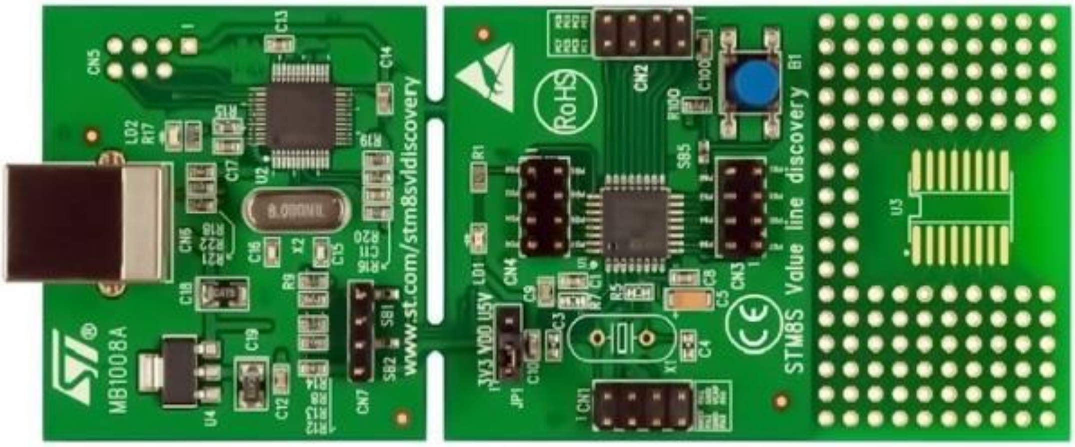
Starting STM8 Microcontrollers

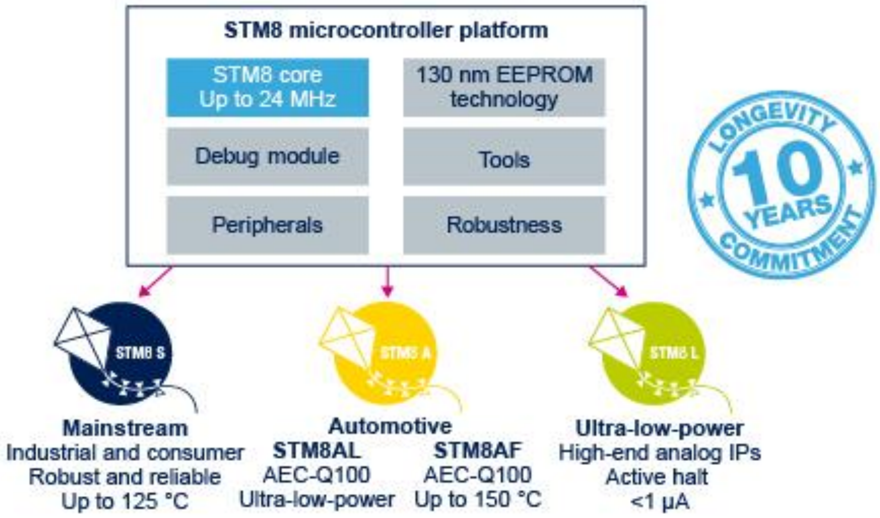
STM8 microcontrollers are 8-bit general purpose microcontrollers from **STMicroelectronics (STM)**. STM is famous mainly for its line of 32-bit ARM Cortex microcontrollers – the STM32s. STM8 microcontrollers are rarely discussed in that context. However, STM8 MCUs are robust and most importantly they come packed with lots of hardware features. Except for the ARM core, 32-bit architecture, performance and some minor differences, STM8s have many peripheral similarities to STM32s. In my opinion, STM8s are equally or sometimes more matched than the popular PICs and AVRs in all areas. Unlike PICs and AVRs however, I have seen STM8s mostly in various SMD packages. Only a handful of STM8 chips are available in PDIP/through-hole packages. I think it is a big reason for which most small industries and hobbyists don't play with them as much as with other 8-bit families. People like to setup their test projects in breadboards, trial PCBs or strip-boards first, prototype and then develop for production. To cope with this issue, STM has provided several affordable STM8 Discovery (Disco) boards to get started with. Besides there are many cheap STM8 breakout-boards from China.



I have experience playing with AVRs, PICs, 8051s, STM32s, MSP430s, TivaC and so on. To be honest, I thought learning about STM8 micros is a pure waste of time and energy. The learning curve will be steep. Things and tools would be different and thus difficult. However, gradually I found these MCUs very useful and there's literally no complexity at all. The main drive factor for learning STM8s is the price factor. They are hell cheap. When it comes down to other things, I have not found any book on STM8s written in English. There's literally no 100% complete blog post on the internet that shows the basics. Similarly, same story with tools. I have been using MikroC for AVRs, 8051s and ARMs and it is my favourite but at the time of writing, there's no MikroC compiler for STM8 family. I have also not stumbled upon any Arduino-like IDE that supports STM8 micros. Arduino-based solutions are also not my favourite as they don't go deep and have several limitations. Maybe it is not my luck. After much study and search, I found out that there are a few C compilers for STM8s. However, any new tool is both different and difficult at first. It is not always easy to adapt to new environments. You may never know what unanticipated challenges and harshness a new environment may throw at you even when you reach certain levels of expertise. I also don't want to use any pirated software and so a free compiler was a major requirement. I found out ST Visual Develop and Cosmic COSC compiler are both free tools. Cosmic used to be a paid tool but now it is absolutely free. The only easy thing till then was buying the STM8S Value Line Discovery board for just a few dollars and downloading the stuffs.

The STM8 Family

There are over a hundred STM8 microcontrollers available today. The STM8 family can be simplified into three categorical groups as shown below.

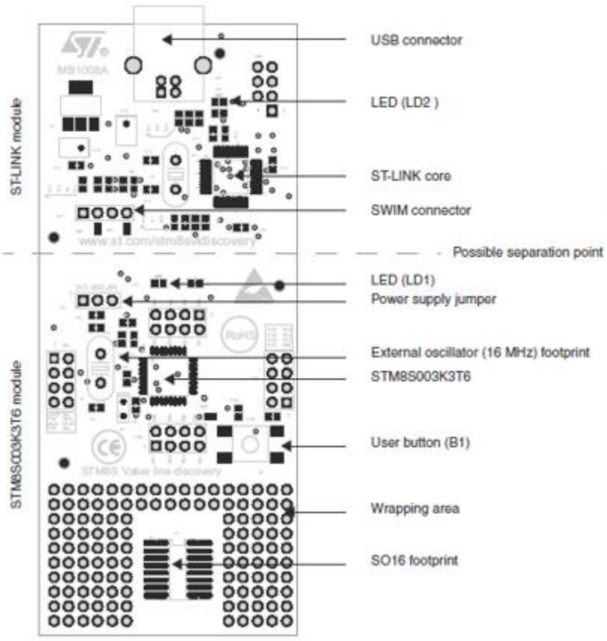


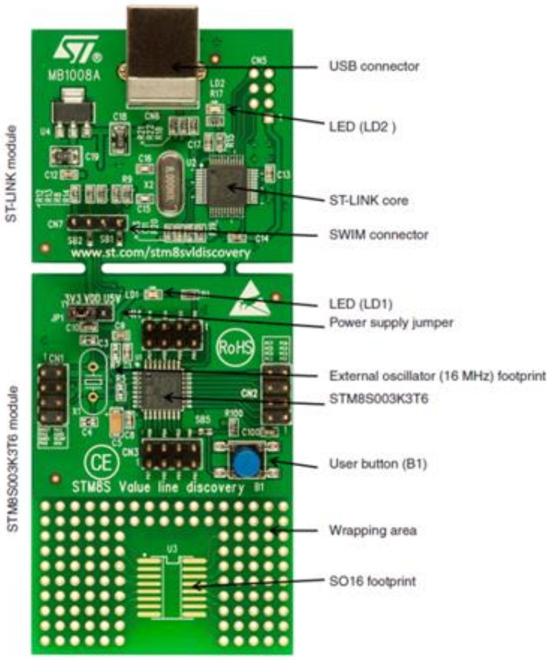
There are subgroups within these groups but broadly speaking these three groups are what by which we can define the entire family. STM8S micros are general purpose robust and reliable micros that can be employed in almost all scopes. This is the most commonly used group and in fact we will be exploring it in this article. They are also cheap and smart. The second group – the STM8A family is intended mainly for automotive industries. This group is packed with additional hardware interfaces like CAN and LIN that are musts according to present-day automotive industry doctrine. The STM8As are also very robust and are designed to withstand the harsh extremes of an automobile. For instance, STM8As can withstand high temperatures, in excess of 100°C. The last group consists of STM8L micros which are crafted for low power or battery-backed applications. Virtually they consume no power in idle mode. Thus, if you need high power savings or energy cuts in your projects, this group is the best choice. There are also low power versions of automotive-standard STM8 micros that are labelled STM8AL. Apart from all these there is also one version of STM8 micros that are specifically designed for capacitive touch applications. These are called STM8Ts.

The features and benefits of STM8 micros are numerous and can't simply be expressed in few words. The more you explore, the more you will feel. STM8s can be powered with 3.3V or 5V DC power supplies and have built-in brownout detection circuitry. The low power editions can operate at much lower voltages than these values. Official STM8 Discovery boards come with voltage selection jumpers to allow users to select operating voltage level as required. There is very minimum risk of program corruption due to EMI or some other similar unprecedented factors. There is fail-safe security for the clock system which ensures that a system based on a STM8 micro won't stop working or stuck up should its external clock source fail. All internal hardware possesses more features than any other competitive 8-bit microcontroller families that are widely available in the market. The best part is the price benefit. You pay less for the most. All these features are well-suited for extremely harsh industrial environments. STM8s are designed with maximum possible combinations of features. Beyond your wildest wet dream, there are many extraordinary stuffs waiting to be unboxed.

Overview of the Discovery Board

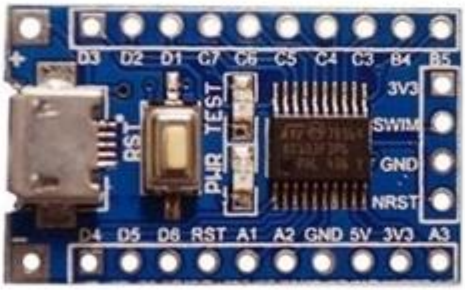
For getting started with STM8s, STM has provided several STM8 Disco boards. There are also other third party boards too. However, I strongly recommend Disco boards for learning and experimental purposes. There are several reasons for this recommendation. One main reason is the fact that all Disco boards come with on-board detachable ST-Link programmers and they are extremely cheap. Shown below is the top layout of a STM8S discovery board.

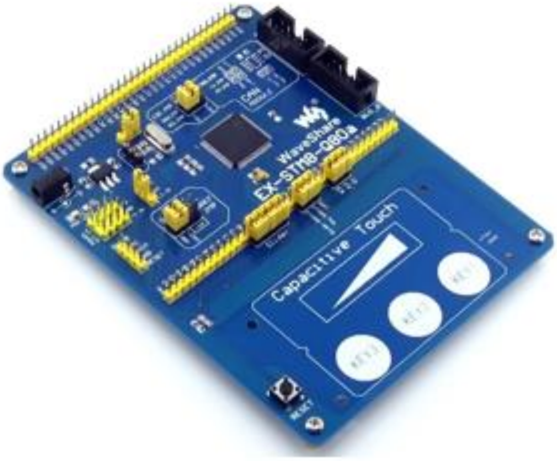




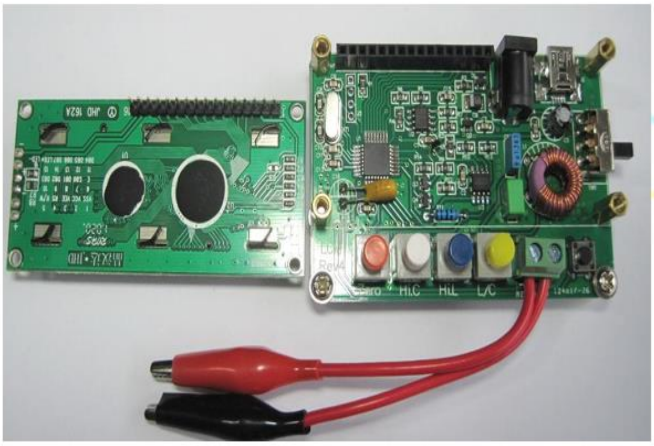
The board I used here in this article hosts a STM8S003K3T6 micro. It is an 32 pin entry-level micro with 8kB flash, 1kB RAM and 128 byte true data EEPROM. It comes with some additional hardware – a LED connected to PD0 and a push button connected to PB7. Just as I said it also houses a detachable ST-Link programmer. However, I don't recommend separating the programmer from the whole package. The board also has a prototyping area should one needs to prototype something. The overall board has a small form-factor and is a bit longer than a standard credit card. There are several other similar and popular STM8 Discovery boards like the STM8S105 Discovery.

There are also bulks of cheap Chinese minimum system STM8 dev boards hosting different STM8 chips. Overall the boards and the chips are so cheap that many simple cheap gadgets from China are based on STM8 MCUs.


D3 D2 D1 C7 C6 C5 C4 C3 B4 B5
3V3
SWIM
GND
NRST
RST
PWR TEST
D4 D5 D6 RST A1 A2 GND 5V 3V3 A3



Some cheap STM8-based simple products are shown below:



The first one is a cheap DIY LC meter LC-100A. The other one is a simple DC panel meter. These are just simple examples. There are many industrial and sophisticated products based on STM8 micros.

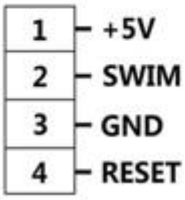
Hardware Tools

The list of hardware tools needed is not very long. We will obviously need a STM8 board and I prefer a Discovery board over other boards since it comes with a built-in ST-Link programmer/debugger hardware. If you have some other board like the ones I already showed, you will need a ST-Link programmer. I recommend an additional ST-Link programmer apart from the one available on board.

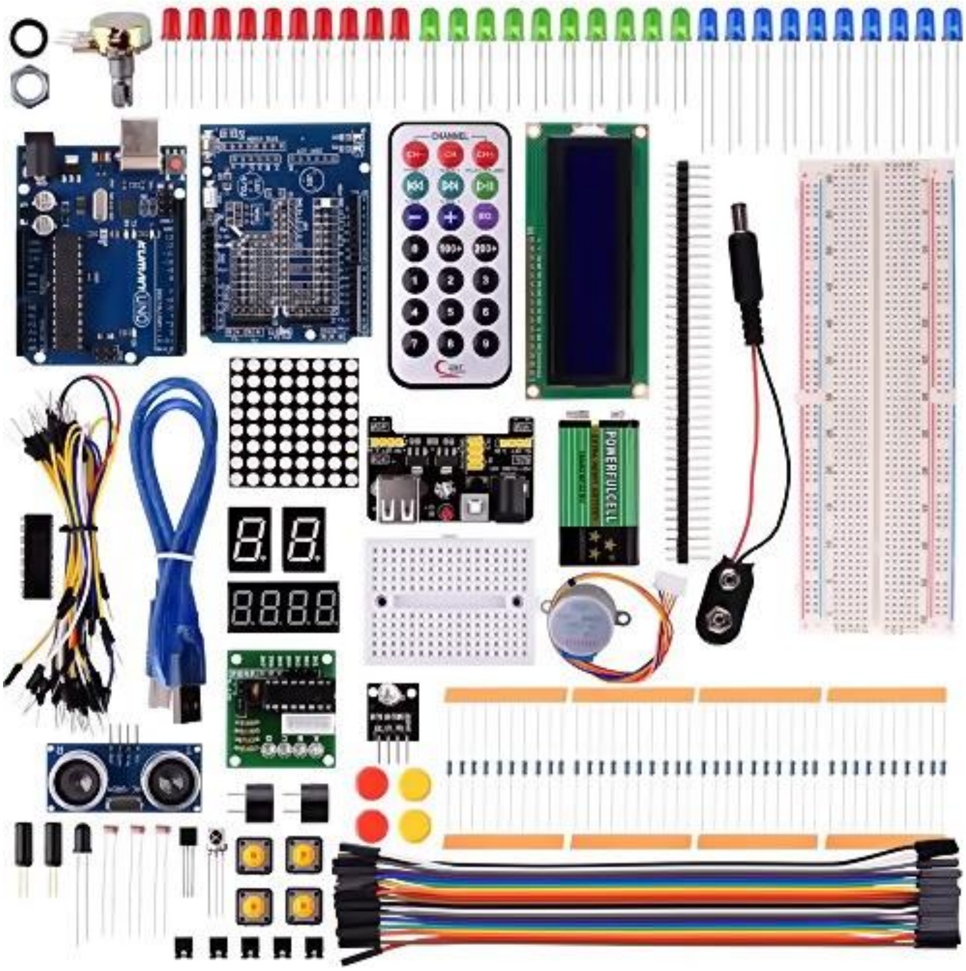
ST-LINK/V2

ST-LINK/V2-ISOL

ST-Link programmers/debuggers communicate with target STM8 micros via SWIM interface. This interface is the standard for all STM8 micros. Basically, it is a four-wire interface with two wire (VDD and GND) being used for powering the target. The rest two are reset I/O and SWIM I/O. In the official ST-Link V2 programmer unlike other ST-Link programmers, there is a dedicated port for SWIM interface with STM8 inscribed near it. Cheap USB flash drive-sized ST-Links are also available in the market and they are portable and as good as the official ones.



Apart from these we will also require some basic electronic lab stuffs like a USB-to-serial converter, connecting/jumper wires, LEDs, buttons, various types of sensors, etc. that are typically found in a common Arduino starter kit.



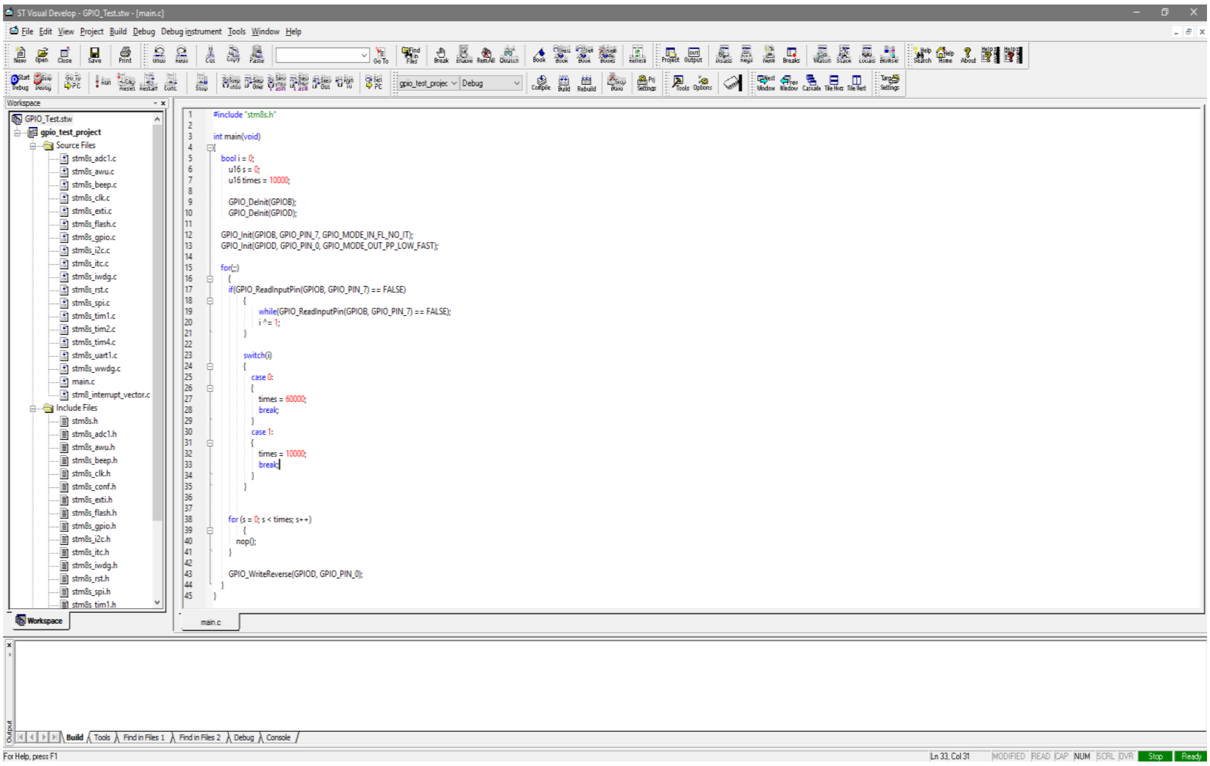
It is yet better if you have either a logic analyser or oscilloscope. A good multimeter and a well-regulated DC power supply/source are must haves. You can also use a cell phone charging power bank as a power source since Disco boards have USB ports.

Software Tools

Just like any other software developer, my choice of language for software development is C language. I don't want to spend time coding complex stuffs in assembly language. Apart from that I chose C language for the fact that STMicroelectronics has provided a ***Standard Peripheral Library (SPL)*** that is very easy to use. With SPL, it becomes totally unnecessary to program each register with meaningless numbers and maintain coding sequence. We will never need to access registers for any reasons as everything is done under the hood of SPL. All sequences are deal inside the SPL. All that we will ever need is the clear concept of each hardware block, their working principles, their capabilities and limitations.

We will need an ***Integrated Development Environment (IDE)*** and a C-language toolchain. The best stuffs you can get your hands on at zero costs are ***ST Visual Develop (STVD)*** IDE and ***Cosmic*** C compiler. Both are free but a rather difficult to use at first. STVD also comes with a programmer software tool called ***ST Visual Programmer (STVP)***. We'll need STVP to upload codes to target STM8 micros.

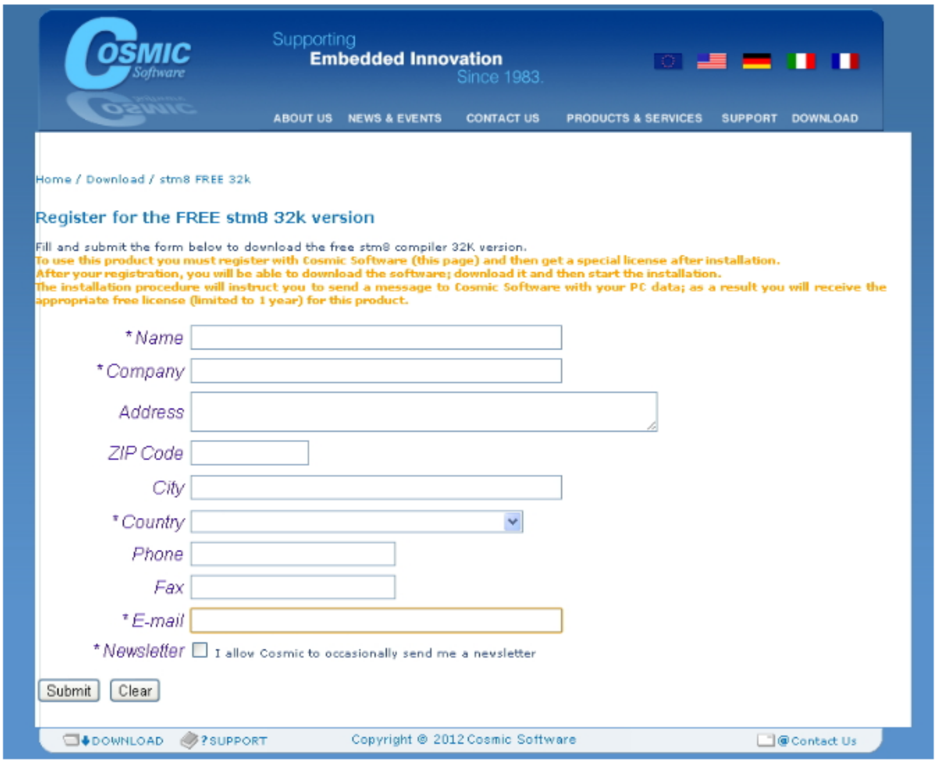
Cosmic used to be a paid tool just like your PC's antivirus software but at the time of writing this article, the Cosmic team has made it absolutely free for STM8 family. However, to use it you will need to register and acquire a license key via email. Usually this procedure of acquiring license and registration is maintained automatically by the software company's server but with Cosmic it is different story. You will need to wait for some guy at Cosmic end to respond to your license request. It may take a few minutes or even a day but still the best part is getting a full version compiler for nothing.



```c
#include "stm8s.h"

int main(void)
{
  bool i = 0;
    u16 s = 0;
    u16 times = 10000;

    GPIO_DeInit(GPIOB);
    GPIO_DeInit(GPIOD);

  GPIO_Init(GPIOB, GPIO_PIN_7, GPIO_MODE_IN_FL_NO_IT);
  GPIO_Init(GPIOD, GPIO_PIN_0, GPIO_MODE_OUT_PP_LOW_FAST);

  for(;;)
    {
    if(GPIO_ReadInputPin(GPIOB, GPIO_PIN_7) == FALSE)
        {
            while(GPIO_ReadInputPin(GPIOB, GPIO_PIN_7) == FALSE);
            i ^= 1;
        }

        switch(i)
        {
          case 0:
          {
            times = 60000;
            break;
          }
          case 1:
          {
            times = 10000;
            break;
          }
        }


    for (s = 0; s < times; s++)
        {
      nop();
    }

    GPIO_WriteReverse(GPIOD, GPIO_PIN_0);
  }
}
```

You can get
STVD from here: http://www.st.com/en/development-tools/stvd-stm8.html and
Cosmic C compiler from here: http://www.cosmic-software.com/download.php.

You need to register in order to download both software. For Cosmic you will also need to acquire a free license for it work. So just fill in some basic info about you.

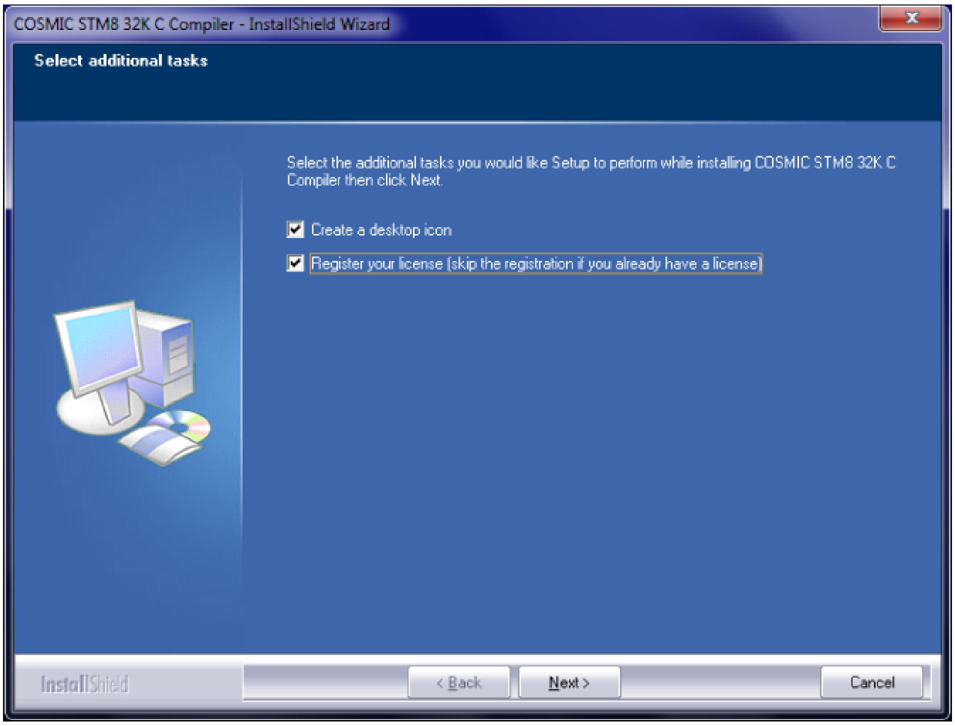


Register for the FREE stm8 32k version

Fill and submit the form below to dovnload the free stm8 compiler 32K version.
To use this product you must register with Cosmic Software (this page) and then get a special license after installation.
After your registration, you will be able to download the software; download it and then start the installation.
The installation procedure will instruct you to send a message to Cosmic Software with your PC data; as a result you will receive the appropriate free license (limited to 1 year) for this product.

*Name
*Company
Address
ZIP Code
City
*Country
Phone
Fax
*E-mail
*Newsletter □ I allow Cosmic to occasionally send me a nevsletter

Firstly, we will need to install STVD. Installation procedure is simple and same as typical software installation. Just click next, next and next. After that we will need to install Cosmic C compiler. Again, just next, next and next until the screen as shown below.

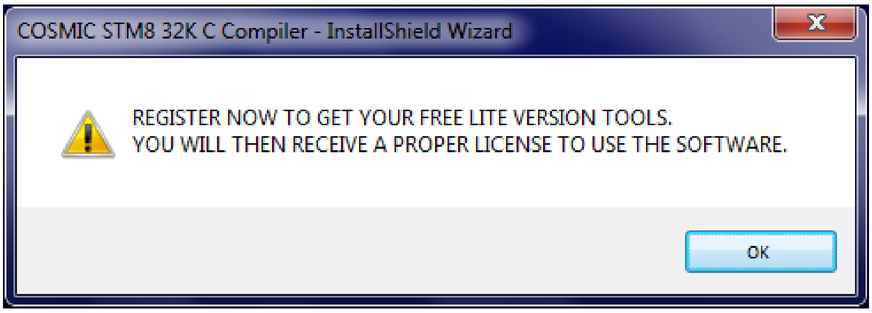


Select the additional tasks you would like Setup to perform while installing COSMIC STM8 32K C Compiler then click Next.

☑ Create a desktop icon

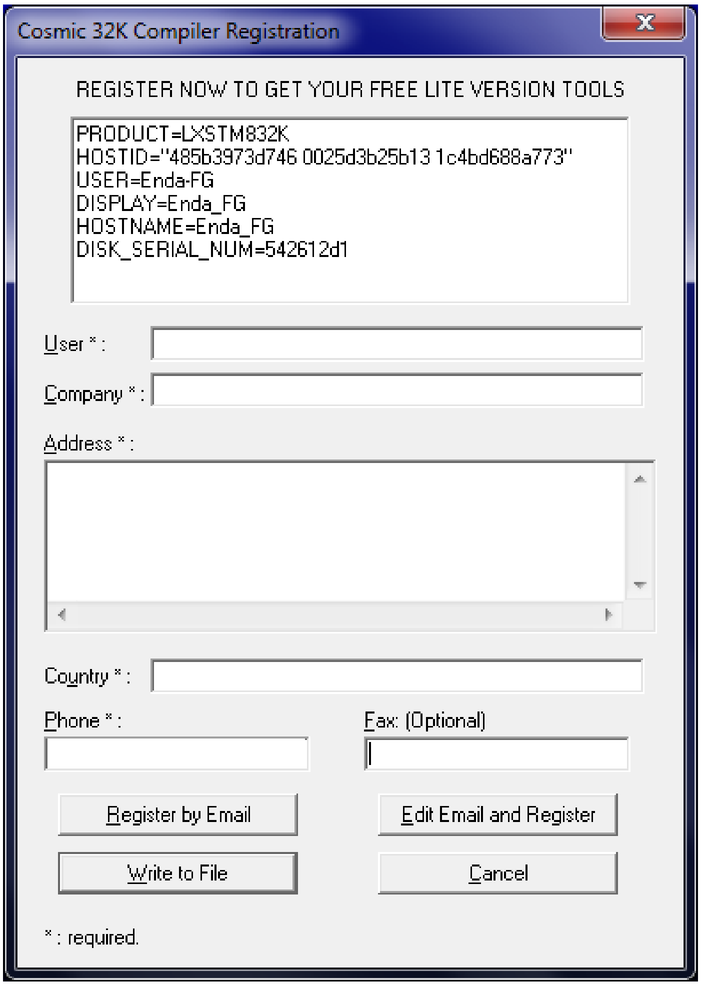
☑ Register your license (skip the registration if you already have a license)

After installation, you'll prompted for licence. You must register your license unless you have already registered. If you have already registered, then you'll be asked if to overwrite registration. You should skip reregistering.


COSMIC STM8 32K C Compiler - InstallShield Wizard

REGISTER NOW TO GET YOUR FREE LITE VERSION TOOLS.
YOU WILL THEN RECEIVE A PROPER LICENSE TO USE THE SOFTWARE.

For the first run, you'll get the following screen looking for a valid license.


Cosmic 32K Compiler Registration

REGISTER NOW TO GET YOUR FREE LITE VERSION TOOLS

```
PRODUCT=LXSTM832K
HOSTID="485b3973d746 0025d3b25b13 1c4bd688a773"
USER=Enda-FG
DISPLAY=Enda_FG
HOSTNAME=Enda_FG
DISK_SERIAL_NUM=542612d1
```

User * :

Company * :

Address * :

Country * :

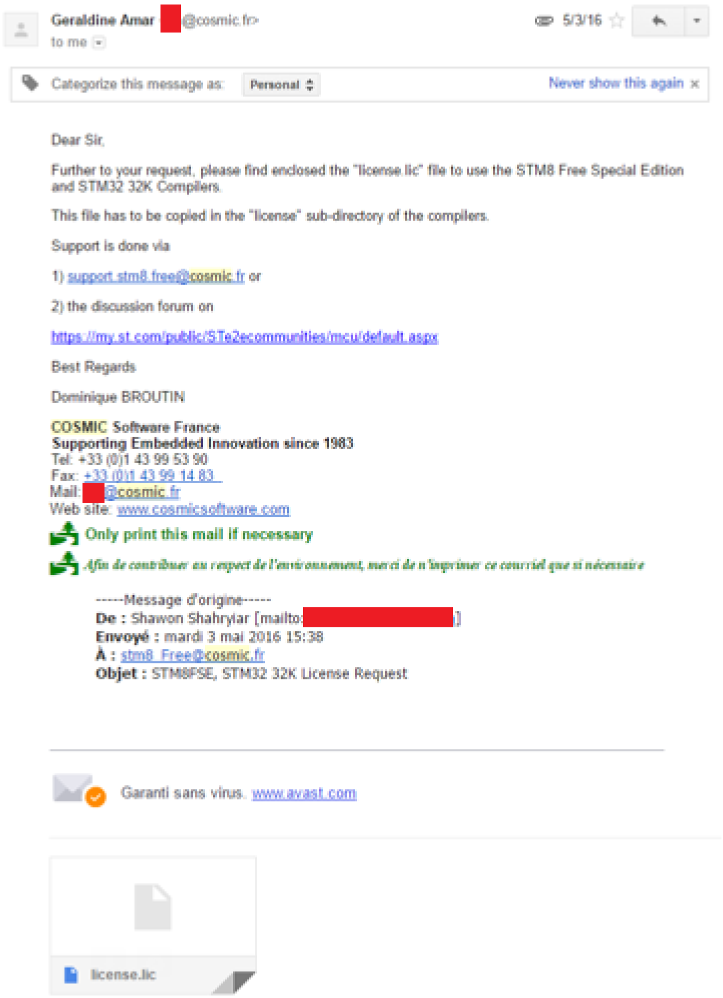
Phone * :

Fax: (Optional)

* : required.

You must fill all the starred (*) points to complete the process of registration. Select ***"Write to File"*** option and save the file as a text (.txt) file. The file name should be ***"CM8_license.txt"***. Send this file to stm8_Free@cosmic.fr with subject "***STM8FSE, STM32 32K License Request***". Now you'll need to wait for the Cosmic team to respond to you. They'll send you an email back with an electronic key license. The file will have a name like ***"license.lic"*** and the email will have some instructions.

This was my emailed license.



Geraldine Amar [redacted]@cosmic.fr> 5/3/16

to me

Categorize this message as: Personal Never show this again ×

Dear Sir,

Further to your request, please find enclosed the "license.lic" file to use the STM8 Free Special Edition and STM32 32K Compilers.

This file has to be copied in the "license" sub-directory of the compilers.

Support is done via

1) support.stm8.free@cosmic.fr or

2) the discussion forum on

https://my.st.com/public/STe2ecommunities/mcu/default.aspx

Best Regards

Dominique BROUTIN

COSMIC Software France
Supporting Embedded Innovation since 1983
Tel: +33 (0)1 43 99 53 90
Fax: +33 (0)1 43 99 14 83
Mail: [redacted]@cosmic.fr
Web site: www.cosmicsoftware.com

Only print this mail if necessary

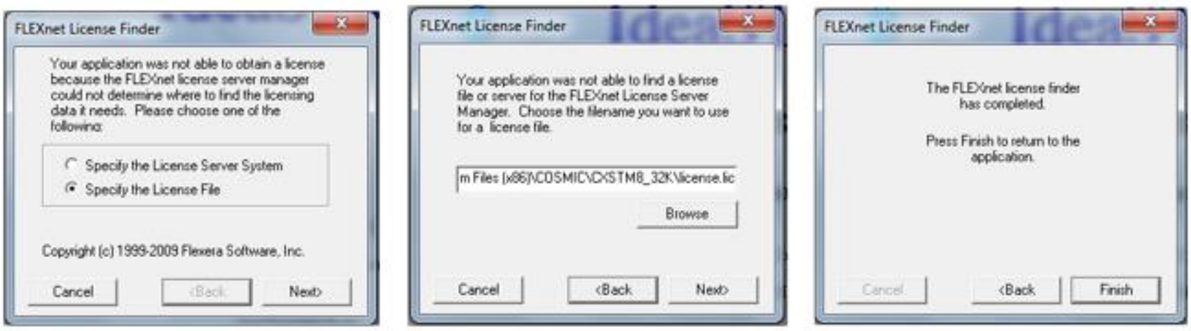
Afin de contribuer au respect de l'environnement, merci de n'imprimer ce courriel que si nécessaire

-----Message d'origine-----
De : Shawon Shahryiar [mailto:[redacted]]
Envoyé : mardi 3 mai 2016 15:38
À : stm8_Free@cosmic.fr
Objet : STM8FSE, STM32 32K License Request

Garanti sans virus. www.avast.com

license.lic

Once you get the license, you'll need to show the software its location and complete the licensing process as shown below. Save the license file in a secured location.



FLEXnet License Finder

Your application was not able to obtain a license because the FLEXnet license server manager could not determine where to find the licensing data it needs. Please choose one of the following:

○ Specify the License Server System
● Specify the License File

Copyright (c) 1999-2009 Flexera Software, Inc.

FLEXnet License Finder

Your application was not able to find a license file or server for the FLEXnet License Server Manager. Choose the filename you want to use for a license file.

m Files (x86)\COSMIC\CXSTM8_32K\license.lic

FLEXnet License Finder

The FLEXnet license finder has completed.

Press Finish to return to the application.

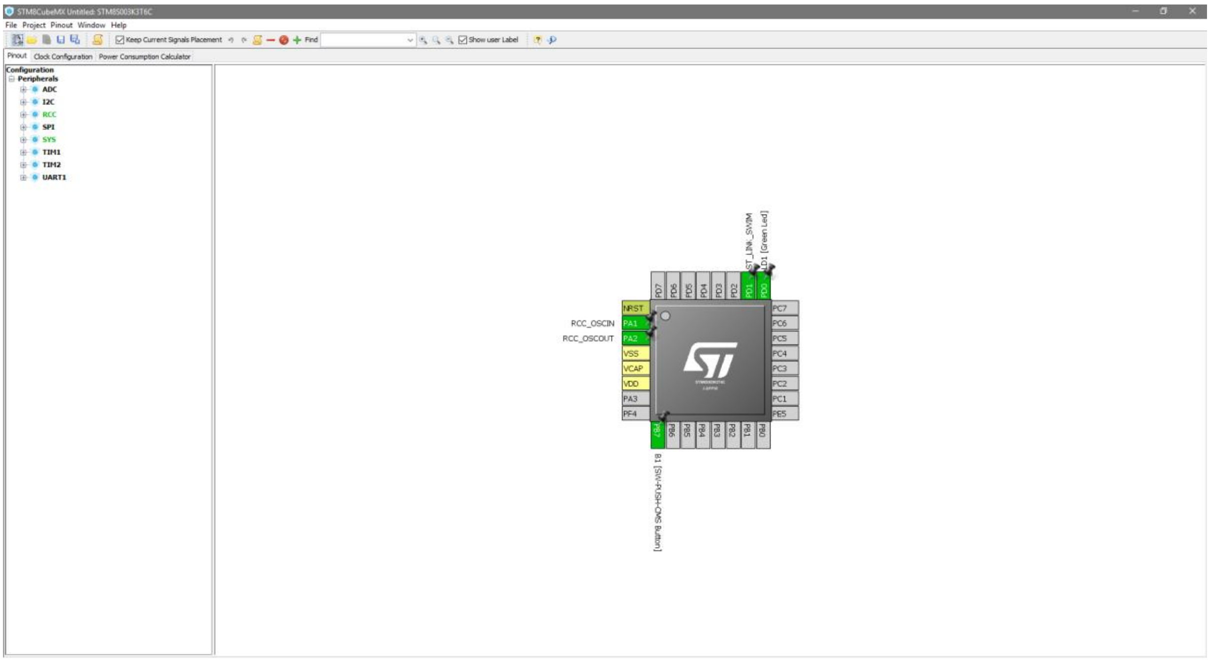
At the end of this process, we can enjoy the compiler without any limitations.

I also recommend that you download Sublime Text (https://www.sublimetext.com/) or Notepad++ (https://notepad-plus-plus.org/) for viewing your code with ease. These are very cool software. This is not mandatory though.

STM8CubeMX

Should I or should I not feel fortunate was my question at the time of writing this article and that’s because STM8CubeMX was released in late February 2017. Yes, that’s the time when I was compiling all these STM8 stuffs together. Prior to that I was wondering about a software similar to STM32CubeMX but for STM8s. Back then, I could not find one and raw documentations were only helpers. Although it is still in its early stages of development and still not as robust as its STM32 cousin in terms of code generation capabilities and other areas, we can expect great innovations in the near future. It reminds me of the early days of STM32CubeMX. Not everyone expected it to overcome all the challenges in a very short period of time. At present, we can use STM8CubeMX for common info on STM8 chips like pin assignments/mapping, basic technical specs like memory capacities, possible clock configurations, etc. I can just wonder the potential future integrations and bug fixes. Power consumption calculator is one such tool hopefully to be integrated. STM, most likely, has some serious big plans for it. Nevertheless, we must thank STM for this cool software.

Visit http://www.st.com/en/development-tools/stm8cubemx.html to download STM8CubeMX.



I recommend using it only as reference. Don’t make yourself dependent on it. There are still many bugs fixes that are yet to be addressed. One example is as in the above screenshot. Notice Timer 3 (TIM3) is missing although STM8S003 has this timer. I’m pretty sure the software developers are working hard on such silly issues. For now, it is more like a promise of good tidings to come. This is why I haven’t included it in the “must-have” software list and wanted it to be discussed separately.

Preparing the Software Tools

Firstly, we need three major documents before starting to program STM8 micros. These are as follows:

1. STM8 Reference Manual.
   http://www.st.com/content/ccc/resource/technical/document/reference_manual/9a/1b/85/07/ca/eb/4f/dd/CD00190271.pdf/files/CD00190271.pdf/jcr:content/translations/en.CD00190271.pdf

2. Datasheet of the MCU (***STM8S003***) that we'll be using.
   http://www.st.com/content/ccc/resource/technical/document/datasheet/42/5a/27/87/ac/5a/44/88/DM00024550.pdf/files/DM00024550.pdf/jcr:content/translations/en.DM00024550.pdf

3. STM8SVLDiscovery Board User Manual.
   http://www.st.com/content/ccc/resource/technical/document/user_manual/c8/37/11/ba/b5/e7/4c/ee/DM00040810.pdf/files/DM00040810.pdf/jcr:content/translations/en.DM00040810.pdf

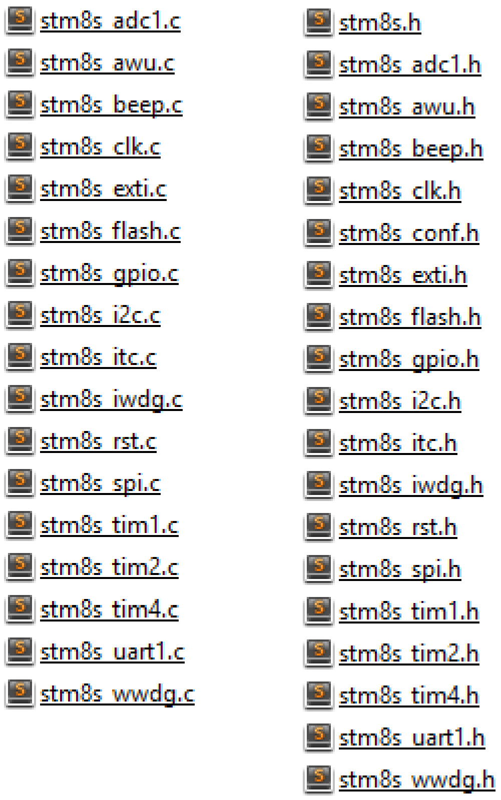
These docs will be needed everywhere during learning session. The reference manual states the use and purpose of all the hardware blocks in details. It includes register descriptions, naming conventions, modes of operation of all hardware, etc. However, it does not specify the specifications of a given STM8 micro and that's because it is a generalized literature for all STM8S and STM8AF micros. As we know even within a family of micros, one MCU differs from another in many aspects. Most commonly this variation comes in the form of memory capacities and I/O pin counts. Sometimes electrical specs also vary and so to know the limits and general specs of our target MCUs we need to checkout their respective datasheets. Lastly the Discovery board user manual is most useful for the hardware schematics, pin assignments and layouts. If you are using some other board then I suggest that you acquire at least its schematic.

