaddr_reg);
  SPI_transfer(addr);
  value = DS3234_read(sramdata_reg);
  DS3234_SS_HIGH;

  return value;
}


void DS3234_get_time(unsigned short hour_format)
{
  unsigned char tmp = 0;

  s = DS3234_read(second_reg);
  s = bcd_to_decimal(s);
  min = DS3234_read(minute_reg);
  min = bcd_to_decimal(min);
  tmp = DS3234_read(hour_reg);

  switch(hour_format)
  {
    case hr12:
    {
       ampm = (tmp & 0x20);
       ampm >>= 5;
       hr = (tmp & 0x1F);
       hr = bcd_to_decimal(hr);
       break;
    }

    default:
    {
       hr = (0x3F & tmp);
       hr = bcd_to_decimal(hr);
       break;
    }
  }
}


void DS3234_get_date(void)
{
  yr = DS3234_read(year_reg);
  yr = bcd_to_decimal(yr);
  mt = (0x1F & DS3234_read(month_reg));
  mt = bcd_to_decimal(mt);
  dt = (0x3F & DS3234_read(date_reg));
  dt = bcd_to_decimal(dt);
  dy = (0x07 & DS3234_read(day_reg));
  dy = bcd_to_decimal(dy);
}
```

```
void DS3234_set_time(unsigned char hSet, unsigned char mSet, unsigned char sSet, unsigned char am_pm_state, u
nsigned char hour_format)
{
  unsigned char tmp = 0;

  DS3234_write(second_reg, (decimal_to_bcd(sSet)));
  DS3234_write(minute_reg, (decimal_to_bcd(mSet)));

  switch(hour_format)
  {
    case hr12:
    {
       switch(am_pm_state)
       {
          case pm:
          {
               tmp = 0x20;
               break;
          }

          default:
          {
               tmp = 0x00;
               break;
          }
       }

       DS3234_write(hour_reg, ((tmp | 0x40 | (0x1F & (decimal_to_bcd(hSet))))));
       break;
    }

    default:
    {
       DS3234_write(hour_reg, (0xBF & (0x3F & (decimal_to_bcd(hSet)))));
       break;
    }
  }
}


void DS3234_set_date(unsigned char daySet, unsigned char dateSet, unsigned char monthSet, unsigned char yearS
et)
{
  DS3234_write(day_reg, (decimal_to_bcd(daySet)));
  DS3234_write(date_reg, (decimal_to_bcd(dateSet)));
  DS3234_write(month_reg, (decimal_to_bcd(monthSet)));
  DS3234_write(year_reg, (decimal_to_bcd(yearSet)));
}
```

main.c

```
#include "STC8xxx.h"
#include "BSP.h"
#include "LCD.c"
#include "lcd_print.c"
#include "DS3234.c"


#define SET           1
#define INC           2
#define SAVE          3

#define keypad_dly    10
#define disp_dly      90


void setup(void);
unsigned char get_keypad(void);
void display_value(unsigned char y_pos, unsigned char x_pos, unsigned char value);
void show_temperature();
void display_time(void);
void get_date_time_data(void);
unsigned char change_value(unsigned char y_pos, unsigned char x_pos, unsigned char value, unsigned char max_v
alue, unsigned char min_value);
void setting(void);
```

```
void show_day(unsigned char value);


void main(void)
{
  setup();


  while(1)
  {
    P55_toggle;
    setting();
    get_date_time_data();
    show_temperature();
    display_time();
    delay_ms(200);
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  P40_input_mode;
  P41_input_mode;
  P42_input_mode;

  P55_open_drain_mode;

  DS3234_init();

  LCD_init();
  LCD_clear_home();
  load_custom_symbol();
}


unsigned char get_keypad()
{
  if(!P40_get_input)
  {
    delay_ms(keypad_dly);
    return SET;
  }

  else if(!P41_get_input)
  {
    delay_ms(keypad_dly);
    return INC;
  }

  else if(!P42_get_input)
  {
    delay_ms(keypad_dly);
    return SAVE;
  }

  else
  {
    return 0;
  }
}


void display_value(unsigned char y_pos, unsigned char x_pos, unsigned char value)
{
  unsigned char ch = 0x00;

  ch = (value / 10);
  LCD_goto((x_pos - 1), (y_pos - 1));
  LCD_putchar((ch + 0x30));
  ch = (value % 10);
  LCD_goto(x_pos, (y_pos - 1));
  LCD_putchar((ch + 0x30));
}
```

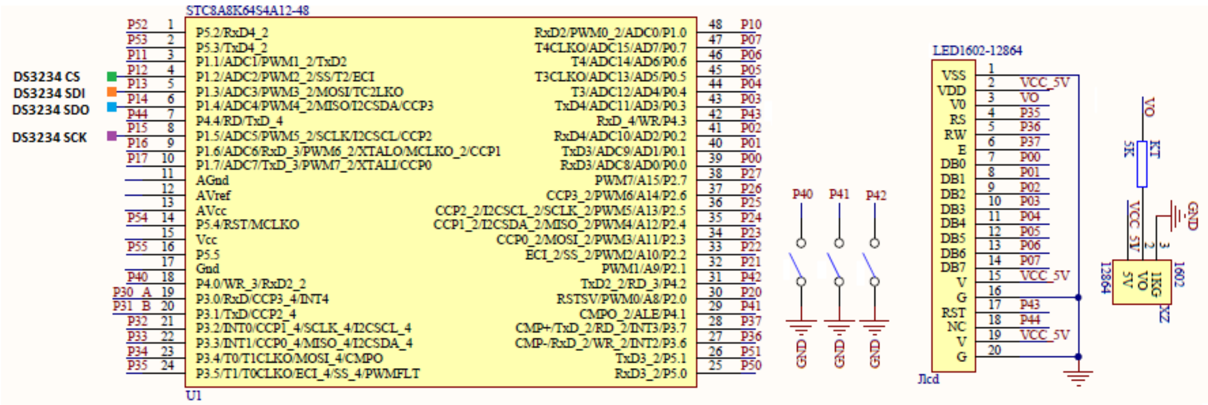
```
void display_time()
{
  display_value(1, 1, hr);
  LCD_goto(2, 0);
  LCD_putstr(":");
  display_value(1, 4, min);
  LCD_goto(5, 0);
  LCD_putstr(":");

  display_value(1, 7, s);

  switch(hr_format)
  {
    case hr12:
    {
      switch(ampm)
      {
        case pm:
        {
          LCD_goto(9, 0);
          LCD_putstr("PM");
          break;
        }
        default:
        {
          LCD_goto(9, 0);
          LCD_putstr("AM");
          break;
        }
      }
      break;
    }
    default:
    {
      LCD_goto(9, 0);
      LCD_putstr("  ");
      break;
    }
  }

  display_value(2, 1, dt);
  LCD_goto(2, 1);
  LCD_putstr("/");
  display_value(2, 4, mt);
  LCD_goto(5, 1);
  LCD_putstr("/");
  display_value(2, 7, yr);
  show_day(dy);
}


void show_temperature()
{
  float t = 0;
  unsigned char temp = 0;

  t = DS3234_get_temp();

  temp = t;
  display_value(2, 10, temp);
  LCD_goto(11, 1);
  LCD_putstr(".");
  temp = ((t - temp) * 100);
  display_value(2, 13, temp);
  print_symbol(14, 1, 0);
  LCD_goto(15, 1);
  LCD_putstr("C");
}


void get_date_time_data()
{
  DS3234_get_date();
  DS3234_get_time(hr_format);
}
```

```
unsigned char change_value(unsigned char y_pos, unsigned char x_pos, unsigned char value, unsigned char max_v
alue, unsigned char min_value)
{
  while(TRUE)
  {
    if(get_keypad() == INC)
    {
      value++;
    }

    if(value > max_value)
    {
      value = min_value;
    }

    LCD_goto((x_pos - 1), (y_pos - 1));
    LCD_putstr("  ");
    delay_ms(disp_dly);

    display_value(y_pos, x_pos, value);
    delay_ms(disp_dly);

    if(get_keypad() == SAVE)
    {
      while(get_keypad() == SAVE);
      return value;
    }
  }
}


void setting()
{
     bit set_cmd;
     unsigned char tmp = 0;

     set_cmd = 0;
     if(get_keypad() == SET)
     {
         while(get_keypad() == SET);
         set_cmd = 1;
     }

     while(set_cmd)
     {
       while(1)
       {
         if(get_keypad() == INC)
         {
           tmp++;
         }
         if(tmp > 2)
         {
           tmp = 0;
         }


         LCD_goto(9, 0);
         LCD_putstr(" ");

         delay_ms(disp_dly);
         switch(tmp)
         {
            case 1:
            {
              LCD_goto(9, 0);
              LCD_putstr("AM");
              ampm = am;
              hr_format = hr12;
              break;
            }

            case 2:
            {
              LCD_goto(9, 0);
              LCD_putstr("PM");
              ampm = pm;
              hr_format = hr12;
              break;
```

```
            }

            default:
            {
              LCD_goto(9, 0);
              LCD_putstr("24");
              hr_format = hr24;
              break;
            }
          }

          delay_ms(disp_dly);

          if(get_keypad() == SAVE)
          {
            break;
          }
         }
         s = change_value(1, 7, s, 59, 0);
         min = change_value(1, 4, min, 59, 0);

         switch(hr_format)
         {
           case hr12:
           {
             hr = change_value(1, 1, hr, 12, 1);
             break;
           }
           default:
           {
             hr = change_value(1, 1, hr, 23, 0);
             break;
           }
         }

         dt = change_value(2, 1, dt, 31, 1);
         mt = change_value(2, 4, mt, 12, 1);
         yr = change_value(2, 7, yr, 99, 0);

         tmp = dy;
         while(1)
         {
           if(get_keypad() == INC)
           {
             tmp++;
           }
           if(tmp > 7)
           {
             tmp = 1;
           }


           LCD_goto(13, 0);
           LCD_putstr("    ");
           delay_ms(disp_dly);
           show_day(tmp);
           delay_ms(disp_dly);

           if(get_keypad() == SAVE)
           {
             dy = tmp;
             break;
           }
         }

         DS3234_set_time(hr, min, s, ampm, hr_format);
         DS3234_set_date(dy, dt, mt, yr);
         set_cmd = 0;
     }
}


void show_day(unsigned char value)
{
  LCD_goto(13, 0);

  switch(value)
  {
    case 1:
```

```
    {
      LCD_putstr("SUN");
      break;
    }
    case 2:
    {
      LCD_putstr("MON");
      break;
    }
    case 3:
    {
      LCD_putstr("TUE");
      break;
    }
    case 4:
    {
      LCD_putstr("WED");
      break;
    }
    case 5:
    {
      LCD_putstr("THR");
      break;
    }
    case 6:
    {
      LCD_putstr("FRI");
      break;
    }
    case 7:
    {
      LCD_putstr("SAT");
      break;
    }
  }
}
```

Schematic



Explanation

As with other hardware, the built-in SPI hardware also has many alternative pin options. The SPI header file describes these pin combinations.

```
/*
SS          MOSI        MISO        SCLK    Hex         Option
P1.2        P1.3        P1.4        P1.5    0x00        option 1
P2.2        P2.3        P2.4        P2.5    0x04        option 2
P7.4        P7.5        P7.6        P7.7    0x08        option 3
P3.5        P3.4        P3.3        P3.2    0x0C        option 4
*/

#define SPI_pin_option(value)                                      do{P_SW1 |= value;}while(0)
```

To use SPI peripheral, we have to set it up first as shown below:

```
SPI_setup(SPI_clk_sysclk_div_16, \
          SPI_master, \
          SPI_MSB_first, \
          SPI_CPHA_trailing, \
          SPI_CLK_CPOL_idle_low, \
          SPI_SS_ignored);

SPI_enable;
```

SPI_setup function sets up SPI bus clock, device role, data sequence, clock phase and polarity alongside built-in hardware slave select pin operation state. After having these set up, the SPI hardware is ready to be enabled.

Since SPI functions like a ring shift-register, only one function is needed for SPI bus data transactions. The following ***SPI_transfer*** function is used for this purpose. First data is written and then read. Reading and writing *SPDAT* register automatically generates SPI bus clock.

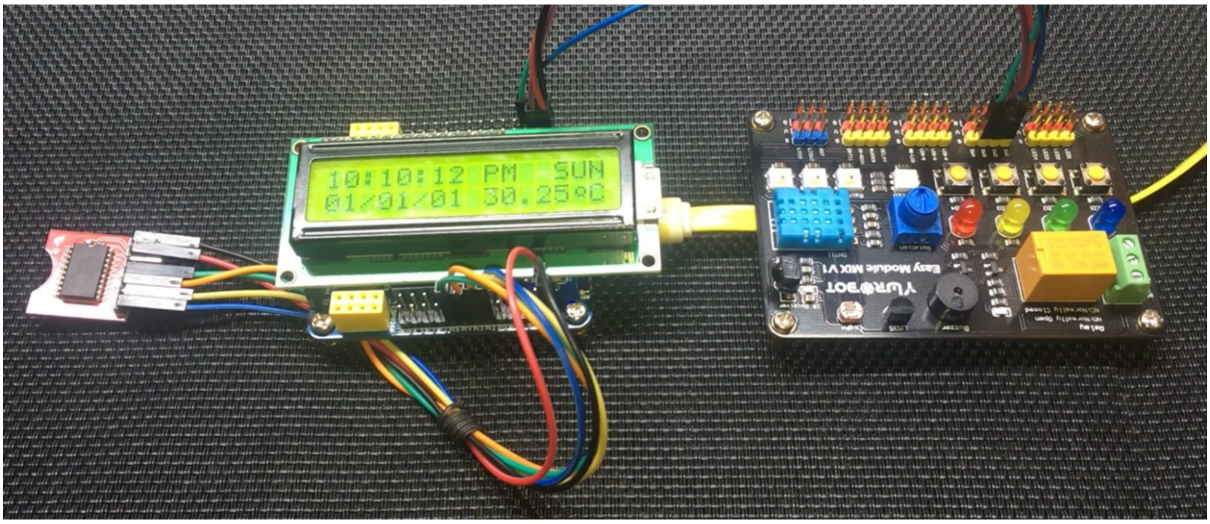
```
unsigned char SPI_transfer(unsigned char write_value)
{
  unsigned char read_value = 0x00;
  unsigned int timeout = SPI_timeout;

  SPDAT = write_value;
  while((!check_SPI_flag) && (timeout > 0))
  {
    timeout--;
    delay_ms(1);
  };

  clear_SPI_flag;
  clear_SPI_write_collision_flag;

  read_value = SPDAT;

  return read_value;
}
```

SPI peripheral is demoed here using DS3234 Real-Time Clock (RTC). The first three functions are what we basically need to fully make the RTC communicate with our STC micro. The rest of the RTC library functions are extensions of the DS3234 read and write functions. The read and write functions are typically what we see with other SPI devices. I recommend readers to check the timing diagrams of DS3234 from its datasheet and check the code in order to properly realize the SPI working principle in conjunction with the code presented here. Note that since the hardware slave select pin is not used, slave selection is done in code by changing the state of a GPIO pin.