Now with Cosmic, STVD and STVP installed our software tool setup is almost ready. There are two approaches to STM8 programming. The first uses the traditional concepts of register-access-based coding, meaning you'll have to set up every register on your own. The second way utilizes a specialized method of coding by using standard libraries developed by STM that are both universal and platform independent, meaning your C code will be same for any compiler using these libraries. These libraries are called ***Standard Peripheral Libraries (SPL)***. With these libraries, no one will ever need to get down to register-level access. The libraries are so coded that a coder will only have to know his/her chips' hardware specs and some basics of these hardware. On the coding part, he/she will only have to set properties and desired values. The SPL manages the rest. For instance, when setting up a UART, we will only need to set interrupts, IOs and UART properties like baud rate, parity, etc. All of these setups are done with meaningful numbers and texts.

The STMicroelectronics Standard Peripheral Libraries (SPL) for STM8 microcontrollers can be found here: http://www.st.com/en/embedded-software/stsw-stm8069.html.

I wrote this article using SPL since it will be ridiculous to code STM8s using the old-fashioned way of configuring registers one-by-one manually. Thus, it is a mandatory download item. You should preserve and retain the downloaded SPL zip file fully intact. You may need it when things get messy.

Now make two folders and name them ***"inc"*** and ***"src"***. The ***"inc"*** folder will be filled with all the header files (***".h"*** extension files) from the extracted zip file. Similarly, the ***"src"*** folder will be holding the source files (***".c"*** extension files). For ease of work, it is better to keep these folders secured just like the SPL zip file because every time when we will be making new projects the files in these folders will be needed. You can copy these files to your project folder or you can keep it centrally somewhere. I prefer the former method as doing so will not have any conflicting issue with other projects needing modifications. However, it will cost hard-drive space. This method is however less confusing and trouble-free for beginners. Extract all the files as shown below:



stm8s_adc1.c
stm8s_awu.c
stm8s_beep.c
stm8s_clk.c
stm8s_exti.c
stm8s_flash.c
stm8s_gpio.c
stm8s_i2c.c
stm8s_itc.c
stm8s_iwdg.c
stm8s_rst.c
stm8s_spi.c
stm8s_tim1.c
stm8s_tim2.c
stm8s_tim4.c
stm8s_uart1.c
stm8s_wwdg.c

stm8s.h
stm8s_adc1.h
stm8s_awu.h
stm8s_beep.h
stm8s_clk.h
stm8s_conf.h
stm8s_exti.h
stm8s_flash.h
stm8s_gpio.h
stm8s_i2c.h
stm8s_itc.h
stm8s_iwdg.h
stm8s_rst.h
stm8s_spi.h
stm8s_tim1.h
stm8s_tim2.h
stm8s_tim4.h
stm8s_uart1.h
stm8s_wwdg.h

Note that there are more header files than source files. This is because there are two extra header files – ***stm8s.h*** and ***stm8s_conf.h*** that define processor type and processor properties. To make things work, we will have to comment one line of the ***stm8s_conf.h***. You will find a line at the bottom of this file written as:

#define USE_FULL_ASSERT (1).

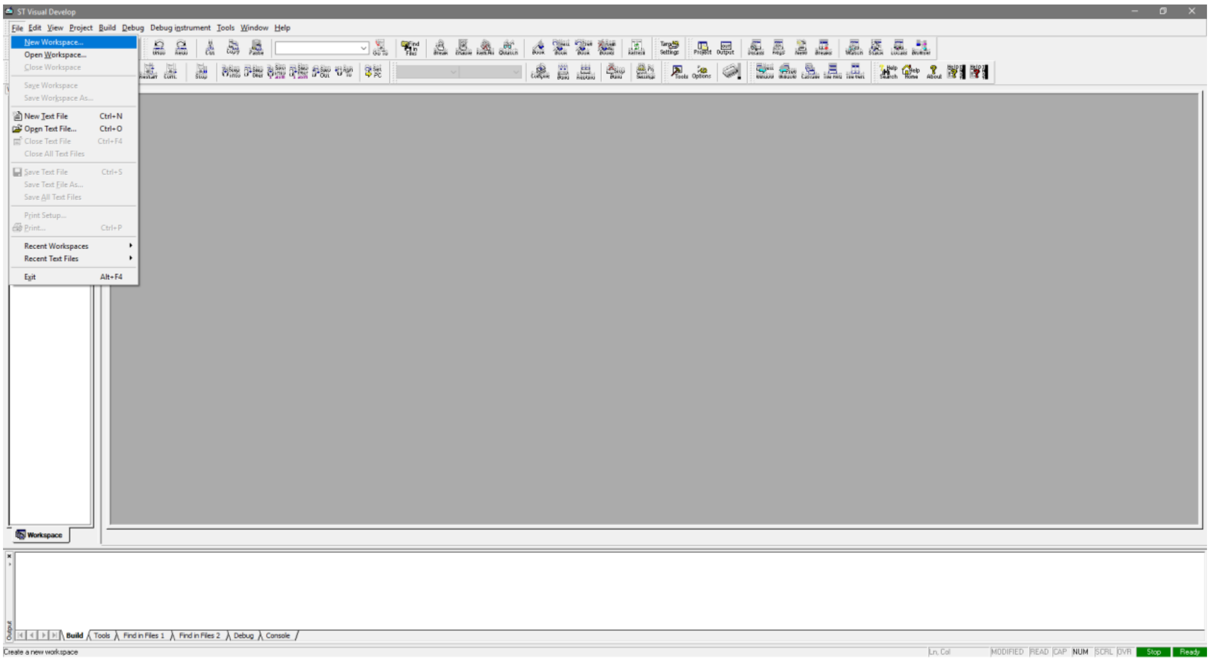
You need to comment or disable this line, otherwise the compiler will throw tons of error messages. Always check this at the start of a project. Surely, we don't want to assert our code.

Creating a New Code Project

Assuming that STVD, STVP and Cosmic are properly installed, we will see how to create a new project.

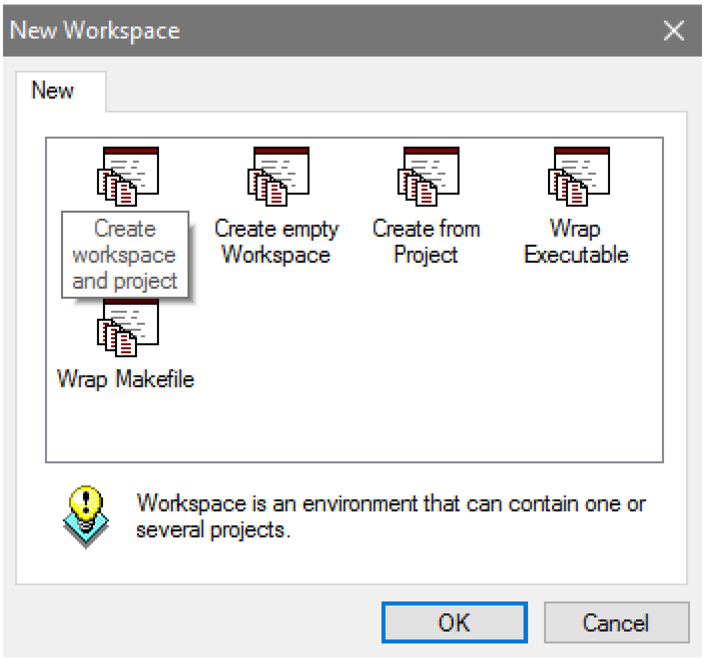
1. Firstly, run STVD.

2. Select ***File >> New Workspace***.



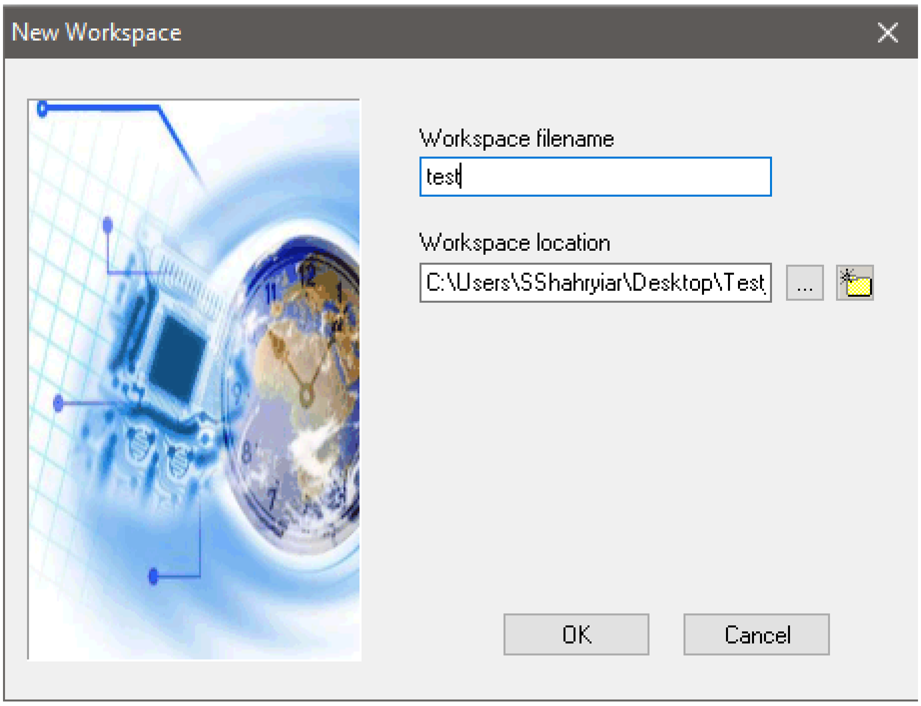
Create a new workspace

3. Select ***Create workspace and project***.



Workspace is an environment that can contain one or several projects.

4. Select workspace folder and workspace name.

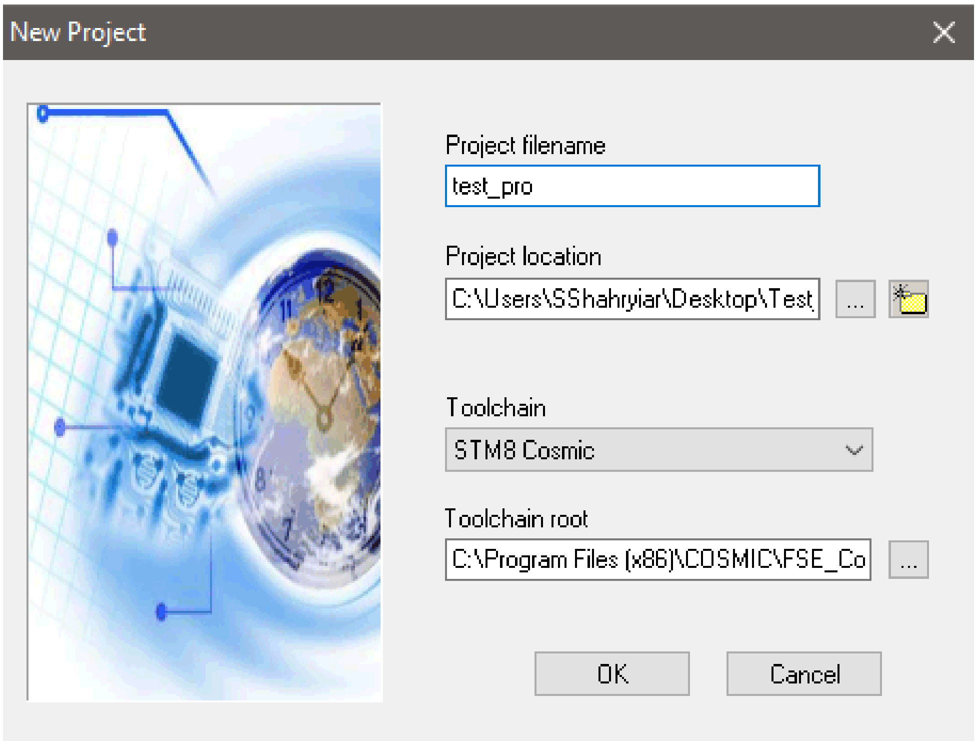


Workspace filename: test

Workspace location: C:\Users\SShahryiar\Desktop\Test

5. Set project name and select toolchain ***STM8 Cosmic***. You may need to set the path of your Cosmic compiler's installation location. In my case, it is:

C:\Program Files (x86)\COSMIC\FSE_Compilers\CXSTM8



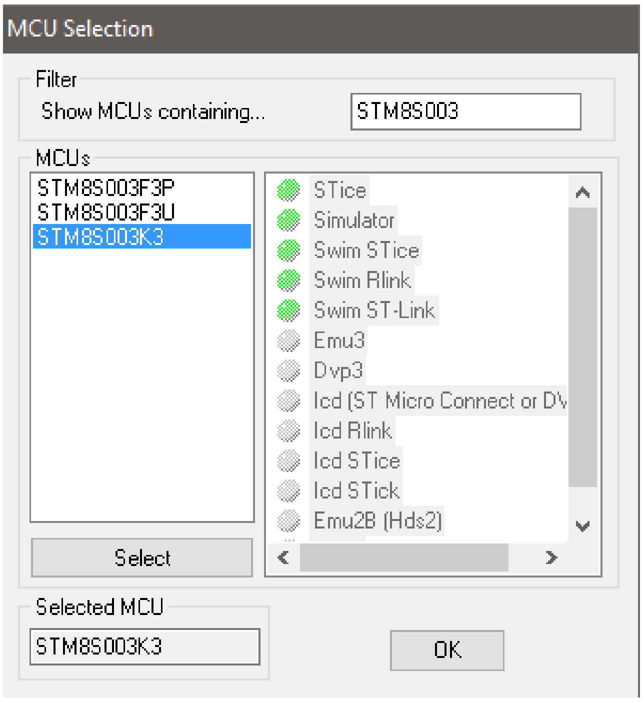
Project filename: test_pro

Project location: C:\Users\SShahryiar\Desktop\Test

Toolchain: STM8 Cosmic

Toolchain root: C:\Program Files (x86)\COSMIC\FSE_Co

6. Type and select target chip part number. Last two or three digits and letters are enough for finding the correct micro.



MCU Selection

Filter

Show MCUs containing... STM8S003

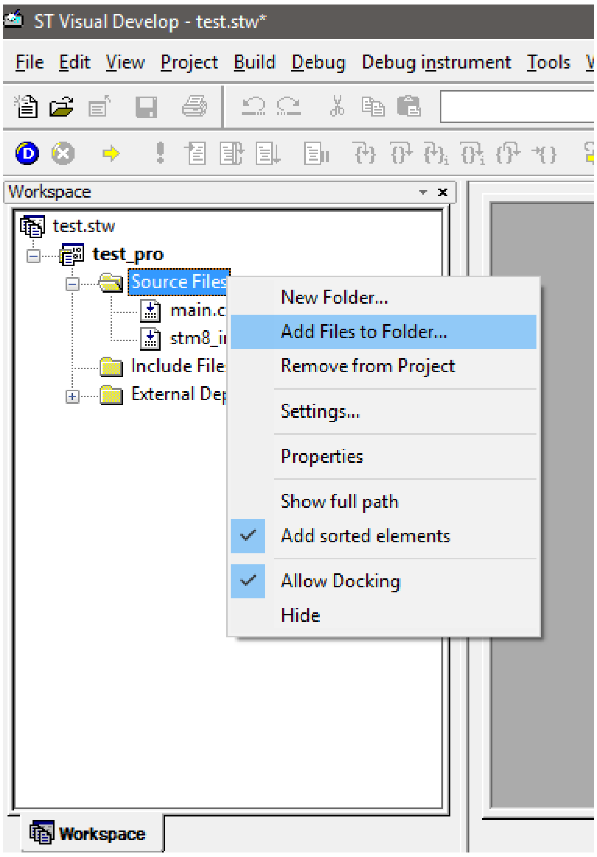
MCUs

STM8S003F3P
STM8S003F3U
STM8S003K3

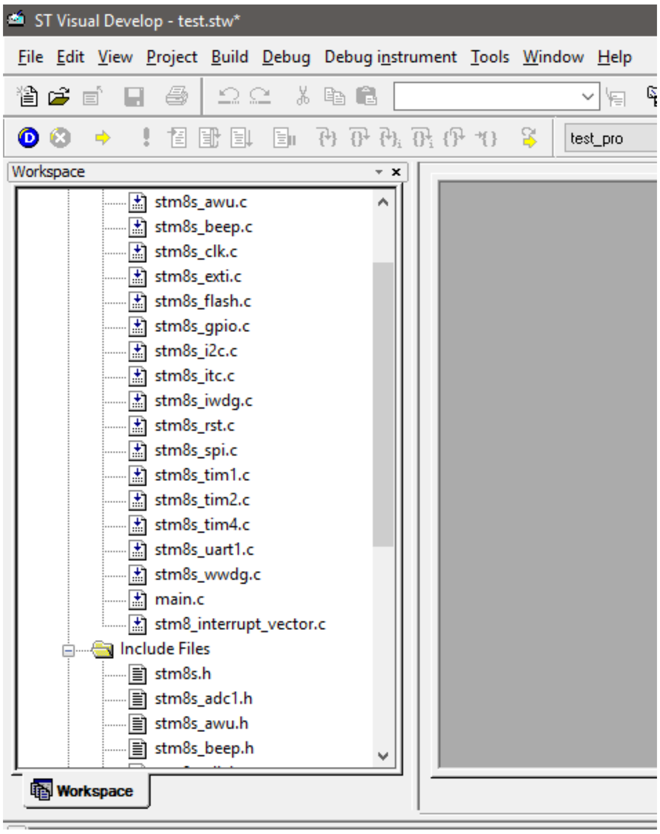
Selected MCU

STM8S003K3

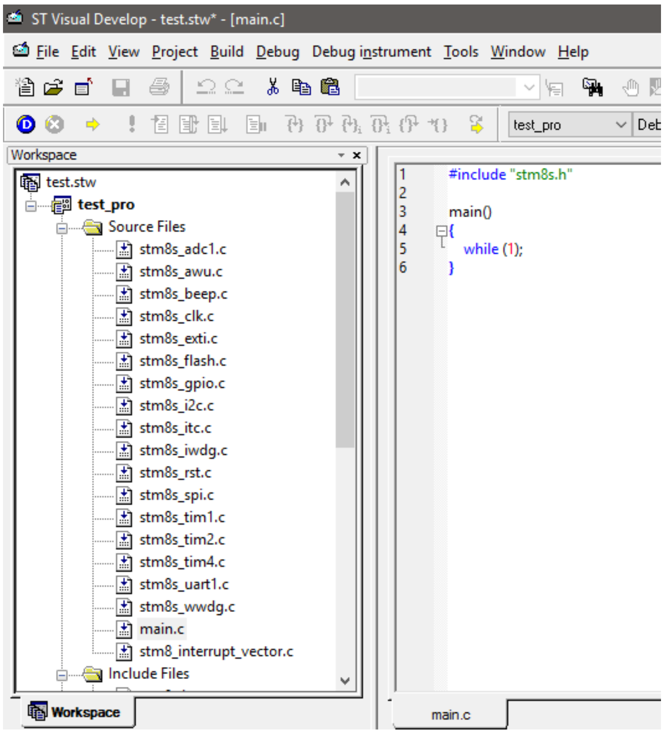
7. Now add the source and header files from the previously mentioned SPL folder.



8. After file inclusions, the workspace tab changes as shown below.

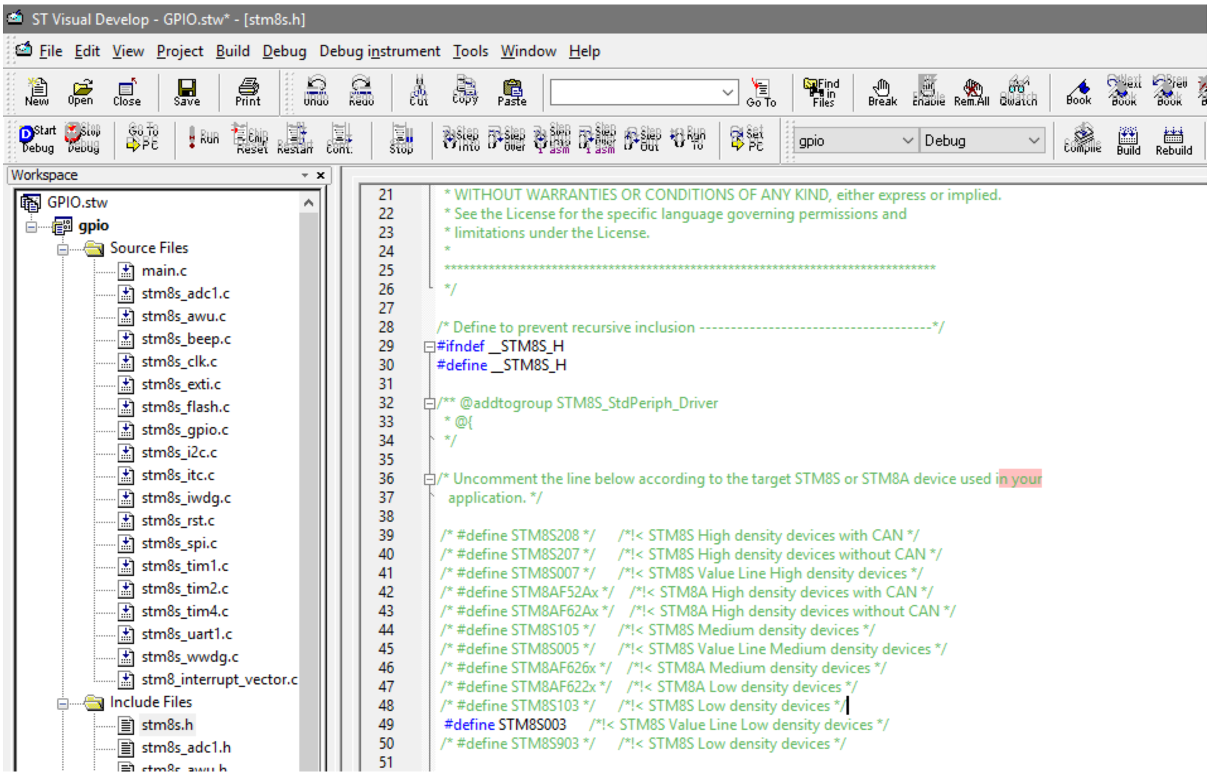


9. Locate and open ***main.c*** file from the source tab, and then type ***#include "stm8s.h"*** at the top as shown below:



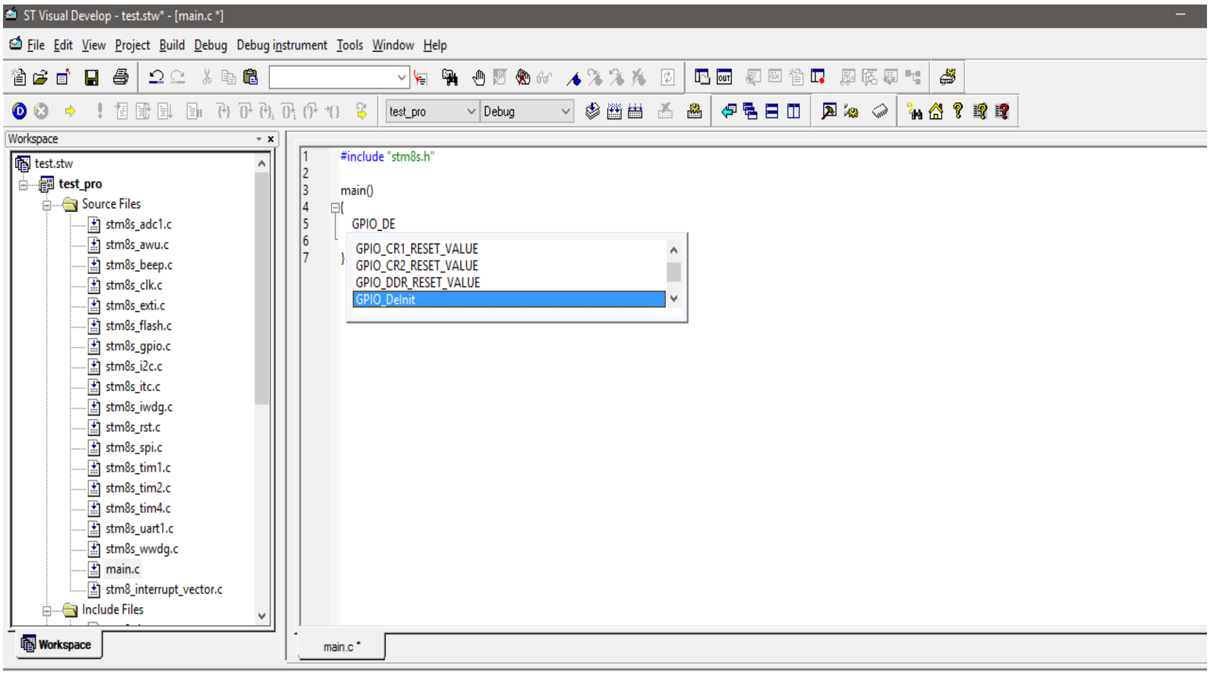
```
#include "stm8s.h"

main()
{
    while (1);
}
```

10. You’ll have to edit the ***STM8S.h*** header file and uncomment the chip number you are going to use as shown below:



```
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 ******************************************************************************
 */

/* Define to prevent recursive inclusion -----------------------------------*/
#ifndef __STM8S_H
#define __STM8S_H

/** @addtogroup STM8S_StdPeriph_Driver
  * @{
  */

/* Uncomment the line below according to the target STM8S or STM8A device used in your
  application. */

 /* #define STM8S208 */      /*!< STM8S High density devices with CAN */
 /* #define STM8S207 */      /*!< STM8S High density devices without CAN */
 /* #define STM8S007 */      /*!< STM8S Value Line High density devices */
 /* #define STM8AF52Ax */    /*!< STM8A High density devices with CAN */
 /* #define STM8AF62Ax */    /*!< STM8A High density devices without CAN */
 /* #define STM8S105 */      /*!< STM8S Medium density devices */
 /* #define STM8S005 */      /*!< STM8S Value Line Medium density devices */
 /* #define STM8AF626x */    /*!< STM8A Medium density devices */
 /* #define STM8AF622x */    /*!< STM8A Low density devices */
 /* #define STM8S103 */      /*!< STM8S Low density devices */
  #define STM8S003     /*!< STM8S Value Line Low density devices */
 /* #define STM8S903 */      /*!< STM8S Low density devices */
```

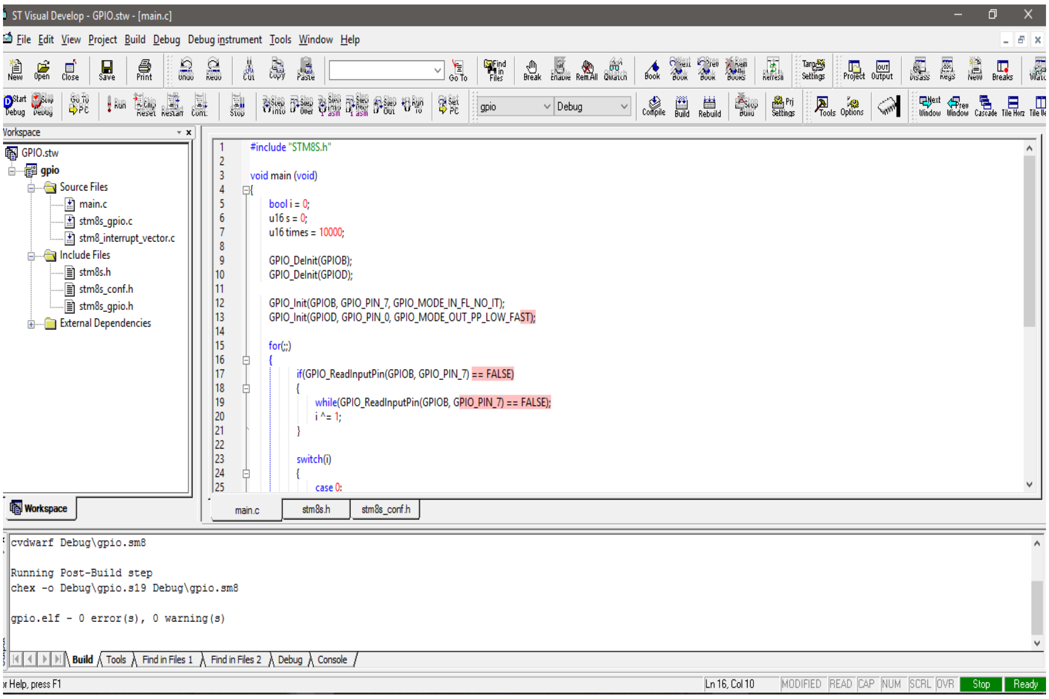
11. Compile the code once using the key combination ***CTRL+F7*** or by pressing the compile button. If everything is okay, there should be no error or warning message. The reason for this blank compilation is to use the compiler’s powerful code assistant feature. With this feature, we can predict or complete a piece of code line by only writing the first few letters and then pressing ***CTRL + SPACE*** keys simultaneously.



```
#include "stm8s.h"

main()
{
  GPIO_DE
}
```

During compilation, you may get tons of errors for hardware files that are not available in your target STM8S micro. For instance, CAN hardware is not available in STM8S003K3 and so if you have added CAN source and header files you will get an error for that. Once identified by the error messages, the corresponding header and source files for that particular hardware must be removed.

Similarly, one more caution must be observed. Unless your code is using any interrupt, interrupt source and header files (***stm8s_it.h*** and ***stm8s_it.c***) must be excluded. Sometimes it is better to add only those files that you will need to complete a project. For example, if your project is just using GPIOs, it is better to add GPIO files only along with ***stm8s.h*** and ***stm8s_conf.h***. However, I recommend this technique only after you have mastered STM8 coding well because in most cases you will need multiple hardware which have dependencies on each other. As an example, when using SPI, you'll need both GPIO and SPI modules. If you understand these dependencies, it is okay to select files as per need. You can, then, comment out unnecessary hardware module files specified in the ***stm8s.h*** header file and get a faster compilation and build process. After compilation, you should always build/rebuild your project by hitting the ***Build*** or ***Rebuild*** button. This will generate the final ***s19*** output file in either ***Debug*** or ***Release*** folder according to the generation mode selected. If things are in order, there should be no error or warning message.



```
#include "STM8S.h"

void main (void)
{
    bool i = 0;
    u16 s = 0;
    u16 times = 10000;

    GPIO_DeInit(GPIOB);
    GPIO_DeInit(GPIOD);

    GPIO_Init(GPIOB, GPIO_PIN_7, GPIO_MODE_IN_FL_NO_IT);
    GPIO_Init(GPIOD, GPIO_PIN_0, GPIO_MODE_OUT_PP_LOW_FAST);

    for(;;)
    {
        if(GPIO_ReadInputPin(GPIOB, GPIO_PIN_7) == FALSE)
        {
            while(GPIO_ReadInputPin(GPIOB, GPIO_PIN_7) == FALSE);
            i ^= 1;
        }

        switch(i)
        {
            case 0:
```

```
cvdwarf Debug\gpio.sm8

Running Post-Build step
chex -o Debug\gpio.s19 Debug\gpio.sm8

gpio.elf - 0 error(s), 0 warning(s)
```

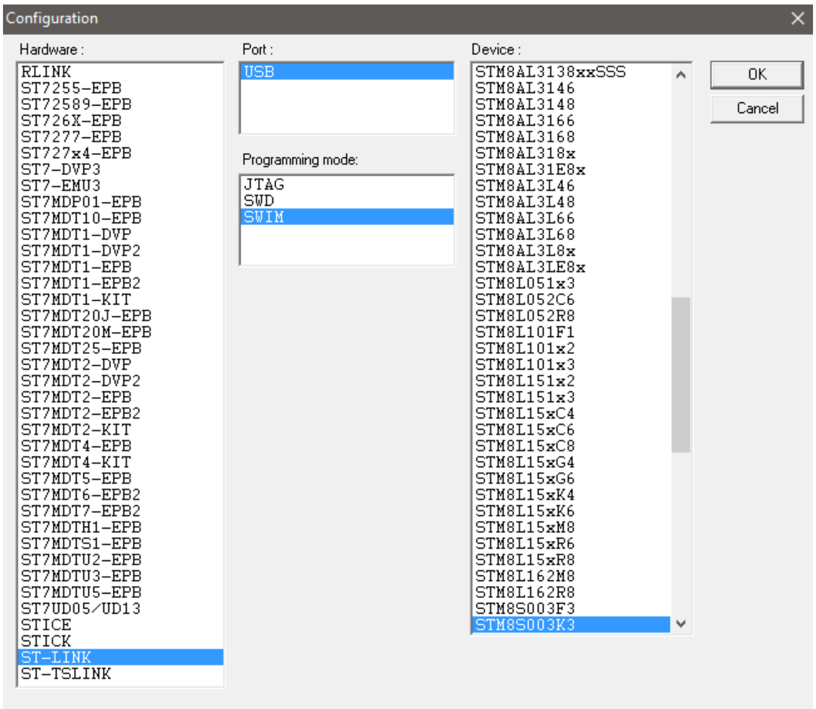
Lastly, I have not found any useful simulation software like Proteus VSM or Electronic Workbench that support STM8 family. Thus, we have to debug our code in real-life with real hardware. It may sound difficult but actually it is not so. We can, however, use such software to make models of STM8 micros and make our PCBs. I don't like simulations as they are not always accurate and real-world type.

One more advice I would like to give to the readers. Please read the SPL help file. It is located in the SPL zip file under the name ***stm8s-a_stdperiph_lib_um.chm***. It explains each function, definition, data structure, all internal hardware modules and how to use them properly. This is a very important document and your best friend in coding STM8 micros. Apart from this document the reference manual is equally important as it details the capabilities of all internal hardware. I won't be detailing the internal hardware much as these docs will be doing so.

Uploading Code

Codes generated by Cosmic C compiler have ***s19*** file extensions. It is similar to typical hex file format, containing user code as hex values. Well since we don't need to modify finally generated output files, it doesn't really matter in which format it is. All we will need is to upload them to our target MCUs. We can do it in two ways – either by using STVP or STVD.

Firstly, let's check the method with STVP. Run STVP software. For the first time the following window will appear. From here we have to select ST-Link programmer, SWIM interface and our target chip.



Hardware :

```
RLINK
ST7255-EPB
ST72589-EPB
ST726X-EPB
ST7277-EPB
ST727x4-EPB
ST7-DVP3
ST7-EMU3
ST7MDP01-EPB
ST7MDT10-EPB
ST7MDT1-DVP
ST7MDT1-DVP2
ST7MDT1-EPB
ST7MDT1-EPB2
ST7MDT1-KIT
ST7MDT20J-EPB
ST7MDT20M-EPB
ST7MDT25-EPB
ST7MDT2-DVP
ST7MDT2-DVP2
ST7MDT2-EPB
ST7MDT2-EPB2
ST7MDT2-KIT
ST7MDT4-EPB
ST7MDT4-KIT
ST7MDT5-EPB
ST7MDT6-EPB2
ST7MDT7-EPB2
ST7MDTH1-EPB
ST7MDTS1-EPB
ST7MDTU2-EPB
ST7MDTU3-EPB
ST7MDTU5-EPB
ST7UD05/UD13
STICE
STICK
ST-LINK
ST-TSLINK
```

Port :

```
USB
```

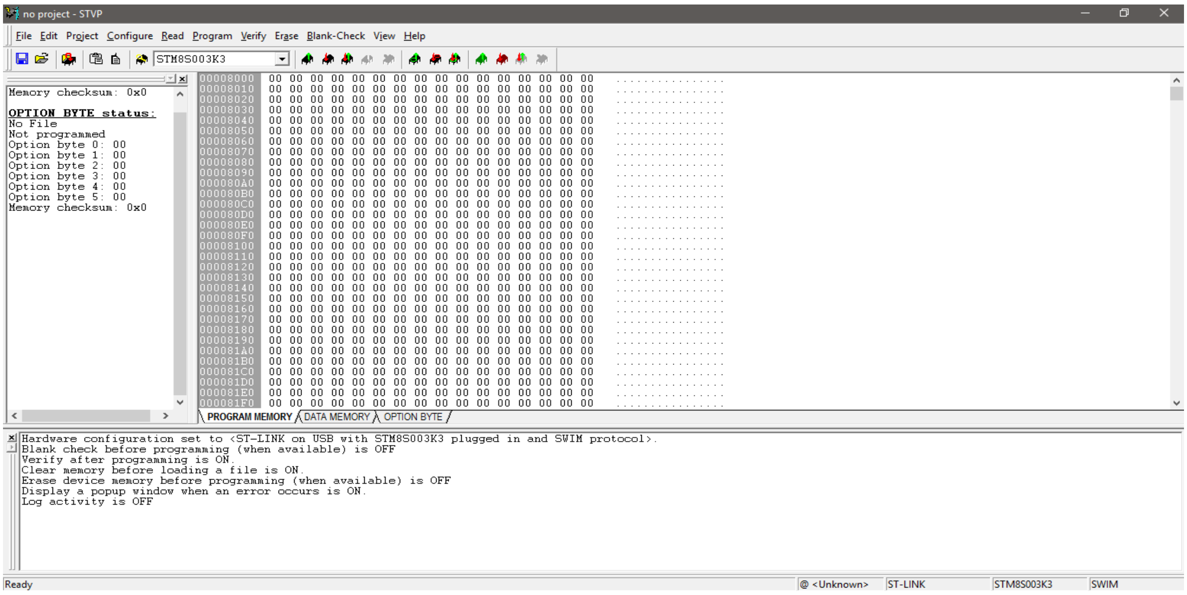
Programming mode:

```
JTAG
SWD
SWIM
```

Device :

```
STM8AL3138xxSSS
STM8AL3146
STM8AL3148
STM8AL3166
STM8AL3168
STM8AL318x
STM8AL31E8x
STM8AL3L46
STM8AL3L48
STM8AL3L66
STM8AL3L68
STM8AL3L8x
STM8AL3LE8x
STM8L051x3
STM8L052C6
STM8L052R8
STM8L101F1
STM8L101x2
STM8L101x3
STM8L151x2
STM8L151x3
STM8L15xC4
STM8L15xC6
STM8L15xC8
STM8L15xG4
STM8L15xG6
STM8L15xK4
STM8L15xK6
STM8L15xM8
STM8L15xR6
STM8L15xR8
STM8L162M8
STM8L162R8
STM8S003F3
STM8S003K3
```

STVP interface looks like any other programmer interface as shown below:


no project - STVP
File Edit Project Configure Read Program Verify Erase Blank-Check View Help
STM8S003K3
PROGRAM MEMORY
DATA MEMORY
OPTION BYTE

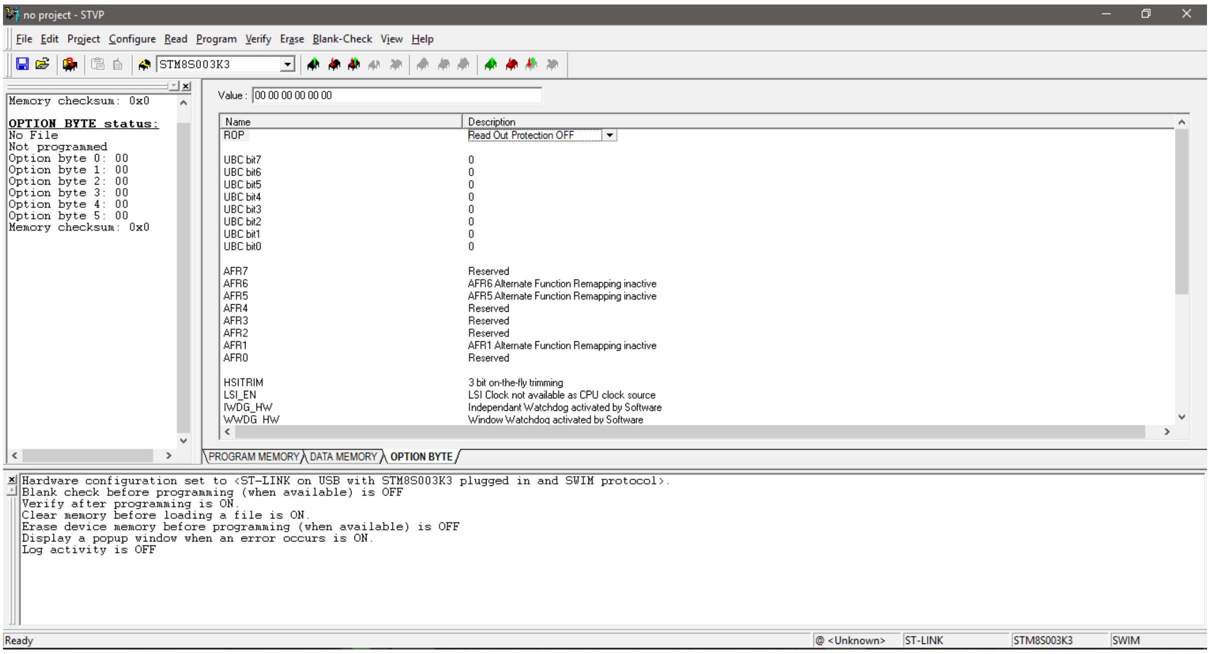
```
Memory checksum: 0x0

OPTION BYTE status:
No File
Not programmed
Option byte 0: 00
Option byte 1: 00
Option byte 2: 00
Option byte 3: 00
Option byte 4: 00
Option byte 5: 00
Memory checksum: 0x0
```

```
00008000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000080A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000080B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000080C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000080D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000080E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000080F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00008190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000081A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000081B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000081C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000081D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000081E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000081F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

```
Hardware configuration set to <ST-LINK on USB with STM8S003K3 plugged in and SWIM protocol>.
Blank check before programming (when available) is OFF
Verify after programming is ON.
Clear memory before loading a file is ON.
Erase device memory before programming (when available) is OFF
Display a popup window when an error occurs is ON.
Log activity is OFF
```

```
Ready                                   @ <Unknown>   ST-LINK   STM8S003K3   SWIM
```

Notice the mid tabs at the bottom of the hex values. From here we can see the hex values for program memory, data/EEPROM memory and configuration settings. The configuration setting bits are intended for setting some special hardware configurations or extending features of the target as well as setting memory readout protection.


no project - STVP
File Edit Project Configure Read Program Verify Erase Blank-Check View Help
STM8S003K3
PROGRAM MEMORY
DATA MEMORY
OPTION BYTE

```
Memory checksum: 0x0

OPTION BYTE status:
No File
Not programmed
Option byte 0: 00
Option byte 1: 00
Option byte 2: 00
Option byte 3: 00
Option byte 4: 00
Option byte 5: 00
Memory checksum: 0x0
```

Value : 00 00 00 00 00 00

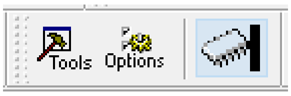
Name	Description
ROP	Read Out Protection OFF
UBC bit7	0
UBC bit6	0
UBC bit5	0
UBC bit4	0
UBC bit3	0
UBC bit2	0
UBC bit1	0
UBC bit0	0
AFR7	Reserved
AFR6	AFR6 Alternate Function Remapping inactive
AFR5	AFR5 Alternate Function Remapping inactive
AFR4	Reserved
AFR3	Reserved
AFR2	Reserved
AFR1	AFR1 Alternate Function Remapping inactive
AFR0	Reserved
HSITRIM	3 bit on-the-fly trimming
LSI_EN	LSI Clock not available as CPU clock source
IWDG_HW	Independant Watchdog activated by Software
WWDG_HW	Window Watchdog activated by Software