```
void DS3234_init(void)
{
  P12_push_pull_mode;
  DS3234_SS_HIGH;

  SPI_pin_option(0x00);

  SPI_setup(SPI_clk_sysclk_div_16, \
            SPI_master, \
            SPI_MSB_first, \
            SPI_CPHA_trailing, \
            SPI_CLK_CPOL_idle_low, \
            SPI_SS_ignored);

  SPI_enable;

  DS3234_write(control_reg, 0x20);
  DS3234_write(status_reg, 0x48);
  DS3234_set_time(hr, min, s, ampm, hr_format);
```

```
  DS3234_set_date(dy, dt, mt, yr);
}


unsigned char DS3234_read(unsigned char addr)
{
  unsigned char value = 0;

  DS3234_SS_LOW;
  SPI_transfer(addr | read_cmd);
  value = SPI_transfer(0x00);
  DS3234_SS_HIGH;

  return value;
}


void DS3234_write(unsigned char addr, unsigned char value)
{
  unsigned long temp = 0;

  DS3234_SS_LOW;
  temp = (addr | write_cmd);
  SPI_transfer(temp);
  SPI_transfer(value);
  DS3234_SS_HIGH;
}
```

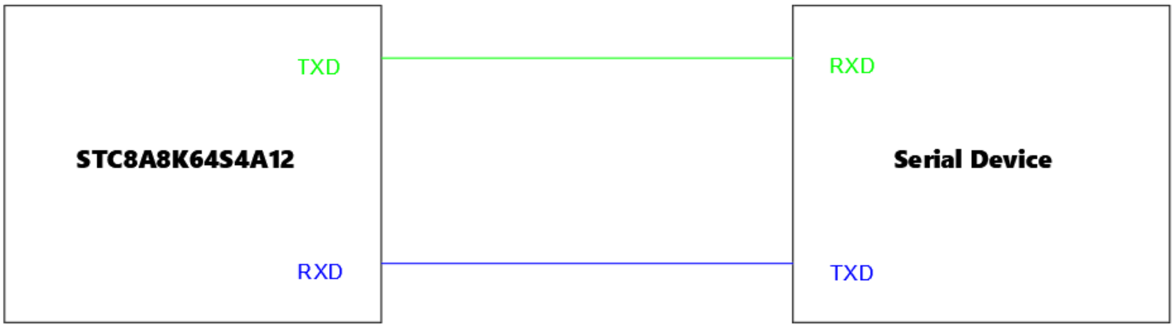
Demo



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

UART

Serial communication is probably the most widely used communication method. It is simple to implement and can be used to communicate with a computer literally directly without needing too many extra external hardware. Serial communication also has the distance advantage over other communication interfaces. Serial communication can be implemented using UART hardware peripherals of STC micros. Many sensors, communication devices like GSM modems and RF devices, RFIDs and other external devices use this simple and time-proven interface for communicating with host micros. UART is also the backbone of other communication methods like MODBUS, IrDA, LIN, etc.



To learn more about UART visit the following link:

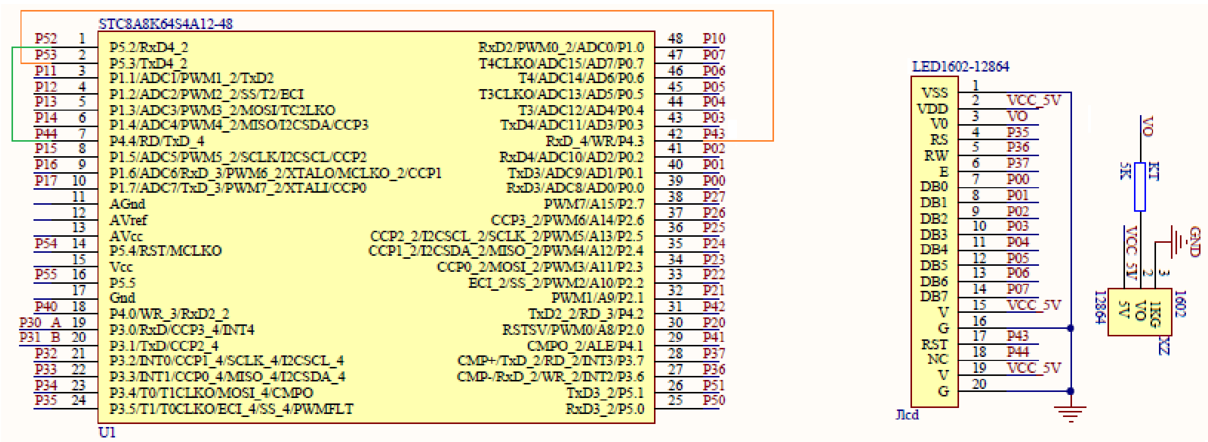
https://learn.mikroe.com/uart-serial-communication

Code

```
#include "STC8xxx.h"
#include "BSP.h"
#include "LCD.c"
#include "lcd_print.c"


void setup(void);


void main(void)
{
  unsigned char msg1[10] = {"MicroArena"};
  unsigned char msg2[10] = {"SShahryiar"};

  char i = 0x00;

  char rcv_1 = 0x00;
  char rcv_4 = 0x00;

  setup();

  LCD_goto(0, 0);
  LCD_putstr("TXD1: ");
  LCD_goto(10, 0);
  LCD_putstr("RXD1: ");

  LCD_goto(0, 1);
  LCD_putstr("TXD4: ");
  LCD_goto(10, 1);
  LCD_putstr("RXD4: ");

  while(1)
  {
```

```
    for(i = 0; i < 10; i++)
    {
      UART1_write_buffer(msg1[i]);
      UART4_write_buffer(msg2[i]);

      LCD_goto(5, 0);
      LCD_putchar(msg1[i]);
      LCD_goto(5, 1);
      LCD_putchar(msg2[i]);

      rcv_1 = UART1_read_buffer();
      rcv_4 = UART4_read_buffer();

      LCD_goto(15, 0);
      LCD_putchar(rcv_1);
      LCD_goto(15, 1);
      LCD_putchar(rcv_4);

      delay_ms(900);
    }
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  UART1_pin_option(0xC0);
  UART1_init(9600, \
             UART1_baud_source_TMR2, \
             UART1_timer_12T, \
             12000000);

  UART4_pin_option(0x04);
  UART4_init(9600, \
             UART4_baud_source_TMR4, \
             UART4_timer_1T, \
             12000000);

  LCD_init();
  LCD_clear_home();
}
```

Schematic



Explanation

STC8A8K64S4A12 has four hardware UART peripherals. Of these four, UART1 has some advanced features that are rarely needed. The rest are pretty much alike. Like other communication hardware peripherals, pin configuration must be selected as alternative pin arrangements are available.

```
/*
RXD         TXD         Hex             Option
P0.2        P0.3        0x00            option 1
P5.2        P5.3        0x04            option 2
*/


#define UART4_pin_option(value)         do{P_SW2 |= value;}while(0)
```

UART, although asynchronous, need to transmit and receive data in timed-frame formats and so when two UART devices need to communicate with each other, they must negotiate a mutually agreed baud rate or else data will not recognized. It is like tuning to the right radio station for listening the music channel that we want to listen. Thus, baud rate generation is a very crucial part of UART peripheral. For baud rate generation, UARTs can either use Timer 2 or other timer having the same number as the UART itself, for example, in the case of UART3, only Timer 2 and Timer 3 can be used as baud rate generator. I personally recommend that we keep Timer 2 reserved for other applications or UART2 and use other timers independently as to avoid conflicts in different UART hardware. Yes, I know it contradicts with my past statement of keeping Timer 2 reserved for UART applications but what else can be done when multiple UARTs are used in an application. BSP functions take care of everything internally and so we can focus on coding.

The system clock is set to 12MHz. This is very important because timers are responsible for baud rate generation and are dependent on system clock.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);
```

To setup UART, we need to specify baud rate, clock source of baud rate generator, i.e., the timer to be used, its prescale factor and system clock speed in hertz.

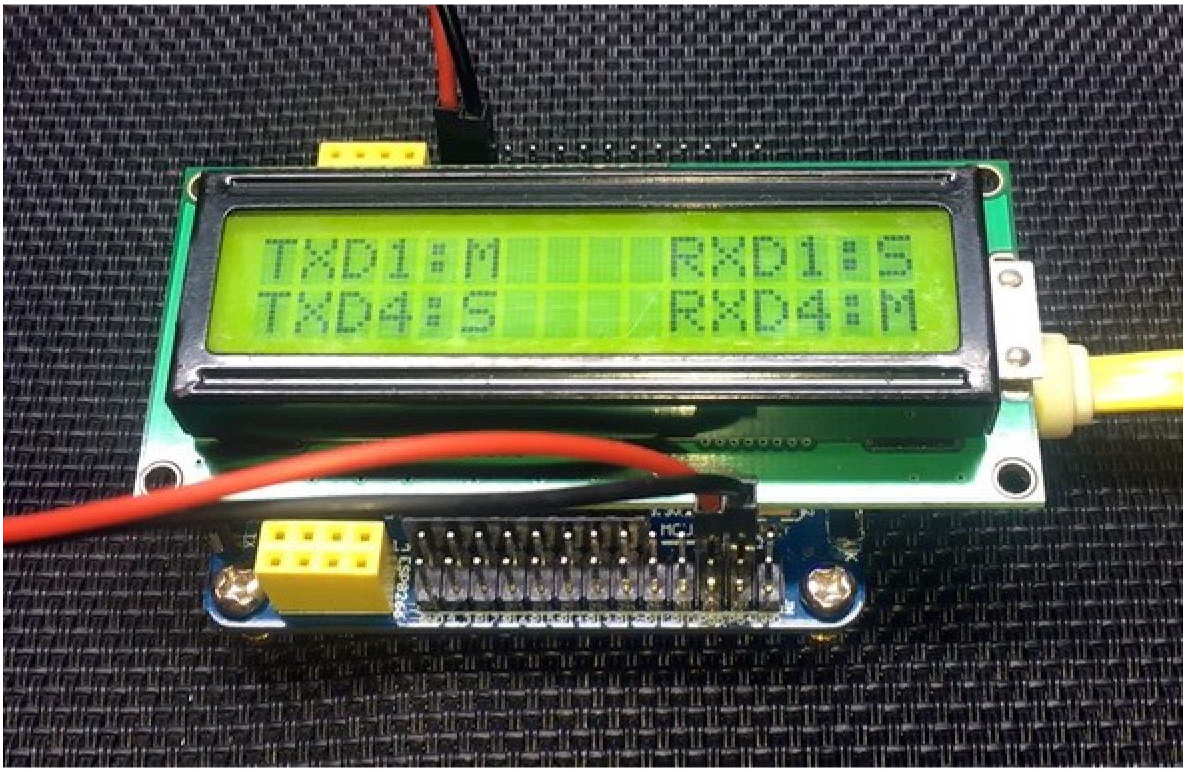
```
UART4_init(9600, \
           UART4_baud_source_TMR4, \
           UART4_timer_1T, \
           12000000);
```

UART reading and writing is done as shown below.

```
UART4_write_buffer(msg2[i]);
....
rcv_4 = UART4_read_buffer();
```

The demo here uses two built-in UARTs - UART1 and UART4, crisscrossed among themselves. Each transmitting and receiving the other's message. Whatever each is sending and receiving is also shown on an LCD display.

Demo

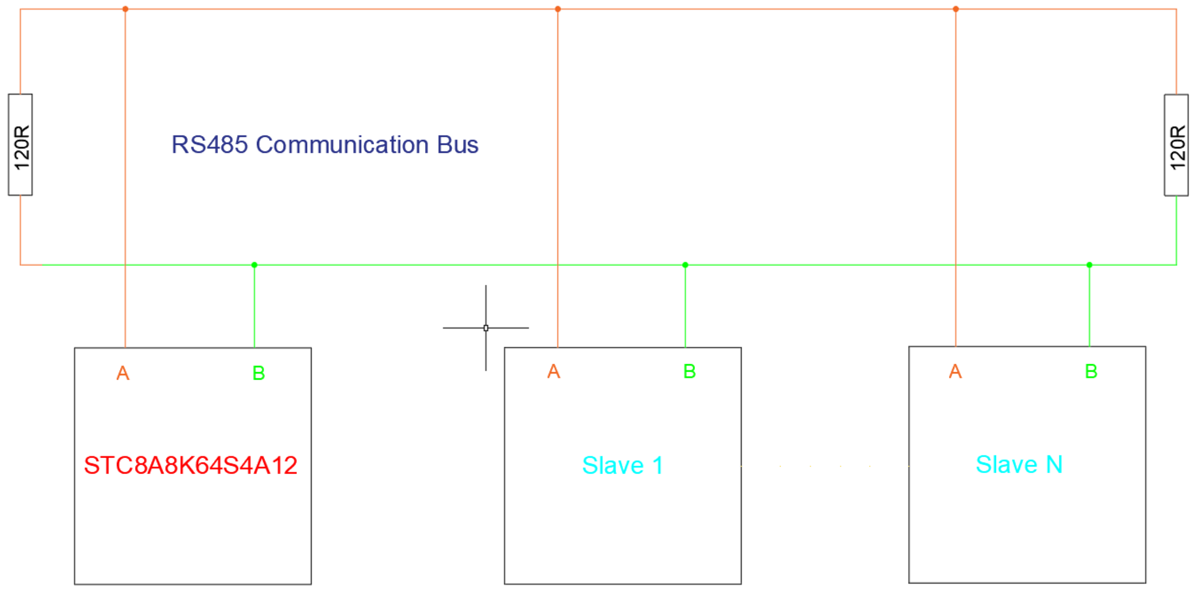


Demo video link: https://youtu.be/5MO5_YcRB_s.

RS485 MODBUS

RS485 is basically a long-range reliable industrial version of serial communication. It is a hardware layer that can support anything from simple serial communication to complex software-based communication layers. MODBUS is such a software layer or protocol that can be used to reliably exchange data between devices. MODBUS (particularly MODBUS RTU) is very popular among industrial devices like energy meters, PLCs and sensor devices. MODBUS has some similarity with I2C in some regards.

Another similar communication medium is the Controller Area Network (CAN). CAN is not supported internally by STC microcontrollers unlike some advanced microcontrollers like the STM32s and other ARMs but with external hardware like MCP2551 and MCP2515 CAN communication can also be achieved. CAN communication is beyond the scope of this tutorial.



To know more about MODBUS, visit these links:

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

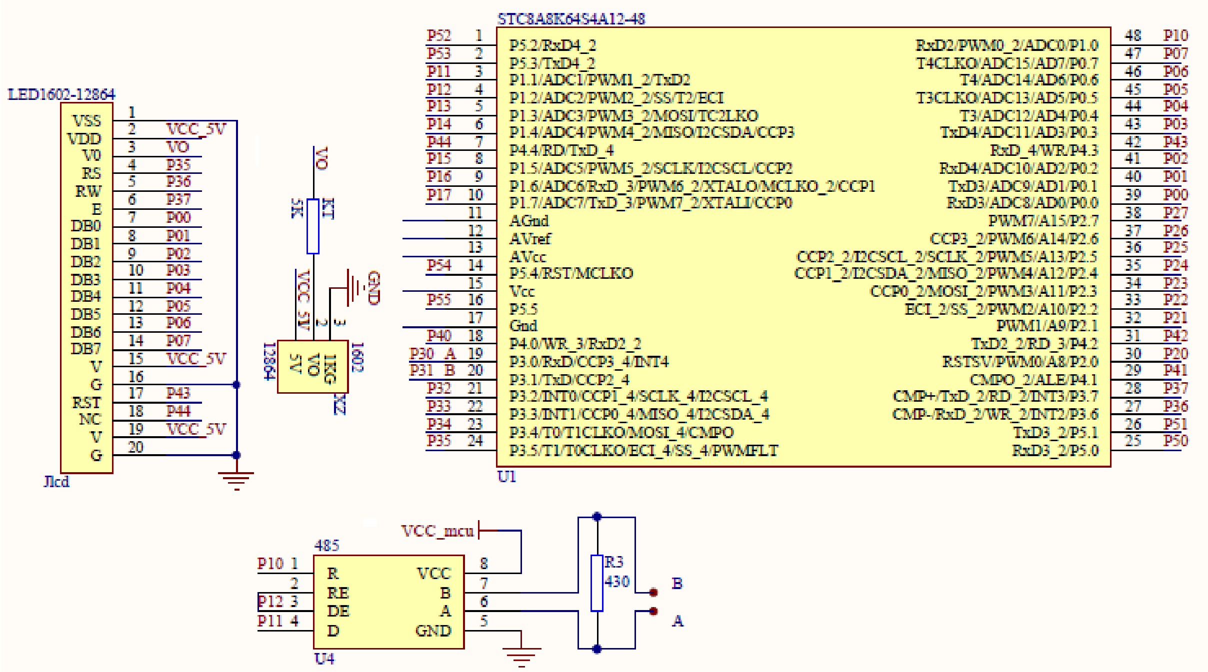
Code

```
#include "STC8xxx.h"
#include "BSP.h"
#include "LCD.c"
#include "lcd_print.c"


#define DIR_RX                                  P12_low
#define DIR_TX                                  P12_high



#define TX_buffer_length                        8
#define RX_buffer_length                        10


unsigned char cnt = 0x00;
unsigned char RX_buffer[RX_buffer_length];

static const unsigned char TX_buffer[TX_buffer_length] = {0x01, 0x03, 0x00, 0x00, 0x00, 0x02, 0xC4, 0x0B};


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


void UART_2_ISR(void)
interrupt 8
{
  if(check_UART_2_RX_flag)
  {
    RX_buffer[cnt++] = UART2_read_buffer();
  }
}


void main(void)
{
  unsigned int value = 0x0000;
  unsigned int CRC_check_1 = 0x0000;
  unsigned int CRC_check_2 = 0x0000;

  setup();

  LCD_goto(0, 0);
  LCD_putstr("R.H / %:");
  LCD_goto(0, 1);
  LCD_putstr("Temp/ C:");
  print_symbol(5, 1, 0);

  while(1)
  {
      send_read_command();

      CRC_check_1 = MODBUS_RTU_CRC16(RX_buffer, 7);
      CRC_check_2 = make_word(RX_buffer[8], RX_buffer[7]);

      if(CRC_check_1 == CRC_check_2)
      {
         value = make_word(RX_buffer[5], RX_buffer[6]);
         print_F(11, 0, (value / 10.0), 1);

         value = make_word(RX_buffer[3], RX_buffer[4]);
         print_F(11, 1, (value / 10.0), 1);
      }

      else
      {
        LCD_goto(12, 0);
        LCD_putstr("--.-");
```

```
        LCD_goto(12, 1);
        LCD_putstr("--.-");
      }

      P55_toggle;
      delay_ms(1000);
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  P12_push_pull_mode;
  P55_open_drain_mode;

  LCD_init();
  LCD_clear_home();
  load_custom_symbol();

  UART2_pin_option(0x00);

  UART2_init(9600, \
             UART2_timer_1T, \
             12000000);

  _enable_UART_2_interrupt;
  _enable_global_interrupt;
}


void flush_RX_buffer(void)
{
  signed char i = (RX_buffer_length - 1);

  while(i > -1)
  {
    RX_buffer[i] = 0x00;
    i--;
  };
}


void send_read_command(void)
{
  unsigned char i = 0x00;

  flush_RX_buffer();

  DIR_TX;

  for(i = 0; i < TX_buffer_length; i++)
  {
    UART2_write_buffer(TX_buffer[i]);
  }

  cnt = 0;
  DIR_RX;

  delay_ms(600);
}


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

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

  return tmp;
}


void get_HB_LB(unsigned int value, unsigned char *HB, unsigned char *LB)
```

```c
{
  *LB = (value & 0x00FF);
  *HB = ((value & 0xFF00) >> 8);
}


unsigned int MODBUS_RTU_CRC16(unsigned char *data_input, unsigned char data_length)
{
  static const unsigned int CRC_table[256] =
  {
     0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
     0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
     0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
     0x0A00, 0xCAC1, 0xCB81, 0x0B40, 0xC901, 0x09C0, 0x0880, 0xC841,
     0xD801, 0x18C0, 0x1980, 0xD941, 0x1B00, 0xDBC1, 0xDA81, 0x1A40,
     0x1E00, 0xDEC1, 0xDF81, 0x1F40, 0xDD01, 0x1DC0, 0x1C80, 0xDC41,
     0x1400, 0xD4C1, 0xD581, 0x1540, 0xD701, 0x17C0, 0x1680, 0xD641,
     0xD201, 0x12C0, 0x1380, 0xD341, 0x1100, 0xD1C1, 0xD081, 0x1040,
     0xF001, 0x30C0, 0x3180, 0xF141, 0x3300, 0xF3C1, 0xF281, 0x3240,
     0x3600, 0xF6C1, 0xF781, 0x3740, 0xF501, 0x35C0, 0x3480, 0xF441,
     0x3C00, 0xFCC1, 0xFD81, 0x3D40, 0xFF01, 0x3FC0, 0x3E80, 0xFE41,
     0xFA01, 0x3AC0, 0x3B80, 0xFB41, 0x3900, 0xF9C1, 0xF881, 0x3840,
     0x2800, 0xE8C1, 0xE981, 0x2940, 0xEB01, 0x2BC0, 0x2A80, 0xEA41,
     0xEE01, 0x2EC0, 0x2F80, 0xEF41, 0x2D00, 0xEDC1, 0xEC81, 0x2C40,
     0xE401, 0x24C0, 0x2580, 0xE541, 0x2700, 0xE7C1, 0xE681, 0x2640,
     0x2200, 0xE2C1, 0xE381, 0x2340, 0xE101, 0x21C0, 0x2080, 0xE041,
     0xA001, 0x60C0, 0x6180, 0xA141, 0x6300, 0xA3C1, 0xA281, 0x6240,
     0x6600, 0xA6C1, 0xA781, 0x6740, 0xA501, 0x65C0, 0x6480, 0xA441,
     0x6C00, 0xACC1, 0xAD81, 0x6D40, 0xAF01, 0x6FC0, 0x6E80, 0xAE41,
     0xAA01, 0x6AC0, 0x6B80, 0xAB41, 0x6900, 0xA9C1, 0xA881, 0x6840,
     0x7800, 0xB8C1, 0xB981, 0x7940, 0xBB01, 0x7BC0, 0x7A80, 0xBA41,
     0xBE01, 0x7EC0, 0x7F80, 0xBF41, 0x7D00, 0xBDC1, 0xBC81, 0x7C40,
     0xB401, 0x74C0, 0x7580, 0xB541, 0x7700, 0xB7C1, 0xB681, 0x7640,
     0x7200, 0xB2C1, 0xB381, 0x7340, 0xB101, 0x71C0, 0x7080, 0xB041,
     0x5000, 0x90C1, 0x9181, 0x5140, 0x9301, 0x53C0, 0x5280, 0x9241,
     0x9601, 0x56C0, 0x5780, 0x9741, 0x5500, 0x95C1, 0x9481, 0x5440,
     0x9C01, 0x5CC0, 0x5D80, 0x9D41, 0x5F00, 0x9FC1, 0x9E81, 0x5E40,
     0x5A00, 0x9AC1, 0x9B81, 0x5B40, 0x9901, 0x59C0, 0x5880, 0x9841,
     0x8801, 0x48C0, 0x4980, 0x8941, 0x4B00, 0x8BC1, 0x8A81, 0x4A40,
     0x4E00, 0x8EC1, 0x8F81, 0x4F40, 0x8D01, 0x4DC0, 0x4C80, 0x8C41,
     0x4400, 0x84C1, 0x8581, 0x4540, 0x8701, 0x47C0, 0x4680, 0x8641,
     0x8201, 0x42C0, 0x4380, 0x8341, 0x4100, 0x81C1, 0x8081, 0x4040
   };

  unsigned char temp = 0;
  unsigned int CRC_word = 0xFFFF;

  while(data_length--)
  {
      temp = *data_input++ ^ CRC_word;
      CRC_word >>= 8;
      CRC_word ^= CRC_table[temp];
  }

  return CRC_word;
}
```

Schematic



Explanation

In this example, a MODBUS-based SHT20 relative humidity and temperature sensor is read with a STC micro. The sensor accepts MODBUS RTU data frames.

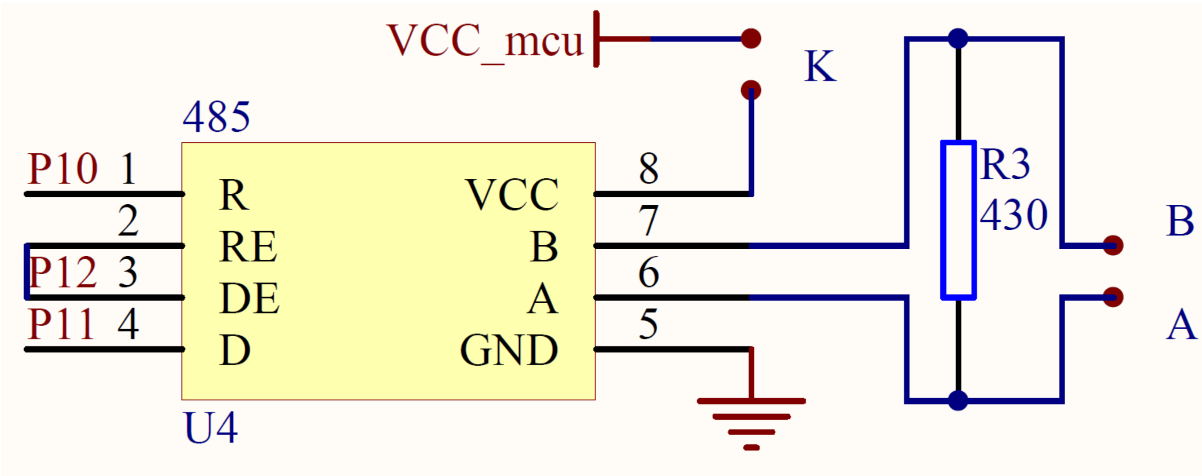
The setup is similar to the one we have already seen in the UART example and it should easy by now. The only exceptions are the UART module and the use of its interrupt.

```
CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

P12_push_pull_mode;
P55_open_drain_mode;

UART2_pin_option(0x00);

UART2_init(9600, \
           UART2_timer_1T, \
           12000000);

_enable_UART_2_interrupt;
_enable_global_interrupt;
```

The UART's interrupt is used for receiving data only.

```
void UART_2_ISR(void)
interrupt 8
{
  if(check_UART_2_RX_flag)
  {
    RX_buffer[cnt++] = UART2_read_buffer();
  }
}
```

For sending data, the following function is used. In this function, previously received data are cleared first in order to make the micro ready to receive new batch of data and to make sure that past data do not make any conflict with the new ones. The onboard MAX485's data direction is set to transmission mode and the transmission (TX) buffer data are sent via UART. MAX485's mode operation is reset back to reception mode in order to receive new data.

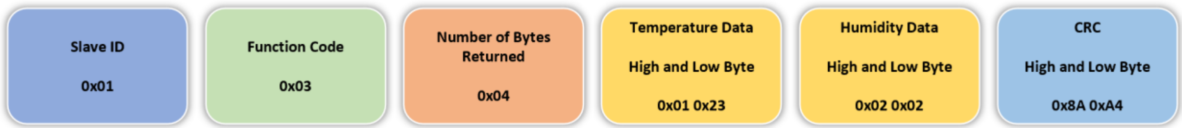



```
static const unsigned char TX_buffer[TX_buffer_length] = {0x01, 0x03, 0x00, 0x00, 0x00, 0x02, 0xC4, 0x0B};

....

void send_read_command(void)
{
  unsigned char i = 0x00;

  flush_RX_buffer();

  DIR_TX;

  for(i = 0; i < TX_buffer_length; i++)
  {
    UART2_write_buffer(TX_buffer[i]);
  }

  cnt = 0;
  DIR_RX;

  delay_ms(600);
}
```

The TX buffer contains the following information in the following order. This is the standard frame that should be sent by a host device to read holding registers. MODBUS RTU strictly prohibits other data frame formats and doing so will only lead to errors.

Slave ID	Function Code	Start Address	Data Length	CRC
		High and Low Byte	High and Low Byte	High and Low Byte
0x01	0x03	0x00 0x00	0x00 0x02	0xC4 0x0B

When the sensor receives these bytes in this order, it responds back with a reception or RX frame which is something like the frame shown below. This frame is similar to the TX frame but the important stuffs are the relative humidity and temperature data. These are the stuff that we mainly need. The values in frame shown below is just for giving an example.



As per MODBUS RTU frame shown above, the slave ID is 0x01, the function code is 0x03 and 4 data bytes have been sent by the sensor along with the CRC fields. Function code 0x03 stands for reading holding registers. The temperature and the relative humidity are computed according to the calculation shown below:

Data Field	High Byte	Low Byte	Word	Integer (I)	Decimal (D = I/10)	Unit
Temperature	0x01	0x23	0x0123	291	29.1	^{o}C
Relative Humidity	0x02	0x02	0x0202	514	51.4	%

The table above shows that the high and low bytes are combined to make word and the word is divided by 10 to get desired outputs.

For more information on MODBUS, please refer to the docs mentioned in the beginning of this section. MODBUS RTU, itself, is a big topic and it is not possible to fully cover it in this tutorial.

The code runs by querying the SHT20 sensor. After receiving all data from sensor, two Cyclic Redundancy Checks (CRC) are performed in order to ensure that valid and error-free data have been received. One CRC is the embedded with the RX frame and other is calculated using a CRC lookup table. Details of CRC lookup table and formula are covered in MODBUS RTU documentations. If CRCs match then the RX frame is considered to contain valid data and with the RX data, we can compute relative humidity and temperature values.

```
send_read_command();

CRC_check_1 = MODBUS_RTU_CRC16(RX_buffer, 7);
CRC_check_2 = make_word(RX_buffer[8], RX_buffer[7]);

if(CRC_check_1 == CRC_check_2)
{
     value = make_word(RX_buffer[5], RX_buffer[6]);
     print_F(11, 0, (value / 10.0), 1);

     value = make_word(RX_buffer[3], RX_buffer[4]);
     print_F(11, 1, (value / 10.0), 1);
}

else
{
     LCD_goto(12, 0);
     LCD_putstr("--.-");
     LCD_goto(12, 1);
     LCD_putstr("--.-");
}

P55_toggle;
delay_ms(1000);
```

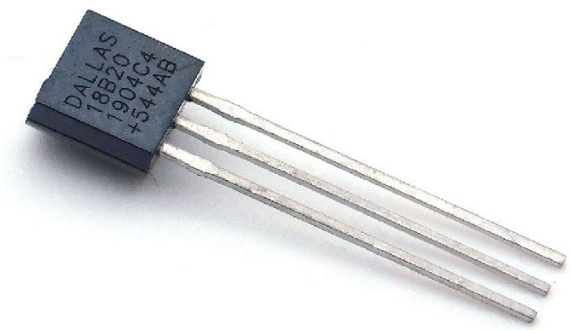
Demo

Demo video link: https://youtu.be/e9XLeAYte_Q.

One Wire and Bit-Banging Technique

One wire is not a standard system of communication unlike I2C, SPI and UART because it varies from device to device. Thus, this form of communication needs more coding rather than hardware-design consideration.

DS18B20 is a digital temperature sensor that uses one wire communication to communicate with a host microcontroller. One wire technique relies on time-slotting mechanism in which logical ones and zeroes are represented by pulses of variable widths. The same technique is used in infrared remote controllers, various sensors like SONAR sensors, relative humidity and temperature sensors like DHT22, etc and many other devices. The strategic advantage of one-wire technique is low pin count but the disadvantage is heavy software implementation and background processing.



Like one wire devices, sometimes we have to deal with hardware devices that do not follow conventional communication techniques as mentioned earlier and yet again we have to drive these devices using bit-banging techniques. TM1637 seven-segment display driver is such a device. Apparently, it uses something similar to I2C as two wires (clock and data) are needed for communication but in practice it has no similarity with I2C communication at all.

The best way to deal with such cases is to study the device datasheet and look for its timing diagrams. Timing diagrams along with proprietary protocols are all that would be needed to make libraries for these devices on our own.