```
Hardware configuration set to <ST-LINK on USB with STM8S003K3 plugged in and SWIM protocol>.
Blank check before programming (when available) is OFF
Verify after programming is ON.
Clear memory before loading a file is ON.
Erase device memory before programming (when available) is OFF
Display a popup window when an error occurs is ON.
Log activity is OFF
```

Ready
@ <Unknown>
ST-LINK
STM8S003K3
SWIM

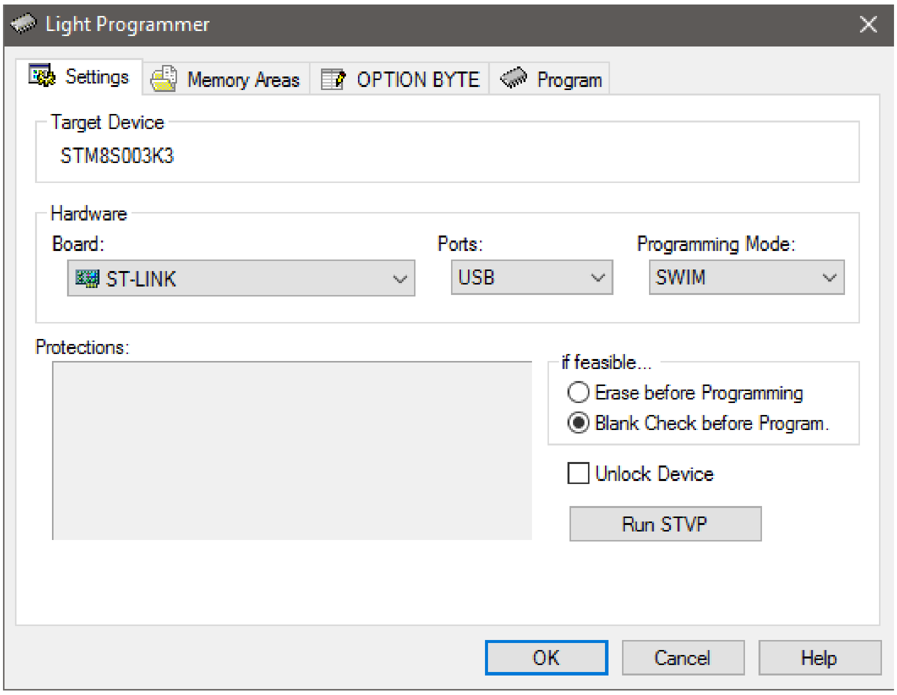
Try not to mess with security or protection bit at first or during tests as it will lock your chip up, rendering it useless. You won't simply be able to write it again until you unlock it. Unless needed, we won't be changing any default configuration bit. One thing to note is the fact that upon new compilation and build, the newly generated output file is automatically reloaded. The rest of the stuffs like loading or saving a s19 file, reading, writing and others are as simple as like with other programmers. I won't be explaining these steps as I assume that readers of this article already know how to do all these from their previous experiences with other MCUs.

Now we will explore how we can upload a code to our target using STVD. After compiling and building a project successfully without any error, the compiler will generate a s19 output file either in ***Debug*** or ***Release*** folder depending on which mode of compilation selected. By default, ***Debug*** mode is selected unless the coder changed it and so our desired s19 file will be in this folder. First, we need to open the programmer interface. We can do that either by clicking the icon as shown below:



or we can go to ***Tools >> Programmer***.

We will get a new window as shown below:



Light Programmer

Settings | Memory Areas | OPTION BYTE | Program

Target Device: STM8S003K3

Hardware

Board: ST-LINK

Ports: USB

Programming Mode: SWIM

Protections:

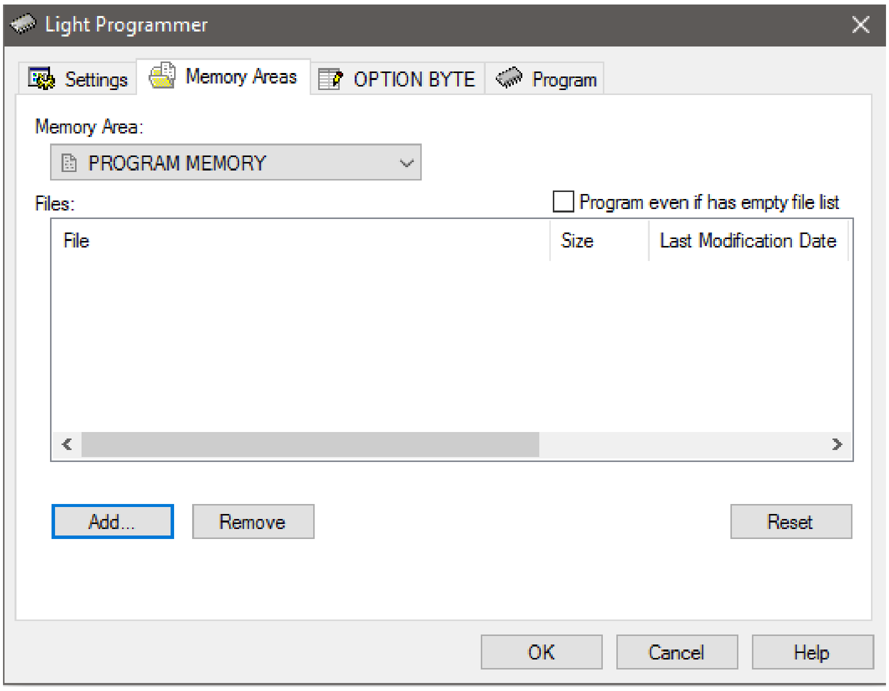
if feasible...

○ Erase before Programming

● Blank Check before Program.

□ Unlock Device

As the name of the new window suggests, it is a light-weight programmer interface but good enough for our purpose. Notice that there are many options and four tabs. Here again we need to select programmer, programming interface (SWIM) and erase/blank check options. Then we go to the next tab to select files for EEPROM (if any) and Program (also Flash/Code) memory as shown below. You can add/remove files just as usually.



Light Programmer

Settings | Memory Areas | OPTION BYTE | Program

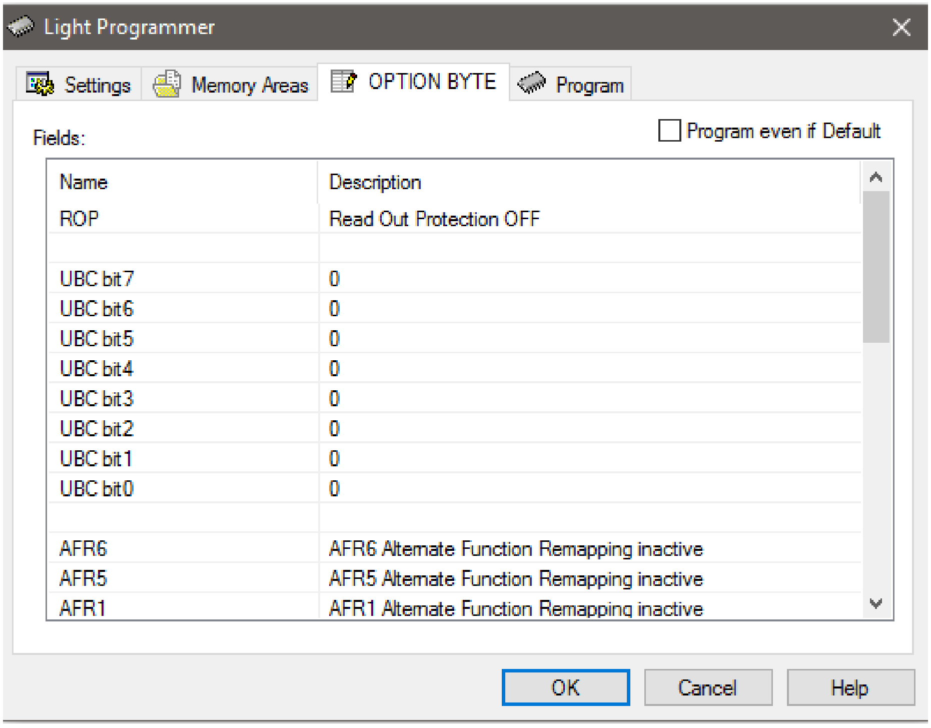
Memory Area: PROGRAM MEMORY

Files:

□ Program even if has empty file list

File	Size	Last Modification Date

Next, we set configuration bits if needed from the tab as shown below:

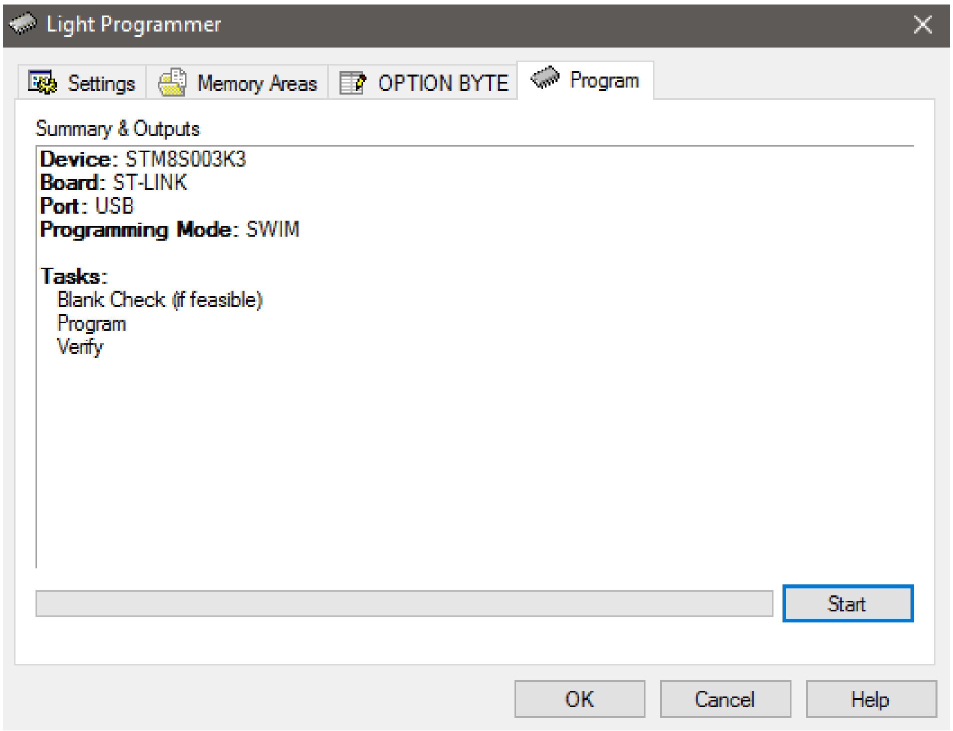


Fields:

□ Program even if Default

Name	Description
ROP	Read Out Protection OFF
UBC bit7	0
UBC bit6	0
UBC bit5	0
UBC bit4	0
UBC bit3	0
UBC bit2	0
UBC bit1	0
UBC bit0	0
AFR6	AFR6 Alternate Function Remapping inactive
AFR5	AFR5 Alternate Function Remapping inactive
AFR1	AFR1 Alternate Function Remapping inactive

Finally, we are ready to upload code. Just hit the start button and wait for the process to finish.



Summary & Outputs

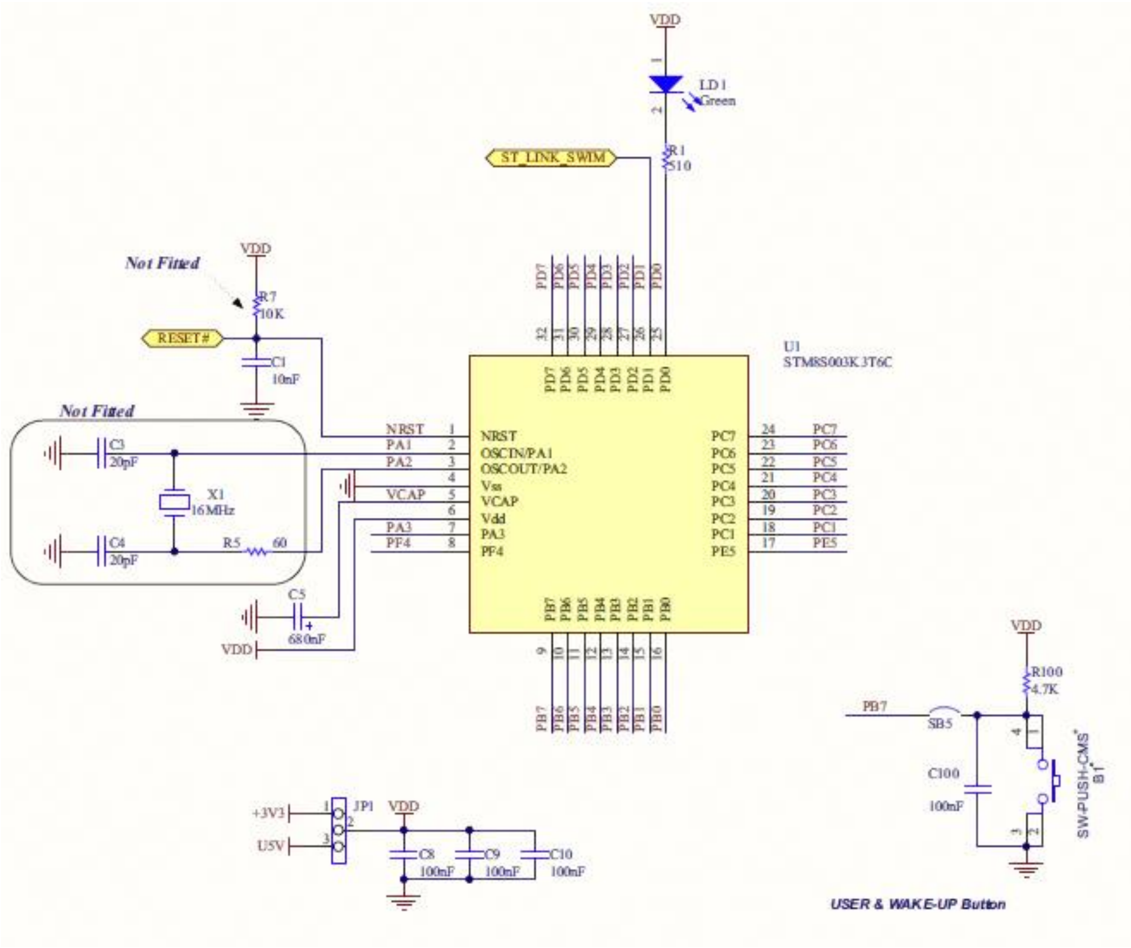
Device: STM8S003K3
Board: ST-LINK
Port: USB
Programming Mode: SWIM

Tasks:
Blank Check (if feasible)
Program
Verify

Every time a code is programmed, it is verified automatically.

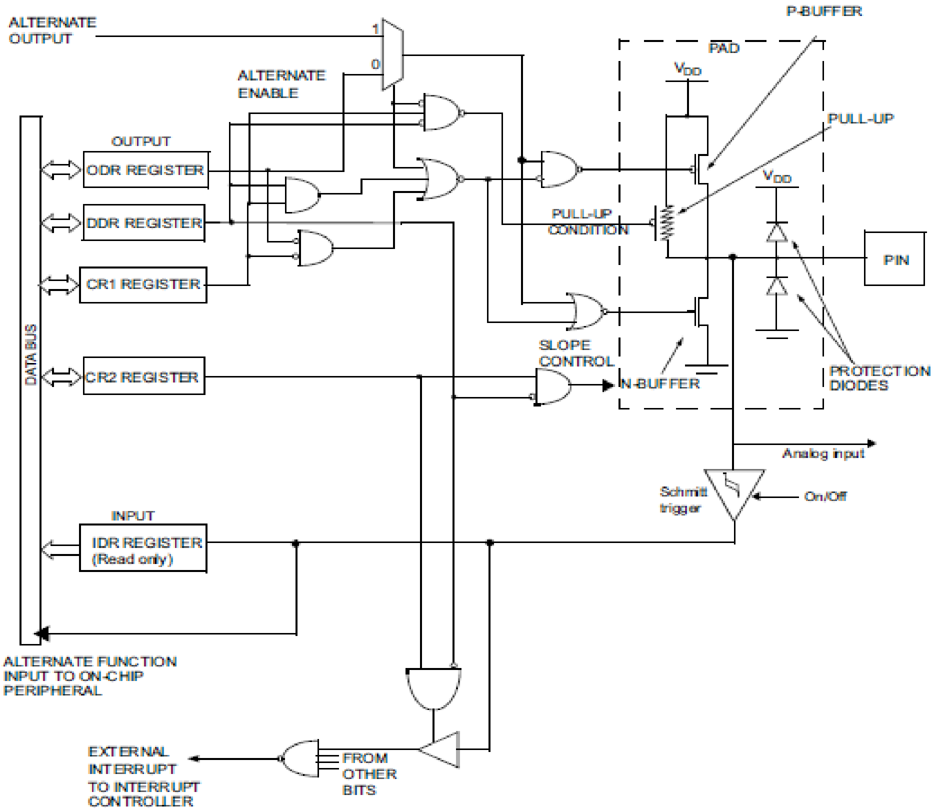
General Purpose Input Output (GPIO)

The very first *"Hello World"* project that we do with every new embedded device is a simple LED blinking program. Here we do the same. We will be basically testing both input and output function by making a variable flash rate blinking LED. Check the schematic of the Disco board. Check the pins with which the on-board LED and the push button are connected.

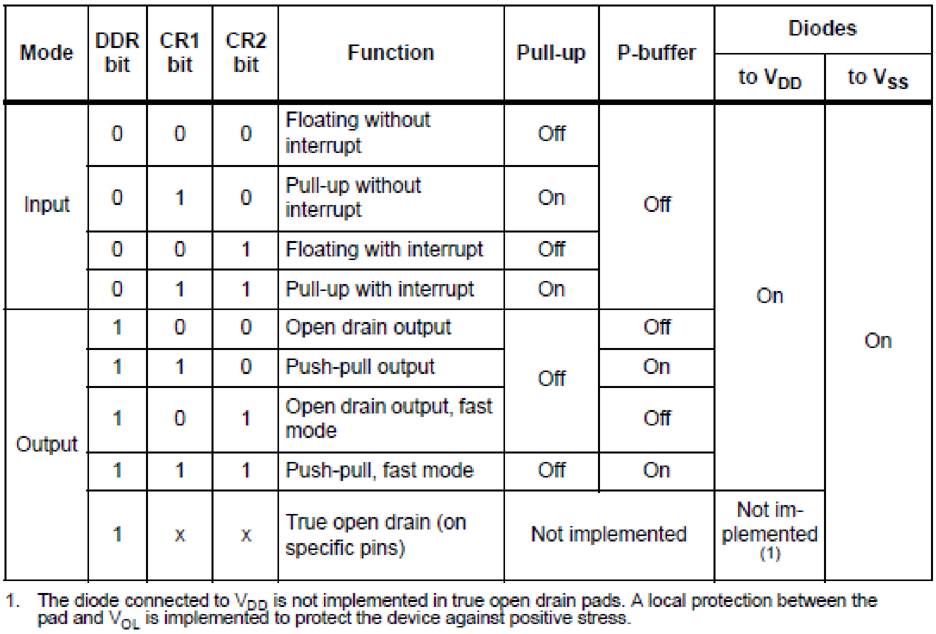


You can also use the STM8CubeMX in board selector mode for this too.

Shown below is the internal block diagram of GPIO pins:



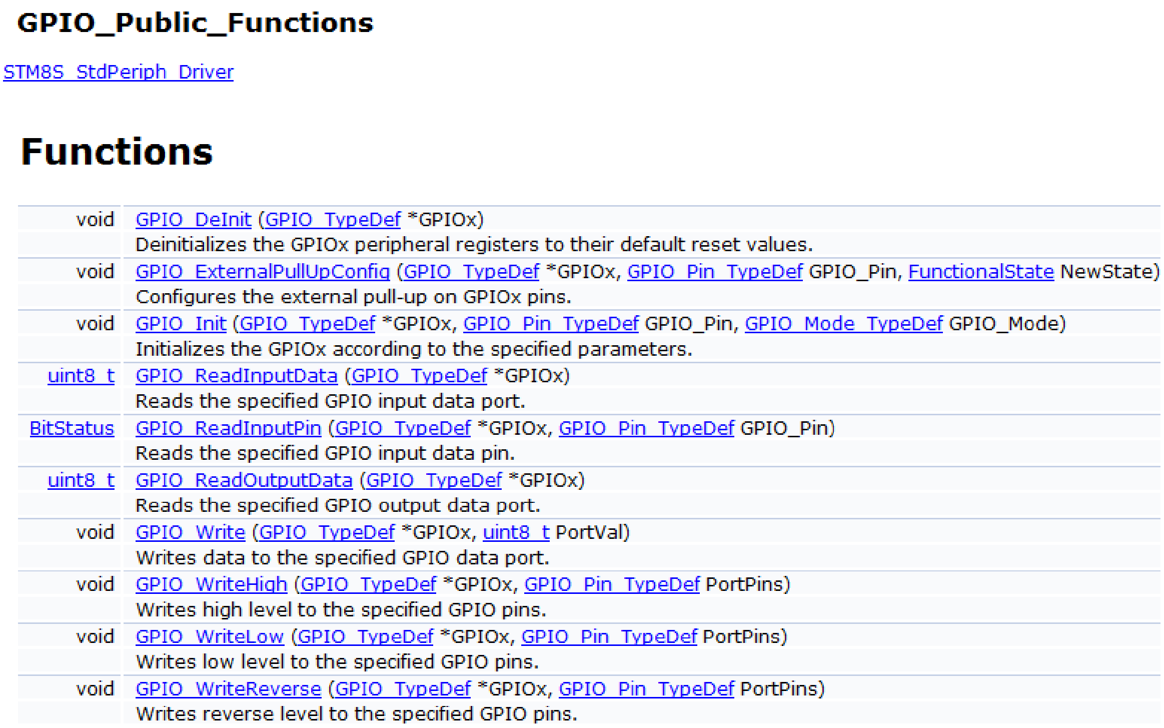
Because each I/O is independently configurable and have many options associated with it, its block looks complex at first sight. Check the various options every I/O possess:



Mode	DDR bit	CR1 bit	CR2 bit	Function	Pull-up	P-buffer	Diodes	
							to V_{DD}	to V_{SS}
Input	0	0	0	Floating without interrupt	Off	Off	On	On
	0	1	0	Pull-up without interrupt	On			
	0	0	1	Floating with interrupt	Off			
	0	1	1	Pull-up with interrupt	On			
Output	1	0	0	Open drain output	Off	Off		
	1	1	0	Push-pull output		On		
	1	0	1	Open drain output, fast mode		Off		
	1	1	1	Push-pull, fast mode	Off	On		
	1	x	x	True open drain (on specific pins)	Not implemented		Not implemented (1)	

1. The diode connected to V_{DD} is not implemented in true open drain pads. A local protection between the pad and V_{OL} is implemented to protect the device against positive stress.

Shown below are the SPL functions associated with the GPIO module.



GPIO_Public_Functions

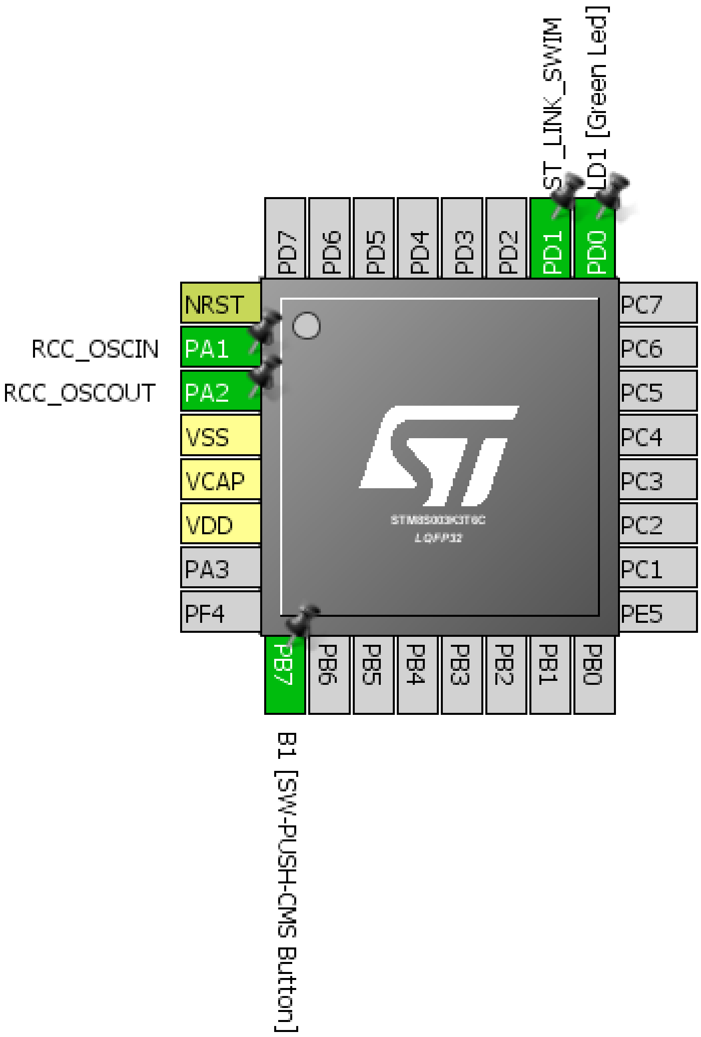
STM8S_StdPeriph_Driver

Functions

void	GPIO_DeInit (GPIO_TypeDef *GPIOx) Deinitializes the GPIOx peripheral registers to their default reset values.
void	GPIO_ExternalPullUpConfig (GPIO_TypeDef *GPIOx, GPIO_Pin_TypeDef GPIO_Pin, FunctionalState NewState) Configures the external pull-up on GPIOx pins.
void	GPIO_Init (GPIO_TypeDef *GPIOx, GPIO_Pin_TypeDef GPIO_Pin, GPIO_Mode_TypeDef GPIO_Mode) Initializes the GPIOx according to the specified parameters.
uint8_t	GPIO_ReadInputData (GPIO_TypeDef *GPIOx) Reads the specified GPIO input data port.
BitStatus	GPIO_ReadInputPin (GPIO_TypeDef *GPIOx, GPIO_Pin_TypeDef GPIO_Pin) Reads the specified GPIO input data pin.
uint8_t	GPIO_ReadOutputData (GPIO_TypeDef *GPIOx) Reads the specified GPIO output data port.
void	GPIO_Write (GPIO_TypeDef *GPIOx, uint8_t PortVal) Writes data to the specified GPIO data port.
void	GPIO_WriteHigh (GPIO_TypeDef *GPIOx, GPIO_Pin_TypeDef PortPins) Writes high level to the specified GPIO pins.
void	GPIO_WriteLow (GPIO_TypeDef *GPIOx, GPIO_Pin_TypeDef PortPins) Writes low level to the specified GPIO pins.
void	GPIO_WriteReverse (GPIO_TypeDef *GPIOx, GPIO_Pin_TypeDef PortPins) Writes reverse level to the specified GPIO pins.

Observe the code below. This is the power of the ST's SPL. The code is written with no traditional register access. Everything here has a meaningful nomenclature, just like regular naming/words of the reference manual/comments. There shouldn't be any issue understanding the code. The code is almost Arduino-like. Here we are polling an input pin's state to alter the blink rate of a LED.

Hardware Connection



Code Example

```
#include "STM8S.h"


void main (void)
{
     bool i = 0;

     GPIO_DeInit(GPIOB);
    GPIO_DeInit(GPIOD);

    GPIO_Init(GPIOB, GPIO_PIN_7, GPIO_MODE_IN_FL_NO_IT);
    GPIO_Init(GPIOD, GPIO_PIN_0, GPIO_MODE_OUT_PP_LOW_FAST);

    for(;;)
   {
               if(GPIO_ReadInputPin(GPIOB, GPIO_PIN_7) == FALSE)
               {
                     while(GPIO_ReadInputPin(GPIOB, GPIO_PIN_7) == FALSE);
```

```
                i ^= 1;
            }

            switch(i)
            {
                case 0:
                {
                        delay_ms(1000);
                        break;
                }
               case 1:
               {
                        delay_ms(200);
                        break;
               }
            }

            GPIO_WriteReverse(GPIOD, GPIO_PIN_0);
    }
}
```

Explanation

The following lines deinitialize the GPIOs we used. Every time you reconfigure or setup a hardware peripheral for the first time you must deinitialize it before using it. Though it is not mandatory, it will remove any chance of wrong/conflicting configurations.

```
 GPIO_DeInit(GPIOB);
 GPIO_DeInit(GPIOD);
```

After deinitialization, we are good to go for initializing or setting up the GPIOs. Inputs can be with or without internal pull-up resistors. Outputs can be either push-pull totem-pole or open drain types. Each pin can be individually configured and does not have any dependency on another. The following codes set GPIO PB7 as a floating input with no interrupt capability and GPIO PD0 as a fast push-pull output. PB7 is set up as a floating input rather than an internally pulled-up input because the button on the Disco board is already pulled up externally.

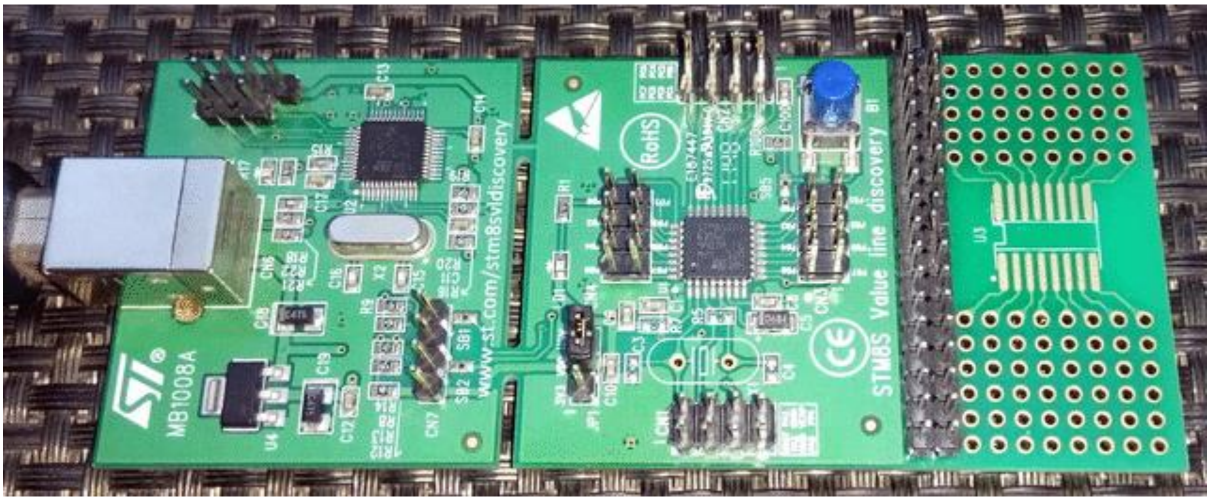
```
GPIO_Init(GPIOB, GPIO_PIN_7, GPIO_MODE_IN_FL_NO_IT);
GPIO_Init(GPIOD, GPIO_PIN_0, GPIO_MODE_OUT_PP_LOW_FAST);
```

The remaining part of the code in the main loop is just polling the button's state and altering the delay time for toggling the LED if the button is pressed.

```
for(;;)
{
            if(GPIO_ReadInputPin(GPIOB, GPIO_PIN_7) == FALSE)
            {
                while(GPIO_ReadInputPin(GPIOB, GPIO_PIN_7) == FALSE);
                i ^= 1;
            }

            switch(i)
            {
                case 0:
                {
                        delay_ms(1000);
                        break;
                }
```

```
          case 1:
          {
                delay_ms(200);
                break;
          }
        }

        GPIO_WriteReverse(GPIOD, GPIO_PIN_0);
}
```

Demo

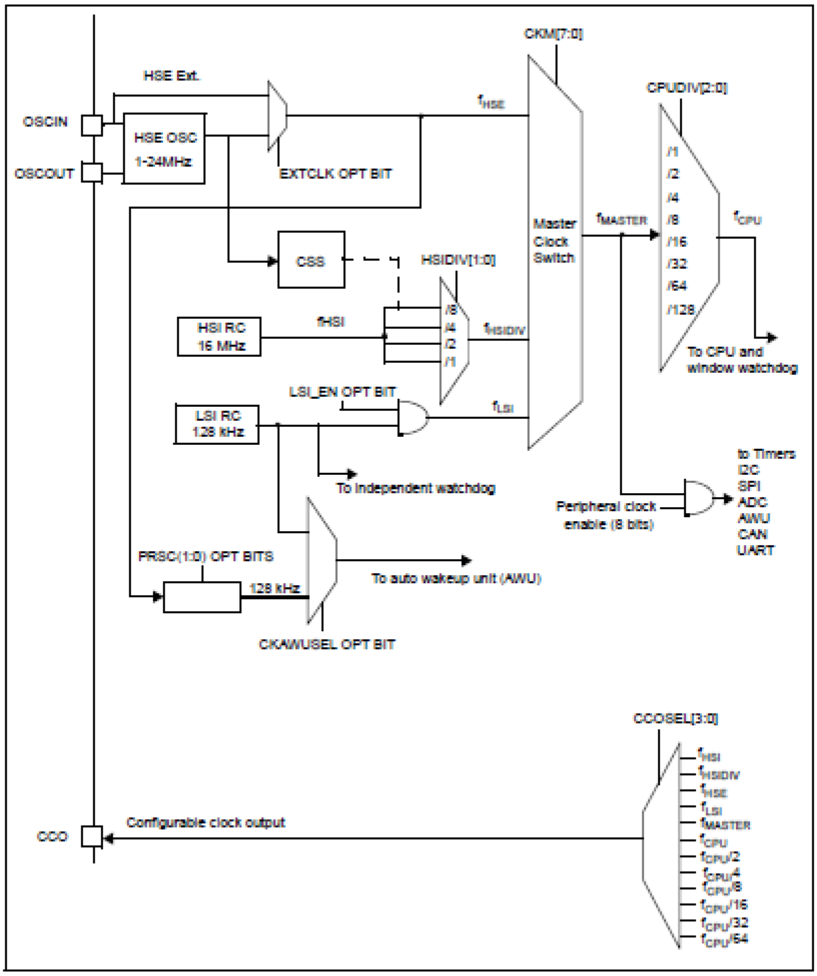


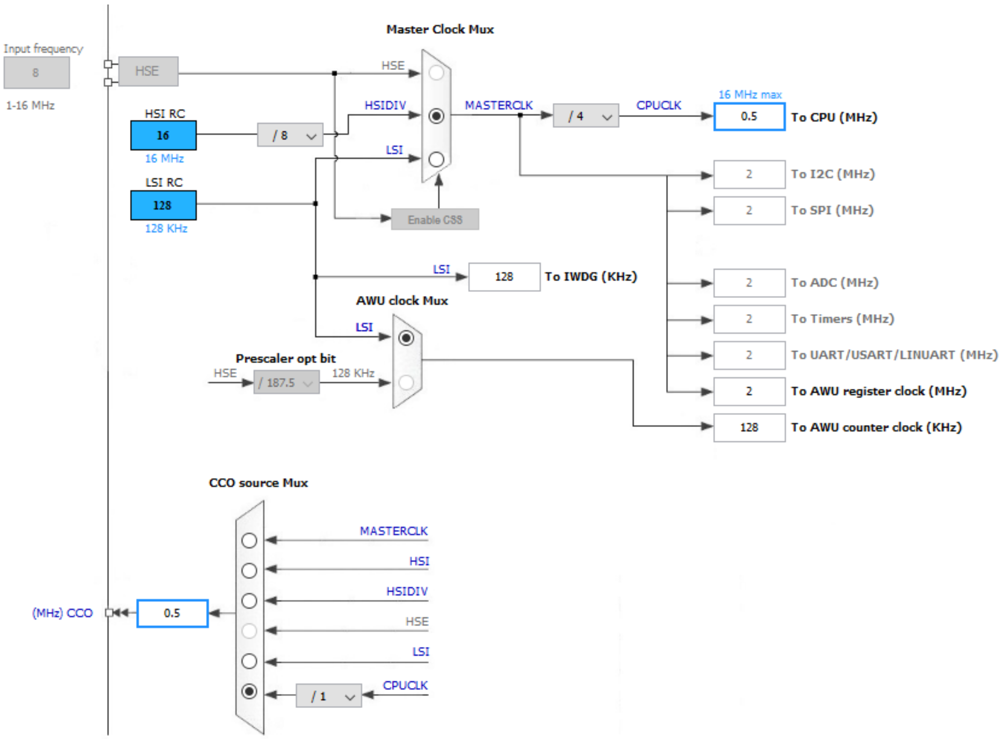
Video link: https://www.youtube.com/watch?v=Rr1vpfoze4w

Clock System (CLK)

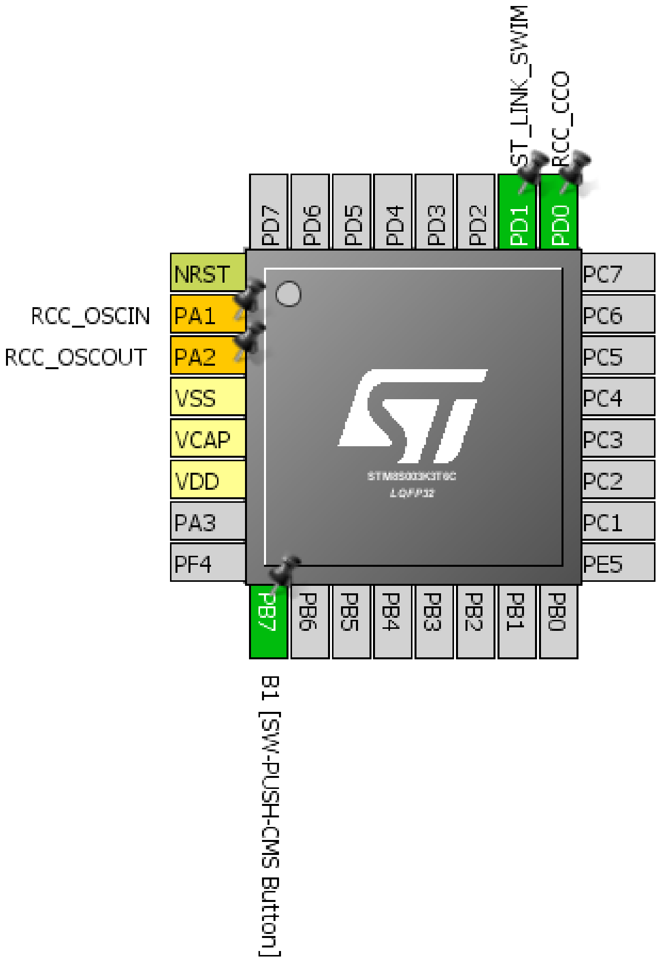
The internal network of STM8 clock system allows us to tune up operating speeds of peripherals and CPU according to our needs. Software delays and power consumption depend on how the clock system is set.

In STM8 micros, there are three main clock sources – ***High Speed Internal Clock (HSI)***, ***High Speed External Clock (HSE)*** and ***Low Speed Internal Clock (LSI)***. The HSI has an oscillating frequency of 16MHz and is an internal RC oscillator with good precision – about 1% tolerant over a wide temperature range. HSE can be an external clock circuitry, temperature-compensated crystal oscillator (TCXO) or ordinary crystal resonator. It accepts all frequencies from 1MHz to 24MHz. Lastly, LSI clock is also an independent internal RC oscillator-based clock source that is mainly intended for idle or low power operating modes and the independent watchdog timer (IWDG). It has a fixed factory calibrated operating frequency of 128kHz and is not as accurate as HSI or HSE. There are also clock dividers/prescalers at various points to scale clocks as per requirement. Mainly two prescalers are what we need – the HSI prescaler and the CPU divider. Peripherals are directly feed by the main clock source. Additionally, there is a clock output pin (CCO) that outputs a given clock frequency. It can be used to clock another micro, generate clock for other devices like logic ICs. It can also be used as a free oscillator or perform clock performance tests. There’s fail-safe clock security that makes HSI backup of HSE. Should HSE fail, HSI takes over automatically. Check the internal block diagram below:





Hardware Connection



Code Example

The code example below demonstrates how to run the CPU at 2MHz clock using HSI and extract 500kHz clock output from CCO pin using CCO output selection. HSE is divided by 8, i.e. 16MHz divided by 8 equals 2MHz. This 2MHz is the master clock source and further divided four times to get 500kHz.

Note CCO pin is only available in some pins. For example, in STM8S003K3 this pin is either PD0 pin or PC4. We will need to override the default function of PD0 pin to favour for CCO output. To do so, we will need to change Alternate Function (***AFR5***) configuration bit during code upload.