Code

one_wire.h

```
#define DS18B20_GPIO_init()             P30_quasi_bidirectional_mode

#define DS18B20_IN()                    P30_get_input

#define DS18B20_OUT_LOW()               P30_low
#define DS18B20_OUT_HIGH()              P30_high

#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 "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.c"


#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 "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);
}
```

TM1637.h

```
#define TM1637_CLK_HIGH              P41_high
#define TM1637_CLK_LOW               P41_low

#define TM1637_DAT_HIGH              P42_high
#define TM1637_DAT_LOW               P42_low

#define TM1637_DELAY_US              4

#define TM1637_BRIGHTNESS_MIN        0
#define TM1637_BRIGHTNESS_1          1
#define TM1637_BRIGHTNESS_2          2
#define TM1637_BRIGHTNESS_3          3
#define TM1637_BRIGHTNESS_4          4
#define TM1637_BRIGHTNESS_5          5
#define TM1637_BRIGHTNESS_6          6
#define TM1637_BRIGHTNESS_MAX        7

#define TM1637_POSITION_MAX          4

#define TM1637_CMD_SET_DATA       0x40
#define TM1637_CMD_SET_ADDR       0xC0
#define TM1637_CMD_SET_DSIPLAY    0x80

#define TM1637_SET_DATA_WRITE     0x00
#define TM1637_SET_DATA_READ      0x02
#define TM1637_SET_DATA_A_ADDR    0x00
#define TM1637_SET_DATA_F_ADDR    0x04
#define TM1637_SET_DATA_M_NORM    0x00
#define TM1637_SET_DATA_M_TEST    0x10
#define TM1637_SET_DISPLAY_OFF    0x00
#define TM1637_SET_DISPLAY_ON     0x08


const unsigned char seg_data[10] =
{
    0x3F, // 0
    0x06, // 1
    0x5B, // 2
    0x4F, // 3
    0x66, // 4
    0x6D, // 5
    0x7D, // 6
    0x07, // 7
    0x7F, // 8
    0x6F  // 9
};


void TM1637_init(void);
void TM1637_start(void);
void TM1637_stop(void);
unsigned char TM1637_write_byte(unsigned char value);
void TM1637_send_command(unsigned char value);
void TM1637_clear(void);
void TM1637_display_segments(unsigned char position, unsigned char segment_value, unsigned char colon_state);
```

TM1637.c

```
#include "TM1637.h"


void TM1637_init(void)
{
    P41_push_pull_mode;
    P42_push_pull_mode;

    TM1637_DAT_LOW;
    TM1637_CLK_LOW;
    TM1637_send_command(TM1637_CMD_SET_DSIPLAY | TM1637_BRIGHTNESS_2 | TM1637_SET_DISPLAY_ON);
    TM1637_clear();
}
```

```
void TM1637_start(void)
{
    TM1637_DAT_HIGH;
    TM1637_CLK_HIGH;
    delay_us(TM1637_DELAY_US);
    TM1637_DAT_LOW;
}


void TM1637_stop(void)
{

    TM1637_CLK_LOW;
    delay_us(TM1637_DELAY_US);

    TM1637_DAT_LOW;
    delay_us(TM1637_DELAY_US);

    TM1637_CLK_HIGH;
    delay_us(TM1637_DELAY_US);

    TM1637_DAT_HIGH;
}


unsigned char TM1637_write_byte(unsigned char value)
{
    unsigned char i = 0x08;
    unsigned char ack = 0x00;

    while(i)
    {
        TM1637_CLK_LOW;
        delay_us(TM1637_DELAY_US);

        if(value & 0x01)
        {
            TM1637_DAT_HIGH;
        }

        else
        {
            TM1637_DAT_LOW;
        }

        TM1637_CLK_HIGH;
        delay_us(TM1637_DELAY_US);

        value >>= 1;
        i--;
    }

    TM1637_CLK_LOW;

    delay_us(TM1637_DELAY_US);

    ack = P42_get_input;

    if(ack != 0)
    {
        TM1637_DAT_LOW;
    }

    delay_us(TM1637_DELAY_US);

    TM1637_CLK_HIGH;
    delay_us(TM1637_DELAY_US);

    TM1637_CLK_LOW;
    delay_us(TM1637_DELAY_US);


    return (ack);
}


void TM1637_send_command(unsigned char value)
{
```

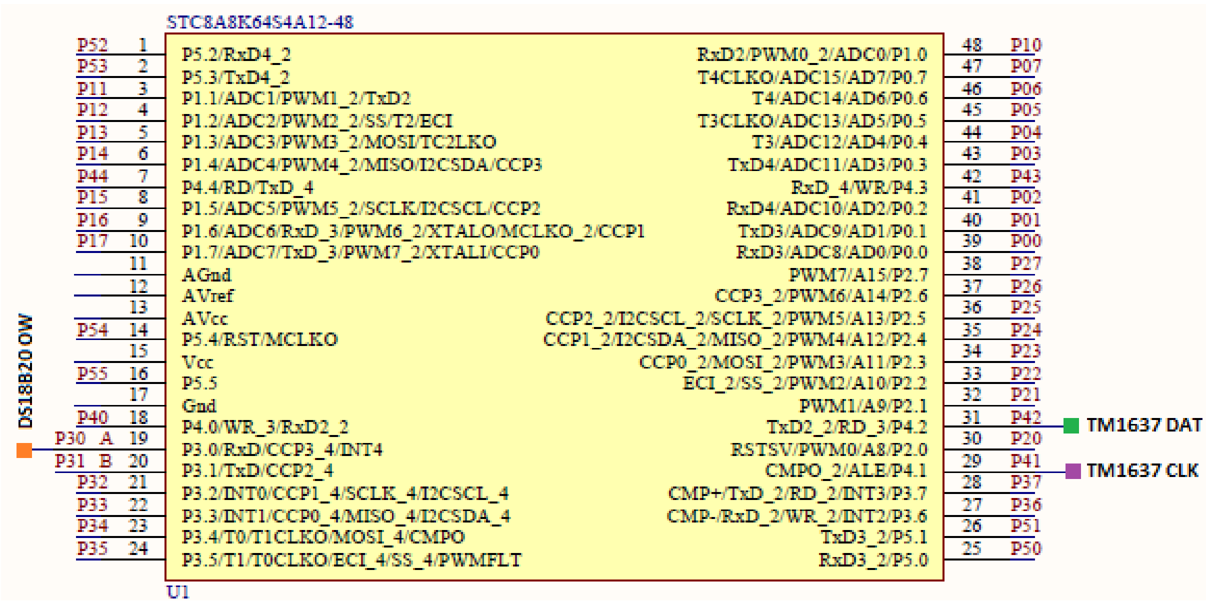
```
    TM1637_start();
    TM1637_write_byte(value);
    TM1637_stop();
}


void TM1637_clear(void)
{
    signed char i = (TM1637_POSITION_MAX - 1);

    while(i > -1)
    {
        TM1637_display_segments(i, 0x00, 0x00);
        i--;
    };
}



void TM1637_display_segments(unsigned char position, unsigned char segment_value, unsigned char colon_state)
{
    if(position == 1)
    {
        switch(colon_state)
        {
            case 1:
            {
                segment_value |= 0x80;
                break;
            }

            default:
            {
                segment_value &= 0x7F;
                break;
            }
        }
    }

    TM1637_send_command(TM1637_CMD_SET_DATA | TM1637_SET_DATA_F_ADDR);
    TM1637_start();
    TM1637_write_byte(TM1637_CMD_SET_ADDR | (position & (TM1637_POSITION_MAX - 1)));
    TM1637_write_byte(segment_value);
    TM1637_stop();
}
```

main.c

```
#include "STC8xxx.h"
#include "BSP.h"
#include "DS18B20.c"
#include "TM1637.c"


void setup(void);


void main(void)
{
  signed long t = 0;

  setup();


  while(1)
  {
    t = ((signed long)DS18B20_get_temperature());


    if((t > 999) && (t >= 0))

    {
        TM1637_display_segments(0, 0x40, 0);
        TM1637_display_segments(1, 0x40, 0);
```

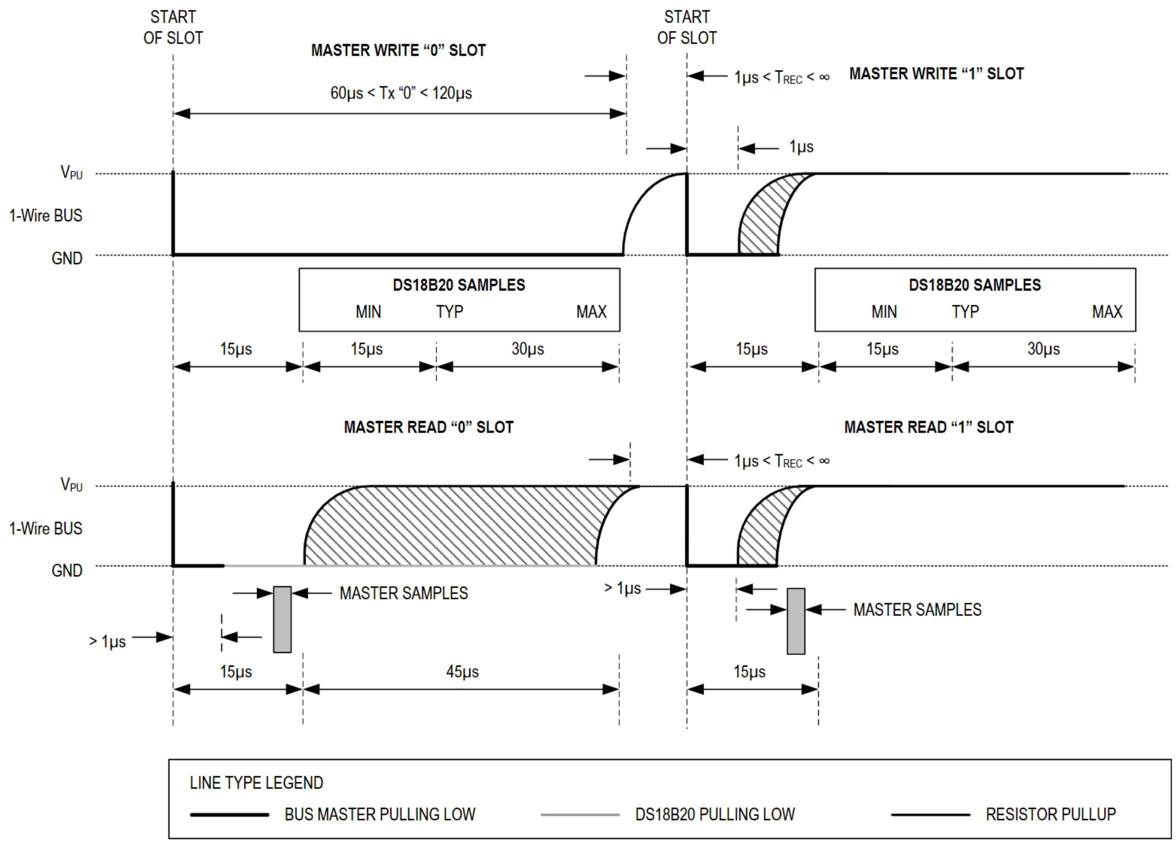
```
    }

    else
    {
        TM1637_display_segments(0, seg_data[(t / 10)], 0);
        TM1637_display_segments(1, seg_data[(t % 10)], 0);
    }

    TM1637_display_segments(2, 0x63, 0);
    TM1637_display_segments(3, 0x39, 0);

    delay_ms(600);
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  TM1637_init();
  DS18B20_init();
}
```

Schematic



Explanation

DS18B20's one communication basics are shown below. Notice how time-slotting technique is being employed.



These timing diagrams are what that have been use to compose the following codes:

```
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;
}
```

Details of DS18B20's one wire communication can be found in the following application notes from Maxim.

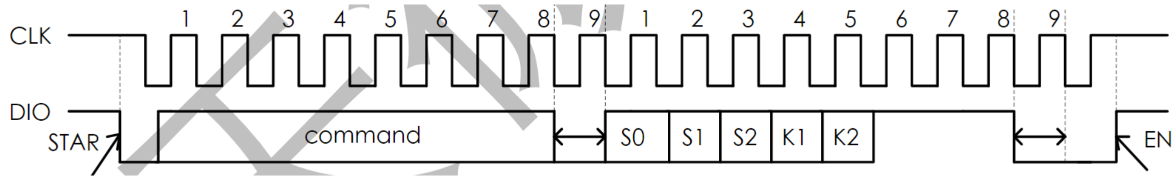
https://www.maximintegrated.com/en/app-notes/index.mvp/id/126
https://www.maximintegrated.com/en/app-notes/index.mvp/id/162

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.

It is worth mentioning that pin declarations should be checked before actually hooking any device.

```
#define DS18B20_GPIO_init()             P30_quasi_bidirectional_mode

#define DS18B20_IN()                    P30_get_input

#define DS18B20_OUT_LOW()               P30_low
#define DS18B20_OUT_HIGH()              P30_high
```

Bit-banging TM1637 is achieved by implementing what have been documented in the ***Interface interpretation*** section of the datasheet.



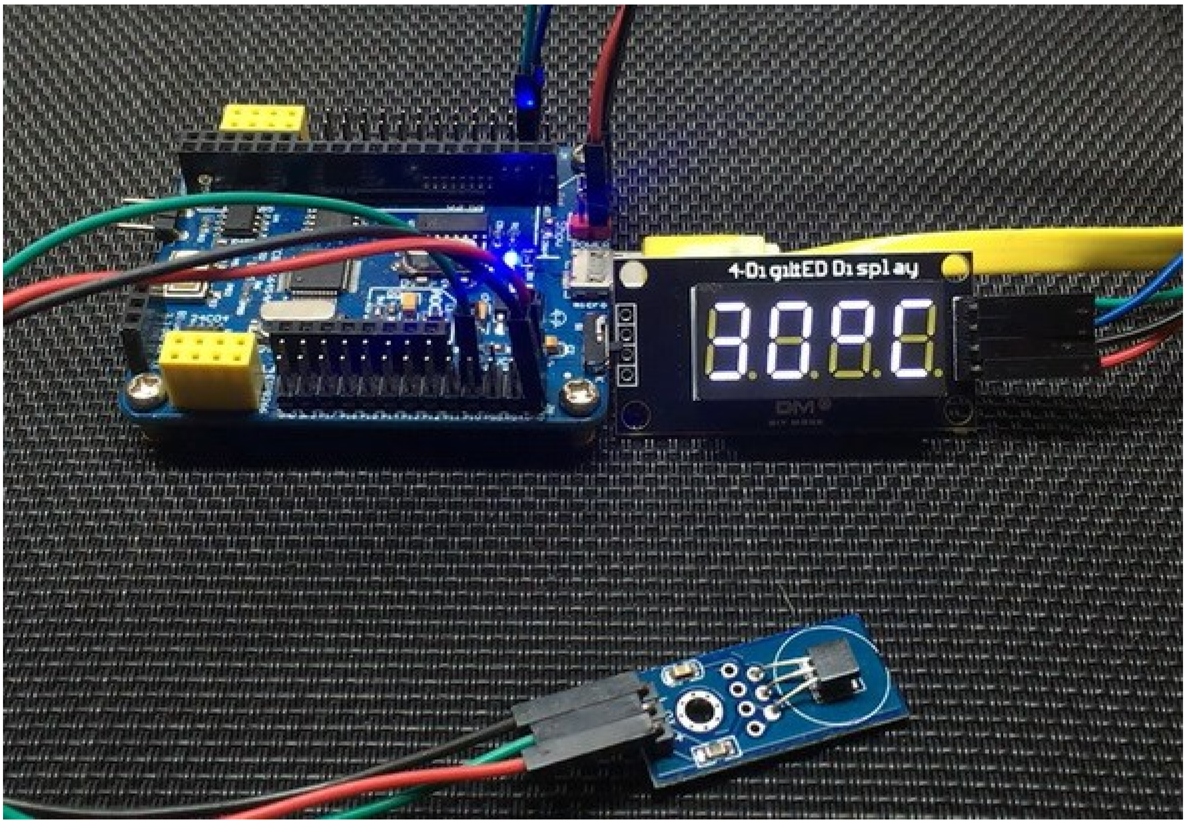
The following codes are the coded representation of the communication timing diagram of the device. Again, I insist readers to go through device datasheet for details and explanation.

```
void TM1637_start(void)
{
    TM1637_DAT_HIGH;
    TM1637_CLK_HIGH;
    delay_us(TM1637_DELAY_US);
    TM1637_DAT_LOW;
}


void TM1637_stop(void)
{

    TM1637_CLK_LOW;
    delay_us(TM1637_DELAY_US);

    TM1637_DAT_LOW;
    delay_us(TM1637_DELAY_US);

    TM1637_CLK_HIGH;
    delay_us(TM1637_DELAY_US);

    TM1637_DAT_HIGH;
}


unsigned char TM1637_write_byte(unsigned char value)
{
    unsigned char i = 0x08;
    unsigned char ack = 0x00;

    while(i)
    {
        TM1637_CLK_LOW;
        delay_us(TM1637_DELAY_US);

        if(value & 0x01)
        {
            TM1637_DAT_HIGH;
        }

        else
        {
            TM1637_DAT_LOW;
        }

        TM1637_CLK_HIGH;
        delay_us(TM1637_DELAY_US);

        value >>= 1;
        i--;
    }

    TM1637_CLK_LOW;

    delay_us(TM1637_DELAY_US);

    ack = P42_get_input;

    if(ack != 0)
```

```
    {
        TM1637_DAT_LOW;
    }

    delay_us(TM1637_DELAY_US);

    TM1637_CLK_HIGH;
    delay_us(TM1637_DELAY_US);

    TM1637_CLK_LOW;
    delay_us(TM1637_DELAY_US);


    return (ack);
}
```

The demo here is a simple DS18B20-based thermometer. The temperature sensed by the DS18B20 sensor is displayed on a TM1637-based seven segment displays.

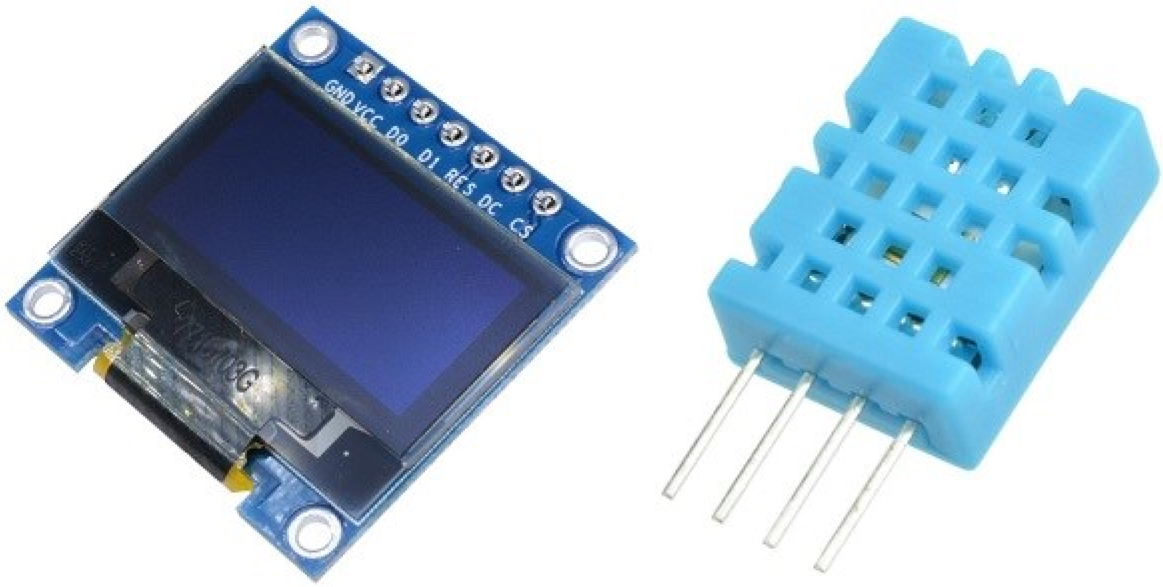
Demo



Demo video link: https://youtu.be/2SP1hiXuvfM.

Software SPI and One Wire Bit-Banging

Software SPI is basically bit-banging ordinary GPIO pins to reproduce SPI signals. Software SPI is not needed unless hardware SPI peripheral is absent in the device or hardware SPI pins are used up for some other tasks. Though it is slow and require extra coding compared to hardware SPI, it is helpful for beginners to understand SPI-based hardware as it offers full control over all SPI signals.



Likewise, bit-banging can also be applied to communicate with one wire devices such as DHT11, DHT22, etc.

In this section, we will see how to use bit-banging to read a DHT11 sensor and show temperature and relative humidity data on a SPI-based SSD1306 OLED display.

Code

DHT11.h

```
#define DHT11_pin_init          P10_quasi_bidirectional_mode

#define DHT11_pin_HIGH          P10_high
#define DHT11_pin_LOW           P10_low

#define DHT11_pin_IN            P10_get_input


unsigned char values[5];


void DHT11_init(void);
unsigned char DHT11_get_byte(void);
unsigned char DHT11_get_data(void);
```

DHT11.c

```c
#include "DHT11.h"


extern unsigned char values[5] = {0x00, 0x00, 0x00, 0x00, 0x00};


void DHT11_init(void)
{
   DHT11_pin_init;
   delay_ms(1000);
}


unsigned char DHT11_get_byte(void)
{
   unsigned char s = 0x08;
   unsigned char value = 0x00;

   while(s > 0)
   {
      value <<= 1;
      while(DHT11_pin_IN == LOW);
      large_delay_TMR_0(30);

      if(DHT11_pin_IN == HIGH)
      {
          value |= 1;
      }

      while(DHT11_pin_IN == HIGH);
      s--;
   }
   return value;
}


unsigned char DHT11_get_data(void)
{
   unsigned char s = 0;
   unsigned char check_sum = 0;

   DHT11_pin_HIGH;
   DHT11_pin_LOW;
   large_delay_TMR_0(18000);
   DHT11_pin_HIGH;
   large_delay_TMR_0(26);

   if(DHT11_pin_IN == HIGH)
   {
      return 1;
   }

   large_delay_TMR_0(80);

   if(DHT11_pin_IN == LOW)
   {
      return 2;
   }

   large_delay_TMR_0(80);

   for(s = 0; s <= 4; s++)
   {
       values[s] = DHT11_get_byte();
   }

   DHT11_pin_HIGH;

   for(s = 0; s < 4; s++)
   {
       check_sum += values[s];
   }

   if(check_sum != values[4])
```

```
    {
        return 3;
    }
    else
    {
        return 0;
    }
}
```

font.c

```
const unsigned char font_regular[92][6] =
{
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   // sp
    0x00, 0x00, 0x00, 0x2f, 0x00, 0x00,   // !
    0x00, 0x00, 0x07, 0x00, 0x07, 0x00,   // "
    0x00, 0x14, 0x7f, 0x14, 0x7f, 0x14,   // #
    0x00, 0x24, 0x2a, 0x7f, 0x2a, 0x12,   // $
    0x00, 0x62, 0x64, 0x08, 0x13, 0x23,   // %
    0x00, 0x36, 0x49, 0x55, 0x22, 0x50,   // &
    0x00, 0x00, 0x05, 0x03, 0x00, 0x00,   // '
    0x00, 0x00, 0x1c, 0x22, 0x41, 0x00,   // (
    0x00, 0x00, 0x41, 0x22, 0x1c, 0x00,   // )
    0x00, 0x14, 0x08, 0x3E, 0x08, 0x14,   // *
    0x00, 0x08, 0x08, 0x3E, 0x08, 0x08,   // +
    0x00, 0x00, 0x00, 0xA0, 0x60, 0x00,   // ,
    0x00, 0x08, 0x08, 0x08, 0x08, 0x08,   // -
    0x00, 0x00, 0x60, 0x60, 0x00, 0x00,   // .
    0x00, 0x20, 0x10, 0x08, 0x04, 0x02,   // /
    0x00, 0x3E, 0x51, 0x49, 0x45, 0x3E,   // 0
    0x00, 0x00, 0x42, 0x7F, 0x40, 0x00,   // 1
    0x00, 0x42, 0x61, 0x51, 0x49, 0x46,   // 2
    0x00, 0x21, 0x41, 0x45, 0x4B, 0x31,   // 3
    0x00, 0x18, 0x14, 0x12, 0x7F, 0x10,   // 4
    0x00, 0x27, 0x45, 0x45, 0x45, 0x39,   // 5
    0x00, 0x3C, 0x4A, 0x49, 0x49, 0x30,   // 6
    0x00, 0x01, 0x71, 0x09, 0x05, 0x03,   // 7
    0x00, 0x36, 0x49, 0x49, 0x49, 0x36,   // 8
    0x00, 0x06, 0x49, 0x49, 0x29, 0x1E,   // 9
    0x00, 0x00, 0x36, 0x36, 0x00, 0x00,   // :
    0x00, 0x00, 0x56, 0x36, 0x00, 0x00,   // ;
    0x00, 0x08, 0x14, 0x22, 0x41, 0x00,   // <
    0x00, 0x14, 0x14, 0x14, 0x14, 0x14,   // =
    0x00, 0x00, 0x41, 0x22, 0x14, 0x08,   // >
    0x00, 0x02, 0x01, 0x51, 0x09, 0x06,   // ?
    0x00, 0x32, 0x49, 0x59, 0x51, 0x3E,   // @
    0x00, 0x7C, 0x12, 0x11, 0x12, 0x7C,   // A
    0x00, 0x7F, 0x49, 0x49, 0x49, 0x36,   // B
    0x00, 0x3E, 0x41, 0x41, 0x41, 0x22,   // C
    0x00, 0x7F, 0x41, 0x41, 0x22, 0x1C,   // D
    0x00, 0x7F, 0x49, 0x49, 0x49, 0x41,   // E
    0x00, 0x7F, 0x09, 0x09, 0x09, 0x01,   // F
    0x00, 0x3E, 0x41, 0x49, 0x49, 0x7A,   // G
    0x00, 0x7F, 0x08, 0x08, 0x08, 0x7F,   // H
    0x00, 0x00, 0x41, 0x7F, 0x41, 0x00,   // I
    0x00, 0x20, 0x40, 0x41, 0x3F, 0x01,   // J
    0x00, 0x7F, 0x08, 0x14, 0x22, 0x41,   // K
    0x00, 0x7F, 0x40, 0x40, 0x40, 0x40,   // L
    0x00, 0x7F, 0x02, 0x0C, 0x02, 0x7F,   // M
    0x00, 0x7F, 0x04, 0x08, 0x10, 0x7F,   // N
    0x00, 0x3E, 0x41, 0x41, 0x41, 0x3E,   // O
    0x00, 0x7F, 0x09, 0x09, 0x09, 0x06,   // P
    0x00, 0x3E, 0x41, 0x51, 0x21, 0x5E,   // Q
    0x00, 0x7F, 0x09, 0x19, 0x29, 0x46,   // R
    0x00, 0x46, 0x49, 0x49, 0x49, 0x31,   // S
    0x00, 0x01, 0x01, 0x7F, 0x01, 0x01,   // T
    0x00, 0x3F, 0x40, 0x40, 0x40, 0x3F,   // U
    0x00, 0x1F, 0x20, 0x40, 0x20, 0x1F,   // V
    0x00, 0x3F, 0x40, 0x38, 0x40, 0x3F,   // W
    0x00, 0x63, 0x14, 0x08, 0x14, 0x63,   // X
    0x00, 0x07, 0x08, 0x70, 0x08, 0x07,   // Y
    0x00, 0x61, 0x51, 0x49, 0x45, 0x43,   // Z
    0x00, 0x00, 0x7F, 0x41, 0x41, 0x00,   // [ 91
    0x00, 0x02, 0x04, 0x08, 0x10, 0x20,    // \92
    0x00, 0x00, 0x41, 0x41, 0x7F, 0x00,   // ]
    0x00, 0x04, 0x02, 0x01, 0x02, 0x04,   // ^
    0x00, 0x40, 0x40, 0x40, 0x40, 0x40,   // _
```

```
    0x00, 0x00, 0x01, 0x02, 0x04, 0x00,   // '
    0x00, 0x20, 0x54, 0x54, 0x54, 0x78,   // a
    0x00, 0x7F, 0x48, 0x44, 0x44, 0x38,   // b
    0x00, 0x38, 0x44, 0x44, 0x44, 0x20,   // c
    0x00, 0x38, 0x44, 0x44, 0x48, 0x7F,   // d
    0x00, 0x38, 0x54, 0x54, 0x54, 0x18,   // e
    0x00, 0x08, 0x7E, 0x09, 0x01, 0x02,   // f
    0x00, 0x18, 0xA4, 0xA4, 0xA4, 0x7C,   // g
    0x00, 0x7F, 0x08, 0x04, 0x04, 0x78,   // h
    0x00, 0x00, 0x44, 0x7D, 0x40, 0x00,   // i
    0x00, 0x40, 0x80, 0x84, 0x7D, 0x00,   // j
    0x00, 0x7F, 0x10, 0x28, 0x44, 0x00,   // k
    0x00, 0x00, 0x41, 0x7F, 0x40, 0x00,   // l
    0x00, 0x7C, 0x04, 0x18, 0x04, 0x78,   // m
    0x00, 0x7C, 0x08, 0x04, 0x04, 0x78,   // n
    0x00, 0x38, 0x44, 0x44, 0x44, 0x38,   // o
    0x00, 0xFC, 0x24, 0x24, 0x24, 0x18,   // p
    0x00, 0x18, 0x24, 0x24, 0x18, 0xFC,   // q
    0x00, 0x7C, 0x08, 0x04, 0x04, 0x08,   // r
    0x00, 0x48, 0x54, 0x54, 0x54, 0x20,   // s
    0x00, 0x04, 0x3F, 0x44, 0x40, 0x20,   // t
    0x00, 0x3C, 0x40, 0x40, 0x20, 0x7C,   // u
    0x00, 0x1C, 0x20, 0x40, 0x20, 0x1C,   // v
    0x00, 0x3C, 0x40, 0x30, 0x40, 0x3C,   // w
    0x00, 0x44, 0x28, 0x10, 0x28, 0x44,   // x
    0x00, 0x1C, 0xA0, 0xA0, 0xA0, 0x7C,   // y
    0x00, 0x44, 0x64, 0x54, 0x4C, 0x44,   // z
    0x14, 0x14, 0x14, 0x14, 0x14, 0x14    // horiz lines
};
```