```
#include "STM8S.h"


#define LED_pin                       GPIO_PIN_0
#define LED_port                      GPIOD


void setup(void);
void clock_setup(void);
void GPIO_setup(void);


void main(void)
{
        setup();
        GPIO_WriteLow(LED_port, LED_pin);

        while(TRUE)
        {
        };
}


void setup(void)
{
        clock_setup();
        GPIO_setup();
}


void clock_setup(void)
{
        CLK_DeInit();

        CLK_HSECmd(DISABLE);
        CLK_LSICmd(DISABLE);
        CLK_HSICmd(ENABLE);
        while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

        CLK_ClockSwitchCmd(ENABLE);
        CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
        CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV4);

        CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
        DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

        CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
```

```
                CLK_CCOConfig(CLK_OUTPUT_CPU);
                CLK_CCOCmd(ENABLE);
                while(CLK_GetFlagStatus(CLK_FLAG_CCORDY) == FALSE);
}


void GPIO_setup(void)
{
                GPIO_DeInit(LED_port);
                GPIO_Init(LED_port, LED_pin, GPIO_MODE_OUT_OD_HIZ_FAST);
}
```

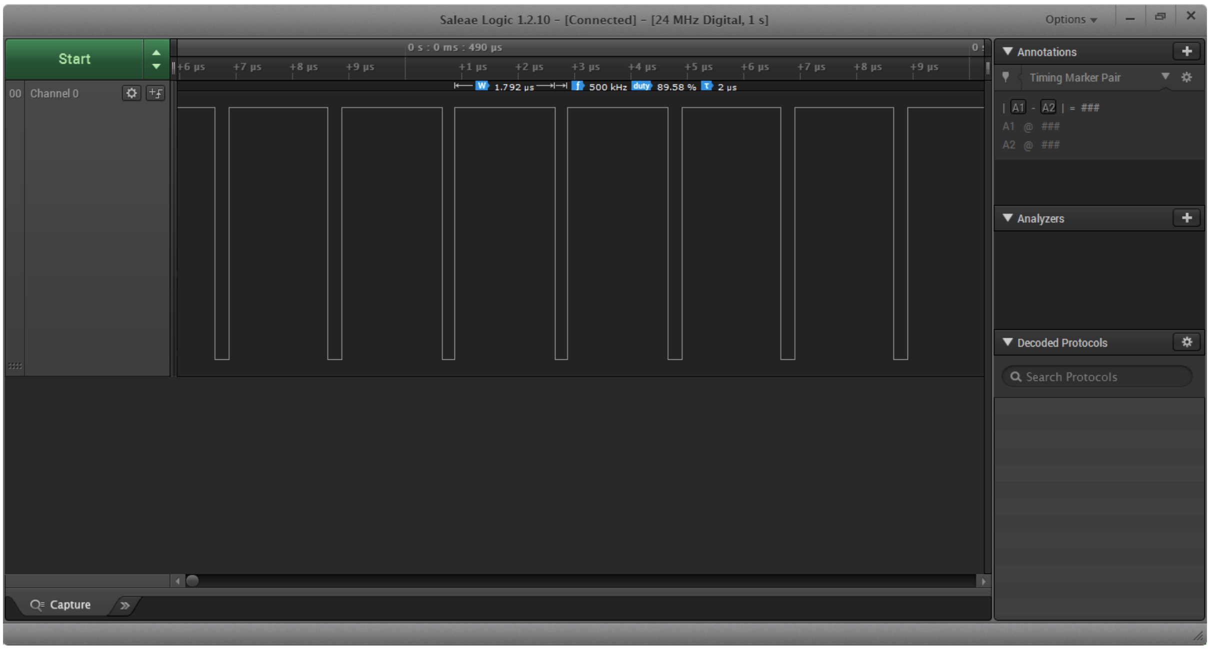
Explanation

The full explanation of this code is given in the last segment of this article. The only thing I'll describe here is this part:

```
CLK_CCOConfig(CLK_OUTPUT_CPU);
CLK_CCOCmd(ENABLE);
while(CLK_GetFlagStatus(CLK_FLAG_CCORDY) == FALSE);
```

These lines select the clock source that the CCO pin will output, enable the CCO module and wait for it to stabilize. Here I selected the CCO to output CPU clock.

Demo

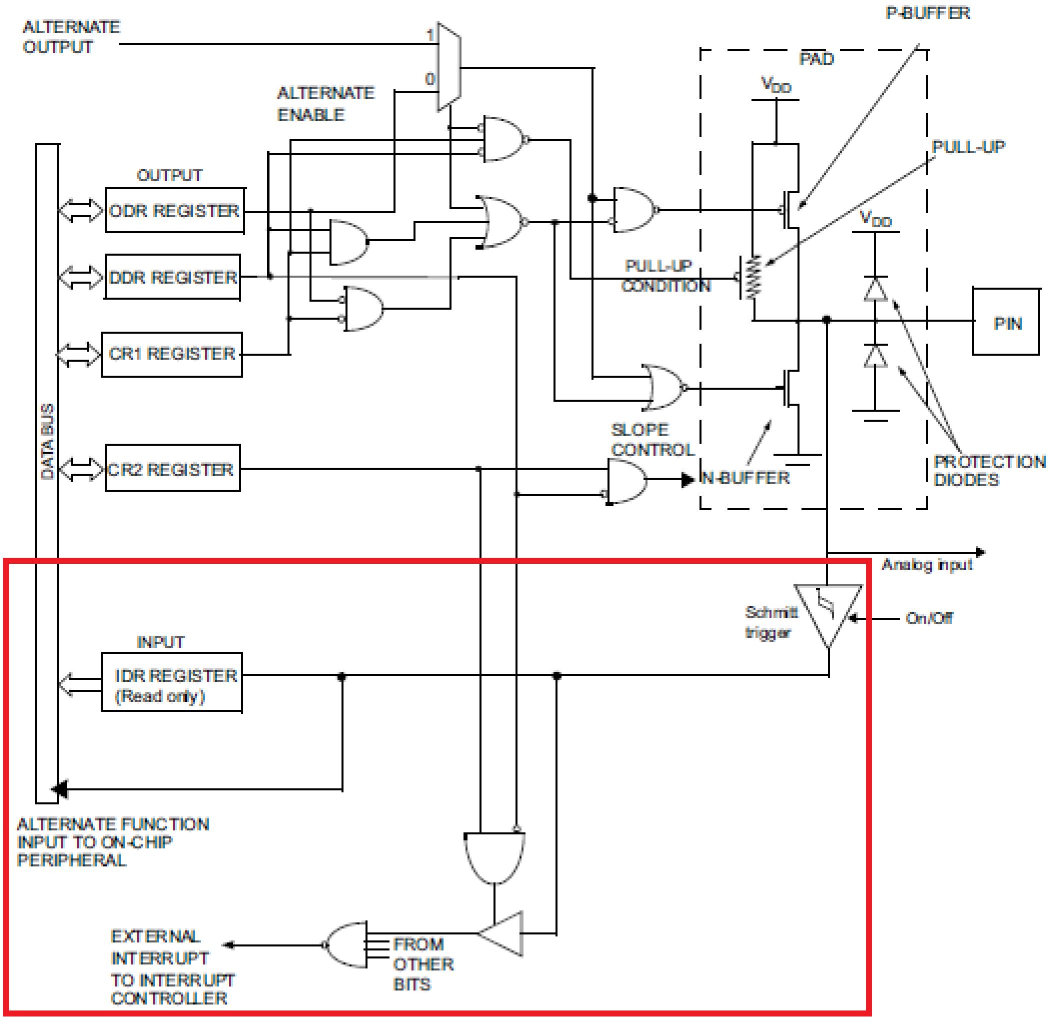


Video link: https://www.youtube.com/watch?v=IeLUc_s3jBE

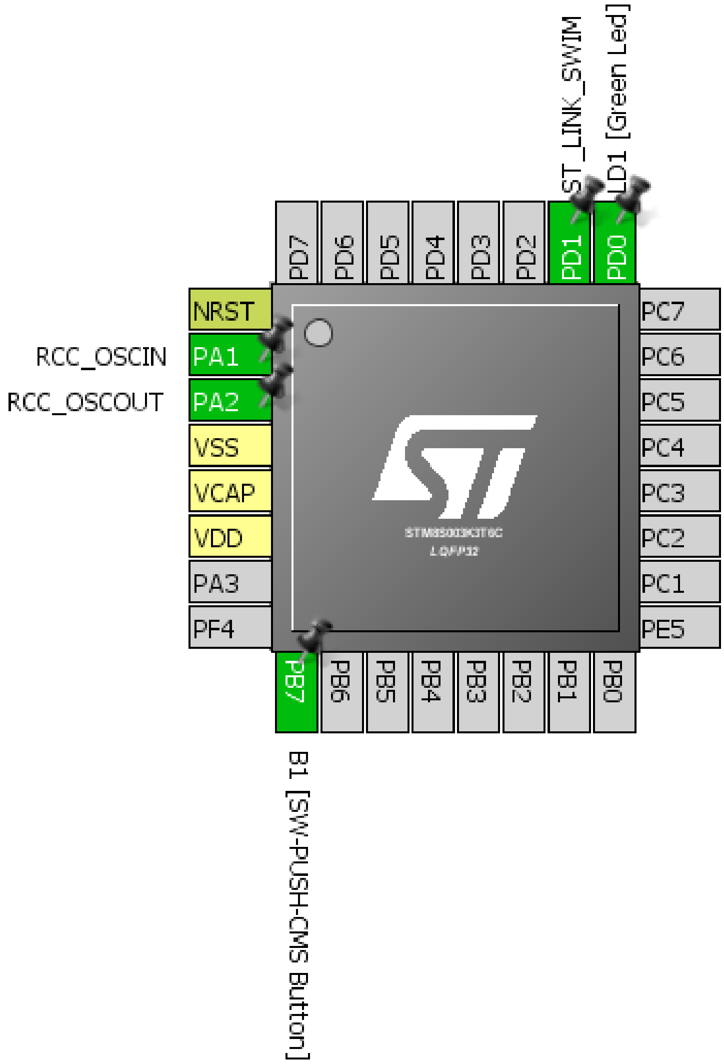
External Interrupt (EXTI)

External interrupt is an additional feature of GPIOs in input mode. It makes a micro to respond instantly to changes done on it GPIO input pin(s) by an external events/triggers, skipping other tasks. External interrupts are useful in many scenarios. For instance, an emergency button. Consider the case of a treadmill. You are running at a speed and suddenly you get an ache in one of your ankles. You'll surely want to stop running immediately. Rather decreasing speed in small steps, you can hit the emergency button to quickly stop the treadmill. The emergency button will interrupt all other processes and immediately instruct the treadmill's CPU to decrease speed faster than otherwise possible. Thus, it has high priority over other processes.

In most 8-bit micros, there are few external interrupt pins and very limited options are available for them but that's not the case with STM's micros. In STM8s, almost all GPIO pins have independent external interrupt capability with input Schmitt triggers. Additionally, there's interrupt controller to set interrupt priority.



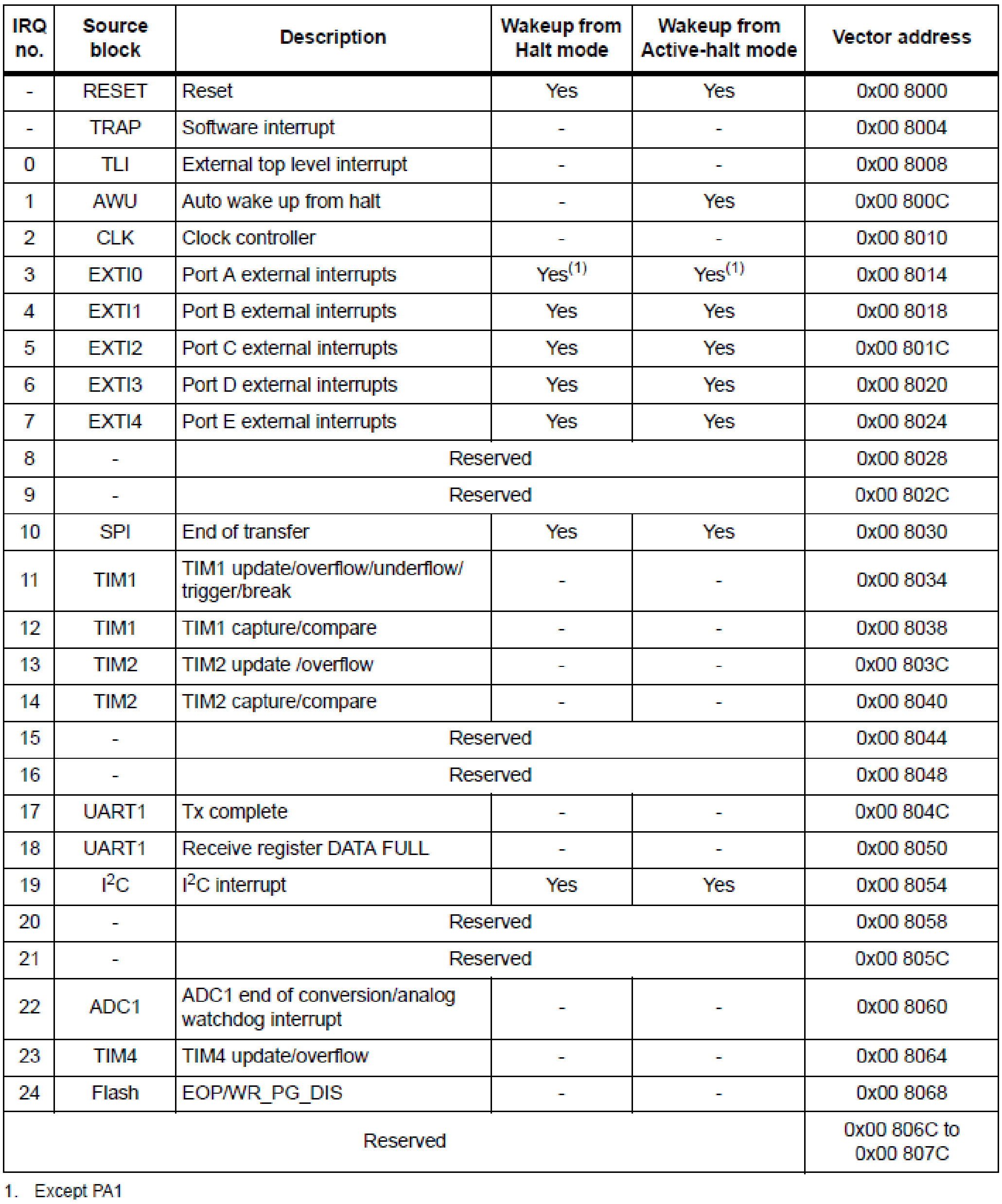
Hardware Connection



Code Example

We will do the same variable flash rate LED blinking example as in the GPIO example but this time with the DISCO board's button in external interrupt mode. The code here needs special attention as now we will see how interrupts are configured and coded. This code is way different than anything I saw before. I'm saying so because you'll need to follow certain steps unlike other compilers. In other compilers, all we do is create the interrupt function and tell the compiler the interrupt vector address. Same here too but too many steps involved.

Now check the interrupt vector table of STM8S003 below:



IRQ no.	Source block	Description	Wakeup from Halt mode	Wakeup from Active-halt mode	Vector address
-	RESET	Reset	Yes	Yes	0x00 8000
-	TRAP	Software interrupt	-	-	0x00 8004
0	TLI	External top level interrupt	-	-	0x00 8008
1	AWU	Auto wake up from halt	-	Yes	0x00 800C
2	CLK	Clock controller	-	-	0x00 8010
3	EXTI0	Port A external interrupts	Yes(1)	Yes(1)	0x00 8014
4	EXTI1	Port B external interrupts	Yes	Yes	0x00 8018
5	EXTI2	Port C external interrupts	Yes	Yes	0x00 801C
6	EXTI3	Port D external interrupts	Yes	Yes	0x00 8020
7	EXTI4	Port E external interrupts	Yes	Yes	0x00 8024
8	-	Reserved			0x00 8028
9	-	Reserved			0x00 802C
10	SPI	End of transfer	Yes	Yes	0x00 8030
11	TIM1	TIM1 update/overflow/underflow/ trigger/break	-	-	0x00 8034
12	TIM1	TIM1 capture/compare	-	-	0x00 8038
13	TIM2	TIM2 update /overflow	-	-	0x00 803C
14	TIM2	TIM2 capture/compare	-	-	0x00 8040
15	-	Reserved			0x00 8044
16	-	Reserved			0x00 8048
17	UART1	Tx complete	-	-	0x00 804C
18	UART1	Receive register DATA FULL	-	-	0x00 8050
19	I^{2}C	I^{2}C interrupt	Yes	Yes	0x00 8054
20	-	Reserved			0x00 8058
21	-	Reserved			0x00 805C
22	ADC1	ADC1 end of conversion/analog watchdog interrupt	-	-	0x00 8060
23	TIM4	TIM4 update/overflow	-	-	0x00 8064
24	Flash	EOP/WR_PG_DIS	-	-	0x00 8068
Reserved					0x00 806C to 0x00 807C

1. Except PA1

You'll find this table not in the reference manual but in the device's datasheet. This table varies with devices and so be sure of correct datasheet. The DISCO board's button is connected to PB7 and so clearly, we will need ***IRQ4***, i.e. EXTI1 or PORTB external interrupts. All external interrupts on GPIOB pin are masked in this vector address.

Please note that codes that use peripheral interrupts need ***stm8s_it.h*** and ***stm8s_it.c*** files. Therefore, add them if you are to use interrupts.

main.c

```
#include "stm8s.h"


bool state = FALSE;


void GPIO_setup(void);
void EXTI_setup(void);
void clock_setup(void);


void main(void)
{
        GPIO_setup();
        EXTI_setup();
        clock_setup();

        do
        {
                GPIO_WriteReverse(GPIOD, GPIO_PIN_0);
                if(state == TRUE)
                {
                        delay_ms(100);
                }
                else
                {
                        delay_ms(1000);
                }
        }while (TRUE);
}


void GPIO_setup(void)
{
        GPIO_DeInit(GPIOB);
        GPIO_Init(GPIOB, GPIO_PIN_7, GPIO_MODE_IN_PU_IT);

        GPIO_DeInit(GPIOD);
        GPIO_Init(GPIOD, GPIO_PIN_0, GPIO_MODE_OUT_PP_LOW_FAST);
}


void EXTI_setup(void)
{
        ITC_DeInit();
        ITC_SetSoftwarePriority(ITC_IRQ_PORTB, ITC_PRIORITYLEVEL_0);

        EXTI_DeInit();
        EXTI_SetExtIntSensitivity(EXTI_PORT_GPIOB, EXTI_SENSITIVITY_FALL_ONLY);
        EXTI_SetTLISensitivity(EXTI_TLISENSITIVITY_FALL_ONLY);

        enableInterrupts();
}


void clock_setup(void)
{
        CLK_DeInit();

        CLK_HSECmd(DISABLE);
        CLK_LSICmd(DISABLE);
        CLK_HSICmd(ENABLE);
        while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);
```

```
                CLK_ClockSwitchCmd(ENABLE);
                CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);
                CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);

                CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
                DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

                CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
                CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
                CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
                CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
                CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
                CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
                CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
                CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}
```

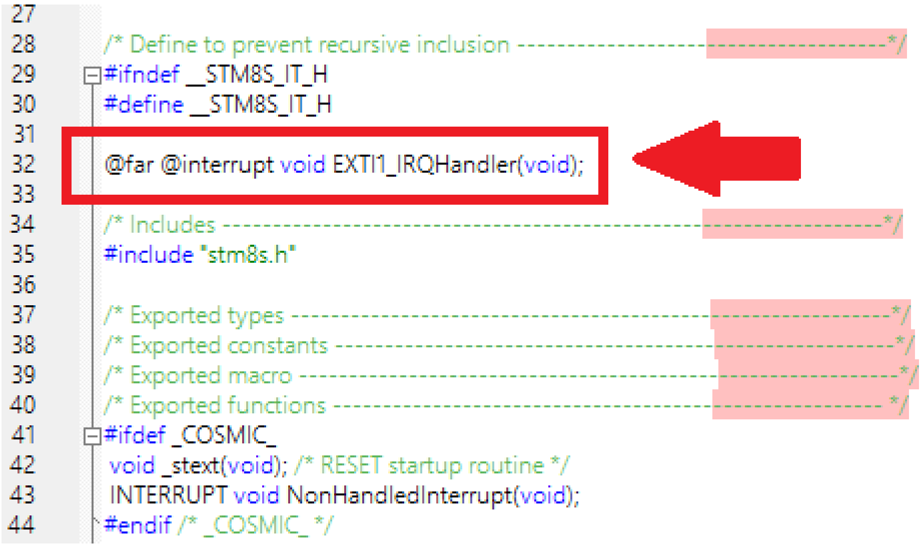
stm8_interrupt_vector.c

```
/*              BASIC INTERRUPT VECTOR TABLE FOR STM8 devices
 *              Copyright (c) 2007 STMicroelectronics
 */

 #include "stm8s_it.h"


typedef void @far (*interrupt_handler_t)(void);

struct interrupt_vector {
                unsigned char interrupt_instruction;
                interrupt_handler_t interrupt_handler;
};

//@far @interrupt void NonHandledInterrupt (void)
//{
                /* in order to detect unexpected events during development,
                   it is recommended to set a breakpoint on the following instruction
                */
//              return;
//}

extern void _stext();     /* startup routine */


struct interrupt_vector const _vectab[] = {
                {0x82, (interrupt_handler_t)_stext}, /* reset */
                {0x82, NonHandledInterrupt}, /* trap  */
                {0x82, NonHandledInterrupt}, /* irq0  */
                {0x82, NonHandledInterrupt}, /* irq1  */
                {0x82, NonHandledInterrupt}, /* irq2  */
                {0x82, NonHandledInterrupt}, /* irq3  */
                {0x82, (interrupt_handler_t)EXTI1_IRQHandler}, /* irq4  */
                {0x82, NonHandledInterrupt}, /* irq5  */
                {0x82, NonHandledInterrupt}, /* irq6  */
                {0x82, NonHandledInterrupt}, /* irq7  */
                {0x82, NonHandledInterrupt}, /* irq8  */
                {0x82, NonHandledInterrupt}, /* irq9  */
                {0x82, NonHandledInterrupt}, /* irq10 */
                {0x82, NonHandledInterrupt}, /* irq11 */
                {0x82, NonHandledInterrupt}, /* irq12 */
                {0x82, NonHandledInterrupt}, /* irq13 */
                {0x82, NonHandledInterrupt}, /* irq14 */
                {0x82, NonHandledInterrupt}, /* irq15 */
                {0x82, NonHandledInterrupt}, /* irq16 */
                {0x82, NonHandledInterrupt}, /* irq17 */
```

```
        {0x82, NonHandledInterrupt}, /* irq18 */
        {0x82, NonHandledInterrupt}, /* irq19 */
        {0x82, NonHandledInterrupt}, /* irq20 */
        {0x82, NonHandledInterrupt}, /* irq21 */
        {0x82, NonHandledInterrupt}, /* irq22 */
        {0x82, NonHandledInterrupt}, /* irq23 */
        {0x82, NonHandledInterrupt}, /* irq24 */
        {0x82, NonHandledInterrupt}, /* irq25 */
        {0x82, NonHandledInterrupt}, /* irq26 */
        {0x82, NonHandledInterrupt}, /* irq27 */
        {0x82, NonHandledInterrupt}, /* irq28 */
        {0x82, NonHandledInterrupt}, /* irq29 */
};
```

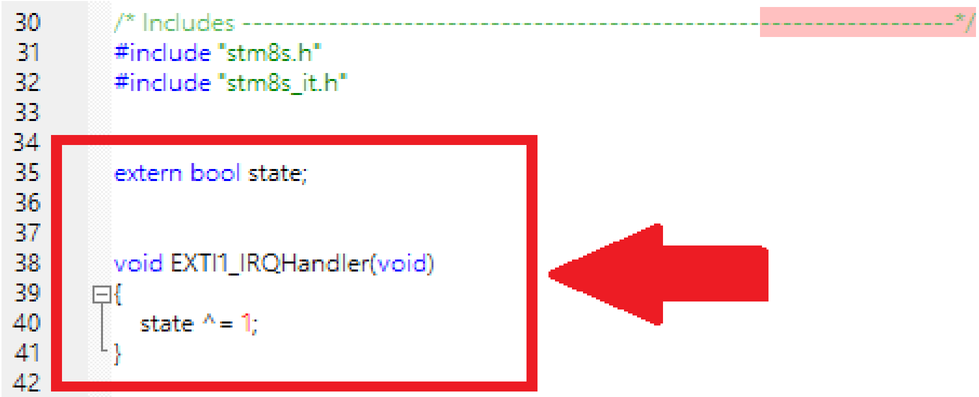
Now check the top parts of the ***stm8s_it.h*** and ***stm8s_it.c*** files respectively.

stm8s_it.h



```

/* Define to prevent recursive inclusion -------------------------------------*/
#ifndef __STM8S_IT_H
#define __STM8S_IT_H

@far @interrupt void EXTI1_IRQHandler(void);

/* Includes ------------------------------------------------------------------*/
#include "stm8s.h"

/* Exported types ------------------------------------------------------------*/
/* Exported constants --------------------------------------------------------*/
/* Exported macro ------------------------------------------------------------*/
/* Exported functions ------------------------------------------------------- */
#ifdef _COSMIC_
 void _stext(void); /* RESET startup routine */
 INTERRUPT void NonHandledInterrupt(void);
#endif /* _COSMIC_ */
```

stm8s_it.c



```
/* Includes ------------------------------------------------------------------*/
#include "stm8s.h"
#include "stm8s_it.h"


extern bool state;


void EXTI1_IRQHandler(void)
{
    state ^= 1;
}

```

These must be coded.

Explanation

Most part of the code is same as previous codes and so I won't be going through them again. However, there's something new:

```
void EXTI_setup(void)
{
        ITC_DeInit();
        ITC_SetSoftwarePriority(ITC_IRQ_PORTB, ITC_PRIORITYLEVEL_0);

        EXTI_DeInit();
        EXTI_SetExtIntSensitivity(EXTI_PORT_GPIOB, EXTI_SENSITIVITY_FALL_ONLY);
        EXTI_SetTLISensitivity(EXTI_TLISENSITIVITY_FALL_ONLY);

        enableInterrupts();
}
```

This function is where we are setting up the external interrupt. The first two lines deinitiate the interrupt controller and set priority while initiating it. It is not mandatory unless you want to set interrupt priority. Then we configure the external interrupt on PORTB pins. We also set the edge that will invoke an interrupt. Finally, we enable global interrupt. There goes the ***main.c*** file

Now it's time to explain the ***stm8_interrupt_vector.c*** file. The top part of this file must include this line `#include "stm8s_it.h"`. It must also have the following section commented out:

```
//@far @interrupt void NonHandledInterrupt (void)
//{
        /* in order to detect unexpected events during development,
           it is recommended to set a breakpoint on the following instruction
        */
//      return;
//}
```

We need to let our compiler know the name of the function that it should call when a particular interrupt is triggered. There are two parts for that. Firstly, the interrupt vector address and secondly the name of the function. This is reason for this line:

```
{0x82, (interrupt_handler_t)EXTI1_IRQHandler}, /* irq4  */
```

Lastly, the ***stm8s_it.h*** and ***stm8s_it.c*** files contain the prototype and the function that will execute the interrupt service routine (ISR). In our case, the ISR will change the logic state of the Boolean variable ***state***. This will alter that flashing rate in the main loop.

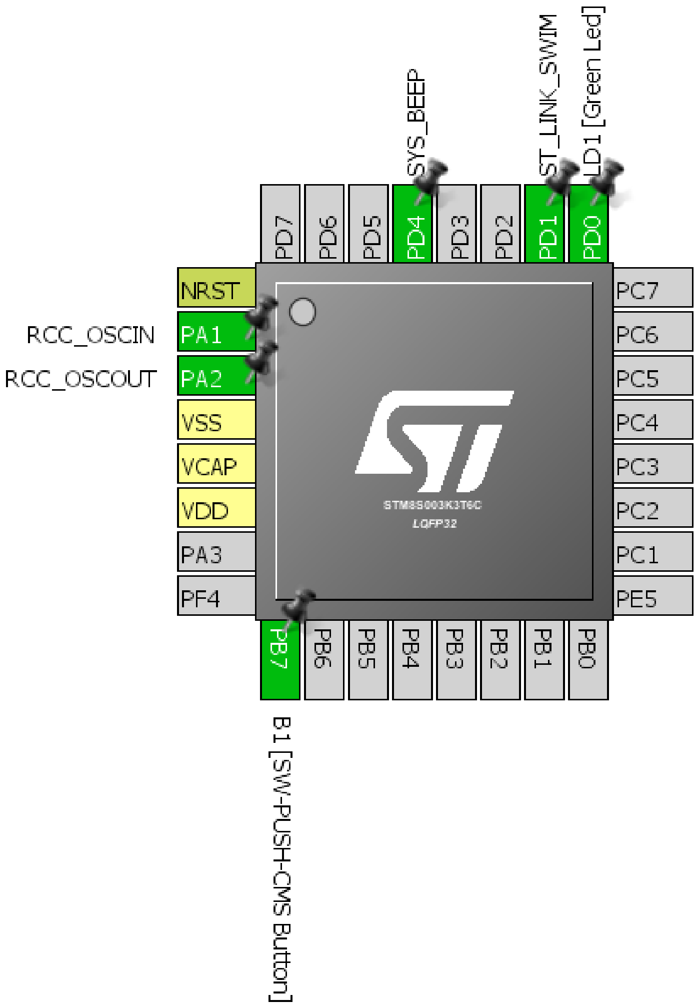
Demo

Video link: https://www.youtube.com/watch?v=P6qdmWgH-Ls

Beeper (BEEP)

The beeper hardware is a sound generation unit. This is a hardware not found in other micros and is useful in scenarios where we need an audible output. An alarm is a good example. The beeper unit uses LSI to generate 1kHz, 2kHz and 4kHz square wave outputs that can be directly feed to a small piezo tweeter (not buzzer). In most STM8 micros, the beeper module's I/O pin (PD4) is not accessible unless alternate function configuration bit is altered during code upload. However, there are few exceptional chips like the STM8S003 in which we don't need to change any configuration bit at all. The beeper module has dependencies with the Auto-Wake-Up (AWU) module.

Hardware Connection



Code Example

```
#include "STM8S.h"


void clock_setup(void);
void GPIO_setup(void);
void beeper_setup(void);


void main(void)
{
    clock_setup();
    GPIO_setup();
    beeper_setup();

    while(TRUE)
    {
       GPIO_WriteLow(GPIOD, GPIO_PIN_0);
       BEEP_Cmd(ENABLE);
       delay_ms(200);

       GPIO_WriteHigh(GPIOD, GPIO_PIN_0);
       BEEP_Cmd(DISABLE);
       delay_ms(200);
    };
}


void clock_setup(void)
{
    CLK_DeInit();

    CLK_HSECmd(DISABLE);
    CLK_LSICmd(ENABLE);
    while(CLK_GetFlagStatus(CLK_FLAG_LSIRDY) == FALSE);
    CLK_HSICmd(ENABLE);
    while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

    CLK_ClockSwitchCmd(ENABLE);
    CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);
    CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);

    CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
    DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

     CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
     CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
     CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
     CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, ENABLE);
     CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
     CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
     CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
     CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}


void GPIO_setup(void)
{
     GPIO_DeInit(GPIOD);
     GPIO_Init(GPIOD, GPIO_PIN_0, GPIO_MODE_OUT_OD_LOW_FAST);
     GPIO_Init(GPIOD, GPIO_PIN_4, GPIO_MODE_OUT_PP_HIGH_FAST);
}


void beeper_setup(void)
```

```
{
   BEEP_DeInit();
   BEEP_LSICalibrationConfig(128000);
   BEEP_Init(BEEP_FREQUENCY_2KHZ);
}
```

Explanation

As stated earlier beeper module is dependent on the AWU module and so we need to enable this module's peripheral clock:

```
CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, ENABLE);
```

We need to set the beeper port pin as an output pin:

```
GPIO_Init(GPIOD, GPIO_PIN_4, GPIO_MODE_OUT_PP_HIGH_FAST);
```

Configuring the beeper is straight. Just like other peripherals, we deinitialize it first and set both LSI and beep frequency. Optionally we can calibrate the LSI.

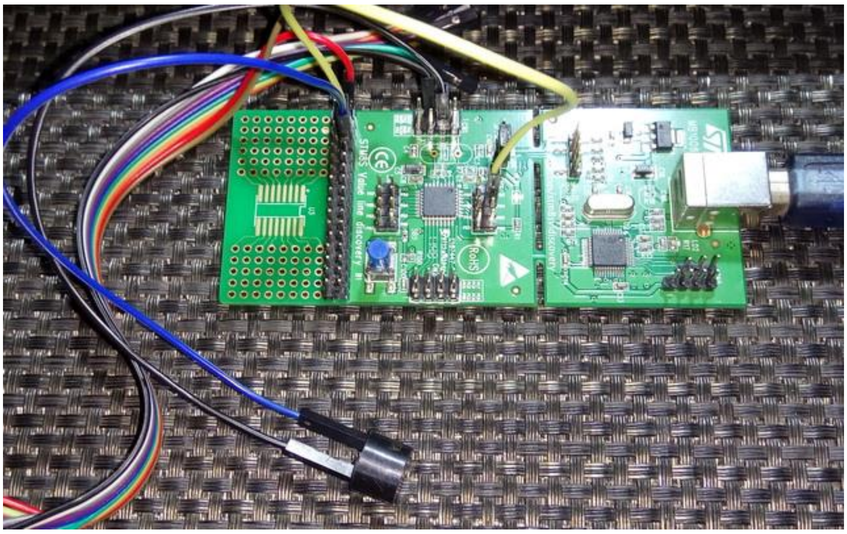
```
void beeper_setup(void)
{
   BEEP_DeInit();
   BEEP_LSICalibrationConfig(128000);
   BEEP_Init(BEEP_FREQUENCY_2KHZ);
}
```

To activate/deactivate the beeper we need to use the following instructions:

```
BEEP_Cmd(ENABLE);
```

```
BEEP_Cmd(DISABLE);
```

Demo



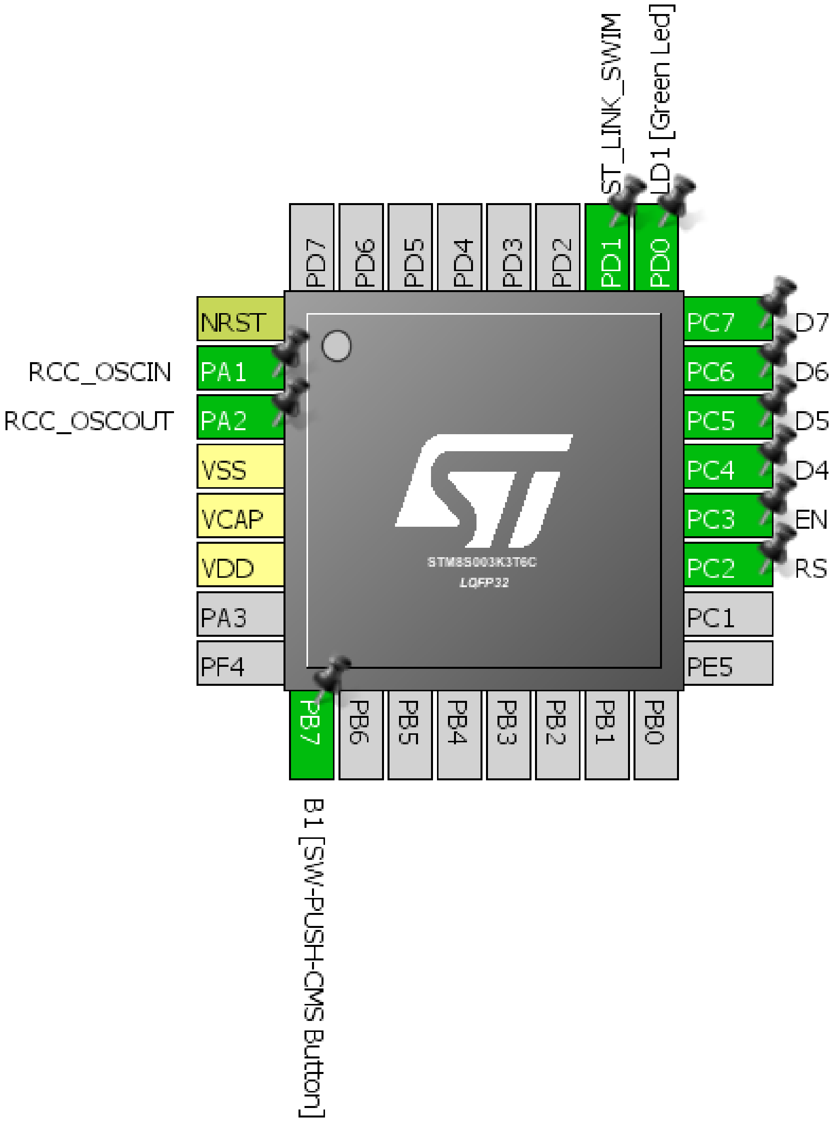
Video link: https://www.youtube.com/watch?v=LDPtULsJao8

Alphanumerical LCD

Alphanumerical displays are the most common and basic form of displays after seven segments and LED displays. They are useful for projecting multiple data quickly in ways that are otherwise difficult with other kinds of displays.

To interface a LCD with a STM8 micro, we need LCD library. The STM8 SPL does not have a library for such displays and we need to code it on our own. Interfacing a LCDs are not difficult tasks as no special hardware is required to drive LCDs other than GPIOs. However, there are some tasks needed in the software end. We need to include the library files. The process of library inclusion is discussed in the later part of this article as it needs some special attentions. The example I’m sharing here uses 6 GPIO pins from GPIOC. The read-write (R/W) pin of the LCD is connected to ground. The layout is as shown below.

Hardware Connection



Code Example

lcd.h

```
#include "stm8s.h"


#define LCD_PORT                                GPIOD

#define LCD_RS                                  GPIO_PIN_2
#define LCD_EN                                  GPIO_PIN_3
#define LCD_DB4                                 GPIO_PIN_4
#define LCD_DB5                                 GPIO_PIN_5
#define LCD_DB6                                 GPIO_PIN_6
#define LCD_DB7                                 GPIO_PIN_7

#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_GPIO_init(void);
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);
void toggle_io(unsigned char lcd_data, unsigned char bit_pos, unsigned char pin_num);
```

lcd.c

```
#include "lcd.h"


void LCD_GPIO_init(void)
{
    GPIO_Init(LCD_PORT, LCD_RS, GPIO_MODE_OUT_PP_HIGH_FAST);
    GPIO_Init(LCD_PORT, LCD_EN, GPIO_MODE_OUT_PP_HIGH_FAST);
    GPIO_Init(LCD_PORT, LCD_DB4, GPIO_MODE_OUT_PP_HIGH_FAST);
    GPIO_Init(LCD_PORT, LCD_DB5, GPIO_MODE_OUT_PP_HIGH_FAST);
    GPIO_Init(LCD_PORT, LCD_DB6, GPIO_MODE_OUT_PP_HIGH_FAST);
    GPIO_Init(LCD_PORT, LCD_DB7, GPIO_MODE_OUT_PP_HIGH_FAST);
    delay_ms(10);
}


void LCD_init(void)
{
     LCD_GPIO_init();
    toggle_EN_pin();

    GPIO_WriteLow(LCD_PORT, LCD_RS);
    GPIO_WriteLow(LCD_PORT, LCD_DB7);
    GPIO_WriteLow(LCD_PORT, LCD_DB6);
    GPIO_WriteHigh(LCD_PORT, LCD_DB5);
    GPIO_WriteHigh(LCD_PORT, LCD_DB4);
    toggle_EN_pin();

    GPIO_WriteLow(LCD_PORT, LCD_DB7);
    GPIO_WriteLow(LCD_PORT, LCD_DB6);
    GPIO_WriteHigh(LCD_PORT, LCD_DB5);
    GPIO_WriteHigh(LCD_PORT, LCD_DB4);
    toggle_EN_pin();

    GPIO_WriteLow(LCD_PORT, LCD_DB7);
    GPIO_WriteLow(LCD_PORT, LCD_DB6);
    GPIO_WriteHigh(LCD_PORT, LCD_DB5);
    GPIO_WriteHigh(LCD_PORT, LCD_DB4);
    toggle_EN_pin();

    GPIO_WriteLow(LCD_PORT, LCD_DB7);
    GPIO_WriteLow(LCD_PORT, LCD_DB6);
    GPIO_WriteHigh(LCD_PORT, LCD_DB5);
    GPIO_WriteLow(LCD_PORT, LCD_DB4);
    toggle_EN_pin();

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


void LCD_send(unsigned char value, unsigned char mode)
{
    switch(mode)
    {
        case DAT:
        {
            GPIO_WriteHigh(LCD_PORT, LCD_RS);
            break;
        }
        case CMD:
        {
             GPIO_WriteLow(LCD_PORT, LCD_RS);
```

```
                break;
            }
        }

        LCD_4bit_send(value);
}


void LCD_4bit_send(unsigned char lcd_data)
{
    toggle_io(lcd_data, 7, LCD_DB7);
    toggle_io(lcd_data, 6, LCD_DB6);
    toggle_io(lcd_data, 5, LCD_DB5);
    toggle_io(lcd_data, 4, LCD_DB4);
    toggle_EN_pin();
    toggle_io(lcd_data, 3, LCD_DB7);
    toggle_io(lcd_data, 2, LCD_DB6);
    toggle_io(lcd_data, 1, LCD_DB5);
    toggle_io(lcd_data, 0, LCD_DB4);
    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)
{
    GPIO_WriteHigh(LCD_PORT, LCD_EN);
    delay_ms(2);
    GPIO_WriteLow(LCD_PORT,LCD_EN);
}


void toggle_io(unsigned char lcd_data, unsigned char bit_pos, unsigned char pin_num)
```

```
{
    bool temp = FALSE;

    temp = (0x01 & (lcd_data >> bit_pos));

    switch(temp)
    {
        case TRUE:
        {
            GPIO_WriteHigh(LCD_PORT, pin_num);
            break;
        }

        default:
        {
            GPIO_WriteLow(LCD_PORT, pin_num);
            break;
         }
      }
}
```

main.c

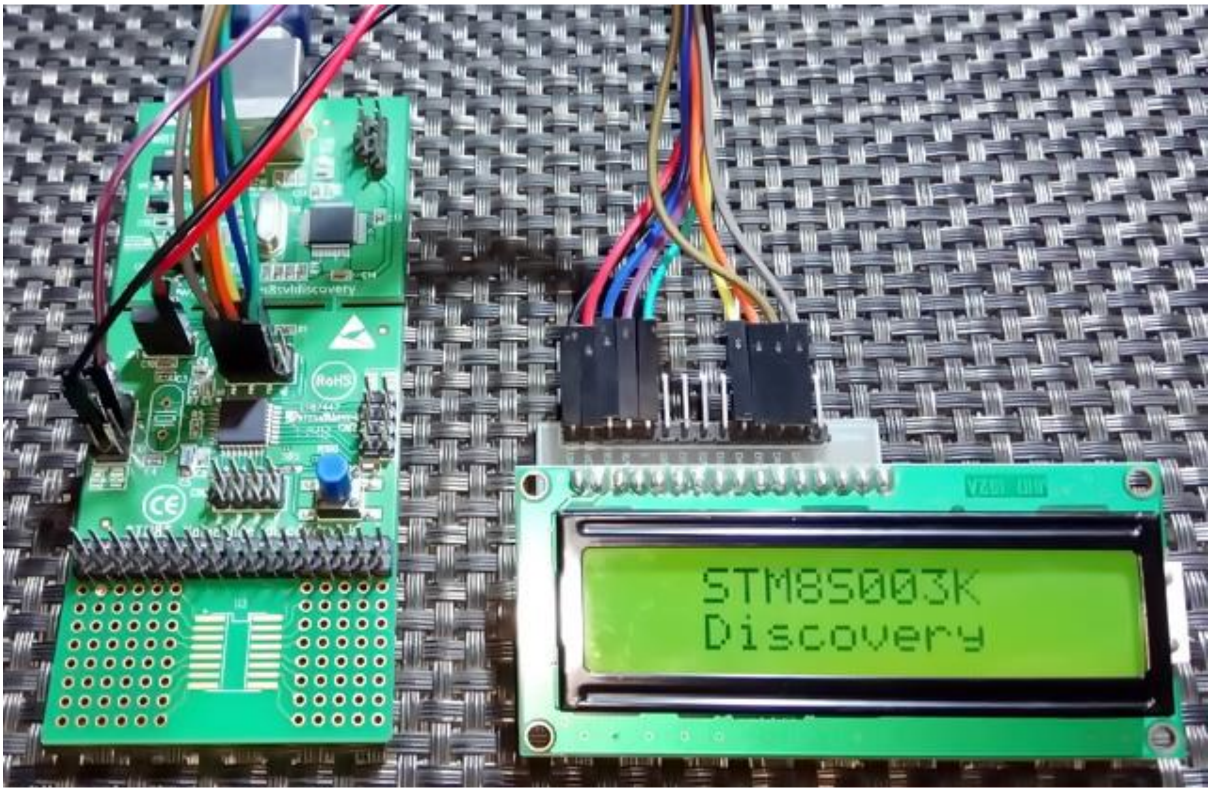
```
#include "STM8S.h"
#include "lcd.h"


void clock_setup(void);
void GPIO_setup(void);


void main(void)
{
    const char txt1[] = {"MICROARENA"};
    const char txt2[] = {"SShahryiar"};
    const char txt3[] = {"STM8S003K"};
    const char txt4[] = {"Discovery"};

    unsigned char s = 0x00;

    clock_setup();
    GPIO_setup();

    LCD_init();
    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 < 9; s++)
    {
        LCD_goto((3 + s), 0);
        LCD_putchar(txt3[s]);
        delay_ms(90);
    }
   for(s = 0; s < 9; s++)
   {
        LCD_goto((3 + s), 1);
        LCD_putchar(txt4[s]);
        delay_ms(90);
    }

    while (TRUE);
}


void clock_setup(void)
{
    CLK_DeInit();

    CLK_HSECmd(DISABLE);
    CLK_LSICmd(DISABLE);
    CLK_HSICmd(ENABLE);
    while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

    CLK_ClockSwitchCmd(ENABLE);
    CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
    CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);

    CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
    DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

    CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}


void GPIO_setup(void)
{
    GPIO_DeInit(LCD_PORT);
}
```

Explanation

There's little to explain this code as it involves GPIOs only. The codes for the LCD are coded using all available info on its datasheet, just initialization and working principle. One thing to note, however, is the CPU clock speed. If the CPU clock is too fast, LCDs may not work. This is because most LCDs have a maximum working frequency of 250kHz. It is best to keep this frequency below 200kHz.