SSD1306.h

```
#define SSD1306_CS_PIN_HIGH                   P03_high
#define SSD1306_CS_PIN_LOW                    P03_low

#define SSD1306_DC_PIN_HIGH                   P04_high
#define SSD1306_DC_PIN_LOW                    P04_low

#define SSD1306_RST_PIN_HIGH                  P05_high
#define SSD1306_RST_PIN_LOW                   P05_low

#define SSD1306_SDA_PIN_HIGH                  P06_high
#define SSD1306_SDA_PIN_LOW                   P06_low

#define SSD1306_SCK_PIN_HIGH                  P07_high
#define SSD1306_SCK_PIN_LOW                   P07_low

#define DAT                                         1
#define CMD                                         0

#define Set_Lower_Column_Start_Address_CMD       0x00
#define Set_Higher_Column_Start_Address_CMD      0x10
#define Set_Memory_Addressing_Mode_CMD           0x20
#define Set_Column_Address_CMD                   0x21
#define Set_Page_Address_CMD                     0x22
#define Set_Display_Start_Line_CMD               0x40
#define Set_Contrast_Control_CMD                 0x81
#define Set_Charge_Pump_CMD                      0x8D
#define Set_Segment_Remap_CMD                    0xA0
#define Set_Entire_Display_ON_CMD                0xA4
#define Set_Normal_or_Inverse_Display_CMD        0xA6
#define Set_Multiplex_Ratio_CMD                  0xA8
#define Set_Display_ON_or_OFF_CMD                0xAE
#define Set_Page_Start_Address_CMD               0xB0
#define Set_COM_Output_Scan_Direction_CMD        0xC0

#define Set_Display_Offset_CMD                   0xD3
#define Set_Display_Clock_CMD                    0xD5
#define Set_Pre_charge_Period_CMD                0xD9
#define Set_Common_HW_Config_CMD                 0xDA
#define Set_VCOMH_Level_CMD                      0xDB
#define Set_NOP_CMD                              0xE3

#define Horizontal_Addressing_Mode               0x00
#define Vertical_Addressing_Mode                 0x01
#define Page_Addressing_Mode                     0x02
```

```
#define Disable_Charge_Pump                       0x00
#define Enable_Charge_Pump                        0x04

#define Column_Address_0_Mapped_to_SEG0           0x00
#define Column_Address_0_Mapped_to_SEG127         0x01

#define Normal_Display                            0x00
#define Entire_Display_ON                         0x01

#define Non_Inverted_Display                      0x00
#define Inverted_Display                          0x01

#define Display_OFF                               0x00
#define Display_ON                                0x01

#define Scan_from_COM0_to_63                      0x00
#define Scan_from_COM63_to_0                      0x08

#define x_size                                    128
#define x_max                                     x_size
#define x_min                                     0
#define y_size                                    64
#define y_max                                     8
#define y_min                                     0

#define ON                                        1
#define OFF                                       0

#define ROUND                                     1
#define SQUARE                                    0

#define buffer_size                               1024//(x_max * y_max)


unsigned char buffer[buffer_size];


void OLED_init(void);
void OLED_reset_sequence(void);
void OLED_write(unsigned char value, unsigned char type);
void OLED_gotoxy(unsigned char x_pos, unsigned char y_pos);
void OLED_fill(unsigned char bmp_data);
void OLED_clear_screen(void);
void OLED_clear_buffer(void);
void OLED_cursor(unsigned char x_pos, unsigned char y_pos);
void OLED_draw_bitmap(unsigned char xb, unsigned char yb, unsigned char xe, unsigned char ye, unsigned char b
mp_img[]);
void OLED_print_char(unsigned char x_pos, unsigned char y_pos, unsigned char ch);
void OLED_print_string(unsigned char x_pos, unsigned char y_pos, unsigned char *ch);
void OLED_print_chr(unsigned char x_pos, unsigned char y_pos, signed long value);
void OLED_print_int(unsigned char x_pos, unsigned char y_pos, signed long value);
void OLED_print_decimal(unsigned char x_pos, unsigned char y_pos, unsigned long value, unsigned char points);
void OLED_print_float(unsigned char x_pos, unsigned char y_pos, float value, unsigned char points);
void Draw_Pixel(unsigned char x_pos, unsigned char y_pos, short colour);
void Draw_Line(signed long x1, signed long y1, signed long x2, signed long y2, short colour);
void Draw_Rectangle(signed long x1, signed long y1, signed long x2, signed long y2, short fill, short colour,
 short type);
void Draw_Circle(signed long xc, signed long yc, signed long radius, short fill, short colour);
```

SSD1306.c

```
#include "SSD1306.h"
#include "fonts.c"


void OLED_init(void)
{
    P03_push_pull_mode;
    P04_push_pull_mode;
    P05_push_pull_mode;
    P06_push_pull_mode;
    P07_push_pull_mode;

    OLED_clear_buffer();

    OLED_reset_sequence();
```

```
    OLED_write((Set_Display_ON_or_OFF_CMD + Display_OFF) , CMD);;

    OLED_write(Set_Display_Clock_CMD, CMD);
    OLED_write(0x80, CMD);

    OLED_write(Set_Multiplex_Ratio_CMD, CMD);
    OLED_write(0x3F, CMD);

    OLED_write(Set_Display_Offset_CMD, CMD);
    OLED_write(0x00, CMD);

    OLED_write((Set_Display_Start_Line_CMD | 0x00), CMD);


    OLED_write(Set_Charge_Pump_CMD, CMD);
    OLED_write((Set_Higher_Column_Start_Address_CMD | 0x04), CMD);

    OLED_write(Set_Memory_Addressing_Mode_CMD, CMD);
    OLED_write(Page_Addressing_Mode, CMD);

    OLED_write((Set_Segment_Remap_CMD | Column_Address_0_Mapped_to_SEG127), CMD);

    OLED_write((Set_COM_Output_Scan_Direction_CMD | Scan_from_COM63_to_0), CMD);

    OLED_write(Set_Common_HW_Config_CMD, CMD);
    OLED_write(0x12, CMD);

    OLED_write(Set_Contrast_Control_CMD, CMD);
    OLED_write(0xCF, CMD);

    OLED_write(Set_Pre_charge_Period_CMD, CMD);
    OLED_write(0xF1, CMD);

    OLED_write(Set_VCOMH_Level_CMD, CMD);
    OLED_write(0x40, CMD);

    OLED_write((Set_Entire_Display_ON_CMD | Normal_Display), CMD);

    OLED_write((Set_Normal_or_Inverse_Display_CMD | Non_Inverted_Display), CMD);

    OLED_write((Set_Display_ON_or_OFF_CMD + Display_ON) , CMD);

    OLED_gotoxy(0, 0);

    OLED_clear_screen();
}


void OLED_reset_sequence(void)
{
    SSD1306_SCK_PIN_HIGH;
    SSD1306_RST_PIN_LOW;
    delay_ms(60);
    SSD1306_RST_PIN_HIGH;
    SSD1306_SCK_PIN_LOW;
    delay_ms(60);
}


void OLED_write(unsigned char value, unsigned char type)
{
    unsigned char s = 0x08;

    SSD1306_CS_PIN_LOW;

    switch(type)
    {
      case DAT:
      {
        SSD1306_DC_PIN_HIGH;
        break;
      }

      default:
      {
        SSD1306_DC_PIN_LOW;
        break;
```

```
        }
    }

    while(s > 0)
    {
      if((value & 0x80) != 0x00)
      {
        SSD1306_SDA_PIN_HIGH;
      }
      else
      {
        SSD1306_SDA_PIN_LOW;
      }

      SSD1306_SCK_PIN_HIGH;
      SSD1306_SCK_PIN_LOW;

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

    SSD1306_CS_PIN_HIGH;
}


void OLED_gotoxy(unsigned char x_pos, unsigned char y_pos)
{
    OLED_write((Set_Page_Start_Address_CMD + y_pos), CMD);
    OLED_write(((x_pos & 0x0F) | Set_Lower_Column_Start_Address_CMD), CMD);
    OLED_write((((x_pos & 0xF0) >> 0x04) | Set_Higher_Column_Start_Address_CMD), CMD);
}


void OLED_fill(unsigned char bmp_data)
{
    unsigned char x_pos = 0x00;
    unsigned char page = 0x00;

    for(page = y_min; page < y_max; page++)
    {
        OLED_write((Set_Page_Start_Address_CMD + page), CMD);
        OLED_write(Set_Lower_Column_Start_Address_CMD, CMD);
        OLED_write(Set_Higher_Column_Start_Address_CMD, CMD);

        for(x_pos = x_min; x_pos < x_max; x_pos++)
        {
            OLED_write(bmp_data, DAT);
        }
    }
}


void OLED_clear_screen(void)
{
    OLED_fill(0x00);
}


void OLED_clear_buffer(void)
{
     unsigned long s = 0x00;

     for(s = 0; s < buffer_size; s++)
     {
          buffer[s] = 0x00;
     }
}


void OLED_cursor(unsigned char x_pos, unsigned char y_pos)
{
    unsigned char i = 0x00;

    if(y_pos != 0x00)
    {
        if(x_pos == 1)
        {
```

```
            OLED_gotoxy(0x00, (y_pos + 0x02));
        }
        else
        {
            OLED_gotoxy((0x50 + ((x_pos - 0x02) * 0x06)), (y_pos + 0x02));
        }

        for(i = 0; i < 6; i++)
        {
            OLED_write(0xFF, DAT);
        }
    }
}


void OLED_draw_bitmap(unsigned char xb, unsigned char yb, unsigned char xe, unsigned char ye, unsigned char b
mp_img[])
{
    unsigned long s = 0x0000;
    unsigned char x_pos = 0x00;
    unsigned char y_pos = 0x00;

    for(y_pos = yb; y_pos <= ye; y_pos++)
    {
        OLED_gotoxy(xb, y_pos);
        for(x_pos = xb; x_pos < xe; x_pos++)
        {
            OLED_write(bmp_img[s++], DAT);
        }
    }
}


void OLED_print_char(unsigned char x_pos, unsigned char y_pos, unsigned char ch)
{
    unsigned char chr = 0x00;
    unsigned char s = 0x00;

    chr = (ch - 32);

    if(x_pos > (x_max - 6))
    {
        x_pos = 0;
        y_pos++;
    }
    OLED_gotoxy(x_pos, y_pos);

    for(s = 0x00; s < 0x06; s++)
    {
        OLED_write(font_regular[chr][s], DAT);
    }
}


void OLED_print_string(unsigned char x_pos, unsigned char y_pos, unsigned char *ch)
{
    unsigned char chr = 0x00;
    unsigned char i = 0x00;
    unsigned char j = 0x00;

    while(ch[j] != '\0')
    {
        chr = (ch[j] - 32);

        if(x_pos > (x_max - 0x06))
        {
            x_pos = 0x00;
            y_pos++;
        }
        OLED_gotoxy(x_pos, y_pos);

        for(i = 0x00; i < 0x06; i++)
        {
            OLED_write(font_regular[chr][i], DAT);
        }

        j++;
        x_pos += 6;
     }
```

```
}

void OLED_print_chr(unsigned char x_pos, unsigned char y_pos, signed long value)
{
    unsigned char ch = 0x00;

    if(value < 0x00)
    {
        OLED_print_char(x_pos, y_pos, '-');
        value = -value;
    }
    else
    {
        OLED_print_char(x_pos, y_pos,' ');
    }

     if((value > 99) && (value <= 999))
     {
         ch = (value / 100);
         OLED_print_char((x_pos + 6), y_pos , (48 + ch));
         ch = ((value % 100) / 10);
         OLED_print_char((x_pos + 12), y_pos , (48 + ch));
         ch = (value % 10);
         OLED_print_char((x_pos + 18), y_pos , (48 + ch));
     }
     else if((value > 9) && (value <= 99))
     {
         ch = ((value % 100) / 10);
         OLED_print_char((x_pos + 6), y_pos , (48 + ch));
         ch = (value % 10);
         OLED_print_char((x_pos + 12), y_pos , (48 + ch));
         OLED_print_char((x_pos + 18), y_pos , 32);
     }
     else if((value >= 0) && (value <= 9))
     {
         ch = (value % 10);
         OLED_print_char((x_pos + 6), y_pos , (48 + ch));
         OLED_print_char((x_pos + 12), y_pos , 32);
         OLED_print_char((x_pos + 18), y_pos , 32);
     }
}


void OLED_print_int(unsigned char x_pos, unsigned char y_pos, signed long value)
{
    unsigned char ch = 0x00;

    if(value < 0)
    {
        OLED_print_char(x_pos, y_pos, '-');
        value = -value;
    }
    else
    {
        OLED_print_char(x_pos, y_pos,' ');
    }

    if(value > 9999)
    {
        ch = (value / 10000);
        OLED_print_char((x_pos + 6), y_pos , (48 + ch));

        ch = ((value % 10000)/ 1000);
        OLED_print_char((x_pos + 12), y_pos , (48 + ch));

        ch = ((value % 1000) / 100);
        OLED_print_char((x_pos + 18), y_pos , (48 + ch));

        ch = ((value % 100) / 10);
        OLED_print_char((x_pos + 24), y_pos , (48 + ch));

        ch = (value % 10);
        OLED_print_char((x_pos + 30), y_pos , (48 + ch));
    }

    else if((value > 999) && (value <= 9999))
    {
        ch = ((value % 10000)/ 1000);
```

```
        OLED_print_char((x_pos + 6), y_pos , (48 + ch));

        ch = ((value % 1000) / 100);
        OLED_print_char((x_pos + 12), y_pos , (48 + ch));

        ch = ((value % 100) / 10);
        OLED_print_char((x_pos + 18), y_pos , (48 + ch));

        ch = (value % 10);
        OLED_print_char((x_pos + 24), y_pos , (48 + ch));
        OLED_print_char((x_pos + 30), y_pos , 32);
    }
    else if((value > 99) && (value <= 999))
    {
        ch = ((value % 1000) / 100);
        OLED_print_char((x_pos + 6), y_pos , (48 + ch));

        ch = ((value % 100) / 10);
        OLED_print_char((x_pos + 12), y_pos , (48 + ch));

        ch = (value % 10);
        OLED_print_char((x_pos + 18), y_pos , (48 + ch));
        OLED_print_char((x_pos + 24), y_pos , 32);
        OLED_print_char((x_pos + 30), y_pos , 32);
    }
    else if((value > 9) && (value <= 99))
    {
        ch = ((value % 100) / 10);
        OLED_print_char((x_pos + 6), y_pos , (48 + ch));

        ch = (value % 10);
        OLED_print_char((x_pos + 12), y_pos , (48 + ch));

        OLED_print_char((x_pos + 18), y_pos , 32);
        OLED_print_char((x_pos + 24), y_pos , 32);
        OLED_print_char((x_pos + 30), y_pos , 32);
    }
    else
    {
        ch = (value % 10);
        OLED_print_char((x_pos + 6), y_pos , (48 + ch));
        OLED_print_char((x_pos + 12), y_pos , 32);
        OLED_print_char((x_pos + 18), y_pos , 32);
        OLED_print_char((x_pos + 24), y_pos , 32);
        OLED_print_char((x_pos + 30), y_pos , 32);
    }
}


void OLED_print_decimal(unsigned char x_pos, unsigned char y_pos, unsigned long value, unsigned char points)
{
    unsigned char ch = 0x00;

    OLED_print_char(x_pos, y_pos, '.');

    ch = (value / 1000);
    OLED_print_char((x_pos + 6), y_pos , (48 + ch));

    if(points > 1)
    {
        ch = ((value % 1000) / 100);
        OLED_print_char((x_pos + 12), y_pos , (48 + ch));


        if(points > 2)
        {
            ch = ((value % 100) / 10);
            OLED_print_char((x_pos + 18), y_pos , (48 + ch));

            if(points > 3)
            {
                ch = (value % 10);
                OLED_print_char((x_pos + 24), y_pos , (48 + ch));
            }
        }
    }
}
```

```
void OLED_print_float(unsigned char x_pos, unsigned char y_pos, float value, unsigned char points)
{
    signed long tmp = 0x0000;

    tmp = value;
    OLED_print_int(x_pos, y_pos, tmp);
    tmp = ((value - tmp) * 10000);

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

    if((value >= 9999) && (value < 99999))
    {
        OLED_print_decimal((x_pos + 36), y_pos, tmp, points);
    }
    else if((value >= 999) && (value < 9999))
    {
        OLED_print_decimal((x_pos + 30), y_pos, tmp, points);
    }
    else if((value >= 99) && (value < 999))
    {
        OLED_print_decimal((x_pos + 24), y_pos, tmp, points);
    }
    else if((value >= 9) && (value < 99))
    {
        OLED_print_decimal((x_pos + 18), y_pos, tmp, points);
    }
    else if(value < 9)
    {
        OLED_print_decimal((x_pos + 12), y_pos, tmp, points);
        if((value) < 0)
        {
            OLED_print_char(x_pos, y_pos, '-');
        }
        else
        {
            OLED_print_char(x_pos, y_pos, ' ');
        }
    }
}


void Draw_Pixel(unsigned char x_pos, unsigned char y_pos, short colour)
{
    unsigned char value = 0x00;
    unsigned char page = 0x00;
    unsigned char bit_pos = 0x00;

    page = (y_pos / y_max);
    bit_pos = (y_pos - (page * y_max));
    value = buffer[((page * x_max) + x_pos)];

    if((colour & 0x01) != 0)
    {
        value |= (1 << bit_pos);
    }
    else
    {
        value &= (~(1 << bit_pos));
    }

    buffer[((page * x_max) + x_pos)] = value;
    OLED_gotoxy(x_pos, page);
    OLED_write(value, DAT);
}


void Draw_Line(signed long x1, signed long y1, signed long x2, signed long y2, short colour)
{
    signed long dx = 0x0000;
    signed long dy = 0x0000;
    signed long stepx = 0x0000;
    signed long stepy = 0x0000;
    signed long fraction = 0x0000;
```

```
    dy = (y2 - y1);
    dx = (x2 - x1);

    if (dy < 0)
    {
        dy = -dy;
        stepy--;
    }
    else
    {
        stepy = 1;
    }

    if (dx < 0)
    {
        dx = -dx;
        stepx--;
    }
    else
    {
        stepx = 1;
    }

    dx <<= 1;
    dy <<= 1;

    Draw_Pixel(((unsigned char)x1), ((unsigned char)y1), colour);

    if(dx > dy)
    {
        fraction = (dy - (dx >> 1));
        while (x1 != x2)
        {
            if(fraction >= 0)
            {
                y1 += stepy;
                fraction -= dx;
            }

            x1 += stepx;
            fraction += dy;

            Draw_Pixel(((unsigned char)x1), ((unsigned char)y1), colour);
        }
    }
    else
    {
        fraction = (dx - (dy >> 1));
        while (y1 != y2)
        {
            if (fraction >= 0)
            {
                x1 += stepx;
                fraction -= dy;
            }

            y1 += stepy;
            fraction += dx;

            Draw_Pixel(((unsigned char)x1), ((unsigned char)y1), colour);
        }
    }
}


void Draw_Rectangle(signed long x1, signed long y1, signed long x2, signed long y2, short fill, short colour,
 short type)
{
     unsigned char i = 0x00;
     unsigned char xmin = 0x00;
     unsigned char xmax = 0x00;
     unsigned char ymin = 0x00;
     unsigned char ymax = 0x00;

     if(fill == ON)
     {
        if(x1 < x2)
        {
           xmin = x1;
```