```
CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);
```

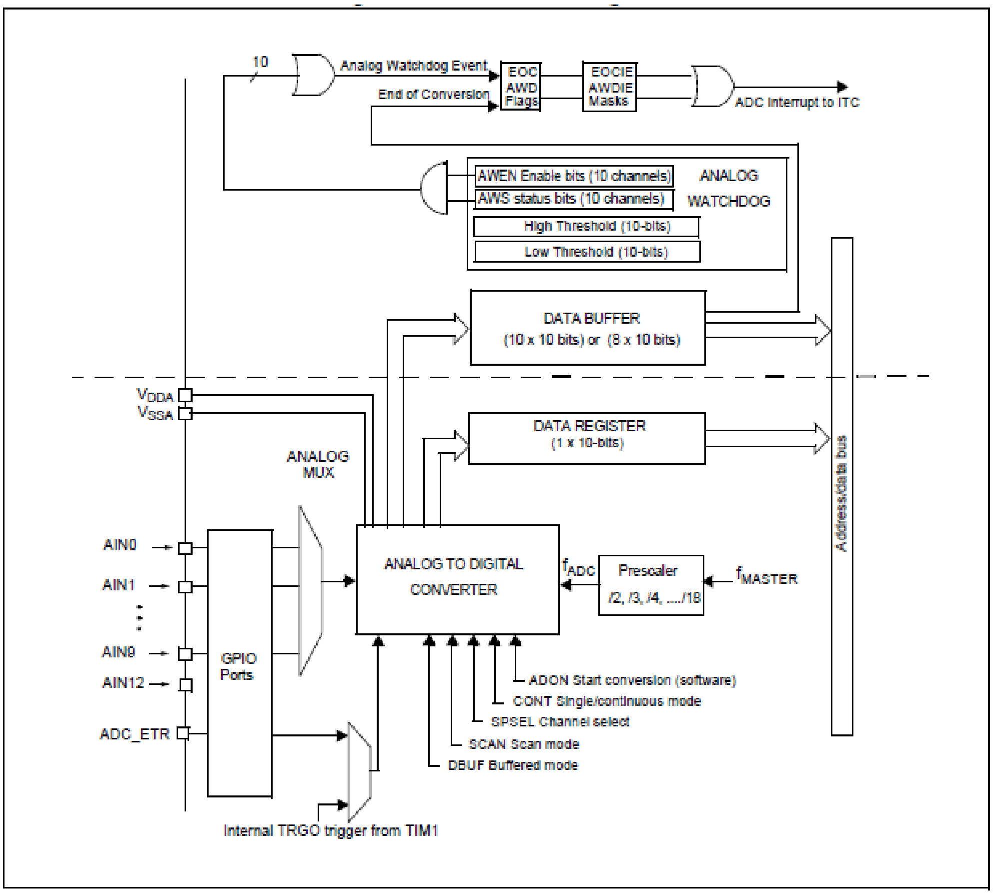
Demo



Video link: https://www.youtube.com/watch?v=TJg2Tuu4QaQ

Analog-to-Digital Converter (ADC)

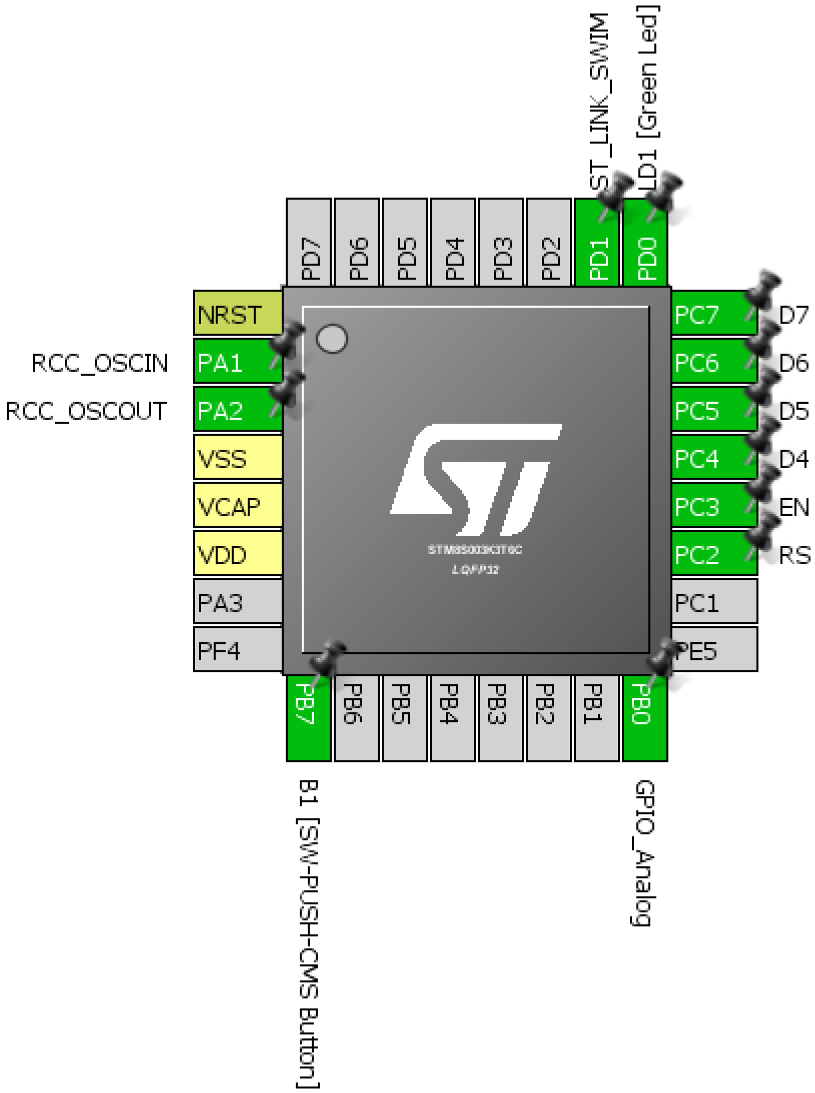
ADC is a very important peripheral in any modern-day microcontroller. It is used to read analogue outputs from sensors, sense voltage levels and so on. For example, we can use an ADC to read a LM35 temperature sensor. The voltage output from the sensor is proportional to temperature and so we can use the voltage info to back-calculate temperature. STM8S003K3 has four ADC channels associated with one ADC block. Other STM8 micros have more ADC channels and ADC blocks. The ADC of STM8 micros is just as same as the ADCs of other micros. There are a few additional features. Shown below is the block diagram of the STM8's ADC peripheral:



A few things must be noted before using the ADC. These enhance performance significantly:

- Input impedance should be less than 10kΩ.
- It is better to keep ADC clock within or less than 4MHz.
- Schmitt triggers must be disabled.
- Opamp-based input buffer and filter circuits are preferred if possible.
- If the ADC has reference source pins, they should be connected to a precision reference source like LM336. It is recommended to use a good LDO regulator chip otherwise.
- Unused ADC pins should not be configured or disabled. This will reduce power consumption.
- Rather taking single samples, ADC readings should be sampled at fixed regular intervals and averaged to get rid of minute fluctuations in readings.
- Right-justified data alignment should be used as it is most convenient to use.
- PCB/wire tracks leading to ADC channels must be short to reduce interference effects.

Hardware Connection



Code Example

```
#include "STM8S.h"


void clock_setup(void);
void GPIO_setup(void);
void ADC1_setup(void);
void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned int value);


void main()
{
        unsigned int A0 = 0x0000;

        clock_setup();
        GPIO_setup();
        ADC1_setup();
```

```
    LCD_init();
    LCD_clear_home();

    LCD_goto(0, 0);
    LCD_putstr("STM8 ADC");
    LCD_goto(0, 1);
    LCD_putstr("A0");

    while(TRUE)
    {
        ADC1_StartConversion();
        while(ADC1_GetFlagStatus(ADC1_FLAG_EOC) == FALSE);

        A0 = ADC1_GetConversionValue();
        ADC1_ClearFlag(ADC1_FLAG_EOC);

        lcd_print(4, 1, A0);
        delay_ms(90);
    };
}


void clock_setup(void)
{
    CLK_DeInit();

    CLK_HSECmd(DISABLE);
    CLK_LSICmd(DISABLE);
    CLK_HSICmd(ENABLE);
    while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

    CLK_ClockSwitchCmd(ENABLE);
    CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV2);
    CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV4);

    CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
    DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

    CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, ENABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}


void GPIO_setup(void)
{
    GPIO_DeInit(GPIOB);
    GPIO_Init(GPIOB, GPIO_PIN_0, GPIO_MODE_IN_FL_NO_IT);

    GPIO_DeInit(GPIOC);

    GPIO_DeInit(GPIOD);
    GPIO_Init(GPIOD, GPIO_PIN_3, GPIO_MODE_IN_PU_NO_IT);
}


void ADC1_setup(void)
{
    ADC1_DeInit();

    ADC1_Init(ADC1_CONVERSIONMODE_CONTINUOUS,
               ADC1_CHANNEL_0,
                          ADC1_PRESSEL_FCPU_D18,
                          ADC1_EXTTRIG_GPIO,
                          DISABLE,
                          ADC1_ALIGN_RIGHT,
                          ADC1_SCHMITTTRIG_CHANNEL0,
                          DISABLE);

             ADC1_Cmd(ENABLE);
}


void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned int value)
{
             char chr = 0x00;

             chr = ((value / 1000) + 0x30);
             LCD_goto(x_pos, y_pos);
             LCD_putchar(chr);

             chr = (((value / 100) % 10) + 0x30);
             LCD_goto((x_pos + 1), y_pos);
             LCD_putchar(chr);

             chr = (((value / 10) % 10) + 0x30);
             LCD_goto((x_pos + 2), y_pos);
             LCD_putchar(chr);

             chr = ((value % 10) + 0x30);
             LCD_goto((x_pos + 3), y_pos);
             LCD_putchar(chr);
}
```

Explanation

First, we need to enable the peripheral clock of the ADC module:

```
CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
```

Secondly, we have to set out ADC pin as a floating GPIO with no interrupt capability:

```
GPIO_Init(GPIOB, GPIO_PIN_0, GPIO_MODE_IN_FL_NO_IT);
```

ADC setup needs a few info regarding the desired ADC channel:

```
void ADC1_setup(void)
{
             ADC1_DeInit();

             ADC1_Init(ADC1_CONVERSIONMODE_CONTINUOUS,
             ADC1_CHANNEL_0,
             ADC1_PRESSEL_FCPU_D18,
             ADC1_EXTTRIG_GPIO,
            DISABLE,
            ADC1_ALIGN_RIGHT,
            ADC1_SCHMITTTRIG_CHANNEL0,
            DISABLE);

             ADC1_Cmd(ENABLE);
}
```

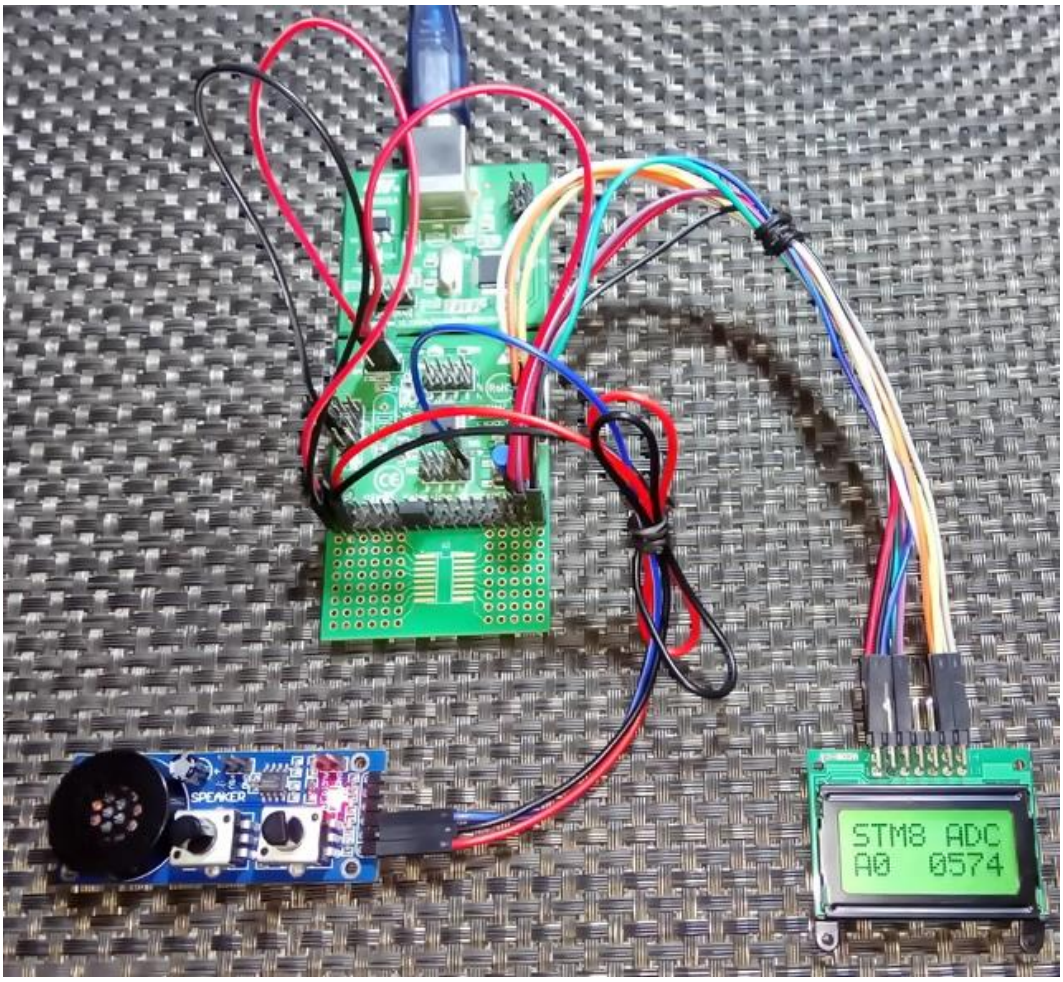
The second line of the above function states that we are going to use ADC channel 0 (PB0) with no Schmitt trigger. We are also not going to use external triggers from timer/GPIO modules. Since the master clock is running at 8MHz, the ADC prescaler divides the master/peripheral clock to get a sampling frequency of 444kHz. We are also going to use continuous conversion mode because we want to continually read the ADC input and don’t want to measure it in certain intervals. Lastly right-justified data alignment is used as it is easy to read from such.

In the main loop, we need to start ADC conversion and wait for the conversion to complete. We are not using interrupt methods and so we need to poll if ADC conversion has completed. At the end of conversion, we can read the ADC and clear ADC End of Conversion (EOC) flag.

```
ADC1_StartConversion();
while(ADC1_GetFlagStatus(ADC1_FLAG_EOC) == FALSE);

A0 = ADC1_GetConversionValue();
ADC1_ClearFlag(ADC1_FLAG_EOC);
```

The rest of the code is about printing the ADC data on a LCD.

Demo

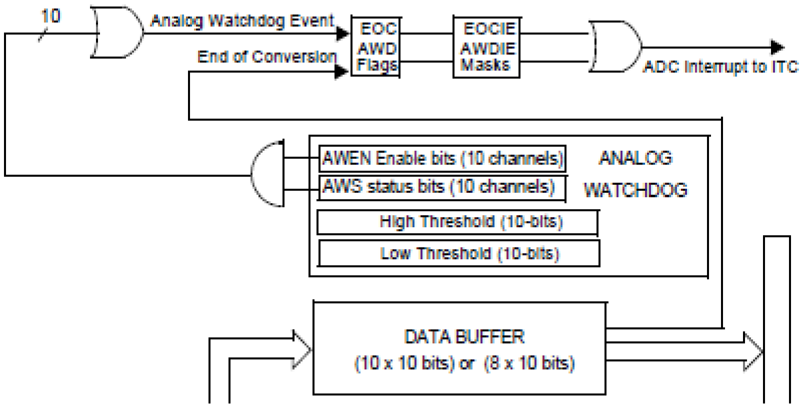


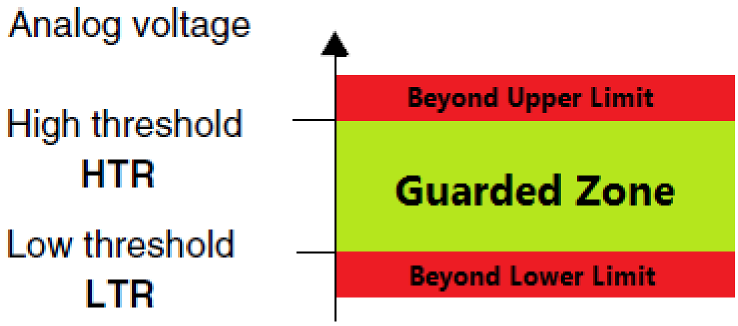
Video link: https://www.youtube.com/watch?v=rx68zPDEZUU

Analog Watchdog (AWD)

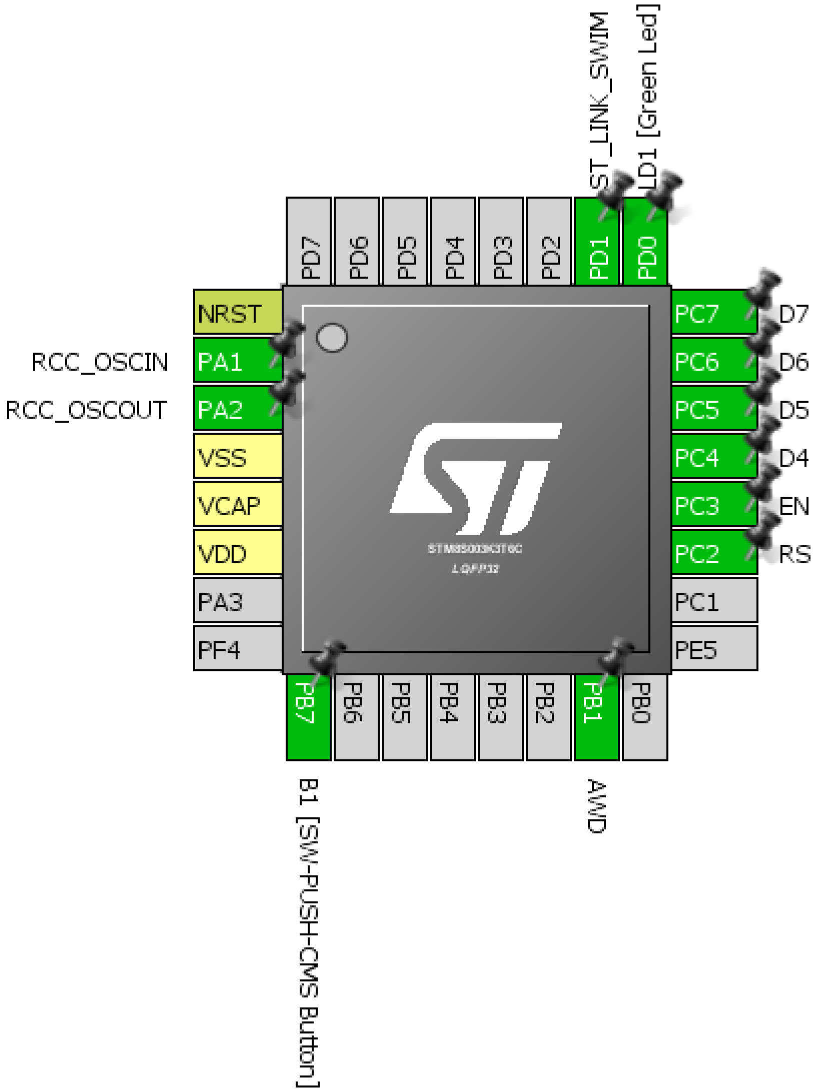
The AWD is one additional feature that most microcontrollers in the market do not have. AWD is more like a comparator but with the exception that we can set both the upper and lower limits of this comparator as per our requirement unlike fixed levels in other micros. The region between the upper and lower limits is called guarded zone. Beyond the boundaries of the guarded zone, the AWD unit kicks off.

The AWD unit is very useful in situations where we need to monitor the output of a sensor for example and take quick actions. For instance, consider a temperature controller. We would want the controller to turn on a heater should temperature fall below some level and turn it off when temperature rises to some high value without complex calculations and constant monitoring in our application firmware. In other microcontrollers, we would have accomplished this simple task using conditional ***IF-ELSE*** statements.





Hardware Connection



Code Example

```
#include "STM8S.h"


void clock_setup(void);
void GPIO_setup(void);
void ADC1_setup(void);
void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned int value);


void main()
{
        unsigned int a1 = 0x0000;

        clock_setup();
        GPIO_setup();
        ADC1_setup();

        LCD_init();
        LCD_clear_home();
```

```
    LCD_goto(0, 0);
    LCD_putstr("STM8 AWD");
    LCD_goto(0, 1);
    LCD_putstr("A1");

    while (TRUE)
    {
        ADC1_ClearFlag(ADC1_FLAG_EOC);

        ADC1_StartConversion();
        while(ADC1_GetFlagStatus(ADC1_FLAG_EOC) == 0);

        a1 = ADC1_GetConversionValue();
        lcd_print(4, 1, a1);

        if(ADC1_GetFlagStatus(ADC1_FLAG_AWD))
        {
            GPIO_WriteReverse(GPIOD, GPIO_PIN_0);
            ADC1_ClearFlag(ADC1_FLAG_AWD);
        }
        else
        {
            GPIO_WriteHigh(GPIOD, GPIO_PIN_0);
        }

        delay_ms(90);
    };
}


void clock_setup(void)
{
    CLK_DeInit();

    CLK_HSECmd(DISABLE);
    CLK_LSICmd(DISABLE);
    CLK_HSICmd(ENABLE);
    while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

    CLK_ClockSwitchCmd(ENABLE);
    CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV2);
    CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV4);

    CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
    DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

    CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, ENABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}


void GPIO_setup(void)
{
    GPIO_DeInit(GPIOB);
    GPIO_Init(GPIOB, GPIO_PIN_1, GPIO_MODE_IN_FL_NO_IT);

    GPIO_DeInit(GPIOD);
    GPIO_Init(GPIOD, GPIO_PIN_0, GPIO_MODE_OUT_PP_HIGH_FAST);
    GPIO_Init(GPIOD, GPIO_PIN_3, GPIO_MODE_IN_PU_NO_IT);
}
```

```
void ADC1_setup(void)
{
        ADC1_DeInit();
        ADC1_Init(ADC1_CONVERSIONMODE_SINGLE,
                                  ADC1_CHANNEL_1,
                                  ADC1_PRESSEL_FCPU_D10,
                                  ADC1_EXTTRIG_GPIO,
                                  DISABLE,
                                  ADC1_ALIGN_RIGHT,
                                  ADC1_SCHMITTTRIG_CHANNEL1,
                                  DISABLE);

        ADC1_AWDChannelConfig(ADC1_CHANNEL_1, ENABLE);
        ADC1_SetHighThreshold(600);
        ADC1_SetLowThreshold(200);

        ADC1_Cmd(ENABLE);
}


void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned int value)
{
        char chr = 0x00;

        chr = ((value / 1000) + 0x30);
        LCD_goto(x_pos, y_pos);
        LCD_putchar(chr);

        chr = (((value / 100) % 10) + 0x30);
        LCD_goto((x_pos + 1), y_pos);
        LCD_putchar(chr);

        chr = (((value / 10) % 10) + 0x30);
        LCD_goto((x_pos + 2), y_pos);
        LCD_putchar(chr);

        chr = ((value % 10) + 0x30);
        LCD_goto((x_pos + 3), y_pos);
        LCD_putchar(chr);
}
```

Explanation

The code for the AWD example is just as the one demonstrated in the ADC example. However, this time the ADC channel is channel 1 (PB1). Setting up the AWD is simple. We just need to set the limits, specify which channel to be monitored and enable the AWD unit.

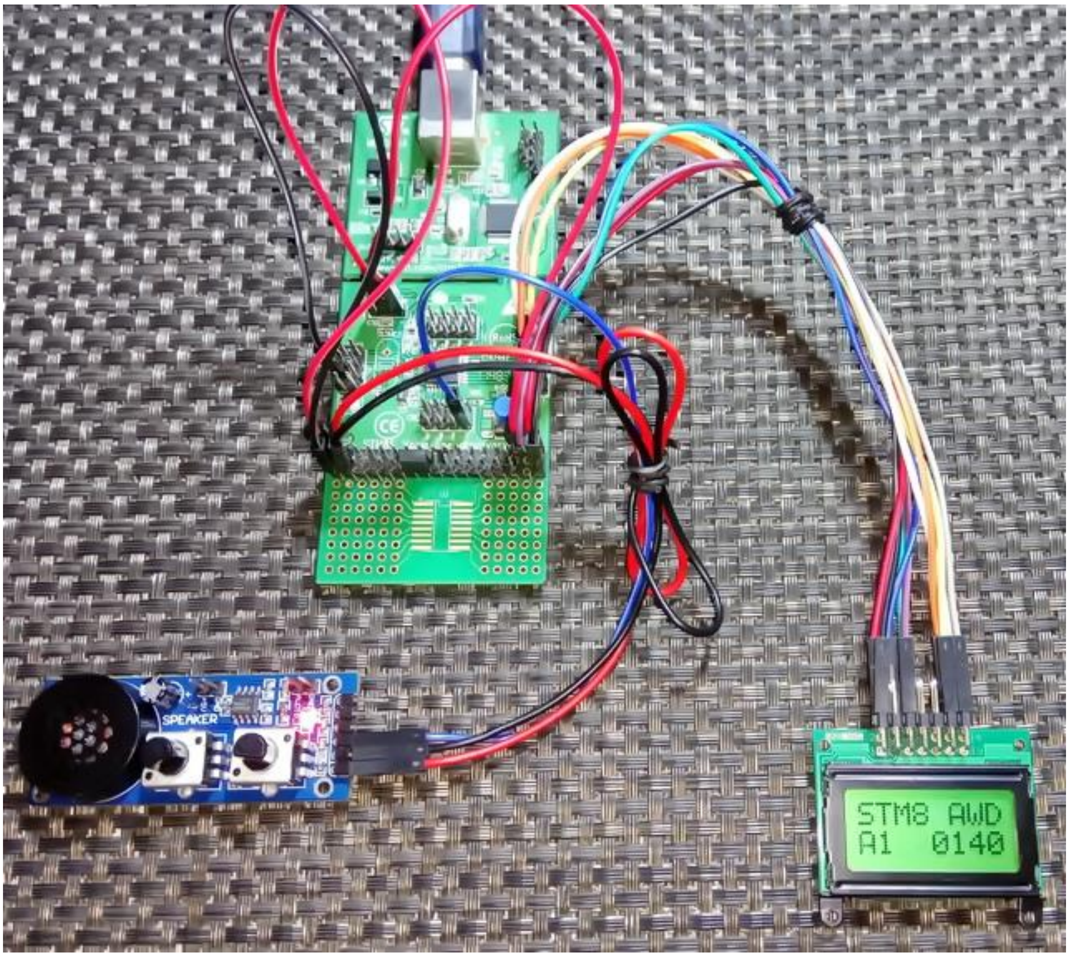
```
ADC1_AWDChannelConfig(ADC1_CHANNEL_1, ENABLE);
ADC1_SetHighThreshold(600);
ADC1_SetLowThreshold(200);
```

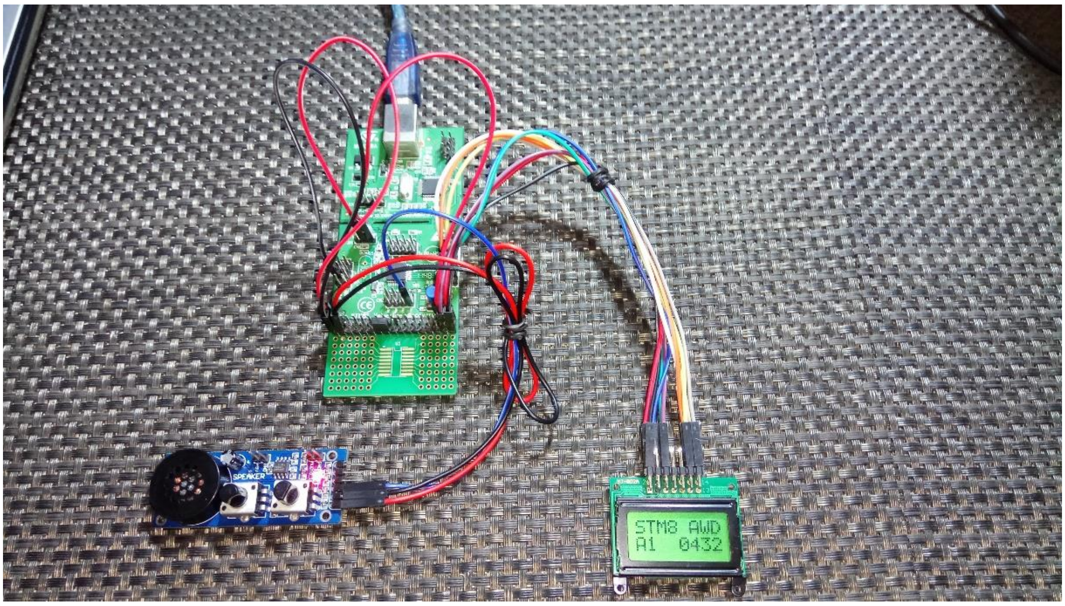
Here we have set 600 and 200 ADC counts as upper and lower limits respectively.

In the main function, we are simply polling AWD flag. If an AWD (beyond boundary zone) event on PB1 pin occurs the LED on PD0 starts flashing. If PB1 senses voltage between 200 and 600 ADC counts, the LED is turned off, indicating guarded zone.

```
if(ADC1_GetFlagStatus(ADC1_FLAG_AWD))
{
            GPIO_WriteReverse(GPIOD, GPIO_PIN_0);
            ADC1_ClearFlag(ADC1_FLAG_AWD);
}
else
{
            GPIO_WriteHigh(GPIOD, GPIO_PIN_0);
}
```

Demo

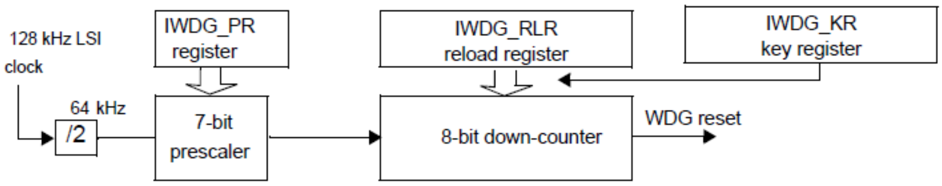


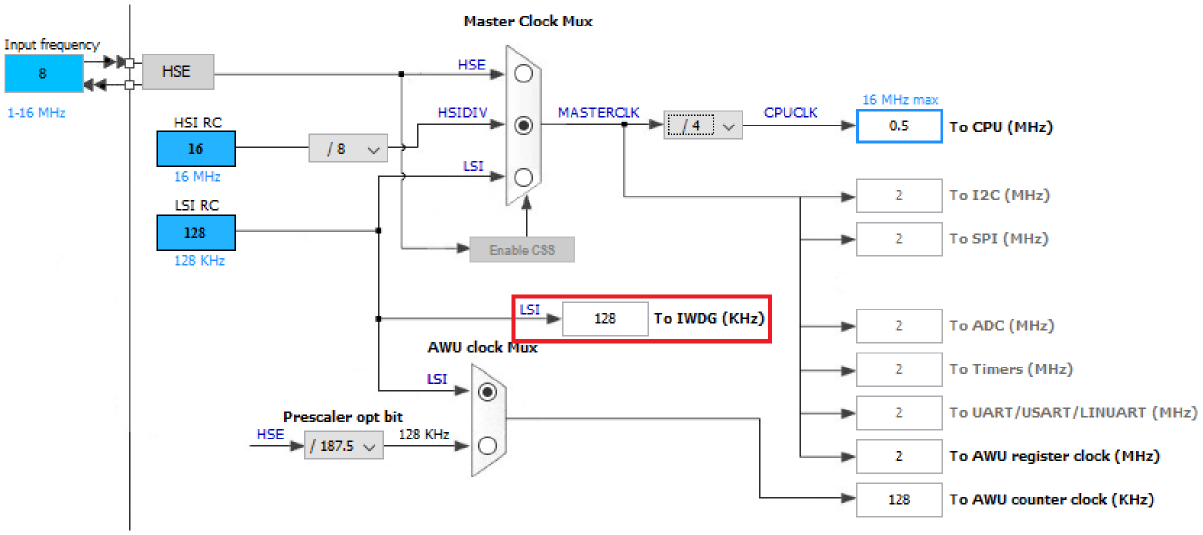


Video link: https://www.youtube.com/watch?v=bvVNuVpeFPk

Independent Watchdog (IWDG)

The IWDG is just the ordinary watchdog timer we usually find in any modern micro. The purpose of this timer is to recover a micro from an unanticipated event that may result in unresponsive or erratic behaviour. As the name suggests, this timer does not share anything with any other internal hardware peripheral and is clocked by LSI (128kHz) only. Thus, it is invulnerable to main clock (HSE or HSI) failure.





The IWDG works by decrementing a counter, counting time in the process. When the counter hits zero, a reset is issued. Usually we would want that this reset never occurs and so the counter is periodically updated in the application firmware. If for some reason, the counter is not refreshed, a reset will occur, recovering the MCU from a disastrous situation.

Configuring the IWDG is very easy with SPL. There are certain steps to follow but SPL manages them well internally. All we'll need is to configure the IWDG and reload it periodically before time runs out.

The formula required to calculate timeout is given below:

$$T = 2 \times T_{LSI} \times P \times R$$

where:

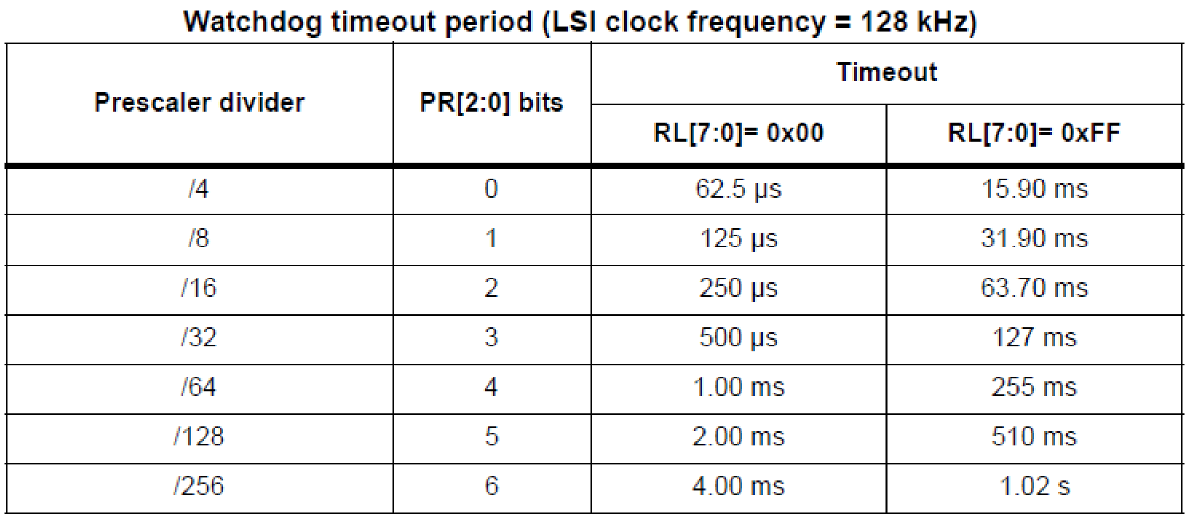
T = Timeout period

$T_{LSI} = 1/f_{LSI}$

$P = 2^{(PR[2:0] + 2)}$

R = RLR[7:0]+1

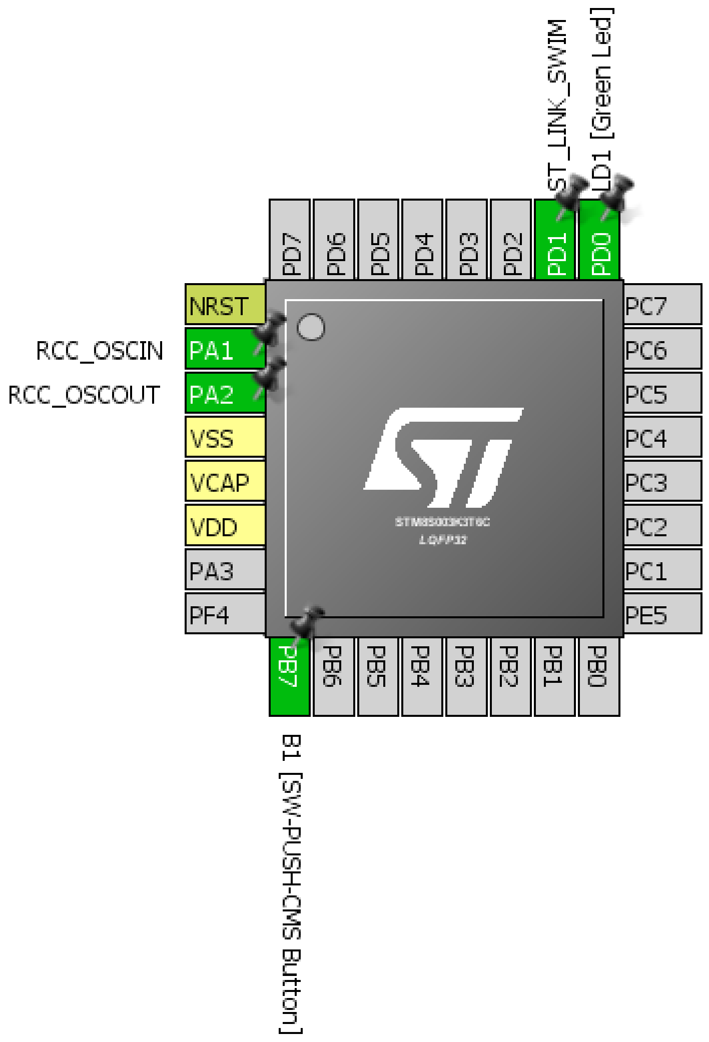
Typical values of timeout are as shown below:



Watchdog timeout period (LSI clock frequency = 128 kHz)

Prescaler divider	PR[2:0] bits	Timeout	
		RL[7:0]= 0x00	RL[7:0]= 0xFF
/4	0	62.5 µs	15.90 ms
/8	1	125 µs	31.90 ms
/16	2	250 µs	63.70 ms
/32	3	500 µs	127 ms
/64	4	1.00 ms	255 ms
/128	5	2.00 ms	510 ms
/256	6	4.00 ms	1.02 s

Hardware Connection



Code Example

```
#include "STM8S.h"


void clock_setup(void);
void GPIO_setup(void);
void IWDG_setup(void);


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

        clock_setup();
        GPIO_setup();

        GPIO_WriteLow(GPIOD, GPIO_PIN_0);
        for(t = 0; t < 60000; t++);

        IWDG_setup();

        while(TRUE)
        {
                GPIO_WriteReverse(GPIOD, GPIO_PIN_0);
                for(t = 0; t < 1000; t++)
                {
                        if(GPIO_ReadInputPin(GPIOB, GPIO_PIN_7) == FALSE)
                        {
                                IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
                                IWDG_ReloadCounter();
                                IWDG_WriteAccessCmd(IWDG_WriteAccess_Disable);
                        }
                }
        };
}


void clock_setup(void)
{
        CLK_DeInit();

        CLK_HSECmd(DISABLE);
        CLK_LSICmd(ENABLE);
        while(CLK_GetFlagStatus(CLK_FLAG_LSIRDY) == FALSE);
        CLK_HSICmd(ENABLE);
        while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

        CLK_ClockSwitchCmd(ENABLE);
        CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
        CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV4);

        CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
        DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

        CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}
```

```
void GPIO_setup(void)
{
            GPIO_DeInit(GPIOB);
            GPIO_DeInit(GPIOD);

            GPIO_Init(GPIOB, GPIO_PIN_7, GPIO_MODE_IN_PU_NO_IT);
            GPIO_Init(GPIOD, GPIO_PIN_0, GPIO_MODE_OUT_PP_LOW_FAST);
}


void IWDG_setup(void)
{
            IWDG_Enable();
            IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
            IWDG_SetPrescaler(IWDG_Prescaler_128);
            IWDG_SetReload(0x99);
            IWDG_WriteAccessCmd(IWDG_WriteAccess_Disable);
}
```

Explanation

In this example, we need not to look at peripheral and CPU clock as IWDG is not dependent on them. Still we can see that the CPU is running at 500kHz speed while the peripherals at 2MHz speed.

```
CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV4);
```

To setup the IWDG, we need to enable it first and then apply ***Write Access Protection*** key (0x55). We just need to set the prescaler and the counter value. The down counter will start from this value and count down to zero unless refreshed. In this example, the prescaler is set to 128 and reload value is set to 153 (0x99). Thus, with these we get a timeout of approximately 300ms. After entering these values we must prevent accidental changes in the firmware and so to do so the write access must be disabled.

```
void IWDG_setup(void)
{
            IWDG_Enable();
            IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
            IWDG_SetPrescaler(IWDG_Prescaler_128);
            IWDG_SetReload(0x99);
            IWDG_WriteAccessCmd(IWDG_WriteAccess_Disable);
}
```

Disco board's user button and LED are used for the demo. At the very beginning, the LED is lit for some time before the IWDG is configured, indicating the start of the firmware. In the main loop, the LED is toggled with some delay arranged by a ***for*** loop. Inside the loop, the button's state is polled. If the button is kept pressed it will always be in logic low state, reloading the IWDG counter. If its state changes to logic high and 300ms passes out, a reset is triggered.

```
GPIO_WriteReverse(GPIOD, GPIO_PIN_0);
for(t = 0; t < 1000; t++)
{
            if(GPIO_ReadInputPin(GPIOB, GPIO_PIN_7) == FALSE)
            {
                        IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
                        IWDG_ReloadCounter();
                        IWDG_WriteAccessCmd(IWDG_WriteAccess_Disable);
            }
}
```

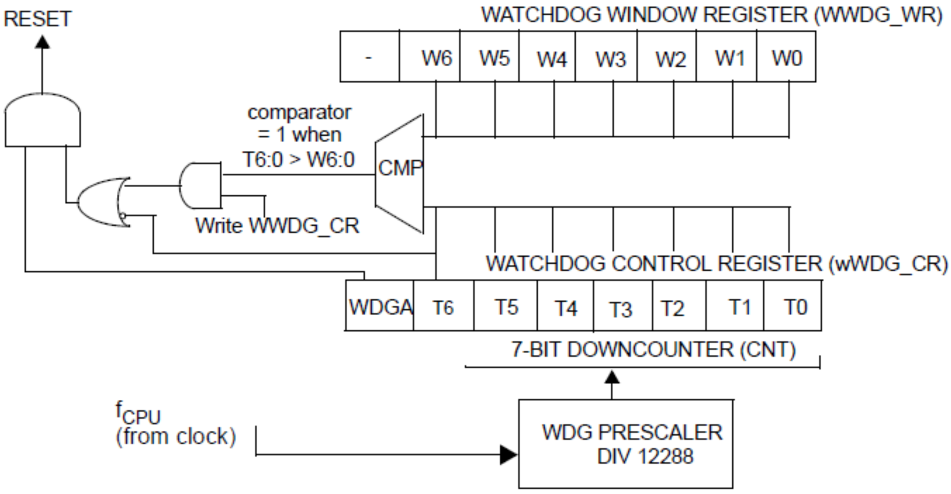
Note it is possible to calibrate LSI. It is however rarely needed.

Demo

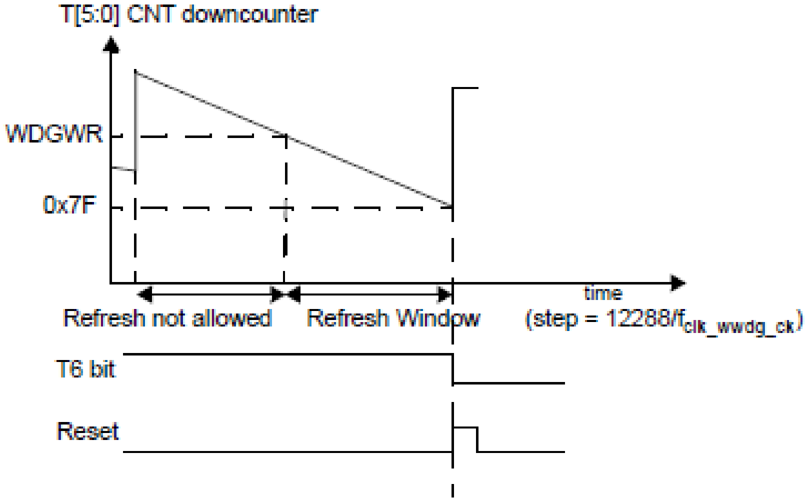
Video link: https://www.youtube.com/watch?v=05XKoy0ieHo

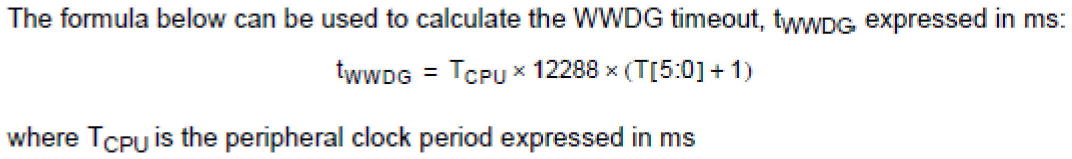
Window Watchdog (WWDG)

The WWDG is a bit more advanced watchdog timer. Unlike the IWDG, it will trigger a reset condition if its counter is reloaded earlier or later than a predefined time window. This kind of timer is usually found in high-end microcontrollers like ARMs, ATXMegas and recently released micros. Cool features like this and others make me feel that indeed STM8s are high-end affordable 8-bit alternatives when compared to traditional 8-bit MCUs.



The WWDG works by comparing a down counter against a window register. The counter can only be refreshed when its value is greater than 0x3F and less than window register value. If the counter is refreshed before the value set on window register or when the counter is less than or equal to 0x3F. If the counter hits the value 0x3F, reset automatically triggers. It is programmer's responsibility to refresh the counter at proper time. Note unlike IWDG, WWDG is not independent of main clock.



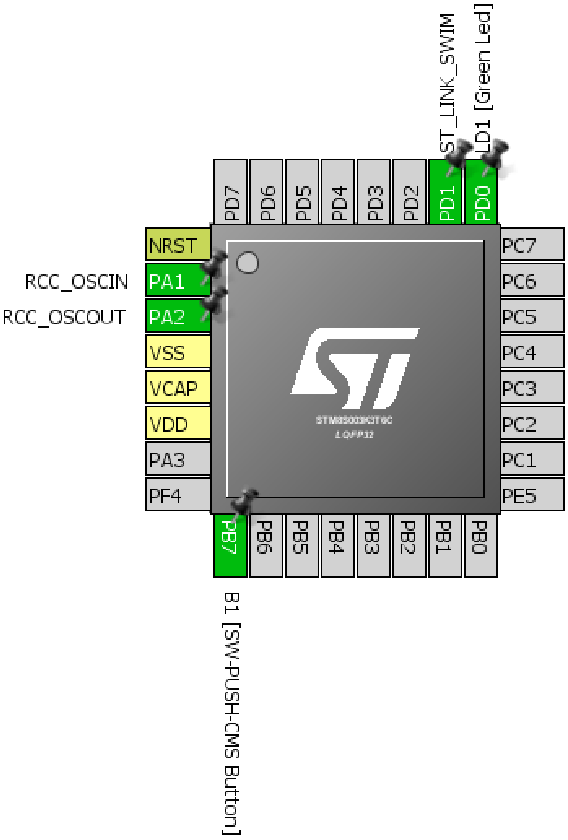


The formula below can be used to calculate the WWDG timeout, t_{WWDG} expressed in ms:

$$t_{WWDG} = T_{CPU} \times 12288 \times (T[5:0] + 1)$$

where T_{CPU} is the peripheral clock period expressed in ms

Hardware Connection



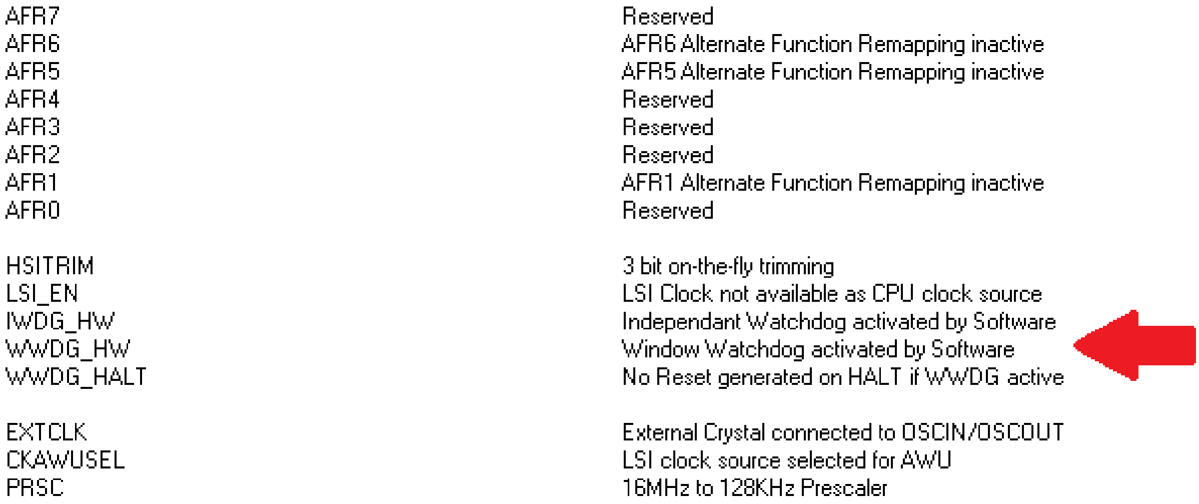
Code Example

The code example here demonstrates WWDG action. Simply Disco board's user LED and button are used. When the code starts executing, the LED starts blinking slowly, indicating the start of the code. When the code executes the main loop, the LED blinks rapidly to indicate main loop execution. If the button is pressed randomly the micro is reset because the counter is refreshed before the allowed time. Sometimes the micro may not reset because the counter may be in the allowed window – hence the name Window Watchdog.

```
#include "STM8S.h"


void clock_setup(void);
void GPIO_setup(void);
void WWDG_setup(void);


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

        clock_setup();
        GPIO_setup();

        for(i = 0x00; i < 0x04; i++)
        {
                GPIO_WriteReverse(GPIOD, GPIO_PIN_0);
                delay_ms(40);
        }

        WWDG_setup();

        while(TRUE)
        {
                if((GPIO_ReadInputPin(GPIOB, GPIO_PIN_7) == FALSE) || ((WWDG_GetCounter() > 0x60) &&
                (WWDG_GetCounter() < 0x7F)))
                {
                        WWDG_SetCounter(0x7F);
                }
                GPIO_WriteReverse(GPIOD, GPIO_PIN_0);
                delay_ms(20);
        };
}


void clock_setup(void)
{
        CLK_DeInit();

        CLK_HSECmd(DISABLE);
        CLK_LSICmd(DISABLE);
        CLK_HSICmd(ENABLE);
        while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

        CLK_ClockSwitchCmd(ENABLE);
        CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
        CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV64);

        CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
        DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

        CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}


void GPIO_setup(void)
{
        GPIO_DeInit(GPIOB);
        GPIO_Init(GPIOB, GPIO_PIN_7, GPIO_MODE_IN_PU_NO_IT);

        GPIO_DeInit(GPIOD);
        GPIO_Init(GPIOD, GPIO_PIN_0, GPIO_MODE_OUT_OD_HIZ_FAST);
}


void WWDG_setup(void)
{
        WWDG_Init(0x7F, 0x60);
}
```

Explanation

There's no way to enable watchdogs manually in software as they are always enabled. However, there are configuration bits to select if the IWDG and the WWDG are enabled in software or hardware. They only come in effect when configured. This is cool.



AFR7	Reserved
AFR6	AFR6 Alternate Function Remapping inactive
AFR5	AFR5 Alternate Function Remapping inactive
AFR4	Reserved
AFR3	Reserved
AFR2	Reserved
AFR1	AFR1 Alternate Function Remapping inactive
AFR0	Reserved
HSITRIM	3 bit on-the-fly trimming
LSI_EN	LSI Clock not available as CPU clock source
IWDG_HW	Independant Watchdog activated by Software
WWDG_HW	Window Watchdog activated by Software
WWDG_HALT	No Reset generated on HALT if WWDG active
EXTCLK	External Crystal connected to OSCIN/OSCOUT
CKAWUSEL	LSI clock source selected for AWU
PRSC	16MHz to 128KHz Prescaler

For WWDG, we just need to set the value of the down counter and the window register value only.

```
void WWDG_setup(void)
{
        WWDG_Init(0x7F, 0x60);
}
```

We need to monitor the WWDG in order to reload it when it is the right time window.

```
while(TRUE)
{
        if((GPIO_ReadInputPin(GPIOB, GPIO_PIN_7) == FALSE) || ((WWDG_GetCounter() > 0x60) &&    (WWDG_GetCounter() <
        0x7F)))
        {
                WWDG_SetCounter(0x7F);
        }
        GPIO_WriteReverse(GPIOD, GPIO_PIN_0);
        delay_ms(20);
};
```

Remember too early of too late will reset the micro.

Demo

Video link: https://www.youtube.com/watch?v=a_JWHJCh_-o

Timer Overview

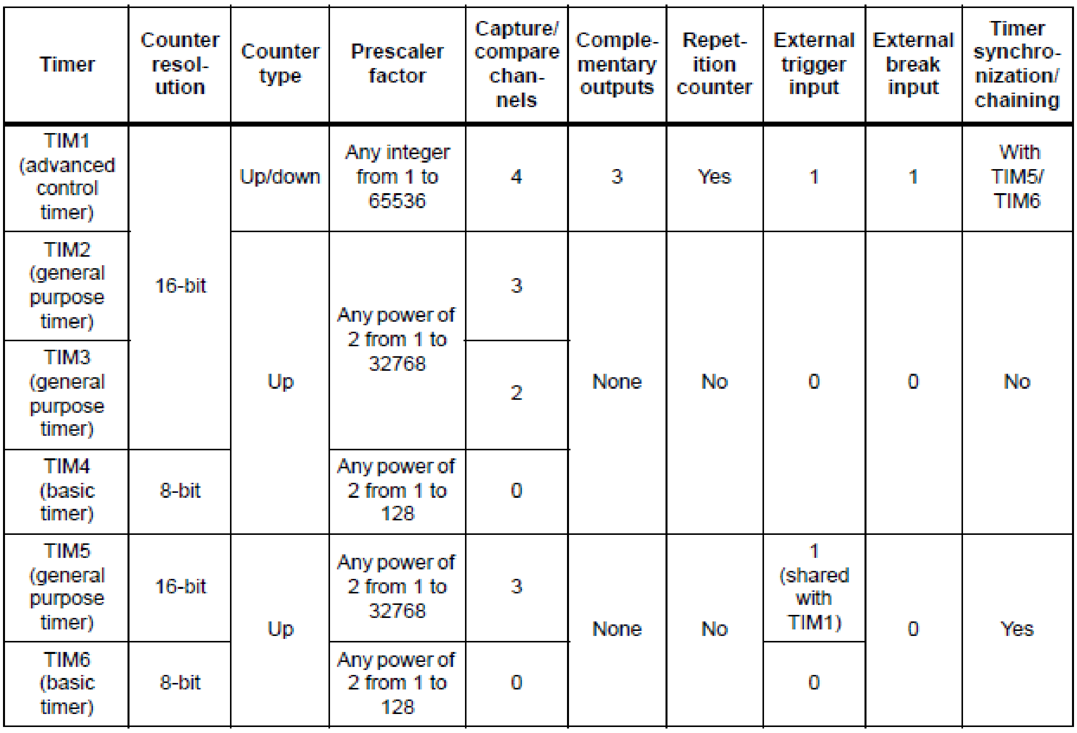
Timers are perhaps the most versatile piece of hardware in any micro. As their name tells, timers are useful for measurement of timed events like frequency, time, phase sequence, etc. and generate time-based events like PWM, waveform, etc.

Timers are also needed for touch sensing applications.

In any STM8 micro, there are three categories of timers. These are:

- Advanced Control Timer (TIM1)
- General Purpose Timers (TIM2, TIM3 & TIM5)
- Basic Timers (TIM4 & TIM6)

The basic working principle of all timers are same with some minor differences. Advanced timers are mainly intended for applications requiring specialized motor control, SMPSs, inverters, waveform generation, pulse width measurements, etc. Then there are general purpose timers that share almost all the features of advanced timer without the advanced features like brake, dead-time control, etc. Basic timers are all same as general purpose timers but lack PWM output/capture input pins and are intended mainly for time base generations. Here's the summary of all timers of STM8 micros:



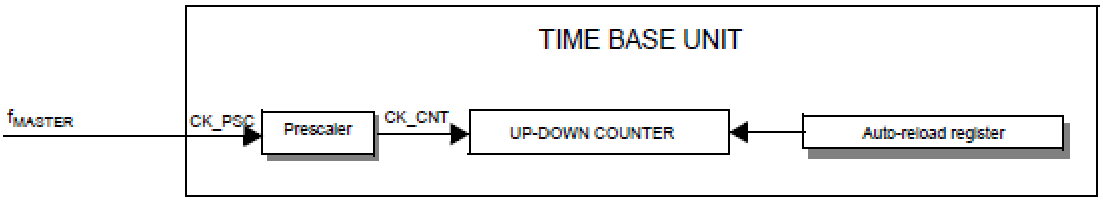
Timer	Counter resolution	Counter type	Prescaler factor	Capture/ compare channels	Complementary outputs	Repetition counter	External trigger input	External break input	Timer synchronization/ chaining
TIM1 (advanced control timer)	16-bit	Up/down	Any integer from 1 to 65536	4	3	Yes	1	1	With TIM5/ TIM6
TIM2 (general purpose timer)		Up	Any power of 2 from 1 to 32768	3	None	No	0	0	No
TIM3 (general purpose timer)				2					
TIM4 (basic timer)	8-bit		Any power of 2 from 1 to 128	0					
TIM5 (general purpose timer)	16-bit	Up	Any power of 2 from 1 to 32768	3	None	No	1 (shared with TIM1)	0	Yes
TIM6 (basic timer)	8-bit		Any power of 2 from 1 to 128	0			0		

Unlike the timers of other micros, STM8 timers have the more functionality that are otherwise only available in some special micros only. Timer cover a significant part of the reference manual. They are so elaborate that it is not possible to describe all of them in just one post. Therefore, here we'll be exploring the basics for now.

Time Base Generation (TIM2)

Time base generation is the most basic property of any timer and is also the most needed requirement in embedded systems. This mode can be used with or without interrupt. We'll first check the method firstly without interrupt and then with interrupt.

With time base generation, we can accurately time stuffs and events that are more precise than using delays, loops or other methods. Time base generation utilizes hardware timers and so work independently from other processes. It has many uses. For instance, with it we can avoid software delays, generate time slots of a Real-Time Operating System (RTOS) and many other tasks.



The time base unit for all timers of STM8 is all same. There are a few differences. For example, Timer 1 (TIM1) has a repetition counter. It is like a counter within another counter. Other timers lack this part. All timers can count up while advance timers can count down too.

The basic theory of time base generation is you have a peripheral clock which you would like to scale according to your need. Thus, you prescale it and use the new clock to run a counter. The counter will tick, incrementing count as time flies. It is just like counting from 0 to 100 and repeating from 0 again after reaching 100. Shown below is the generalized formula for finding an important event called timer reload:

$$Time\ Event\ (Timer\ Reload) = \frac{(Prescaler \times Reptiation\ Counter \times Counts)}{F_{master}}$$

This is the amount of time that will pass before timer overflow event occurs and the timer restarts from its initial count.

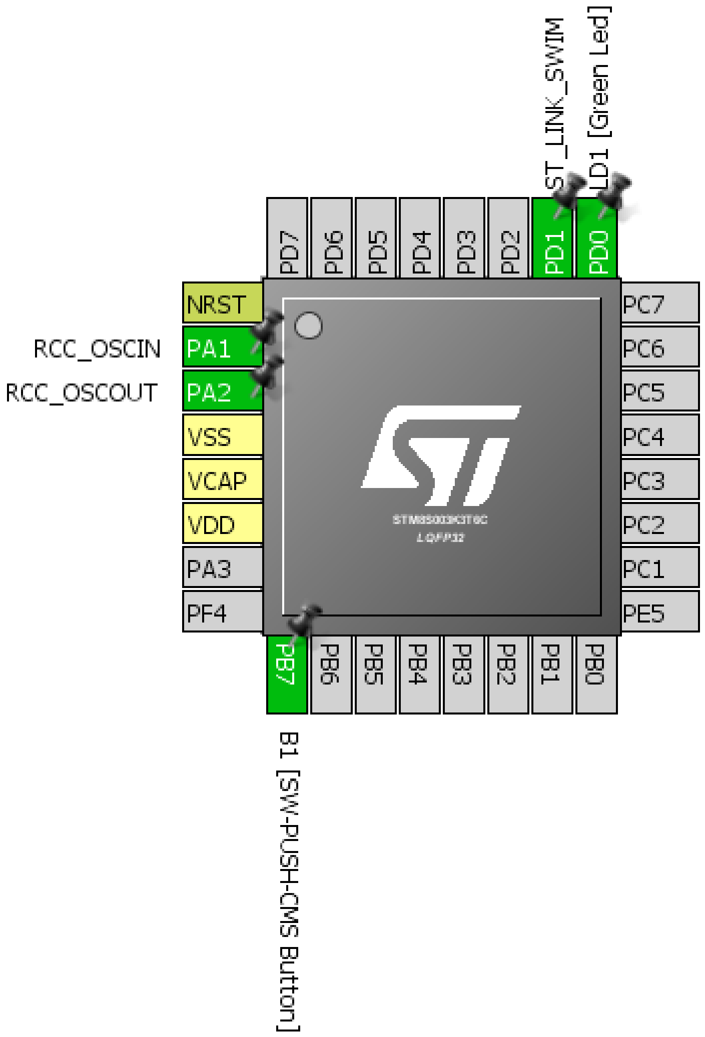
In my example, the peripheral or master clock is set to 2MHz. Thus, to make timer 2 (TIM2) reload after roughly 2 seconds, I have to prescale it by a factor of 2048 and load it with 1952 counts. Note that TIM2 doesn't have a repetition counter and so it is set to 1.

$$Time\ Event\ (Timer\ Reload) = \frac{(2048 \times 1952)}{2 \times 10^6}$$

$$= 1.998\text{s}$$

$$\approx 2\text{s}$$

Hardware Connection



Code Example

In this example, Disco board's user LED is turned-on and off without using any software delay. TIM2 is used to create time delay as such that the code is not stuck in a time-wasting loop.

```
#include "STM8S.h"


void clock_setup(void);
void GPIO_setup(void);
void TIM2_setup(void);


void main(void)
{
        clock_setup();
        GPIO_setup();
        TIM2_setup();

        while(TRUE)
        {
                if(TIM2_GetCounter() > 976)
                {
                        GPIO_WriteHigh(GPIOD, GPIO_PIN_0);
                }
                else
                {
                        GPIO_WriteLow(GPIOD, GPIO_PIN_0);
                }
        };
}


void clock_setup(void)
{
        CLK_DeInit();

        CLK_HSECmd(DISABLE);
        CLK_LSICmd(DISABLE);
        CLK_HSICmd(ENABLE);
        while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

        CLK_ClockSwitchCmd(ENABLE);
        CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
        CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);

        CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
        DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

        CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, ENABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}


void GPIO_setup(void)
{
        GPIO_DeInit(GPIOD);
        GPIO_Init(GPIOD, GPIO_PIN_0, GPIO_MODE_OUT_OD_HIZ_SLOW);
}


void TIM2_setup(void)
{
        TIM2_DeInit();
        TIM2_TimeBaseInit(TIM2_PRESCALER_2048, 1952);
        TIM2_Cmd(ENABLE);
}
```

Explanation

Firstly, the CPU and the peripheral clock is set at 2MHz.

```
CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);
….
….
CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, ENABLE);
```

As explained earlier, to get 2 second timer reload interval we need to prescale the timer by 2048 and load its counter with 1952. This is what should be the setup for TIM2:

```
void TIM2_setup(void)
{
            TIM2_DeInit();
            TIM2_TimeBaseInit(TIM2_PRESCALER_2048, 1952);
            TIM2_Cmd(ENABLE);
}
```

Our goal is to keep the LED on for 1 second and off for 1 second. It takes 1952 TIM2 counts for 2 second interval and so one second passes when this count is 976. Thus, in the main loop we are checking the value of TIM2's counter. From 0 to 976 counts, the LED is on and from 977 to 1952 counts, the LED is off. Note that the LED's positive end is connected to VDD and so it will turn on only PD0 is low.

```
if(TIM2_GetCounter() > 976)
{
     GPIO_WriteHigh(GPIOD, GPIO_PIN_0);
}
else
{
     GPIO_WriteLow(GPIOD, GPIO_PIN_0);
}
```

Demo

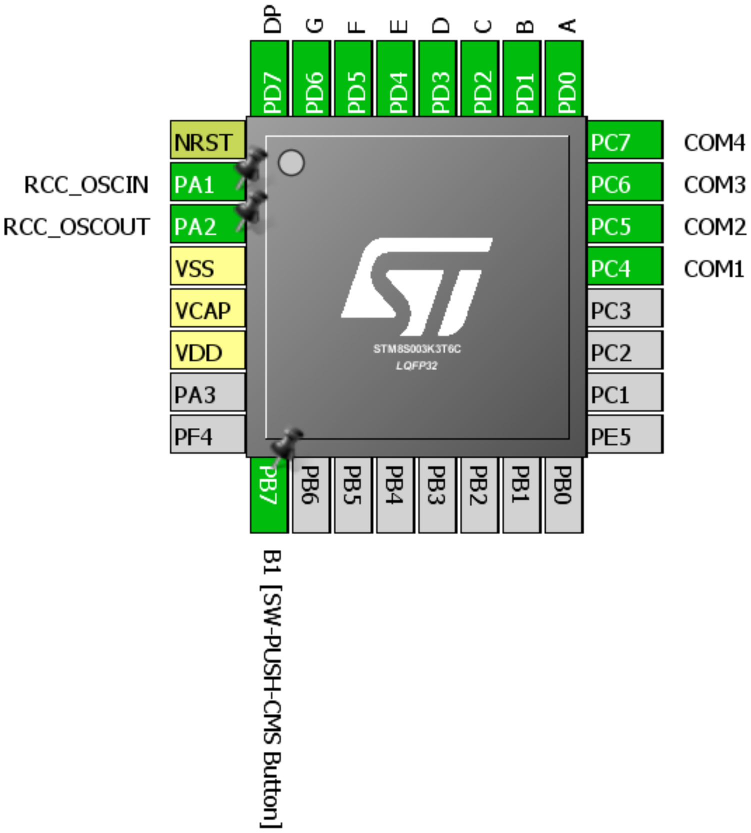
Video link: https://youtu.be/ZstHDHAAHOM

Timer Interrupt (TIM4)

In this example uses the same concepts of the previous example but it is based on timer interrupt – TIM4 interrupt. Timer interrupts are very important interrupts apart from other interrupts in a micro. To me they are highly valuable and useful.

This example demonstrates how to scan and project information on multiple seven segment displays with timer interrupt while the main loop can process the information to be displayed.

Hardware Connection



Code Example

main.c

```
#include "STM8S.h"


unsigned int value = 0x00;

unsigned char n = 0x00;
unsigned char seg = 0x01;
const unsigned char num[0x0A] = {0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8, 0x80, 0x90};
```

```
void GPIO_setup(void);
void clock_setup(void);
void TIM4_setup(void);


void main(void)
{
        GPIO_setup();
        clock_setup();
        TIM4_setup();

        while (TRUE)
        {
                value++;
                delay_ms(999);
        };
}


void GPIO_setup(void)
{
        GPIO_DeInit(GPIOC);
        GPIO_Init(GPIOC, ((GPIO_Pin_TypeDef)(GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7)),
        GPIO_MODE_OUT_PP_HIGH_FAST);

        GPIO_DeInit(GPIOD);
        GPIO_Init(GPIOD, GPIO_PIN_ALL, GPIO_MODE_OUT_PP_HIGH_FAST);
}


void clock_setup(void)
{
        CLK_DeInit();

        CLK_HSECmd(DISABLE);
        CLK_LSICmd(DISABLE);
        CLK_HSICmd(ENABLE);
        while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

        CLK_ClockSwitchCmd(ENABLE);
        CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
        CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);

        CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
        DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

        CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, ENABLE);
}


void TIM4_setup(void)
{
        TIM4_DeInit();
        TIM4_TimeBaseInit(TIM4_PRESCALER_32, 128);
        TIM4_ITConfig(TIM4_IT_UPDATE, ENABLE);
        TIM4_Cmd(ENABLE);
        enableInterrupts();
}
```

stm8s_it.h (top part only)

```
#ifndef __STM8S_IT_H
#define __STM8S_IT_H

@far @interrupt void TIM4_UPD_IRQHandler(void);



/* Includes ------------------------------------------------------------------*/

#include "stm8s.h"
```

stm8s_it.c (top part only)

```
#include "stm8s.h"
#include "stm8s_it.h"


extern unsigned int value;

extern unsigned char n;
extern unsigned char seg;
extern const unsigned char num[10];


void TIM4_UPD_IRQHandler(void)
{
        switch(seg)
        {
                case 1:
                {
                        n = (value / 1000);
                        GPIO_Write(GPIOD, num[n]);
                        GPIO_Write(GPIOC, 0xE0);
                        break;
                }

                case 2:
                {
                        n = ((value / 100) % 10);
                        GPIO_Write(GPIOD, num[n]);
                        GPIO_Write(GPIOC, 0xD0);
                        break;
                }

                case 3:
                {
                        n = ((value / 10) % 10);
                        GPIO_Write(GPIOD, num[n]);
                        GPIO_Write(GPIOC, 0xB0);
                        break;
                }

                case 4:
                {
                        n = (value % 10);
                        GPIO_Write(GPIOD, num[n]);
                        GPIO_Write(GPIOC, 0x70);
                        break;
                }
        }

        seg++;
        if(seg > 4)
        {
                seg = 1;
        }
        TIM4_ClearFlag(TIM4_FLAG_UPDATE);
}
```

stm8_interrupt_vector.c

```
#include "stm8s_it.h"

typedef void @far (*interrupt_handler_t)(void);

struct interrupt_vector {
        unsigned char interrupt_instruction;
        interrupt_handler_t interrupt_handler;
};

//@far @interrupt void NonHandledInterrupt (void)
//{
        /* in order to detect unexpected events during development,
           it is recommended to set a breakpoint on the following instruction
        */
        //return;
//}

extern void _stext();     /* startup routine */


struct interrupt_vector const _vectab[] = {
        {0x82, (interrupt_handler_t)_stext}, /* reset */
        {0x82, NonHandledInterrupt}, /* trap  */
        {0x82, NonHandledInterrupt}, /* irq0  */
        {0x82, NonHandledInterrupt}, /* irq1  */
        {0x82, NonHandledInterrupt}, /* irq2  */
        {0x82, NonHandledInterrupt}, /* irq3  */
        {0x82, NonHandledInterrupt}, /* irq4  */
        {0x82, NonHandledInterrupt}, /* irq5  */
        {0x82, NonHandledInterrupt}, /* irq6  */
        {0x82, NonHandledInterrupt}, /* irq7  */
        {0x82, NonHandledInterrupt}, /* irq8  */
        {0x82, NonHandledInterrupt}, /* irq9  */
        {0x82, NonHandledInterrupt}, /* irq10 */
        {0x82, NonHandledInterrupt}, /* irq11 */
        {0x82, NonHandledInterrupt}, /* irq12 */
        {0x82, NonHandledInterrupt}, /* irq13 */
        {0x82, NonHandledInterrupt}, /* irq14 */
        {0x82, NonHandledInterrupt}, /* irq15 */
        {0x82, NonHandledInterrupt}, /* irq16 */
        {0x82, NonHandledInterrupt}, /* irq17 */
        {0x82, NonHandledInterrupt}, /* irq18 */
        {0x82, NonHandledInterrupt}, /* irq19 */
        {0x82, NonHandledInterrupt}, /* irq20 */
        {0x82, NonHandledInterrupt}, /* irq21 */
        {0x82, NonHandledInterrupt}, /* irq22 */
        {0x82, (interrupt_handler_t)TIM4_UPD_IRQHandler}, /* irq23 */
        {0x82, NonHandledInterrupt}, /* irq24 */
        {0x82, NonHandledInterrupt}, /* irq25 */
        {0x82, NonHandledInterrupt}, /* irq26 */
        {0x82, NonHandledInterrupt}, /* irq27 */
        {0x82, NonHandledInterrupt}, /* irq28 */
        {0x82, NonHandledInterrupt}, /* irq29 */
};
```

Explanation

Both the peripheral and CPU clocks are running at 2MHz.

```
CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);
....
….
CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, ENABLE);
```

TIM4 is a basic timer and so it is better to use it for such tasks. We initialize it by setting its prescaler to 32 and loading its counter with 128. These values will make TIM4 overflow and interrupt every 2ms – enough time to project info in one seven segment. There are 4 seven segment displays and so within 8ms all four are updated and your eyes see it as if all projecting info at the same time – a trick of vision. Lastly, we need to enable what kind of interrupt we are expecting from the timer and finally enable the global interrupt.

```
void TIM4_setup(void)
{
        TIM4_DeInit();
        TIM4_TimeBaseInit(TIM4_PRESCALER_32, 128);
        TIM4_ITConfig(TIM4_IT_UPDATE, ENABLE);
        TIM4_Cmd(ENABLE);
        enableInterrupts();
}
```

Remember the first interrupt example? We have to let the compiler know which interrupt we are using. If you look at the datasheet, you'll see that TIM4 update/overflow is located in IRQ23. We need this and so we should make the following change in the ***stm8_interrupt_vector.c*** file:

```
{0x82, (interrupt_handler_t)TIM4_UPD_IRQHandler}, /* irq23 */
```

Remember to add the interrupt header and source files as we are going to use interrupt here. Inside the ISR, we do the scanning of each seven segment. Every time an overflow interrupt occurs, a seven segment is changed. At the end of the ISR a counter is incremented to select the next display when new overflow event occurs. Inside the ***Switch-Case***, we turn on the seven segment and decide the value that seven segment should show. Finally, the timer overflow/update flag is cleared.

```
switch(seg)
{
        case 1:
        {
                n = (value / 1000);
                GPIO_Write(GPIOD, num[n]);
                GPIO_Write(GPIOC, 0xE0);
                break;
        }

        case 2:
        {
                n = ((value / 100) % 10);
                GPIO_Write(GPIOD, num[n]);
                GPIO_Write(GPIOC, 0xD0);
                break;
        }

        case 3:
        {
```

```
                        n = ((value / 10) % 10);
                        GPIO_Write(GPIOD, num[n]);
                        GPIO_Write(GPIOC, 0xB0);
                        break;
            }

            case 4:
            {
                        n = (value % 10);
                        GPIO_Write(GPIOD, num[n]);
                        GPIO_Write(GPIOC, 0x70);
                        break;
            }
}

seg++;
if(seg > 4)
{
            seg = 1;
}
TIM4_ClearFlag(TIM4_FLAG_UPDATE);
```

Demo

Video link: https://youtu.be/Sa20Hf2N4gE

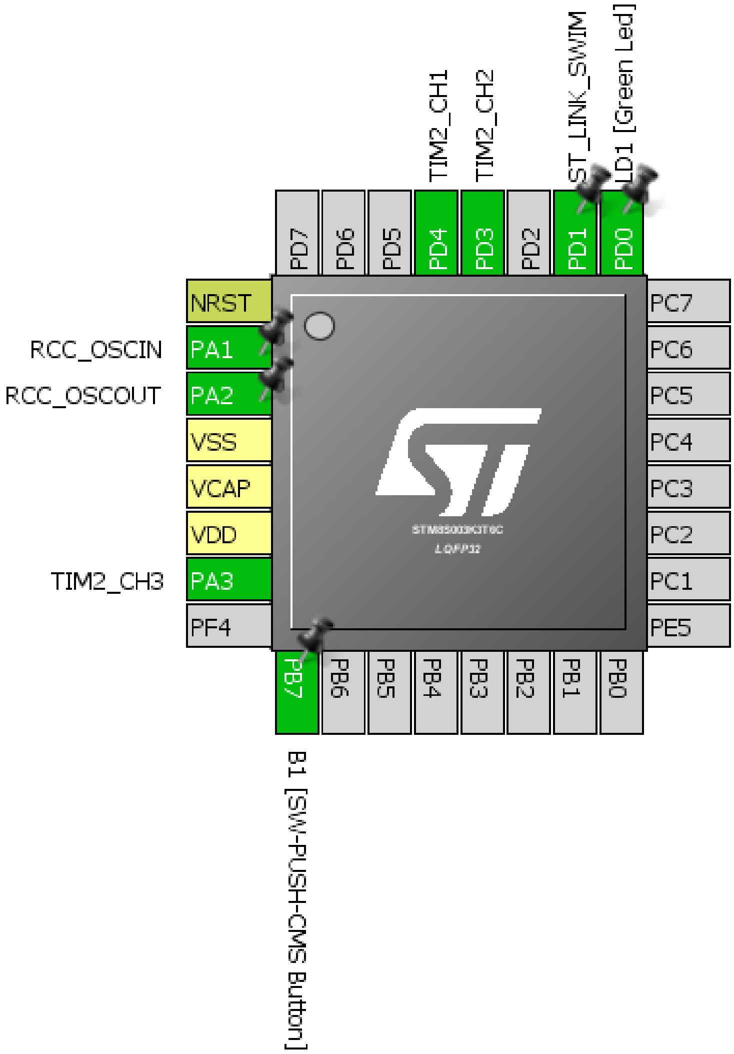
General Purpose Pulse Width Modulation (TIM2 PWM)

Pulse Width Modulation (PWM) is a must-have feature of any microcontroller. PWM has many uses like motor control, SMPSs, lighting control, sound generation, waveform generation, etc. Unlike other micros which have limited PWM channels, STM8 has several PWM channels. For instance, STM8S003K has seven independent PWM channels, three of which belong to TIM2 – a general purpose (GP) timer.

PWMs generated by GP timers are basic PWMs. They can be used for simple tasks like LED brightness control, servo motor control, etc. that don't require advanced features like dead-time, brake or complementary waveform generation. In this section, we will see how to use TIM2 to generate simple PWMs.

Please note that in more advanced STM8 micros, timer I/Os are dependent on alternate function configuration bits. Check those bits before uploading codes. In some STM8 micros, the I/Os are also remappable, meaning that the I/Os can be swapped in different GPIOs. Take the help of STM8CubeMx if needed.

Hardware Connection



Code Example

This is a pretty simple example. Here all three channels of TIM2 are used to smoothly fade and glow three LEDs connected to the timer channels.

```
#include "STM8S.h"


void clock_setup(void);
void GPIO_setup(void);
void TIM2_setup(void);


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

        clock_setup();
        GPIO_setup();
        TIM2_setup();

        while(TRUE)
        {
                for(pwm_duty = 0; pwm_duty < 1000; pwm_duty += 10)
                {
                        TIM2_SetCompare1(pwm_duty);
                        TIM2_SetCompare2(pwm_duty);
                        TIM2_SetCompare3(pwm_duty);
                        delay_ms(10);
                }
                for(pwm_duty = 1000; pwm_duty > 0; pwm_duty -= 10)
                {
                        TIM2_SetCompare1(pwm_duty);
                        TIM2_SetCompare2(pwm_duty);
                        TIM2_SetCompare3(pwm_duty);
                        delay_ms(10);
                }
        };
}


void clock_setup(void)
{
        CLK_DeInit();

        CLK_HSECmd(DISABLE);
        CLK_LSICmd(DISABLE);
        CLK_HSICmd(ENABLE);
        while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

        CLK_ClockSwitchCmd(ENABLE);
        CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
        CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);

        CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
        DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

        CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, ENABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}


void GPIO_setup(void)
{
        GPIO_DeInit(GPIOA);
        GPIO_Init(GPIOA, GPIO_PIN_3, GPIO_MODE_OUT_PP_HIGH_FAST);

        GPIO_DeInit(GPIOD);
        GPIO_Init(GPIOD, ((GPIO_Pin_TypeDef)GPIO_PIN_3 | GPIO_PIN_4), GPIO_MODE_OUT_PP_HIGH_FAST);
}


void TIM2_setup(void)
{
        TIM2_DeInit();
        TIM2_TimeBaseInit(TIM2_PRESCALER_32, 1000);
        TIM2_OC1Init(TIM2_OCMODE_PWM1, TIM2_OUTPUTSTATE_ENABLE, 1000, TIM2_OCPOLARITY_HIGH);
        TIM2_OC2Init(TIM2_OCMODE_PWM1, TIM2_OUTPUTSTATE_ENABLE, 1000, TIM2_OCPOLARITY_LOW);
        TIM2_OC3Init(TIM2_OCMODE_PWM1, TIM2_OUTPUTSTATE_ENABLE, 1000, TIM2_OCPOLARITY_HIGH);
        TIM2_Cmd(ENABLE);
}
```

Explanation

Again, the CPU and the peripheral clock is set at 2MHz.

```
CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);
….
….
CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, ENABLE);
```

Next, we need to configure the PWM GPIOs as outputs.

```
void GPIO_setup(void)
{
        GPIO_DeInit(GPIOA);
        GPIO_Init(GPIOA, GPIO_PIN_3, GPIO_MODE_OUT_PP_HIGH_FAST);

        GPIO_DeInit(GPIOD);
        GPIO_Init(GPIOD, ((GPIO_Pin_TypeDef)GPIO_PIN_3 | GPIO_PIN_4), GPIO_MODE_OUT_PP_HIGH_FAST);
}
```

Just like other microcontrollers, PWM generation involves a timer. Here as said TIM2 is that timer. We need to set time base first before actually configuring the PWM channels.

```
void TIM2_setup(void)
{
        TIM2_DeInit();
        TIM2_TimeBaseInit(TIM2_PRESCALER_32, 1000);
        TIM2_OC1Init(TIM2_OCMODE_PWM1, TIM2_OUTPUTSTATE_ENABLE, 1000, TIM2_OCPOLARITY_HIGH);
        TIM2_OC2Init(TIM2_OCMODE_PWM1, TIM2_OUTPUTSTATE_ENABLE, 1000, TIM2_OCPOLARITY_LOW);
        TIM2_OC3Init(TIM2_OCMODE_PWM1, TIM2_OUTPUTSTATE_ENABLE, 1000, TIM2_OCPOLARITY_HIGH);
        TIM2_Cmd(ENABLE);
}
```

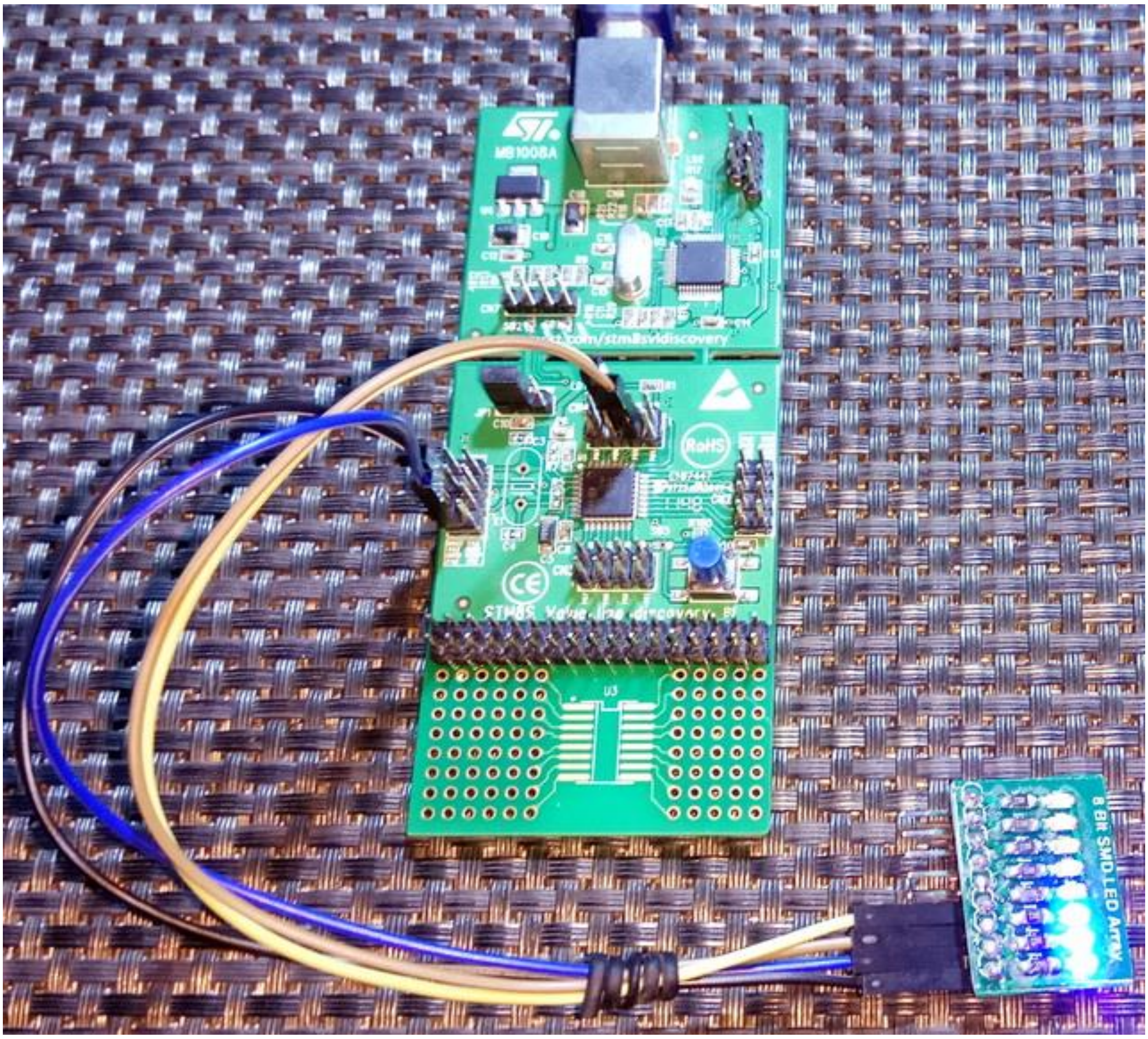
In the codes above, TIM2 has a time base of 16ms or 62.5kHz. This time base is further divided by the ***Output Compare (OC)*** unit. Thus, here the 62.5kHz base is further divided by 1000 to get 62.5Hz PWM frequency. The maximum duty cycle is therefore 1000. Additionally, we can set PWM polarity and command the channel if or if not should it behave in an inverted manner.

To change PWM duty, we need to call the following function:

```
TIM2_SetCompareX(pwm_duty);                    //   where X represents channel ID (1, 2 or 3)
```

Note that in STM8 micros, there is a trade-off between duty cycle and PWM frequency. If the PWM resolution, i.e. duty cycle is big then PWM frequency is small and vice-versa. This is true for all timers.

Demo



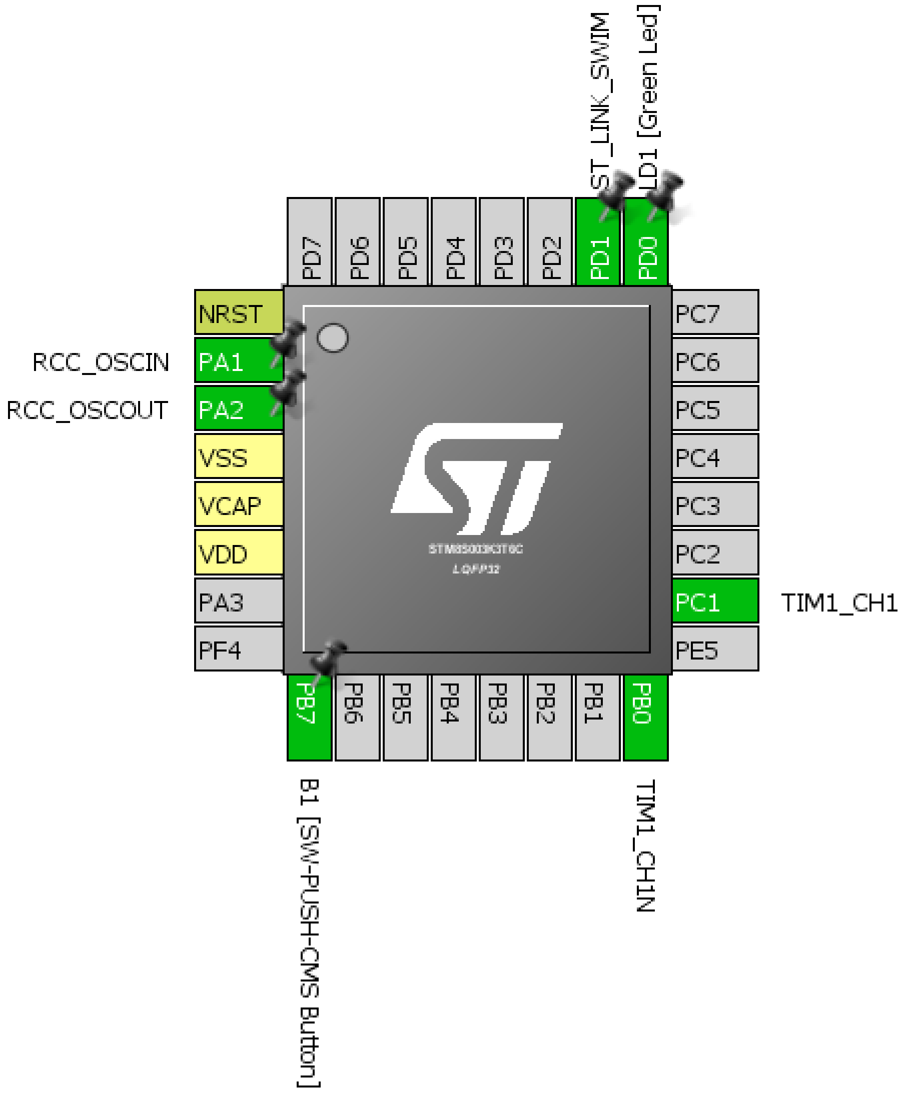
Video link: https://www.youtube.com/watch?v=BPS5unUHDz4

Advanced Pulse Width Modulation (TIM1 PWM)

Timer 1 (TIM1) is an advance timer and so the PWMs generated by it have several additional features that are not available with other timers. For example, it is possible to generate complimentary PWMs with TIM1. Up to three sets of complementary PWMs can be generated. Such PWMs are useful in designing three phase inverters, rectifiers and other power-related tasks. TIM1 PWMs are also very useful for motor control applications. It is also possible to add dead-time and brake. Apart from these TIM1 can also generate PWMs just like GP timers. In this mode however, complimentary PWM outputs are unavailable and up to four independent PWM channels can be made available.

In this example, I have demonstrated how to create complementary PWMs in TIM1 PWM channel 1.

Hardware Connection



Code Example

```
#include "STM8S.h"


void clock_setup(void);
void GPIO_setup(void);
void TIM1_setup(void);


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

        clock_setup();
        GPIO_setup();
        TIM1_setup();

        while(TRUE)
        {
                for(i = 0; i < 1000; i += 1)
                {
                        TIM1_SetCompare1(i);
                        delay_ms(1);
                }
                for(i = 1000; i > 0; i -= 1)
                {
                        TIM1_SetCompare1(i);
                        delay_ms(1);
                }
        };
}


void clock_setup(void)
{
        CLK_DeInit();

        CLK_HSECmd(DISABLE);
        CLK_LSICmd(DISABLE);
        CLK_HSICmd(ENABLE);
        while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

        CLK_ClockSwitchCmd(ENABLE);
        CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);
        CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);

        CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
        DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

        CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, ENABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}


void GPIO_setup(void)
{
        GPIO_DeInit(GPIOB);
        GPIO_Init(GPIOB, GPIO_PIN_0, GPIO_MODE_OUT_PP_HIGH_FAST);

        GPIO_DeInit(GPIOC);
        GPIO_Init(GPIOC, GPIO_PIN_1, GPIO_MODE_OUT_PP_HIGH_FAST);
}


void TIM1_setup(void)
{
        TIM1_DeInit();

        TIM1_TimeBaseInit(16, TIM1_COUNTERMODE_UP, 1000, 1);

        TIM1_OC1Init(TIM1_OCMODE_PWM1,
                     TIM1_OUTPUTSTATE_ENABLE,
                     TIM1_OUTPUTNSTATE_ENABLE,
                     1000,
                     TIM1_OCPOLARITY_LOW,
                     TIM1_OCNPOLARITY_LOW,
                     TIM1_OCIDLESTATE_RESET,
                     TIM1_OCNIDLESTATE_RESET);

        TIM1_CtrlPWMOutputs(ENABLE);
        TIM1_Cmd(ENABLE);
}
```

Explanation

This time we used the full 16MHz speed of HSI both for peripheral and CPU clocks:

```
CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);
CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);
….
….
CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, ENABLE);
```

Like as with the previous example PWM output GPIOs are set as outputs:

```
void GPIO_setup(void)
{
        GPIO_DeInit(GPIOB);
        GPIO_Init(GPIOB, GPIO_PIN_0, GPIO_MODE_OUT_PP_HIGH_FAST);