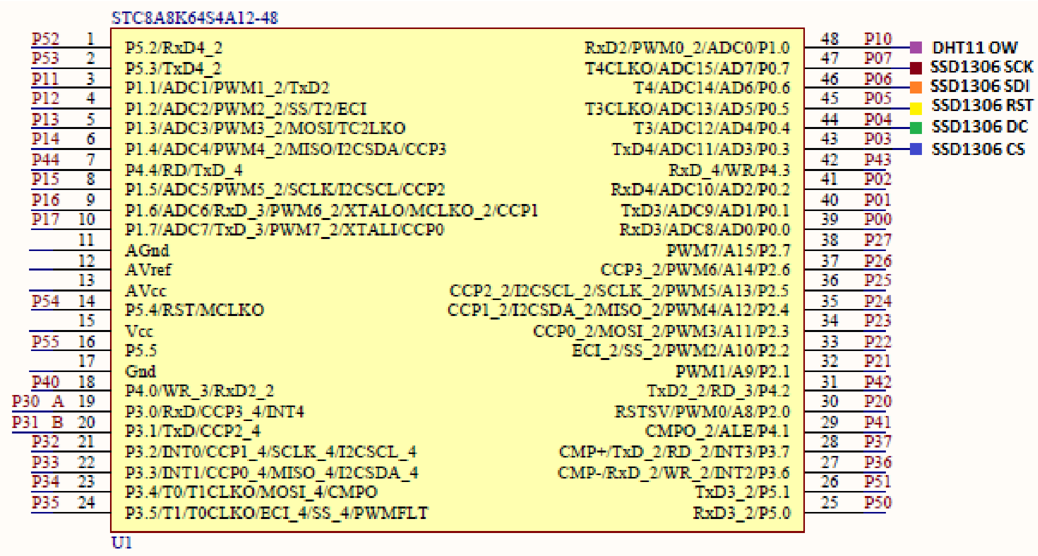
```
            xmax = x2;
        }
        else
        {
            xmin = x2;
            xmax = x1;
        }

        if(y1 < y2)
        {
            ymin = y1;
            ymax = y2;
        }
        else
        {
            ymin = y2;
            ymax = y1;
        }

        for(i = ymin; i <= ymax; ++i)
        {
             Draw_Line(xmin, i, xmax, i, colour);
        }

     }

     else
     {
        Draw_Line(x1, y1, x2, y1, colour);
        Draw_Line(x1, y2, x2, y2, colour);
        Draw_Line(x1, y1, x1, y2, colour);
        Draw_Line(x2, y1, x2, y2, colour);
     }


     if(type != SQUARE)
     {
         Draw_Pixel(((unsigned char)x1), ((unsigned char)y1), ~colour);
         Draw_Pixel(((unsigned char)x1), ((unsigned char)y2), ~colour);
         Draw_Pixel(((unsigned char)x2), ((unsigned char)y1), ~colour);

         Draw_Pixel(((unsigned char)x2), ((unsigned char)y2), ~colour);
     }
}


void Draw_Circle(signed long xc, signed long yc, signed long radius, short fill, short colour)
{
   signed long a = 0x0000;
   signed long b = 0x0000;
   signed long P = 0x0000;

   b = radius;
   P = (1 - b);

   do
   {
        if(fill == ON)
        {
            Draw_Line((xc - a), (yc + b), (xc + a), (yc + b), colour);
            Draw_Line((xc - a), (yc - b), (xc + a), (yc - b), colour);
            Draw_Line((xc - b), (yc + a), (xc + b), (yc + a), colour);
            Draw_Line((xc - b), (yc - a), (xc + b), (yc - a), colour);
        }
        else
        {
            Draw_Pixel((xc + a), (yc + b), colour);
            Draw_Pixel((xc + b), (yc + a), colour);
            Draw_Pixel((xc - a), (yc + b), colour);
            Draw_Pixel((xc - b), (yc + a), colour);
            Draw_Pixel((xc + b), (yc - a), colour);
            Draw_Pixel((xc + a), (yc - b), colour);
            Draw_Pixel((xc - a), (yc - b), colour);
            Draw_Pixel((xc - b), (yc - a), colour);
        }

        if(P < 0)
        {
            P += (3 + (2 * a++));
```

```
        }
        else
        {
           P += (5 + (2 * ((a++) - (b--))));
        }
    }while(a <= b);
}
```

main.c

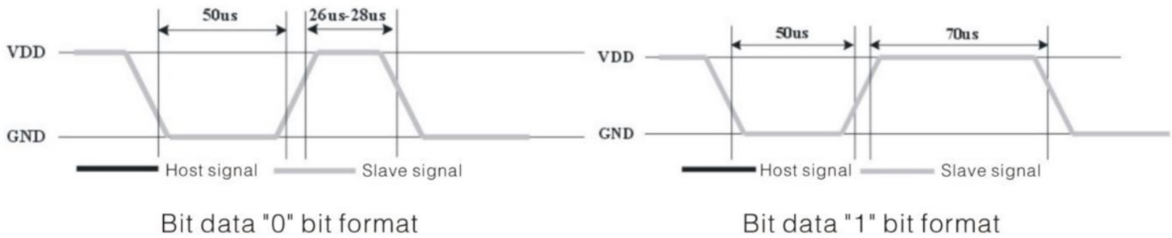
```
#include "STC8xxx.h"
#include "BSP.h"
#include "DHT11.c"
#include "SSD1306.c"


void setup(void);


void main(void)
{
  unsigned char state = 0x00;

  setup();

  OLED_print_string(24, 0, "Temp/Deg C");
  OLED_print_string(24, 4, "Rel.Hum /%");

  while(1)
  {
    state = DHT11_get_data();

    switch(state)
    {
      case 1:
      {
      }
      case 2:
      {
         OLED_print_string(56, 2, "   ");
         OLED_print_string(56, 6, "   ");
         break;
      }
      case 3:
      {
         OLED_print_string(56, 2, "--");
         OLED_print_string(56, 6, "--");
         break;
      }
      default:
      {
         OLED_print_chr(55, 2, values[2]);
         OLED_print_chr(55, 6, values[0]);
         break;
      }
    }

    delay_ms(1000);
  };
}


void setup(void)
{
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_no_output, MCLK_out_P54);

  DHT11_init();
  OLED_init();
}
```

Schematic



Explanation

DHT11's logical one and zero timing diagrams below show that it too uses time-slotting mechanism just like DS18B20. However, its way of communication or protocol is not same as that of DS18B20's and codes are not compatible amongst each other.



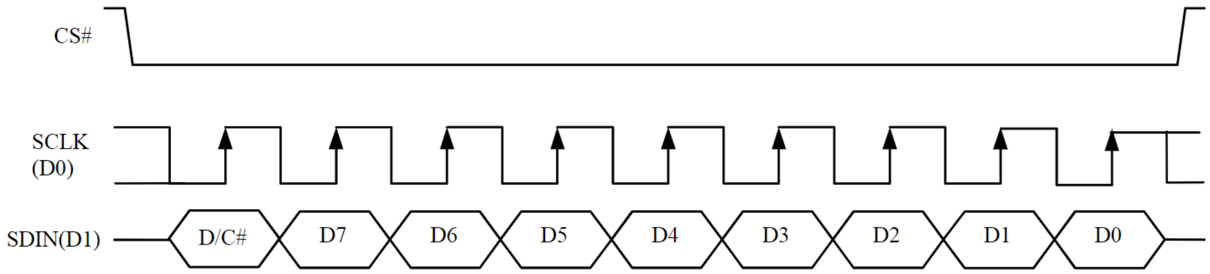
From the above timing diagrams, it is clear that the difference between these two signals is the high time. Thus, after sensing low input we should wait for about 30µs before trying to determine if the sensor sent a *zero* time-slot or a *one* time-slot. This is so because if it sent a logical zero then after 30µs of delay, there should be no floating or logic high signal in DHT11's communication pin. If it is otherwise then a logical *one* time slot should be expected. This is what the following code does.

```
unsigned char DHT11_get_byte(void)
{
   unsigned char s = 0x08;
   unsigned char value = 0x00;

   while(s > 0)
   {
      value <<= 1;
      while(DHT11_pin_IN == LOW);
      large_delay_TMR_0(30);

      if(DHT11_pin_IN == HIGH)
      {
         value |= 1;
      }
```

```
        while(DHT11_pin_IN == HIGH);
        s--;
    }
    return value;
}
```

At this point, it is necessary to mention that instead of traditional software-based delay library that relies on wasting CPU cycles, hardware-based delay library has been used in this code. This library (*tmr_delay.h*) generates more accurate time delays.

Now let's see how SSD1306 OLED display can be bit-banged. First, notice its timing diagram.



The timing diagram suggests:

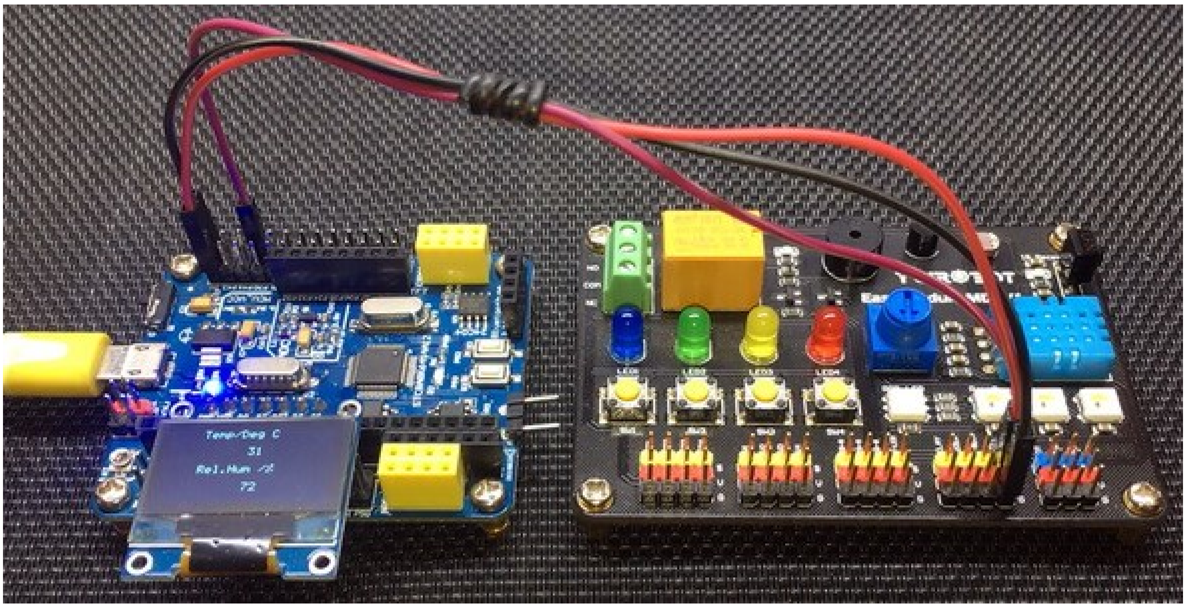
- SPI clock (SCLK) is held low in idle mode.
- SPI data (SDIN) is shifted or clocked during the rising edge or low-to-high transition of SPI clock.
- SPI chip/slave select (CS) should be held low during data transfer only and for the rest of the times, it should be held high.

All these points suggest us that it is a Mode 0 SPI communication and we can craft a code using these data as shown below. It is a mere timing diagram to code translation.

```
void OLED_write(unsigned char value, unsigned char type)
{
    unsigned char s = 0x08;

    SSD1306_CS_PIN_LOW;

    switch(type)
    {
      case DAT:
      {
        SSD1306_DC_PIN_HIGH;
        break;
      }

      default:
      {
        SSD1306_DC_PIN_LOW;
        break;
      }
    }

    while(s > 0)
    {
      if((value & 0x80) != 0x00)
      {
        SSD1306_SDA_PIN_HIGH;
      }
      else
      {
        SSD1306_SDA_PIN_LOW;
```

```
      }

      SSD1306_SCK_PIN_HIGH;
      SSD1306_SCK_PIN_LOW;

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

    SSD1306_CS_PIN_HIGH;
}
```

Demo



Demo video link: https://youtu.be/IP4_gTHhLRI.

Software I2C and Software UART

After trying to grasp the past software-based communication techniques, I think reader should now have some idea of the tricks employed to do the tasks.

At this final stage, we are left with software I2C and software UART. The tricks behind software I2C are similar to our past encounters with one wire communications and TM1637. However, software UART implementation is a bit complicated.

The demo for this example is a simple multi-UART chatting. One UART is hardware-based while the other is based on software. The chatting between the UARTs is displayed on I2C-based serial LCD. This LCD is driven using I2C-based PCF8574 GPIO expander.