        GPIO_DeInit(GPIOC);
        GPIO_Init(GPIOC, GPIO_PIN_1, GPIO_MODE_OUT_PP_HIGH_FAST);
}
```

TIM1 and OC channel initialization is just like the previous example with some minor differences. The time base generation part seems to have some additional arguments. These are because:

- Unlike other timers, TIM1 prescaler value is not a fixed set of multiples of 2.
- The counting mode is not just up mode counting. Counting mode can also be down counting.
- TIM1 has additional repetition counter.
- Except the basic timers all timers in STM8 are 16-bit timer.

If you open the header file for TIM1, you'll see many functions. Many of these functions are not available with other timers, expressing the power of an advance timer.

Likewise, there are some additional info we must feed when configuring the OC channels. We need to set info about complementary channels even if we don't need them. We can additionally set the default idle states of PWMs apart from polarities.

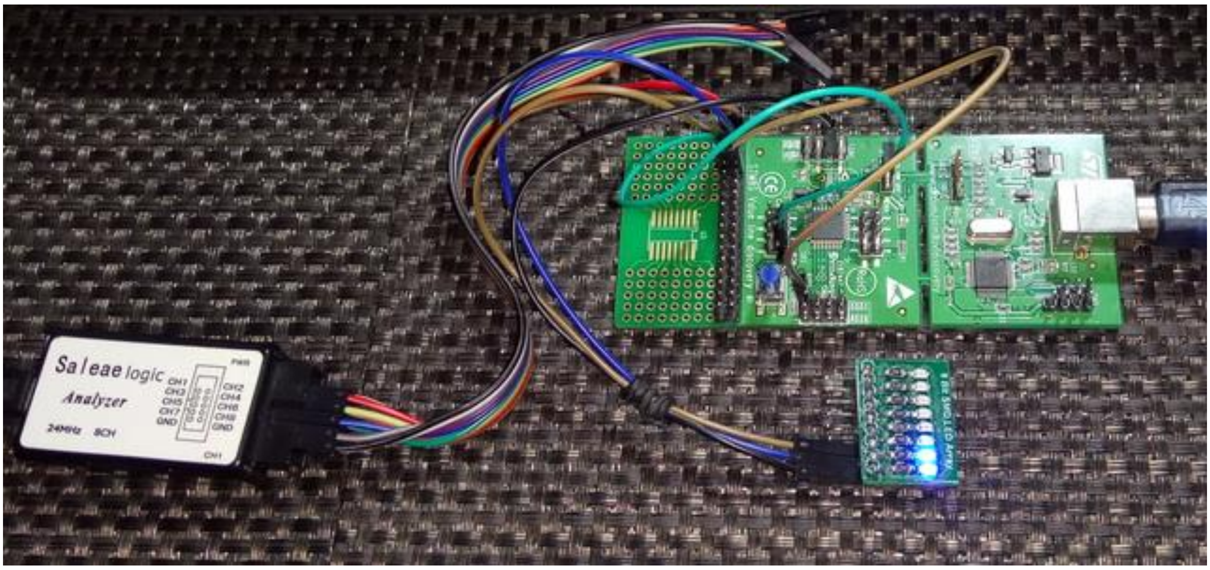
```
void TIM1_setup(void)
{
         TIM1_DeInit();

         TIM1_TimeBaseInit(16, TIM1_COUNTERMODE_UP, 1000, 1);

         TIM1_OC1Init(TIM1_OCMODE_PWM1,
                      TIM1_OUTPUTSTATE_ENABLE,
                      TIM1_OUTPUTNSTATE_ENABLE,
                      1000,
                      TIM1_OCPOLARITY_LOW,
                      TIM1_OCNPOLARITY_LOW,
                      TIM1_OCIDLESTATE_RESET,
                      TIM1_OCNIDLESTATE_RESET);

         TIM1_CtrlPWMOutputs(ENABLE);
         TIM1_Cmd(ENABLE);
}
```

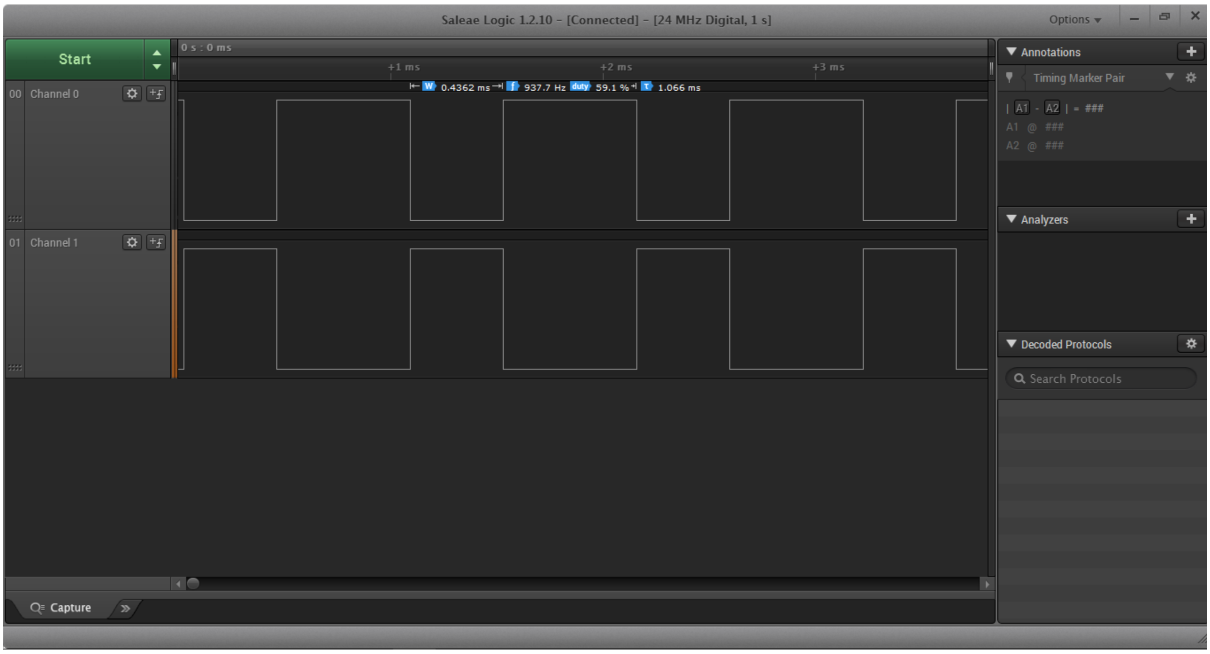
To change the duty cycle of a channel, we need to call this function:

```
TIM1_SetCompareX(duty_cycle);        //  where X represents channel ID (1, 2, 3 or 4)
```

Complementary outputs occur in pairs and so they are interdependent. That's why there is no separate function for outputs labelled ***N***.

Demo





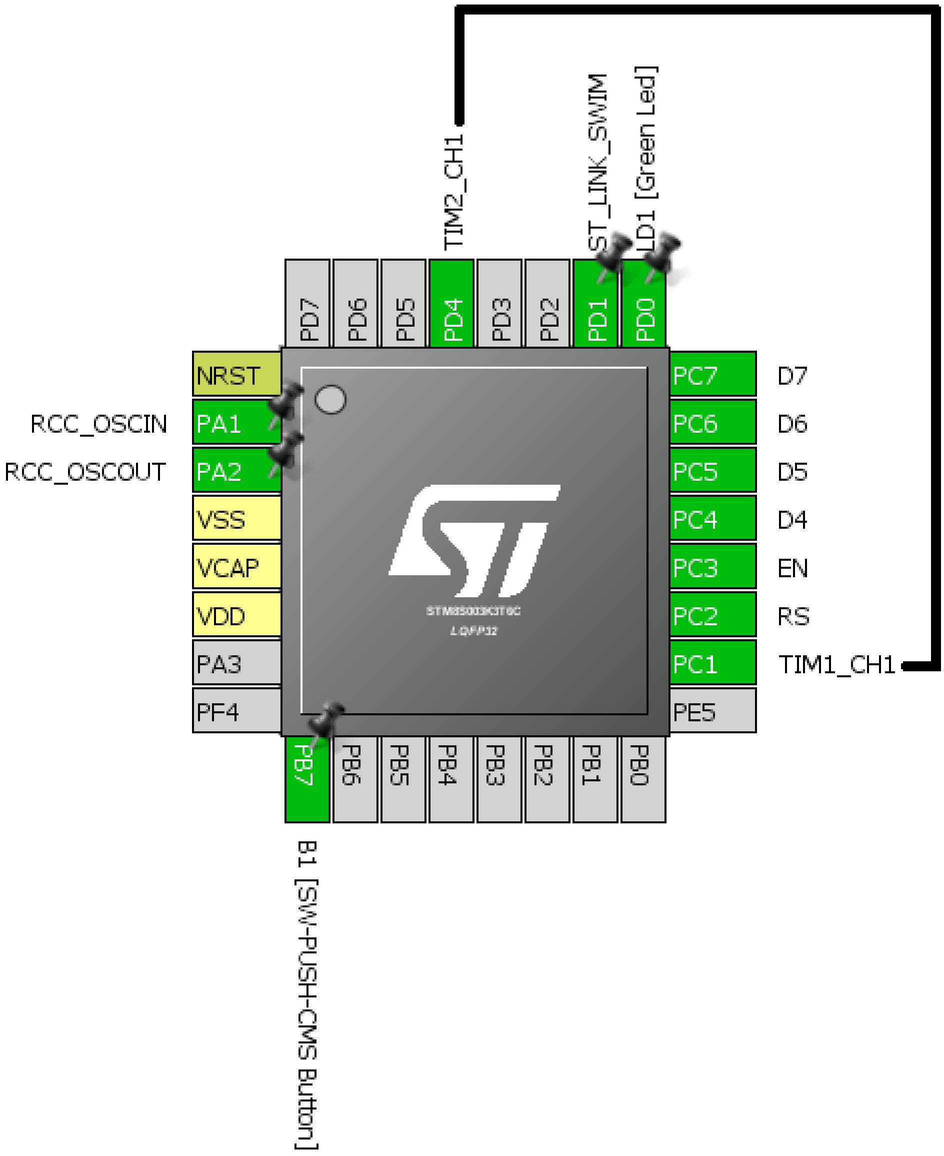
Video link: https://youtu.be/uclCXH1ZPWU

Timer Input Capture (TIM1 & TIM2)

Input capture is needed for measurements of pulses, pulse widths, frequencies, phase detection and similar stuffs. With external interrupts these measurements can be done with some limitations. However, using timer capture has some serious benefits. First of all, accuracy of measurements and secondly timer capture simplifies many tasks as timers themselves time stuffs properly. Dedicated hardware make stuffs like PWM measurement less complex and resource-friendly too.

STM8 timers have several capture channels just like output compare channels (PWM). The number of input capture channels is same as the number of PWM channels. Except basic timers all timers have input capture option.

Hardware Connection



Code Example

In this demo, TIM2 is configured to generate PWM on its CH1 output. TIM1 is configured to capture every rising edge of incoming waveform at its input capture channel CH1. When a capture event occurs, the current time count of TIM1 is saved. By deducting the recent capture count from the previous capture count, we can measure time period of the incoming PWM signal and hence its frequency. If the frequency is too high, TIM1 may overflow and so we need to take care of it too. We, thus, need to check TIM1 overflow event too.

main.c

```
#include "STM8S.h"
#include "lcd.h"


unsigned int overflow_count = 0;
unsigned long pulse_ticks = 0;
unsigned long start_time = 0;
unsigned long end_time = 0;


void clock_setup(void);
void GPIO_setup(void);
void TIM1_setup(void);
void TIM2_setup(void);
void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned long value);


void main()
{
  unsigned long time_period = 0;

        clock_setup();
        GPIO_setup();
        TIM1_setup();
        TIM2_setup();
        LCD_init();

        LCD_clear_home();
        LCD_goto(0, 0);
        LCD_putstr("T/ms:");
        delay_ms(10);

        while(TRUE)
        {
                time_period = pulse_ticks;
                lcd_print(0, 1, time_period);
                delay_ms(400);
        };
}


void clock_setup(void)
{
        CLK_DeInit();

        CLK_HSECmd(DISABLE);
        CLK_LSICmd(DISABLE);
        CLK_HSICmd(ENABLE);
        while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

        CLK_ClockSwitchCmd(ENABLE);
        CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
```

```
    CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV2);

    CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
    DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

    CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, ENABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, ENABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}


void GPIO_setup(void)
{
    GPIO_DeInit(GPIOC);
    GPIO_Init(GPIOC, GPIO_PIN_1, GPIO_MODE_IN_FL_NO_IT);

    GPIO_DeInit(GPIOD);
    GPIO_Init(GPIOD, GPIO_PIN_4, GPIO_MODE_OUT_PP_HIGH_FAST);
}


void TIM1_setup(void)
{
    TIM1_DeInit();
    TIM1_TimeBaseInit(2000, TIM1_COUNTERMODE_UP, 55535, 1);
    TIM1_ICInit(TIM1_CHANNEL_1, TIM1_ICPOLARITY_RISING, TIM1_ICSELECTION_DIRECTTI, 1, 1);
    TIM1_ITConfig(TIM1_IT_UPDATE, ENABLE);
    TIM1_ITConfig(TIM1_IT_CC1, ENABLE);
    TIM1_Cmd(ENABLE);
    enableInterrupts();
}


void TIM2_setup(void)
{
    TIM2_DeInit();
    TIM2_TimeBaseInit(TIM2_PRESCALER_32, 1250);
    TIM2_OC1Init(TIM2_OCMODE_PWM1, TIM2_OUTPUTSTATE_ENABLE, 1000, TIM2_OCPOLARITY_LOW);
    TIM2_SetCompare1(625);
    TIM2_Cmd(ENABLE);
}


void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned long value)
{
    char tmp[6] = {0x20, 0x20, 0x20, 0x20, 0x20, 0x20} ;

    tmp[0] = (((value / 100000) % 10) + 0x30);
    tmp[1] = (((value / 10000) % 10) + 0x30);
    tmp[2] = (((value / 1000) % 10) + 0x30);
    tmp[3] = (((value / 100) % 10) + 0x30);
    tmp[4] = (((value / 10) % 10) + 0x30);
    tmp[5] = ((value % 10) + 0x30);

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

stm8_interrupt_vector.c (Interrupt vector address part only)

```
….
{0x82, (interrupt_handler_t)TIM1_UPD_IRQHandler}, /* irq11 */
{0x82, (interrupt_handler_t)TIM1_CH1_CCP_IRQHandler}, /* irq12 */
….
```

stm8s_it.h (Top part only)

```
#ifndef __STM8S_IT_H
#define __STM8S_IT_H


@far @interrupt void TIM1_UPD_IRQHandler(void);
@far @interrupt void TIM1_CH1_CCP_IRQHandler(void);

/* Includes ------------------------------------------------------------------*/
#include "stm8s.h"
….
```

stm8s_it.c (Top part only)

```
#include "stm8s.h"
#include "stm8s_it.h"


extern unsigned int overflow_count;
extern unsigned long pulse_ticks;
extern unsigned long start_time;
extern unsigned long end_time;


void TIM1_UPD_IRQHandler(void)
{
		overflow_count++;
		TIM1_ClearITPendingBit(TIM1_IT_UPDATE);
		TIM1_ClearFlag(TIM1_FLAG_UPDATE);
}


void TIM1_CH1_CCP_IRQHandler(void)
{
		end_time = TIM1_GetCapture1();
		pulse_ticks = ((overflow_count << 16) - start_time + end_time);
		start_time = end_time;
		overflow_count = 0;
		TIM1_ClearITPendingBit(TIM1_IT_CC1);
		TIM1_ClearFlag(TIM1_FLAG_CC1);

}

….
```

Explanation

The clocks and peripherals are set first. We are using 2MHz peripheral clock and the CPU is running at 0.5MHz.

```
CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV2);
….
….
CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, ENABLE);
CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, ENABLE);
```

GPIOs must be set too. Since TIM2 is to output PWM, its CH1 must be set output. Likewise, TIM1's CH1 GPIO must be set as an input.

```
GPIO_DeInit(GPIOC);
GPIO_Init(GPIOC, GPIO_PIN_1, GPIO_MODE_IN_FL_NO_IT);

GPIO_DeInit(GPIOD);
GPIO_Init(GPIOD, GPIO_PIN_4, GPIO_MODE_OUT_PP_HIGH_FAST);
```

TIM1 need to be configured for input capture. We need to set time base for TIM1 first. It is set as such that TIM1 will overflow every second. Then we set input capture channel by specifying the edge sensitivity, channel, mode and scalars. Since we will be using interrupts, we must enable relevant interrupts.

```
void TIM1_setup(void)
{
        TIM1_DeInit();
        TIM1_TimeBaseInit(2000, TIM1_COUNTERMODE_UP, 55535, 1);
        TIM1_ICInit(TIM1_CHANNEL_1, TIM1_ICPOLARITY_RISING, TIM1_ICSELECTION_DIRECTTI, 1, 1);
        TIM1_ITConfig(TIM1_IT_UPDATE, ENABLE);
        TIM1_ITConfig(TIM1_IT_CC1, ENABLE);
        TIM1_Cmd(ENABLE);
        enableInterrupts();
}
```

TIM2 is set for PWM generation on its CH1. The generated PWM will have a frequency of 50Hz and 50% duty cycle. The setup of TIM2 should be by now self-explanatory:

```
void TIM2_setup(void)
{
        TIM2_DeInit();
        TIM2_TimeBaseInit(TIM2_PRESCALER_32, 1250);
        TIM2_OC1Init(TIM2_OCMODE_PWM1, TIM2_OUTPUTSTATE_ENABLE, 1000, TIM2_OCPOLARITY_LOW);
        TIM2_SetCompare1(625);
        TIM2_Cmd(ENABLE);
}
```

In the vector table of ***stm8_interrupt_vector.c*** file, we need to specify the interrupts we will be using:

```
{0x82, (interrupt_handler_t)TIM1_UPD_IRQHandler}, /* irq11 */
{0x82, (interrupt_handler_t)TIM1_CH1_CCP_IRQHandler}, /* irq12 */
```

We have to specify the interrupt subroutine (ISR) prototype functions in the ***stm8s_it.h*** file. These functions are the places where the code will jump when respective interrupt occurs:

```
@far @interrupt void TIM1_UPD_IRQHandler(void);
@far @interrupt void TIM1_CH1_CCP_IRQHandler(void);
```

The ISR functions are coded in the ***stm8s_it.c*** file:

```
void TIM1_UPD_IRQHandler(void)
{
		overflow_count++;
		TIM1_ClearITPendingBit(TIM1_IT_UPDATE);
		TIM1_ClearFlag(TIM1_FLAG_UPDATE);
}
```

The first part is dealing with TIM1 overflow. If a capture occurs when TIM1 count is near to reset value we need to take this account. This part does so and an overflow counter is incremented.

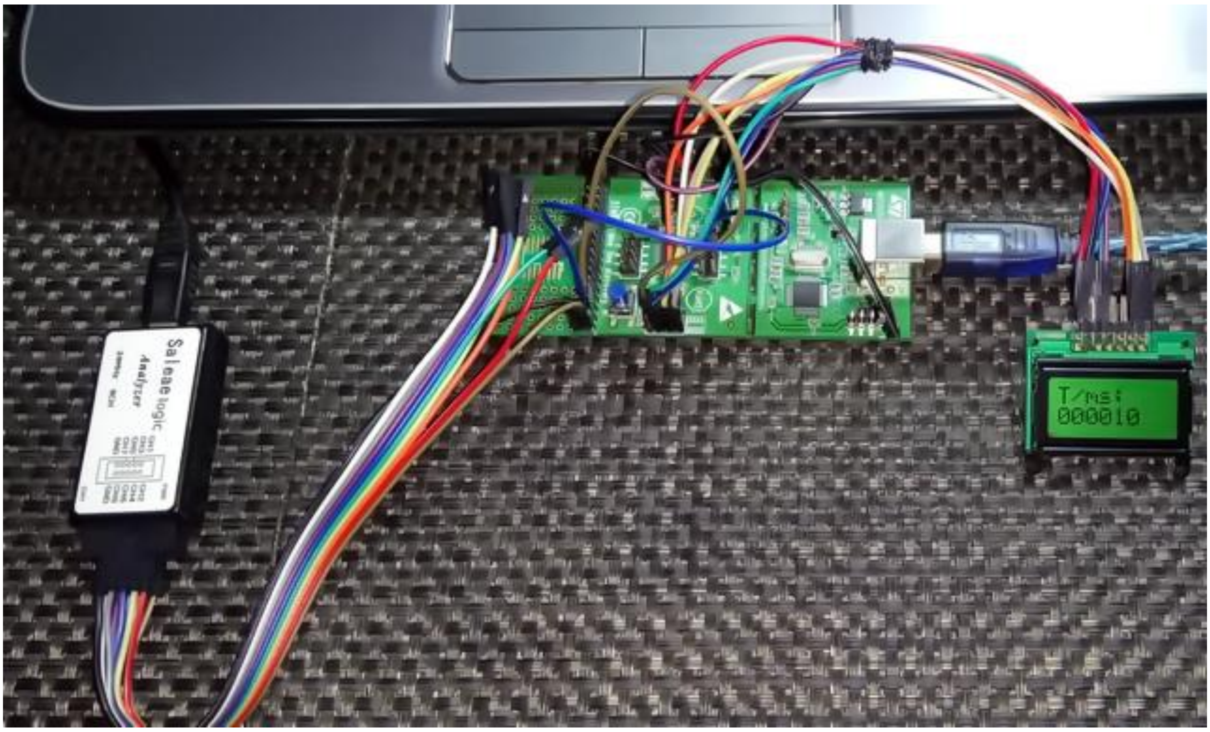
```
void TIM1_CH1_CCP_IRQHandler(void)
{
		end_time = TIM1_GetCapture1();
		pulse_ticks = ((overflow_count << 16) - start_time + end_time);
		start_time = end_time;
		overflow_count = 0;
		TIM1_ClearITPendingBit(TIM1_IT_CC1);
		TIM1_ClearFlag(TIM1_FLAG_CC1);

}
```

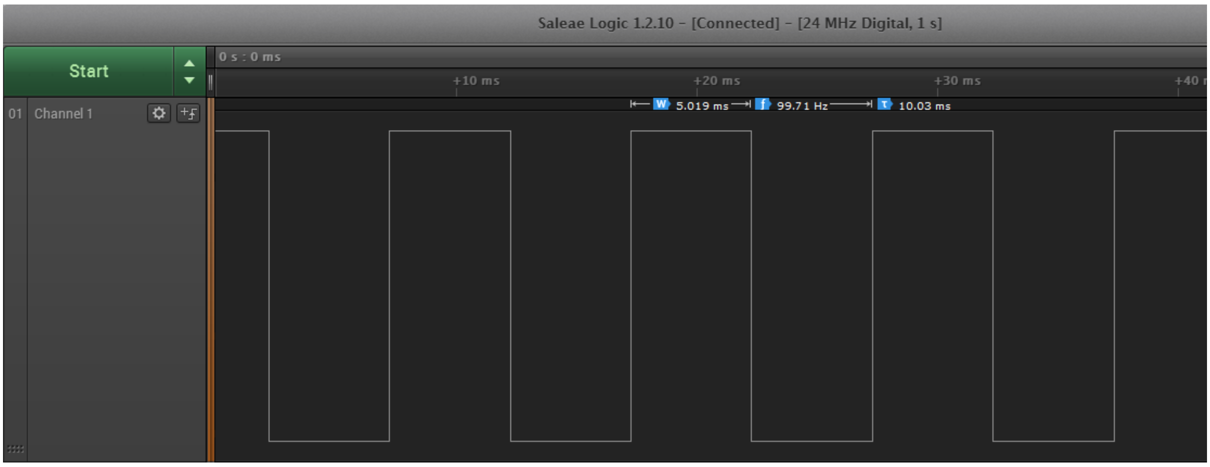
The second part is where TIM1 capture is recorded. Once a rising edge is captured, an interrupt is issued. In the interrupt, we must first save the current TIM1 counter count in the variable named ***end_time***. The formula for pulse tick is then computed. Note how the TIM1 overflow is addressed in the formula. The new start time should be the previous capture time because we need to deduct old capture count from new capture count. Lastly, overflow counter, capture flag and pending interrupts are cleared.

In the main loop, we are just displaying the time period of capture in a LCD while everything is being processed in the background by interrupts:

```
while(TRUE)
{
		time_period = pulse_ticks;
		lcd_print(0, 1, time_period);
		delay_ms(400);
};
```

Demo





Video link: https://youtu.be/bzLUDwuFQTw

Communication Overview

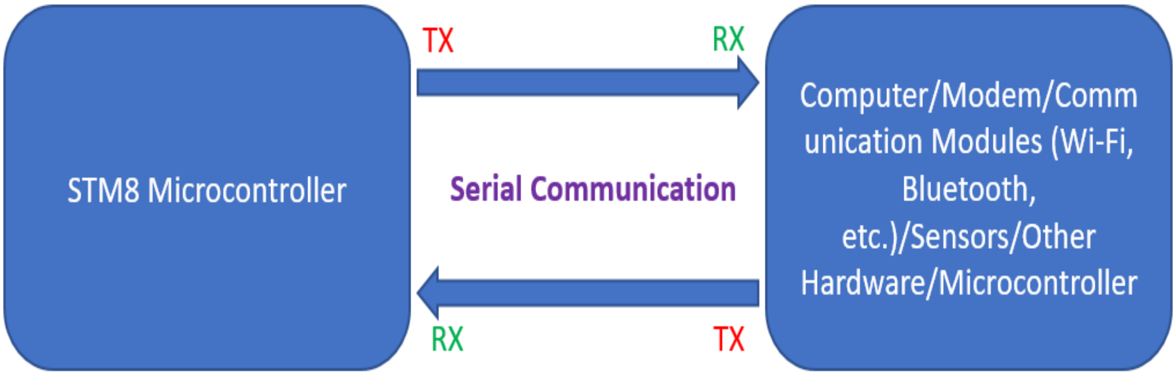
STM8 microcontrollers are packed with several communication interfaces. These interfaces are needed to communicate with external devices like sensors, actuators, drives, etc. The most commonly used ones are Serial Communication (***UART***), Serial Peripheral Interface (***SPI***) and Inter-Integrated Circuit (***I2C***). There are also other additional more robust communication interfaces like Controller Area Network (***CAN***), Local Interconnect Network (***LIN***), Infrared Data Association (***IrDA***) and ***RS-485***. The latter communications will not be discussed here in this article and are kept for future issues. These are methods are, however, not frequently used and are special forms of communications. For example, CAN and LIN are mostly used in automotive industries. Each method communication has its own advantages and disadvantages. Here we'll see the individual basics of various methods of communications.

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	Synchronous master-slave serial communication using one data and one clock line	2	1Mbps	0.5m	127
RS-485	Asynchronous single master differential 2 wire serial communication	2	115.2kbps	1.2km	Several
CAN	Differential communication with multi-master support	2	1Mbps	5km	Several
LIN	Asynchronous 2 wire serial communication similar to UART	2	20kbps	40m	Several
IrDA	Wireless serial communication using infrared medium	2	115.2kbps	<1m	2 (Point-to-Point)

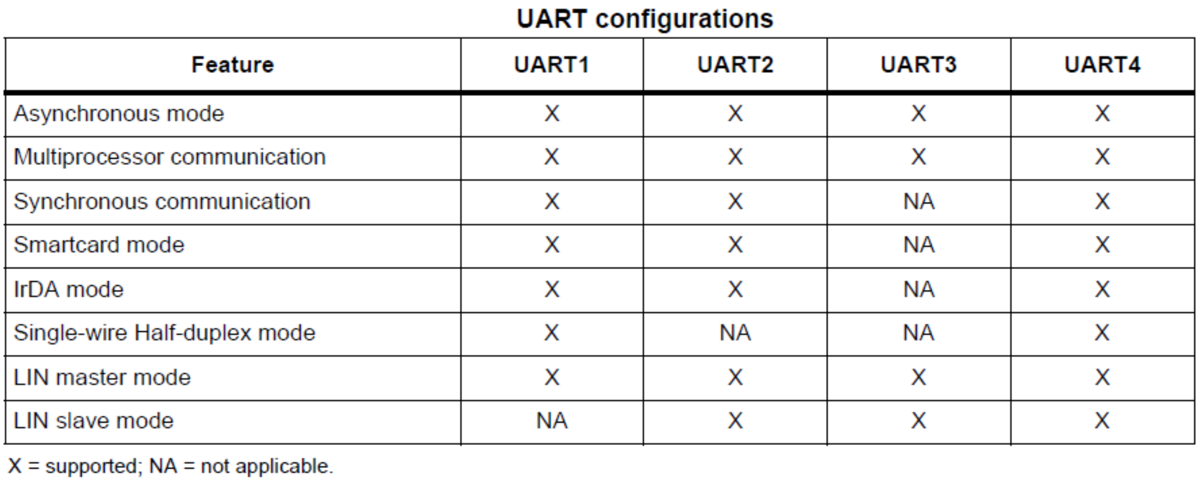
In STM8 microcontrollers, LIN, IrDA, RS-485 and UART all share the UART hardware peripheral. For other communications, there are dedicated separate hardware. We will now be exploring the basic ones here.

Serial Communication (UART)

Serial communication is perhaps the mostly-used classic communication method for interfacing a PC or other machines with a micro. With just two wire, we can achieve a full-duplex point-to-point communication. Owing to its simplicity and wide usage, it is the communication interface backbone that is used with GSM modems, RF modules, Bluetooth devices like RN-52, Wi-Fi devices like the popular ESP8266, etc. It is also widely used in industries. Other communications rely on it, for example, RS-485, LIN, etc.


STM8 Microcontroller
TX
RX
Serial Communication
RX
TX
Computer/Modem/Communication Modules (Wi-Fi, Bluetooth, etc.)/Sensors/Other Hardware/Microcontroller

Most STM8s have at least one UART module. Some have more than one. Different UARTs have different features as shown:



UART configurations

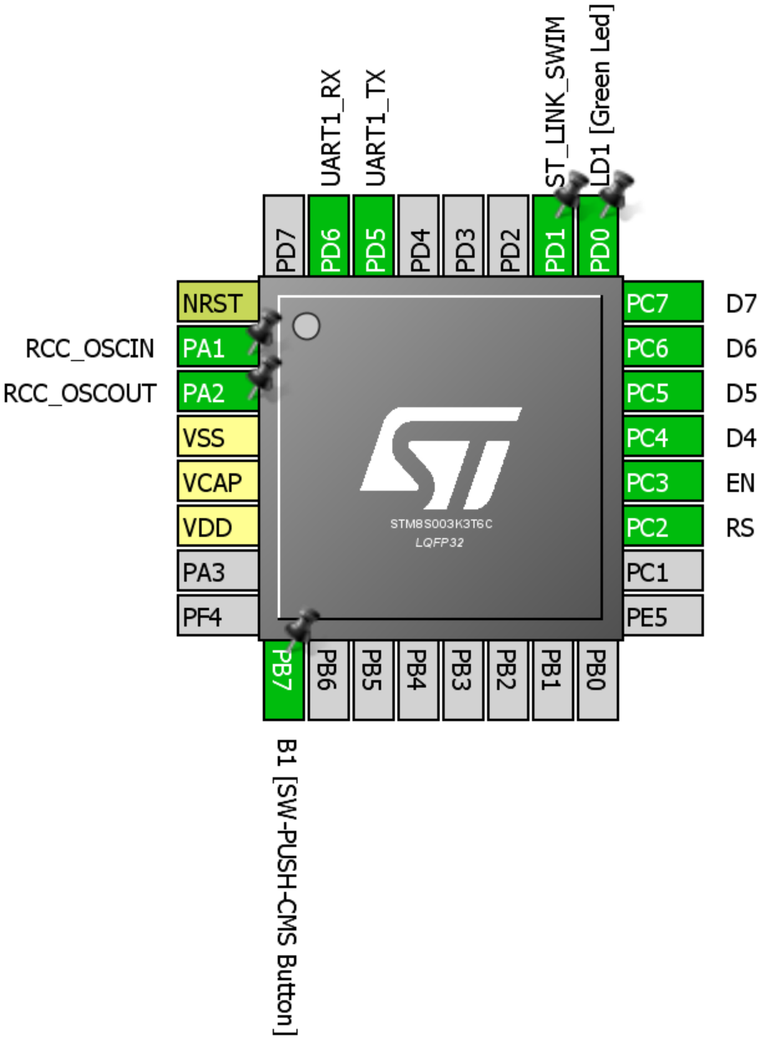
Feature	UART1	UART2	UART3	UART4
Asynchronous mode	X	X	X	X
Multiprocessor communication	X	X	X	X
Synchronous communication	X	X	NA	X
Smartcard mode	X	X	NA	X
IrDA mode	X	X	NA	X
Single-wire Half-duplex mode	X	NA	NA	X
LIN master mode	X	X	X	X
LIN slave mode	NA	X	X	X

X = supported; NA = not applicable.

To learn more about UART visit the following link:
https://learn.mikroe.com/uart-serial-communication/

The UARTs of STM8 micros are so robust and packed with so many features that it is quite impossible to explain them all in this one article. Here we will explore the basic serial communication only. LIN and IRDA will hopefully be covered in future articles.

Hardware Connection



Code Example

```
#include "STM8S.h"


void clock_setup(void);
void GPIO_setup(void);
void UART1_setup(void);


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

                clock_setup();
```

```
        GPIO_setup();
        UART1_setup();
        LCD_init();
        LCD_clear_home();

        LCD_goto(0, 0);
        LCD_putstr("TX:");
        LCD_goto(0, 1);
        LCD_putstr("RX:");

        while(TRUE)
        {
                if(UART1_GetFlagStatus(UART1_FLAG_RXNE) == TRUE)
                {
                        ch = UART1_ReceiveData8();
                        LCD_goto(7, 1);
                        LCD_putchar(ch);
                        UART1_ClearFlag(UART1_FLAG_RXNE);
                        UART1_SendData8(i + 0x30);
                }
                if(UART1_GetFlagStatus(UART1_FLAG_TXE) == FALSE)
                {
                        LCD_goto(7, 0);
                        LCD_putchar(i + 0x30);
                        i++;
                }
        };
}


void clock_setup(void)
{
        CLK_DeInit();

        CLK_HSECmd(DISABLE);
        CLK_LSICmd(DISABLE);
        CLK_HSICmd(ENABLE);
        while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

        CLK_ClockSwitchCmd(ENABLE);
        CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
        CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);

        CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
        DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

        CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, ENABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}


void GPIO_setup(void)
{
        GPIO_DeInit(GPIOD);

        GPIO_Init(GPIOD, GPIO_PIN_5, GPIO_MODE_OUT_PP_HIGH_FAST);
        GPIO_Init(GPIOD, GPIO_PIN_6, GPIO_MODE_IN_PU_NO_IT);
}


void UART1_setup(void)
```

```
{
            UART1_DeInit();

            UART1_Init(9600,
                       UART1_WORDLENGTH_8D,
                       UART1_STOPBITS_1,
                       UART1_PARITY_NO,
                       UART1_SYNCMODE_CLOCK_DISABLE,
                       UART1_MODE_TXRX_ENABLE);

            UART1_Cmd(ENABLE);
}
```

Explanation

The peripheral and CPU clocks are set at 2MHz:

```
CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);
….
….
CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, ENABLE);
```

The TX-RX GPIO pins are set as output and input respectively:

```
GPIO_DeInit(GPIOD);

GPIO_Init(GPIOD, GPIO_PIN_5, GPIO_MODE_OUT_PP_HIGH_FAST);
GPIO_Init(GPIOD, GPIO_PIN_6, GPIO_MODE_IN_PU_NO_IT);
```

UART setup is straightforward. We just need to set baud rate, no. of data bits, no. of stop bit, parity and type of communication (synchronous or asynchronous).

```
void UART1_setup(void)
{
            UART1_DeInit();

            UART1_Init(9600,
                       UART1_WORDLENGTH_8D,
                       UART1_STOPBITS_1,
                       UART1_PARITY_NO,
                       UART1_SYNCMODE_CLOCK_DISABLE,
                       UART1_MODE_TXRX_ENABLE);

            UART1_Cmd(ENABLE);
}
```

In the main code, we are checking both transmission complete and reception complete flags. With these flags, we will know if new data arrived and if it is possible to send a new data.

The first part checks if any new data received. That's why the ***IF*** condition is checking if the RX buffer is empty or not. If it is not empty then new data must have arrived. The new data (a character here) is fetched and displayed on a LCD. Then we clear the RX buffer not empty flag to enable reception of new data. After that we are sending some data to the host PC over the UART.

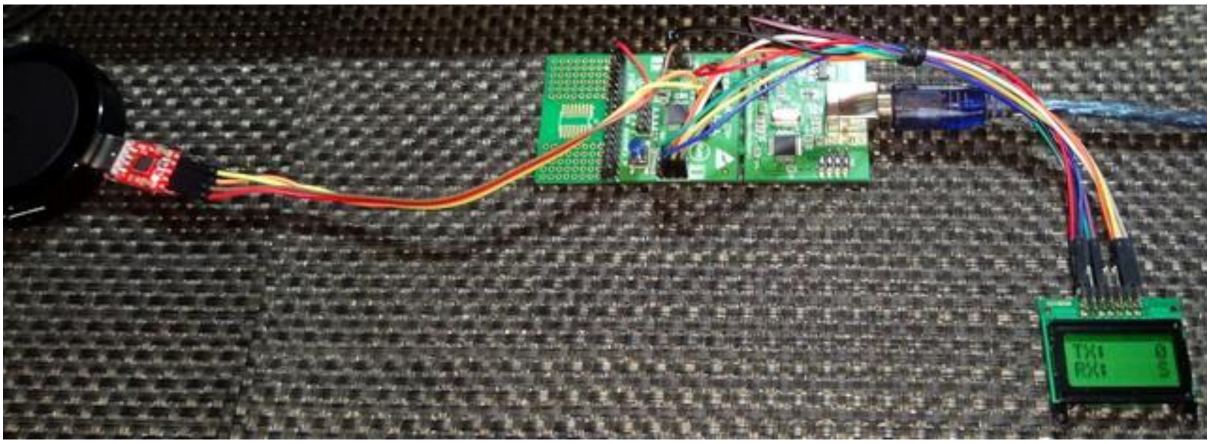
```
if(UART1_GetFlagStatus(UART1_FLAG_RXNE) == TRUE)
{
    ch = UART1_ReceiveData8();
    LCD_goto(7, 1);
    LCD_putchar(ch);
    UART1_ClearFlag(UART1_FLAG_RXNE);
    UART1_SendData8(i + 0x30);
}
```

In the second part, we are checking if the last data was sent from our STM8 micro. The data sent is then displayed on LCD.

```
if(UART1_GetFlagStatus(UART1_FLAG_TXE) == FALSE)
{
    LCD_goto(7, 0);
    LCD_putchar(i + 0x30);
    i++;
}
```

Please note that both of these flags are very important.

Demo



Video link: https://youtu.be/uo2tYDUnMmE

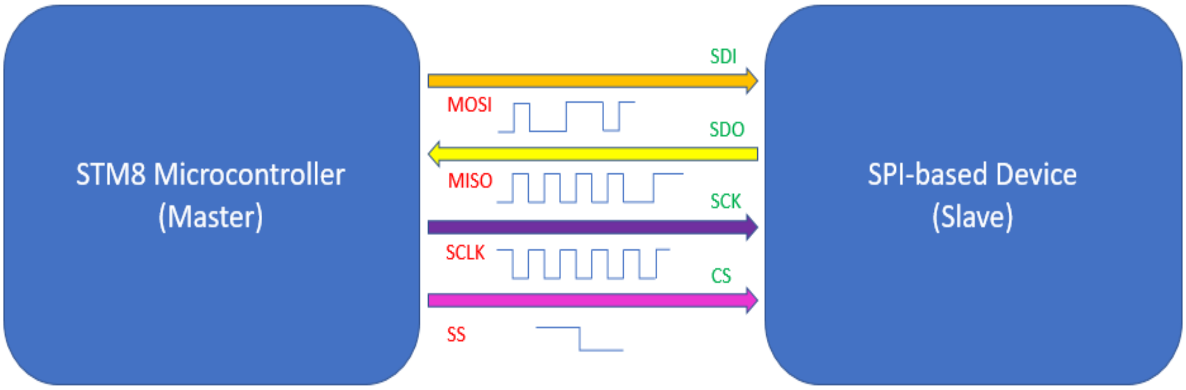
Serial Peripheral Interface (SPI)

SPI communication is an onboard synchronous communication method and is used by a number of devices including sensors, TFT displays, GPIO expanders, PWM controller ICs, memory chips, addon support devices, etc.

There's always one master device in a SPI communication bus which generates clock and select slave(s). Master sends commands to slave(s). Slave(s) responds to commands sent by the master. The number of slaves in a SPI bus is virtually unlimited. Except the chip selection pin, all SPI devices in a bus can share the same clock and data pins.

Typical full-duplex SPI bus requires four basic I/O pins:

- ***Master-Out-Slave-In (MOSI)*** connected to ***Slave-Data-In (SDI)***.
- ***Master-In-Slave-Out (MIS0)*** connected to ***Slave-Data-Out (SDO)***.
- ***Serial Clock (SCLK)*** connected to ***Slave Clock (SCK)***.
- ***Slave Select (SS)*** connected to ***Chip Select (CS)***.

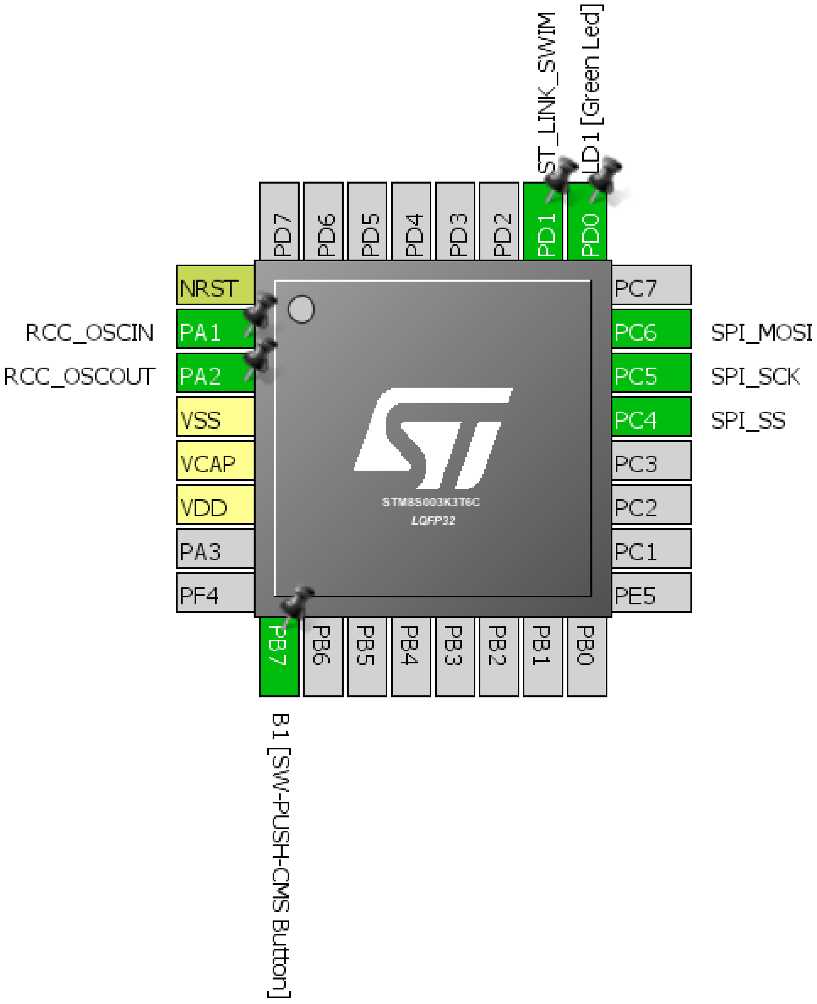


In general, if you wish to know more about SPI bus here are some cool links:

- 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/

STM8s have SPI hardware that are more capable than the SPI hardware found in other micros. An additional feature of STM8's SPI is the hardware CRC. This feature ensures reliable data communication between devices.