Code

SW_I2C.h

```
#define SDA_DIR_OUT()   P10_push_pull_mode
#define SDA_DIR_IN()    P10_input_mode

#define SCL_DIR_OUT()   P11_push_pull_mode
#define SCL_DIR_IN()    P11_input_mode

#define SDA_HIGH()      P10_high
#define SDA_LOW()       P10_low

#define SCL_HIGH()      P11_high
#define SCL_LOW()       P11_low

#define SDA_IN()        P10_get_input

#define I2C_ACK         0xFF
#define I2C_NACK        0x00
```

```
#define I2C_timeout     1000


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

SW_I2C.c

```
#include "SW_I2C.h"


void SW_I2C_init(void)
{
    SDA_DIR_OUT();
    SCL_DIR_OUT();
    delay_ms(10);
    SDA_HIGH();
    SCL_HIGH();
}


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


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


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

    SDA_DIR_IN();

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

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

        delay_us(10);
        i--;
    };

    switch(ack)
```

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

    return j;
}


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

    SDA_DIR_OUT();
    SCL_LOW();

    while(i > 0)
    {

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


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


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

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

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


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

```
    SDA_DIR_IN();

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

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

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

    SCL_LOW();
    return 0;
}
```

PCF8574.h

```
#include "SW_I2C.c"


#define PCF8574_address                  0x4E

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


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

PCF8574.c

```
#include "PCF8574.h"


void PCF8574_init(void)
{
    SW_I2C_init();
    delay_ms(10);
}


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

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

    return port_byte;
}


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

LCD_2_Wire.h

```
#include "PCF8574.c"


#define clear_display                                   0x01
#define goto_home                                       0x02

#define cursor_direction_inc                            (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 "LCD_2_Wire.h"


static unsigned char bl_state;
static unsigned char data_value;


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

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

  delay_ms(10);

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

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

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


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

  switch(bl_state)
  {
     case BL_ON:
     {
        data_value |= 0x08;
        break;
     }
     case BL_OFF:
     {
        data_value &= 0xF7;
        break;
     }
  }

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


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

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

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


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


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

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


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


void LCD_goto(unsigned char x_pos,unsigned char y_pos)
{
  if(y_pos == 0)
  {
    LCD_send((0x80 | x_pos), CMD);
  }
  else
  {
    LCD_send((0x80 | 0x40 | x_pos), CMD);
  }
}
```

soft_UART.h

```
#define tmr_max_cnt     0xFFFF

//sysclk
#define sysclk              12000000

//baud rate
#define baud_rate           9600

//T_value
#define _1T                 0x01
#define _12T                0x0C

#define tmr_load_value      (tmr_max_cnt - (sysclk / 3 / baud_rate / _1T))

sbit RXD_pin = P1^6;
sbit TXD_pin = P1^7;


unsigned char TXing = 0x00;
unsigned char RXing = 0x00;
unsigned char TX_Bit = 0x00;
unsigned char RX_Bit = 0x00;
unsigned char TX_CNT = 0x00;
unsigned char RX_CNT = 0x00;
unsigned char TX_Data = 0x00;
unsigned char RX_Data = 0x00;
unsigned char TX_done = 0x00;
unsigned char RX_done = 0x00;
unsigned char TX_Buffer = 0x00;
unsigned char RX_Buffer = 0x00;


void soft_UART_init(void);
```

soft_UART.c

```
#include "soft_UART.h"


void soft_UART_init(void)
{
  TXing = FALSE;
  RXing = FALSE;
  TX_CNT = 0x00;
  RX_CNT = 0x00;
  TX_done = TRUE;
```

```
  RX_done = FALSE;

  TMR2_setup(TMR2_sysclk, \
             TMR2_clk_prescalar_1T);

  TMR2_load_counter_16(tmr_load_value);

  TMR2_start;

  _enable_TMR_2_interrupt;
  _enable_global_interrupt;
}


void TMR_2_ISR(void)
interrupt 12
{
    if(RXing == TRUE)
    {
        if(--RX_CNT == 0x00)
        {
            RX_CNT = 0x03;

            if(--RX_Bit == 0x00)
            {
                RX_Buffer = RX_Data;
                RXing = FALSE;
                RX_done = TRUE;
            }

            else
            {
                RX_Data >>= 0x01;

                if(RXD_pin == HIGH)
                {
                    RX_Data |= 0x80;
                }
            }
        }
    }

    else if(RXD_pin == FALSE)
    {
        RXing = TRUE;
        RX_CNT = 0x04;
        RX_Bit = 0x09;
    }

    if(--TX_CNT == 0x00)
    {
        TX_CNT = 0x03;

        if(TXing == TRUE)
        {
            if(TX_Bit == 0x00)
            {
                TXD_pin = LOW;
                TX_Data = TX_Buffer;
                TX_Bit = 0x09;
            }

            else
            {
                TX_Data >>= 0x01;

                if(--TX_Bit == 0x00)
                {
                    TXD_pin = HIGH;
                    TXing = FALSE;
                    TX_done = TRUE;
                }

                else
                {
                    TXD_pin = CY;
                }
            }
```

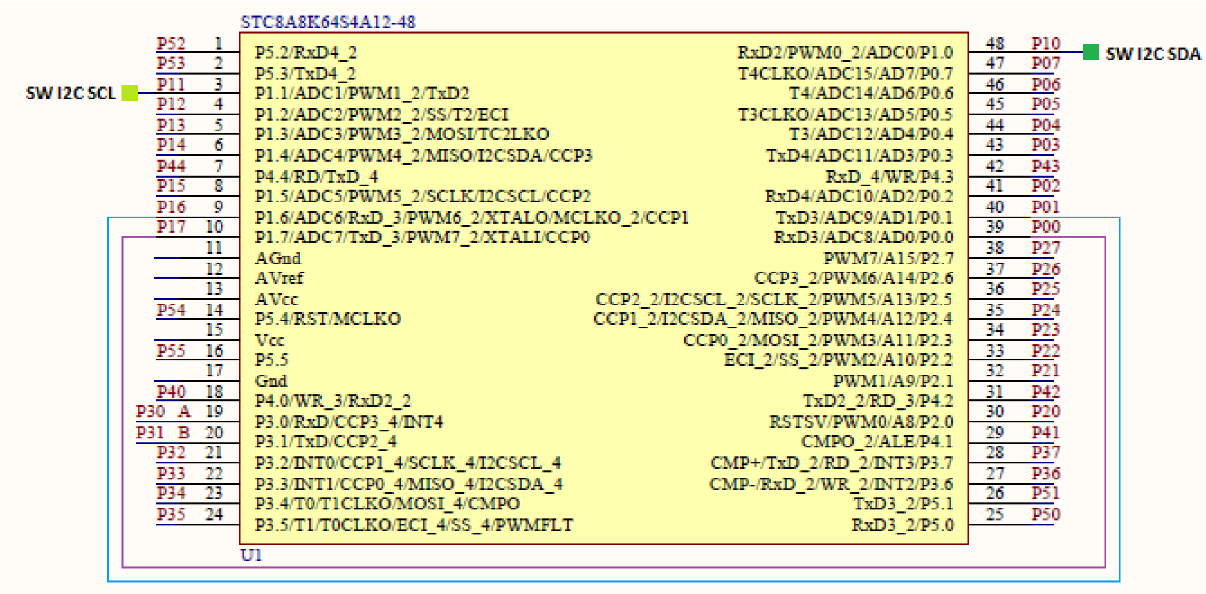
```
        }
    }
}
```

main.c

```
#include "STC8xxx.h"
#include "BSP.h"
#include "LCD_2_Wire.c"
#include "soft_UART.c"


void setup(void);


void main(void)
{
  unsigned char msg1[10] = {"0123456789"};
  unsigned char msg2[10] = {"!@#$%^&*()"};

  char i = 0x00;

  char rcv_s = 0x00;
  char rcv_3 = 0x00;

  setup();

  LCD_goto(0, 0);
  LCD_putstr("TXDs: ");
  LCD_goto(10, 0);
  LCD_putstr("TXD3: ");

  LCD_goto(0, 1);
  LCD_putstr("RXDs: ");
  LCD_goto(10, 1);
  LCD_putstr("RXD3: ");

  while(1)
  {
    for(i = 0; i < 10; i++)
    {
      if(TX_done == TRUE)
      {
          TX_done = FALSE;
          TX_Buffer = msg1[i];
          TXing = TRUE;
      }

      UART3_write_buffer(msg2[i]);

      LCD_goto(5, 0);
      LCD_putchar(msg1[i]);
      LCD_goto(15, 0);
      LCD_putchar(msg2[i]);

      if(RX_done == TRUE)
      {
        RX_done = FALSE;
        rcv_s = RX_Buffer;
      }

      rcv_3 = UART3_read_buffer();

      LCD_goto(5, 1);
      LCD_putchar(rcv_s);
      LCD_goto(15, 1);
      LCD_putchar(rcv_3);

      delay_ms(900);
    }
  };
}


void setup(void)
{
```

```
  CLK_set_sys_clk(IRC_24M, 2, MCLK_SYSCLK_div_1, MCLK_out_P54);

  UART3_pin_option(0x00);

  UART3_init(9600, \
             UART3_baud_source_TMR3, \
             UART3_timer_1T, \
             12000000);

  LCD_init();
  LCD_clear_home();
  soft_UART_init();
}
```

Schematic



Explanation

To use software UART efficiently, the following definitions have to defined properly. Unlike the past software communications that we have seen so far, software UART requires the use of a timer its overflow interrupt. Hardware UARTs also need timers for baud rate generation but here in software UART, a timer has a different but somewhat similar use. UART, itself, is asynchronous in nature and so its data timings must be as accurate as possible in order to reduce data loss, error and corruption. This fact becomes a more serious matter when it comes to software UART because not only we are dealing with an asynchronous communication but we will also be dealing it with software solution instead of actual hardware.

```
//sysclk
#define sysclk              12000000

//baud rate
#define baud_rate           9600

//T_value
#define _1T                 0x01
#define _12T                0x0C

#define tmr_load_value      (tmr_max_cnt - (sysclk / 3 / baud_rate / _1T))

sbit RXD_pin = P1^6;
sbit TXD_pin = P1^7;
```

Initialization of software UART is pretty easy. There are lot of variables that need to be initialized with some defaults and the timer to be used should be initialized and started along with the enabling of the interrupts.

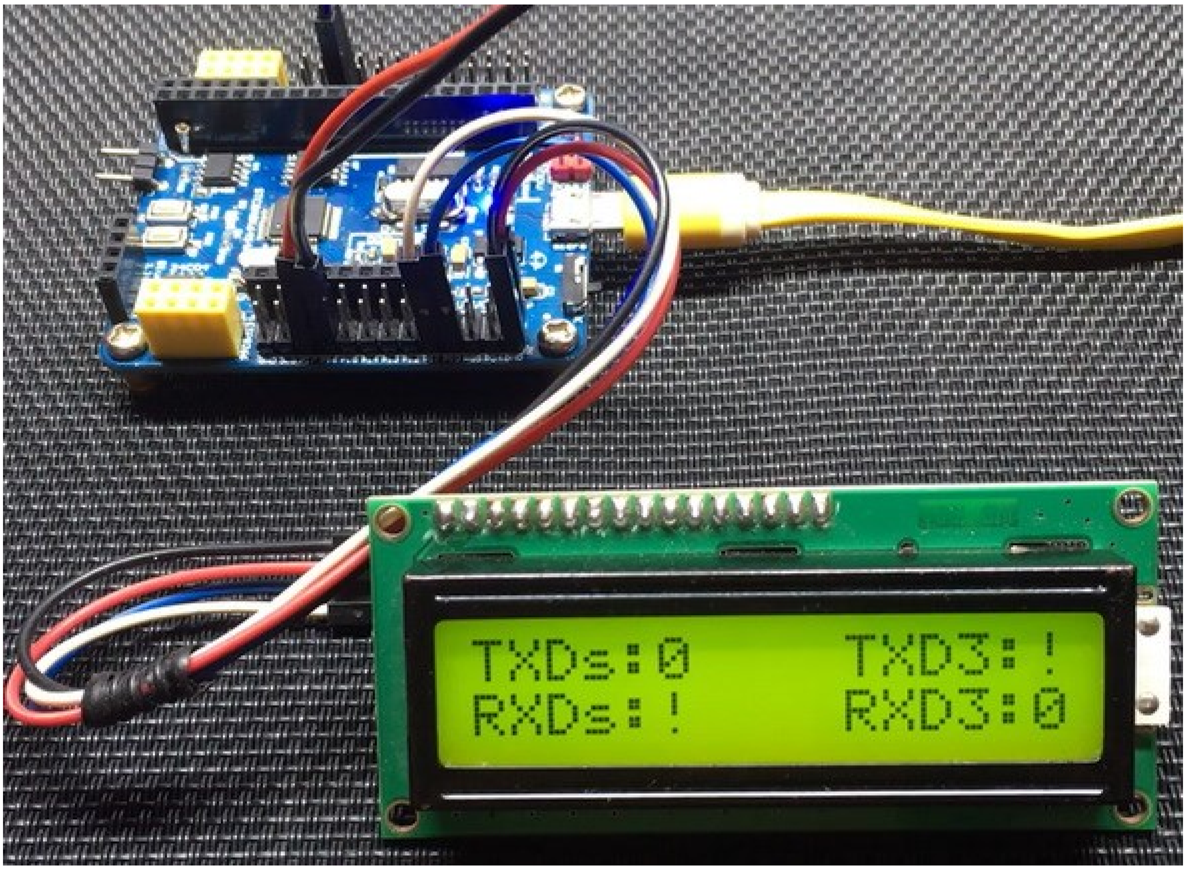
```
void soft_UART_init(void)
{
  TXing = FALSE;
  RXing = FALSE;
  TX_CNT = 0x00;
  RX_CNT = 0x00;
  TX_done = TRUE;
  RX_done = FALSE;

  TMR2_setup(TMR2_sysclk, \
             TMR2_clk_prescalar_1T);

  TMR2_load_counter_16(tmr_load_value);

  TMR2_start;

  _enable_TMR_2_interrupt;
  _enable_global_interrupt;
}
```

Inside the timer interrupt, data transmission and reception operations are done independently of the main loop. The whole operation is done in a similar process as one would expect with a real hardware UART. There are flags and buffers for both TX and RX operations and the timer's interrupt is used to simulate baud rate as perfectly as possible.

```
void TMR_2_ISR(void)
interrupt 12
{
    if(RXing == TRUE)
    {
        if(--RX_CNT == 0x00)
        {
            RX_CNT = 0x03;

            if(--RX_Bit == 0x00)
            {
                RX_Buffer = RX_Data;
                RXing = FALSE;
                RX_done = TRUE;
            }

            else
            {
                RX_Data >>= 0x01;

                if(RXD_pin == HIGH)
                {
                    RX_Data |= 0x80;
                }
            }
        }
    }

    else if(RXD_pin == FALSE)
    {
        RXing = TRUE;
        RX_CNT = 0x04;
        RX_Bit = 0x09;
    }

    if(--TX_CNT == 0x00)
    {
        TX_CNT = 0x03;

        if(TXing == TRUE)
        {
            if(TX_Bit == 0x00)
            {
```

```
                TXD_pin = LOW;
                TX_Data = TX_Buffer;
                TX_Bit = 0x09;
            }

            else
            {
                TX_Data >>= 0x01;

                if(--TX_Bit == 0x00)
                {
                    TXD_pin = HIGH;
                    TXing = FALSE;
                    TX_done = TRUE;
                }

                else
                {
                    TXD_pin = CY;
                }
            }

        }
    }
}
```

Demo



Demo video link: https://youtu.be/7AUjfIpg9a0.

Epilogue

As with any new microcontroller, I enjoyed developing stuffs for STC micros too. Every hardware peripheral was a new experience for me, especially the PCA module. The reasons why I love exploring 8051 core-based MCUs are their relative ease of usage, compatibility and similarity. STC microcontrollers were no exceptions.

There were times during the forging of this work, I did face some odd issues while cherished at some other points and here in the end I would like share some of them.

- While trying out multichannel ADCs, I did notice significant interference amongst channels but later I fixed it by changing ADC clock speed.

- The analogue frontend of STC micros need special attention when designing PCBs because things may not look good if things as stated in the first point happen.

- The more the resolution of an ADC, the more noise it is likely to pick and so external Op-Amp-based buffering and filtering should be considered. Good external reference voltage source should be used for good accuracy.

- As with any 8051-based MCU, memory model is a very important stuff and should be considered in some cases, especially when dealing with pointers and large arrays. The following link describes memory model in details specifically for Keil C51 C compiler: https://www.keil.com/support/man/docs/c51/c51_le_memmodels.htm.

- The programmer GUI is a tool that has many optional functions that are really very helpful. I advise updating the GUI as new releases are uploaded.

- Since STC's portfolio consists of wide variety of 8051-based MCUs, it is possible to use the BSP header files I coded in this document and slightly modify for other STC MCUs.

- STC should invest on a free-compiler of its own. In the past, I have criticized many Chinese semiconductor manufacturers for their lack of software support and STC doesn't stand different from the others. The same is true for official development boards.

Finally, I would like to state that STC micros are game changers in the market of 8051-based microcontrollers. They are reliable, cheap and in fact widely used in Chinese and Asian markets. Personally, I feel delighted to have introduced them to the non-Chinese world. STC micros are without any doubt good micros for any new design.

Code Examples and Libraries used in this tutorial can be downloaded from here.

All demo videos of this tutorial can be found in this Youtube playlist.

Happy coding.

Author: Shawon M. Shahryiar
https://www.facebook.com/groups/microarena
https://www.facebook.com/MicroArena

03.12.2021