Hardware Connection



Code Example

main.c

```
#include "STM8S.h"
#include "MAX72XX.h"


void clock_setup(void);
void GPIO_setup(void);
void SPI_setup(void);


void main()
{
                unsigned char i = 0x00;
   unsigned char j = 0x00;

   volatile unsigned char temp[8] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

   const unsigned char text[96] =
   {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x7E, 0x04, 0x08, 0x08, 0x04, 0x7E, 0x00,     //M
    0x00, 0x42, 0x42, 0x7E, 0x7E, 0x42, 0x42, 0x00,     //I
    0x00, 0x3C, 0x42, 0x42, 0x42, 0x42, 0x24, 0x00,     //C
    0x00, 0x7E, 0x1A, 0x1A, 0x1A, 0x2A, 0x44, 0x00,     //R
    0x00, 0x3C, 0x42, 0x42, 0x42, 0x42, 0x3C, 0x00,     //O
    0x00, 0x7C, 0x12, 0x12, 0x12, 0x12, 0x7C, 0x00,     //A
    0x00, 0x7E, 0x1A, 0x1A, 0x1A, 0x2A, 0x44, 0x00,     //R
    0x00, 0x7E, 0x7E, 0x4A, 0x4A, 0x4A, 0x42, 0x00,     //E
    0x00, 0x7E, 0x04, 0x08, 0x10, 0x20, 0x7E, 0x00,     //N
    0x00, 0x7C, 0x12, 0x12, 0x12, 0x12, 0x7C, 0x00,     //A
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
  };


        clock_setup();
        GPIO_setup();
        SPI_setup();
        MAX72xx_init();

        while(TRUE)
        {
                for(i = 0; i < sizeof(temp); i++)
                {
                        temp[i] = 0x00;
                }

                for(i = 0; i < sizeof(text); i++)
                {
                        for(j = 0; j < sizeof(temp); j++)
                        {
                                temp[j] = text[(i + j)];
                                MAX72xx_write((1 + j), temp[j]);
                                delay_ms(9);
                         }
                 }
        };
}


void clock_setup(void)
{
        CLK_DeInit();

        CLK_HSECmd(DISABLE);
        CLK_LSICmd(DISABLE);
        CLK_HSICmd(ENABLE);
        while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

        CLK_ClockSwitchCmd(ENABLE);
        CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);
        CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);

        CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
        DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

        CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, ENABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
        CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}
```

```
void GPIO_setup(void)
{
        GPIO_DeInit(GPIOC);
        GPIO_Init(GPIOC, ((GPIO_Pin_TypeDef)GPIO_PIN_5 | GPIO_PIN_6), GPIO_MODE_OUT_PP_HIGH_FAST);
}


void SPI_setup(void)
{
        SPI_DeInit();
        SPI_Init(SPI_FIRSTBIT_MSB,
                SPI_BAUDRATEPRESCALER_2,
                SPI_MODE_MASTER,
                SPI_CLOCKPOLARITY_HIGH,
                SPI_CLOCKPHASE_1EDGE,
                SPI_DATADIRECTION_1LINE_TX,
                SPI_NSS_SOFT,
                 0x00);
        SPI_Cmd(ENABLE);
}
```

MAX72xx.h

```
#include "STM8S.h"


#define CS_pin                                  GPIO_PIN_4
#define CS_port                                 GPIOC

#define NOP                                     0x00
#define DIG0                                    0x01
#define DIG1                                    0x02
#define DIG2                                    0x03
#define DIG3                                    0x04
#define DIG4                                    0x05
#define DIG5                                    0x06
#define DIG6                                    0x07
#define DIG7                                    0x08

#define decode_mode_reg                         0x09
#define intensity_reg                           0x0A
#define scan_limit_reg                          0x0B
#define shutdown_reg                            0x0C
#define display_test_reg                        0x0F

#define shutdown_cmd                            0x00
#define run_cmd                                 0x01

#define no_test_cmd                             0x00
#define test_cmd                                0x01

#define digit_0_only                            0x00
#define digit_0_to_1                            0x01
#define digit_0_to_2                            0x02
#define digit_0_to_3                            0x03
#define digit_0_to_4                            0x04
#define digit_0_to_5                            0x05
#define digit_0_to_6                            0x06
#define digit_0_to_7                            0x07

#define No_decode_for_all                       0x00
#define Code_B_decode_digit_0                   0x01
#define Code_B_decode_digit_0_to_3              0x0F
#define Code_B_decode_for_all                   0xFF
```

```
void MAX72xx_init(void);
void MAX72xx_write(unsigned char address, unsigned char value);
```

MAX72xx.c

```
#include "MAX72xx.h"


void MAX72xx_init(void)
{
   GPIO_Init(CS_port, CS_pin, GPIO_MODE_OUT_PP_HIGH_FAST);

   MAX72xx_write(shutdown_reg, run_cmd);
   MAX72xx_write(decode_mode_reg, 0x00);
   MAX72xx_write(scan_limit_reg, 0x07);
   MAX72xx_write(intensity_reg, 0x04);
   MAX72xx_write(display_test_reg, test_cmd);
   delay_ms(10);
   MAX72xx_write(display_test_reg, no_test_cmd);
}


void MAX72xx_write(unsigned char address, unsigned char value)
{
   while(SPI_GetFlagStatus(SPI_FLAG_BSY));
   GPIO_WriteLow(CS_port, CS_pin);

    SPI_SendData(address);
    while(!SPI_GetFlagStatus(SPI_FLAG_TXE));

     SPI_SendData(value);
     while(!SPI_GetFlagStatus(SPI_FLAG_TXE));

     GPIO_WriteHigh(CS_port, CS_pin);
}
```

Explanation

This time we are again using the max peripheral and CPU clock:

```
CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);
CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);
….
….
CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, ENABLE);
```

We also need to set the GPIOs:

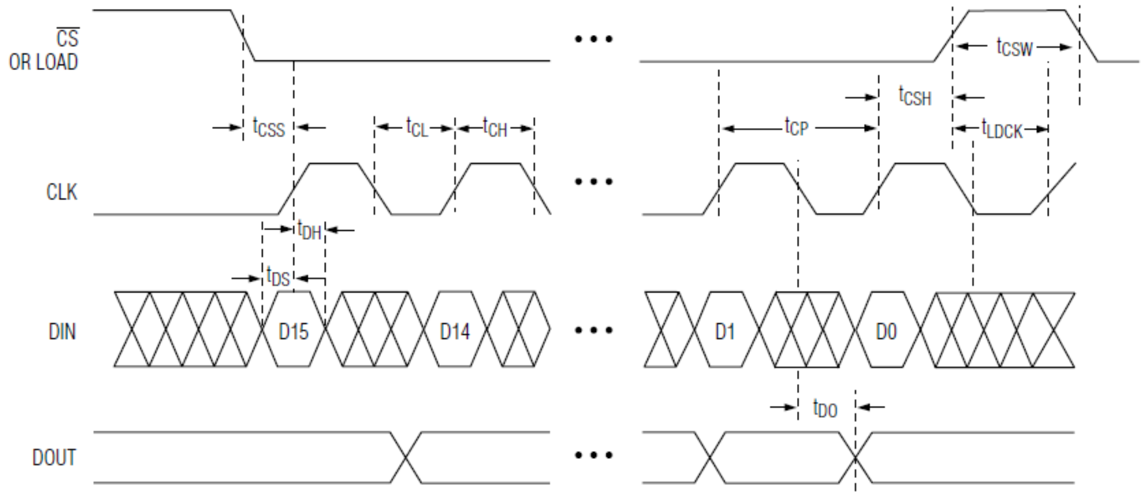
```
#define CS_pin                    GPIO_PIN_4
#define CS_port                   GPIOC
….
….
GPIO_DeInit(GPIOC);
GPIO_Init(CS_port, CS_pin, GPIO_MODE_OUT_PP_HIGH_FAST);
GPIO_Init(GPIOC, ((GPIO_Pin_TypeDef)GPIO_PIN_5 | GPIO_PIN_6), GPIO_MODE_OUT_PP_HIGH_FAST);
```

Note we can use definitions to make things meaningful. The GPIOs should be configured as fast I/Os because SPI communication is faster than simple GPIO operations.

Now for the SPI configuration part. Assuming you know how to interpret timing diagrams and understand device datasheets, SPI configuration should not be a problem. Here in the case of MAX7219, we have configured the SPI port as to send MSB first, we have also selected a fast peripheral clock, we have made the STM8 SPI act like a master with proper SPI mode and we have set the sort of duplex. The last two parameters are not important as we are not using hardware slave select option and CRC feature.

```
void SPI_setup(void)
{
        SPI_DeInit();
        SPI_Init(SPI_FIRSTBIT_MSB,
                 SPI_BAUDRATEPRESCALER_2,
                 SPI_MODE_MASTER,
                 SPI_CLOCKPOLARITY_HIGH,
                 SPI_CLOCKPHASE_1EDGE,
                 SPI_DATADIRECTION_1LINE_TX,
                 SPI_NSS_SOFT,
                 0x00);
        SPI_Cmd(ENABLE);
}
```

The timing diagram of MAX7219 suggests that CS should be low in order for MAX7219 to receive data.



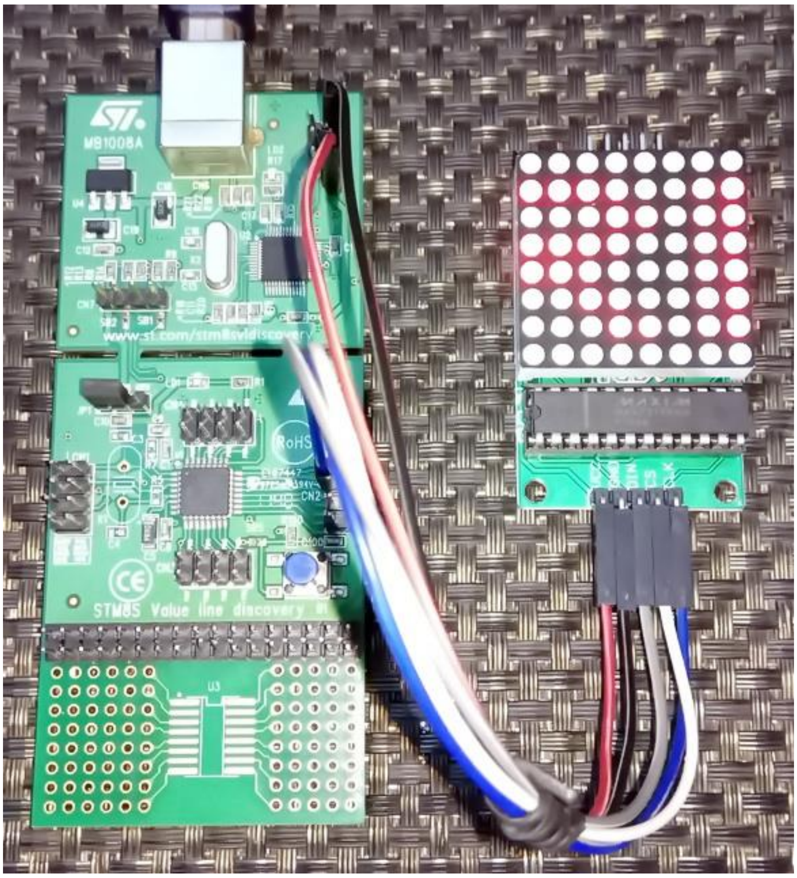
Then it suggests that when idle, clock must be high, data transfer is done on every rising edge of the clock. All these are what required for setting up the SPI hardware.

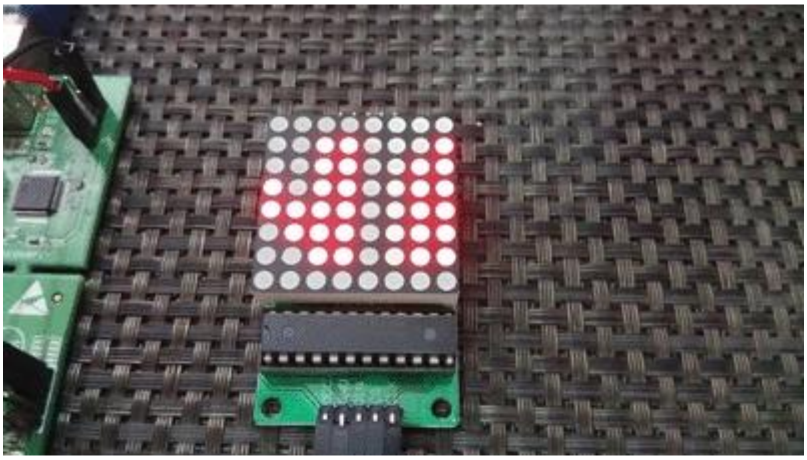
```
void MAX72xx_write(unsigned char address, unsigned char value)
{
  while(SPI_GetFlagStatus(SPI_FLAG_BSY));
  GPIO_WriteLow(CS_port, CS_pin);

   SPI_SendData(address);
   while(!SPI_GetFlagStatus(SPI_FLAG_TXE));

    SPI_SendData(value);
    while(!SPI_GetFlagStatus(SPI_FLAG_TXE));

    GPIO_WriteHigh(CS_port, CS_pin);
}
```

Before sending data to MAX7219, we must check if the SPI hardware is busy for some reason. We set CS low by setting STM8's slave select pin (PC4) low. Then we send address and data. Every time we send something we must wait until it has completely been sent out. Finally, we set CS high to latch sent data. This function is what we will need to set MAX7219 things up and also to update displays.

The demo here is that of a MAX7219- based scrolling dot-matrix display. Letters of ***MICROARENA*** – the name of my Facebook page is scrolled.

Demo

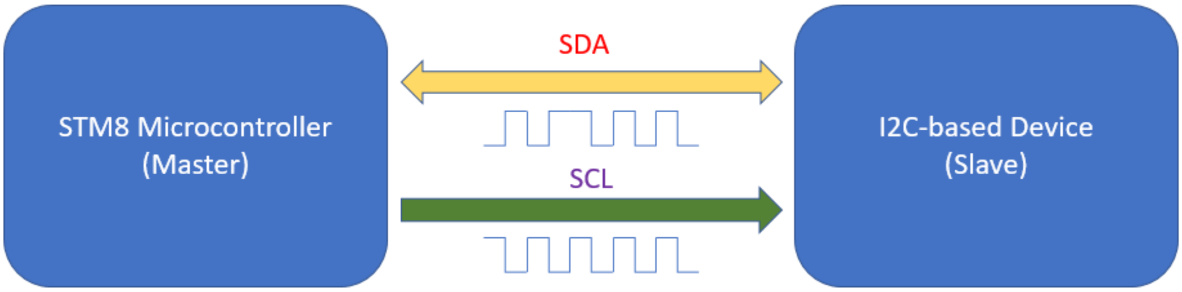




Video link: https://youtu.be/O7mre-bzsGE

Inter-Integrated Circuit (I2C)

I2C is another popular form of on board synchronous serial communication developed by NXP. It just uses two wires for communication and so it is also referred as ***Two Wire Interface (TWI)***. Just like SPI, I2C is widely used in interfacing real-time clocks (RTC), digital sensors, memory chips and so on. It is as much as popular as SPI but compared to SPI it is slower and have some limitations. Up to 127 devices can coexist in an I2C bus. In an I2C bus however it is not possible, by conventional means to interface devices with same device IDs or devices with different logic voltage levels without logic level converters and so on. Still however, I2C is very popular because these issues rarely arise and because of its simplicity. Unlike other communications, there's no pin/wire swapping as two wires connect straight to the bus – SDA to SDA and SCL to SCL.



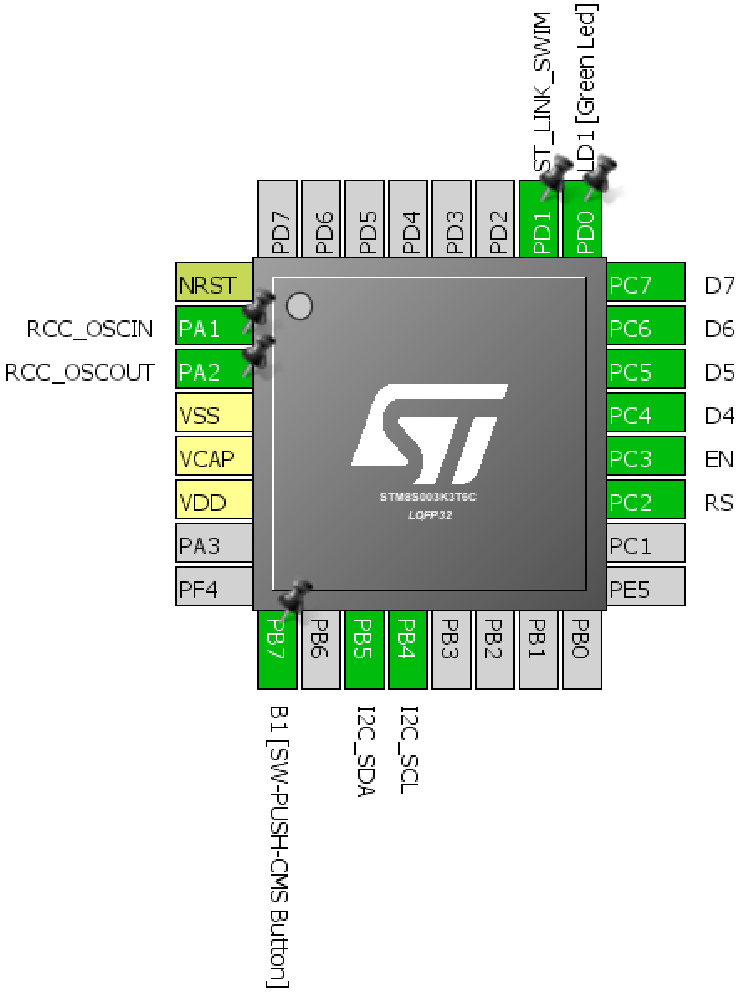
Just like SPI, an I2C bus must contain one master device (usually a microcontroller) and one or more slaves. The master is solely responsible for generating clock signals and initiating communication. Communication starts when master sends out a slave's ID with read/write command. The slave reacts to this command by processing the request from the master and sending out data.

To know more about I2C interface visit the following links:

- 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/

Other protocols like SMBus and I2S have similarities with I2C and so learning about I2C advances learning these too.

Hardware Connection



Code Example

This code demonstrates how to interface BH1750 I2C digital light sensor with STM8S003K3. A LCD is used to display the light sensor’s output in lux.

main.c

```
#include "STM8S.h"
#include "BH1750.h"
#include "lcd.h"


void clock_setup(void);
void GPIO_setup(void);
void I2C_setup(void);
void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned int value);


void main()
{
   unsigned int LX = 0x0000;
```

```
    unsigned int tmp = 0x0000;

    clock_setup();
    GPIO_setup();
    I2C_setup();
    LCD_init();
   BH1750_init();

     LCD_clear_home();
     LCD_goto(0, 0);
     LCD_putstr("STM8 I2C");
     LCD_goto(0, 1);
     LCD_putstr("Lx");
     delay_ms(10);

      while(TRUE)
     {
         tmp = get_lux_value(cont_L_res_mode, 20);

         if(tmp > 10)
         {
             LX = tmp;
         }
         else
         {
             LX = get_lux_value(cont_H_res_mode1, 140);
         }

        lcd_print(3, 1, LX);
        delay_ms(200);
     };
}


void clock_setup(void)
{
      CLK_DeInit();

      CLK_HSECmd(DISABLE);
      CLK_LSICmd(DISABLE);
      CLK_HSICmd(ENABLE);
      while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

      CLK_ClockSwitchCmd(ENABLE);
      CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
      CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV2);

      CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
      DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

      CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
      CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, ENABLE);
      CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
      CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
      CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
      CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
      CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
      CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}


void GPIO_setup(void)
{
    GPIO_DeInit(GPIOB);
    GPIO_Init(GPIOB, GPIO_PIN_4, GPIO_MODE_OUT_OD_HIZ_FAST);
    GPIO_Init(GPIOB, GPIO_PIN_5, GPIO_MODE_OUT_OD_HIZ_FAST);
}
```

```
void I2C_setup(void)
{
   I2C_DeInit();
   I2C_Init(100000,
              BH1750_addr,
              I2C_DUTYCYCLE_2,
              I2C_ACK_CURR,
              I2C_ADDMODE_7BIT,
             (CLK_GetClockFreq() / 1000000));
    I2C_Cmd(ENABLE);
}



void lcd_print(unsigned char x_pos, unsigned char y_pos, unsigned int value)
{
   char tmp[5] = {0x20, 0x20, 0x20, 0x20, 0x20} ;

    tmp[0] = ((value / 10000) + 0x30);
   tmp[1] = (((value / 1000) % 10) + 0x30);
   tmp[2] = (((value / 100) % 10) + 0x30);
   tmp[3] = (((value / 10) % 10) + 0x30);
   tmp[4] = ((value % 10) + 0x30);

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

BH1750.h

```
#include "STM8S.h"


#define   BH1750_addr                                                    0x46

#define  power_down                                                     0x00
#define  power_up                                                       0x01
#define  reset                                                          0x07
#define  cont_H_res_mode1                                               0x10
#define  cont_H_res_mode2                                               0x11
#define  cont_L_res_mode                                                0x13
#define  one_time_H_res_mode1                                           0x20
#define  one_time_H_res_mode2                                           0x21
#define  one_time_L_res_mode                                            0x23


void BH1750_init(void);
void BH1750_write(unsigned char cmd);
unsigned int BH1750_read_word(void);
unsigned int get_lux_value(unsigned char mode, unsigned int delay_time);
```

BH1750.c

```
#include "BH1750.h"


void BH1750_init(void)
{
  delay_ms(10);
  BH1750_write(power_down);
}
```

```
void BH1750_write(unsigned char cmd)
{
   I2C_GenerateSTART(ENABLE);
   while(!I2C_CheckEvent(I2C_EVENT_MASTER_MODE_SELECT));

   I2C_Send7bitAddress(BH1750_addr, I2C_DIRECTION_TX);
   while(!I2C_CheckEvent(I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));

   I2C_SendData(cmd);
   while(!I2C_CheckEvent(I2C_EVENT_MASTER_BYTE_TRANSMITTED));

   I2C_GenerateSTOP(ENABLE);
}


unsigned int BH1750_read_word(void)
{
   unsigned long value = 0x0000;
   unsigned char num_of_bytes = 0x02;
   unsigned char bytes[2] = {0x00, 0x00};

   while(I2C_GetFlagStatus(I2C_FLAG_BUSBUSY));

   I2C_GenerateSTART(ENABLE);
   while(!I2C_CheckEvent(I2C_EVENT_MASTER_MODE_SELECT));

   I2C_Send7bitAddress(BH1750_addr, I2C_DIRECTION_RX);
   while(!I2C_CheckEvent(I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));

   while(num_of_bytes)
   {
        if(I2C_CheckEvent(I2C_EVENT_MASTER_BYTE_RECEIVED))
       {
            if(num_of_bytes == 0)
            {
                 I2C_AcknowledgeConfig(I2C_ACK_NONE);
                 I2C_GenerateSTOP(ENABLE);
             }

            bytes[(num_of_bytes - 1)] = I2C_ReceiveData();
            num_of_bytes--;
        }
   };

   value = ((bytes[1] << 8) | bytes[0]);

   return value;
}


unsigned int get_lux_value(unsigned char mode, unsigned int delay_time)
{
   unsigned long lux_value = 0x00;
   unsigned char dly = 0x00;
   unsigned char s = 0x08;

   while(s)
   {
       BH1750_write(power_up);
       BH1750_write(mode);
       lux_value += BH1750_read_word();
       for(dly = 0; dly < delay_time; dly += 1)
       {
           delay_ms(1);
       }
```

```
    BH1750_write(power_down);
    s--;
  }
  lux_value >>= 3;

  return ((unsigned int)lux_value);
}
```

Explanation

Firstly both the CPU and peripheral clocks are set. Note the CPU is slower than last few examples. This has no significance.

```
CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV8);
CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV2);
….
….
CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, ENABLE);
```

I2C I/Os are set as open drain outputs because they have external pull-up resistors that terminate the bus I/Os to VDD lines. ***SCL*** pin is always an output from host microcontroller's end however ***SDA*** pin's direction varies with reading and writing operations. This is automatically done by the I2C hardware.

```
GPIO_DeInit(GPIOB);
GPIO_Init(GPIOB, GPIO_PIN_4, GPIO_MODE_OUT_OD_HIZ_FAST);
GPIO_Init(GPIOB, GPIO_PIN_5, GPIO_MODE_OUT_OD_HIZ_FAST);
```

I2C setup has many parameters to set, firstly the I2C bus clock speed, then its own ID, clock duty cycle, address mode, acknowledgement type and clock speed of the peripheral. Here the own ID and slave ID are both set same because we are not using our STM8 as a slave. It doesn't matter. You can also set something else.

```
void I2C_setup(void)
{
  I2C_DeInit();
  I2C_Init(100000,
           BH1750_addr,
           I2C_DUTYCYCLE_2,
           I2C_ACK_CURR,
           I2C_ADDMODE_7BIT,
          (CLK_GetClockFreq() / 1000000));
   I2C_Cmd(ENABLE);
}
```

If you have used compilers with built-in I2C library before then you may get some hiccups studying the following part. This is because those built-in libraries accomplish many tasks in the background that you never felt necessary. Flags and acknowledgments are such stuffs that are mostly automatically dealt by the compiler and ignore by most users. Personally, I had to struggle with these before settling this code. Another big difference is fact that the SPL's functions, their operations and nomenclatures for I2C are different than most I2C libraries one has seen before. Lastly, I2C examples with STM's SPL on the internet are rare and most of them demonstrate I2C example with 24 series EEPROMs only. I wanted to do something different and so I used BH1750 I2C digital sensor instead of repeating another EEPROM example.

```
unsigned int BH1750_read_word(void)
{
  unsigned long value = 0x0000;
  unsigned char num_of_bytes = 0x02;
  unsigned char bytes[2] = {0x00, 0x00};

  while(I2C_GetFlagStatus(I2C_FLAG_BUSBUSY));

  I2C_GenerateSTART(ENABLE);
  while(!I2C_CheckEvent(I2C_EVENT_MASTER_MODE_SELECT));

  I2C_Send7bitAddress(BH1750_addr, I2C_DIRECTION_RX);
  while(!I2C_CheckEvent(I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));

  while(num_of_bytes)
  {
      if(I2C_CheckEvent(I2C_EVENT_MASTER_BYTE_RECEIVED))
     {
         if(num_of_bytes == 0)
         {
             I2C_AcknowledgeConfig(I2C_ACK_NONE);
             I2C_GenerateSTOP(ENABLE);
          }

         bytes[(num_of_bytes - 1)] = I2C_ReceiveData();
         num_of_bytes--;
      }
  };

  value = ((bytes[1] << 8) | bytes[0]);

  return value;
}
```

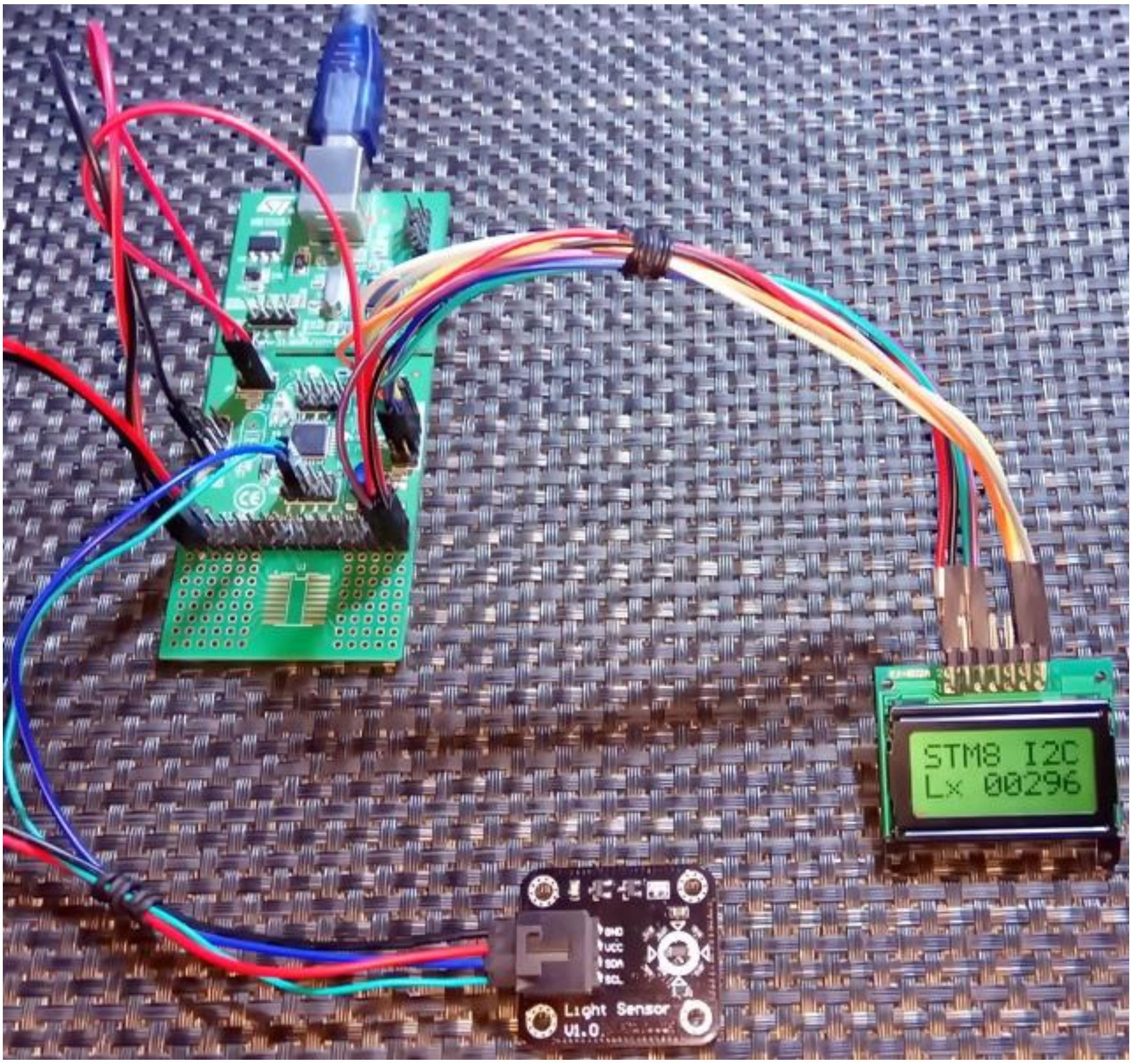
As with SPI, we need to check first if I2C hardware is free. We, then, initiate a I2C start condition and check master/slave mode selection. Next, we send out slave device’s ID or address with read command, signalling that we wish to read from the slave. Again, a flag is checked before continuing. Here the sensor gives 16-bit light output data and so we need to extract two 8-bit data values. This is done in the ***while*** loop. At the end of data extraction process, we must generate stop as well as take care of acknowledgement. Finally, the two bytes are joined to form a word value representing light output.

The following function simplifies the task of determining lux value from the sensor. It selects mode of operation and latency. We will call this function in the main loop to extract average light value in lux.

```
unsigned int get_lux_value(unsigned char mode, unsigned int delay_time)
{
  unsigned long lux_value = 0x00;
  unsigned char dly = 0x00;
  unsigned char s = 0x08;

  while(s)
  {
     BH1750_write(power_up);
     BH1750_write(mode);
     lux_value += BH1750_read_word();
     for(dly = 0; dly < delay_time; dly += 1)
     {
        delay_ms(1);
     }
     BH1750_write(power_down);
```

```
      s--;
   }
   lux_value >>= 3;

   return ((unsigned int)lux_value);
}
```

Demo



Video link: https://youtu.be/bpwki1RCOXU

Some Useful Tips

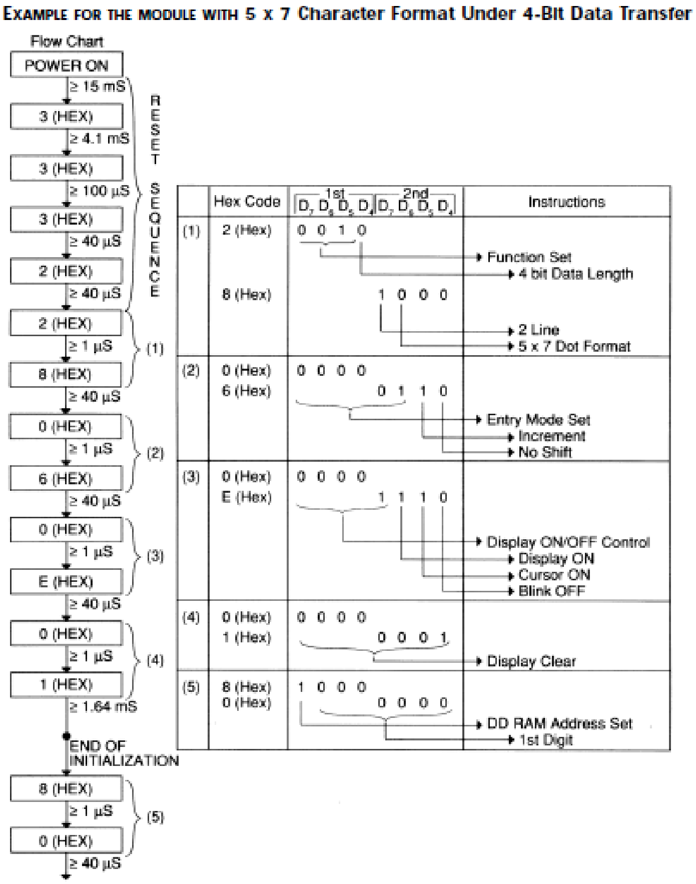
When using a new compiler, I evaluate some certain things. For instance, how do I include my own written library files, interrupt management, what conventions I must follow and what dos and don'ts must be observed.

Creation & Addition of libraries

At some point in working with any microcontroller, you'll need two basic libraries more than anything else. These are LCD and delay libraries. LCDs are great tools for quickly projecting or presenting data apart from debugging a code with a debugger. Similarly, time-wasting delay loops help us slow down things at our liking. Humans are not as fast as machines. Delays can be avoided in many novel ways but delays keep things simple and so are necessities in some areas.

The Standard Peripheral Library only provides libraries for hardware peripherals and surely not for anything else. It is also practically impossible to provide library for all hardware on available on the planet. Thus, whenever when we will be needing new hardware integrations with STM8s, we will have to code and tag our libraries with our projects. So how can we do so?

Earlier in this article I discussed about alphanumerical LCDs and delays. If you check the datasheet of such LCDs, you'll find initialization sequences in some while in others you may also find ready-made codes. These sequences are needed to be translated in code just like what we do with I2C or SPI-based devices. Shown below is such an example:

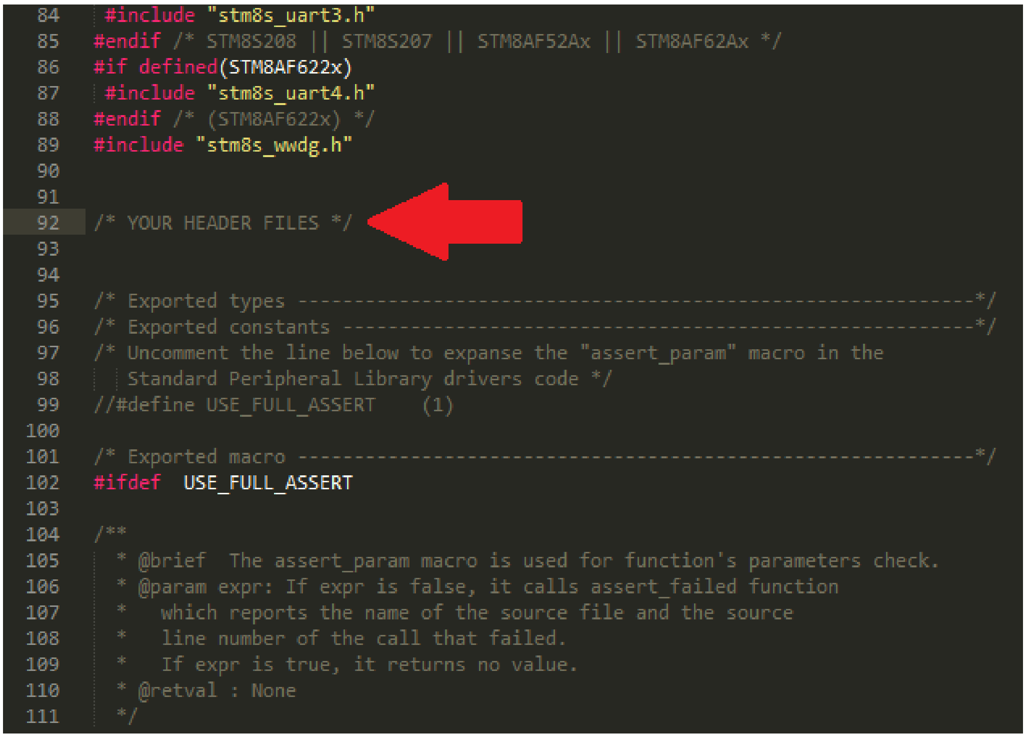


Creating new libraries is simple. Just need to follow the following steps:

- There should be a header file and a source file for every new module. For example, ***lcd.h*** and ***lcd.c***.
- Every header file should start with the inclusion of ***stm8s.h*** header file (***#include "stm8s.h"***). This header is needed because it allows the access to the internal hardware modules available in a STM8 micro. For example, we will need access to GPIOs to develop our LCD library.
- A good practice is that the header files only contain function prototypes, definitions, constants, enumerations and global variables.
- The corresponding source file must only include its header file in beginning.
- The source file should contain the body of codes for all functions declared in the header file.
- When one library is dependent on the functions of another's, the one that will be required in the new library must be included first. For example, we will need delay library in coding the LCD library because there are ***delay_ms*** functions in some parts of the library and so delay library should be included first. This should be the systematic order:

  #include "stm8s_delay.h"
  #include "lcd.h"

  You can include these files at the bottom part of the ***stm8s_conf.h*** header file complying with right precedence as shown below:



```
 #include "stm8s_uart3.h"
#endif /* STM8S208 || STM8S207 || STM8AF52Ax || STM8AF62Ax */
#if defined(STM8AF622x)
 #include "stm8s_uart4.h"
#endif /* (STM8AF622x) */
#include "stm8s_wwdg.h"


/* YOUR HEADER FILES */


/* Exported types ------------------------------------------------------------*/
/* Exported constants --------------------------------------------------------*/
/* Uncomment the line below to expanse the "assert_param" macro in the
   Standard Peripheral Library drivers code */
//#define USE_FULL_ASSERT    (1)

/* Exported macro ------------------------------------------------------------*/
#ifdef  USE_FULL_ASSERT

/**
  * @brief  The assert_param macro is used for function's parameters check.
  * @param expr: If expr is false, it calls assert_failed function
  *   which reports the name of the source file and the source
  *   line number of the call that failed.
  *   If expr is true, it returns no value.
  * @retval : None
  */
```

Alternatively, you can add them after the first line ***#include "stm8s.h"*** in your main source code.

Peripheral Clock Configurations

In most codes revealed so far, I made clock configurations every time. The reasons behind so are

- Selection of right clock source.
- Adjustment of peripheral and system clocks as per requirement. Again, it is mainly intended to balance off both power consumption and overall performance.
- Disabling any unused hardware. This reduces power consumption and help us avoid certain hardware conflicts.

```
void clock_setup(void)
{
   CLK_DeInit();

   CLK_HSECmd(DISABLE);
   CLK_LSICmd(ENABLE);
   while(CLK_GetFlagStatus(CLK_FLAG_LSIRDY) == FALSE);
   CLK_HSICmd(ENABLE);
   while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);

   CLK_ClockSwitchCmd(ENABLE);
   CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);
   CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);

   CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
   DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);

    CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, ENABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
    CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
}
```

The following lines select clock sources:

```
CLK_DeInit();

 CLK_HSECmd(DISABLE);
 CLK_LSICmd(ENABLE);
 while(CLK_GetFlagStatus(CLK_FLAG_LSIRDY) == FALSE);
 CLK_HSICmd(ENABLE);
 while(CLK_GetFlagStatus(CLK_FLAG_HSIRDY) == FALSE);
```

What these lines do are enabling/disabling clock sources and wait for the sources to stabilize.

Then the following lines select clock prescalers and switching:

```
 CLK_ClockSwitchCmd(ENABLE);
 CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);
 CLK_SYSCLKConfig(CLK_PRESCALER_CPUDIV1);

 CLK_ClockSwitchConfig(CLK_SWITCHMODE_AUTO, CLK_SOURCE_HSI,
 DISABLE, CLK_CURRENTCLOCKSTATE_ENABLE);
```

Finally, the last segment enables/disables peripheral clocks:

```
CLK_PeripheralClockConfig(CLK_PERIPHERAL_SPI, DISABLE);
CLK_PeripheralClockConfig(CLK_PERIPHERAL_I2C, DISABLE);
CLK_PeripheralClockConfig(CLK_PERIPHERAL_ADC, DISABLE);
CLK_PeripheralClockConfig(CLK_PERIPHERAL_AWU, DISABLE);
CLK_PeripheralClockConfig(CLK_PERIPHERAL_UART1, DISABLE);
CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER1, DISABLE);
CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER2, DISABLE);
CLK_PeripheralClockConfig(CLK_PERIPHERAL_TIMER4, DISABLE);
```

This segment is very important and should always be rechecked. Different chips have different internal hardware peripheral and so this segment will be different. For instance, STM8S105 has no UART1 module but it has UART2 instead. Research which hardware are available in your target micro and then code this segment. I ended up with several wasted hours of finding trouble in various cases only to find that I didn't enable the required hardware.

Configuring Similar Stuffs Quickly

Sometimes you may end up doing the same stuff over and over again while you could have done it simply with one or two lines of code. For example, in the LCD library, the GPIOs that connect with the LCD share the same configurations. All are fast push-pull outputs.

```
GPIO_Init(LCD_PORT, LCD_RS, GPIO_MODE_OUT_PP_HIGH_FAST);
GPIO_Init(LCD_PORT, LCD_EN, GPIO_MODE_OUT_PP_HIGH_FAST);
GPIO_Init(LCD_PORT, LCD_DB4, GPIO_MODE_OUT_PP_HIGH_FAST);
GPIO_Init(LCD_PORT, LCD_DB5, GPIO_MODE_OUT_PP_HIGH_FAST);
GPIO_Init(LCD_PORT, LCD_DB6, GPIO_MODE_OUT_PP_HIGH_FAST);
GPIO_Init(LCD_PORT, LCD_DB7, GPIO_MODE_OUT_PP_HIGH_FAST);
```

This can be done in a more simplistic manner with just one line of code:

```
GPIO_Init(LCD_PORT, ((GPIO_Pin_TypeDef)(LCD_RS | LCD_EN | LCD_DB4 | LCD_DB5 | LCD_DB6 | LCD_DB7)),
GPIO_MODE_OUT_PP_HIGH_FAST);
```

As you can see it is just a bunch of logical OR operation. The same method is applicable for other peripherals that share the same initialization function.

Some Stuffs About Cosmic C and SPL

- Functions, variables and definitions in Cosmic C are case sensitive.

- Functions with no arguments must not have empty argument areas. For example, you cannot write:

  ```
  void setup ();
  ```

  You should write it as:

  ```
  void setup (void);
  ```

- Definitions and constants can be declared as with any C compiler.

- Wherever there are flags, you need to be careful. You should check and clear flags even if it is cleared by hardware. For instance, when reading ADC, the ADC ***End-Of-Conversion (EOC)*** flag is automatically cleared but still in the code you should check and clear it.

  ```
  ADC1_ClearFlag(ADC1_FLAG_EOC);
  ```

  Flags are so important that unless you check and clear them appropriately, you may not get the right result from your code. Personally, I didn't care much until I got myself into trouble.

- You can mix assembly codes with your C code to enhance performance and optimization. However, you need to have sound knowledge of the assembly instructions. This is a rare requirement. The delay library, for instance, uses no operation assembly instruction to achieve delays. This is written as shown:

  ```
  _asm ("nop");
  ```

- Empty loops are ignored by the compiler as a part of code optimization.

- Long ago, Atmel (now Microchip) published a document regarding ways to efficiently optimize C coding. This document holds true for most microcontrollers. For example, in that document it is stated that a decrementing ***do-while*** loop is much more faster and code efficient than an incrementing ***do-while*** loop. You can apply the methods presented there and other similar tricks with STM8 microcontrollers too. The document can be found here: http://www.atmel.com/images/doc8453.pdf

- Though I don't feel it as a necessity and don't recommend it, you can avoid using the SPL and still code by raw register level access. For example, you can blink a LED with the following code:

  ```
  #include "stm8s.h"


  void main (void)
  {
     GPIOD->DDR |= 0x01;
     GPIOD->CR1 |= 0x01;

      for(;;)
     {
       GPIOD->ODR ^= (1 << 0);
       delay_ms(100);
     };
  }
  ```

  As you can see it is both tedious and meaningless unless you comment every single line.

- Don't mess with configuration (fuse) bits unless needed. Most of the times you will never have to deal with them. ***AFRs*** are used when remapping is needed or to enable special GPIO functionality. These will be of most importance.

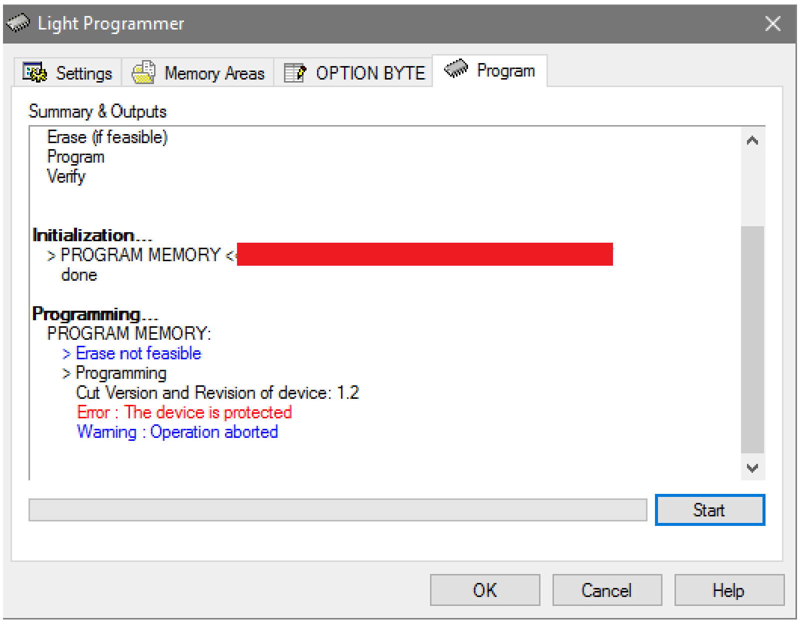
- Bitwise and logic operations are useful. Not only they are fast, they just deal with the designated bits only. SPL has support for such operations but it is still better to know them. Here are some common operations:

```
#define bit_set(reg, bit_val)     reg |= (1 << bit_val)       //For setting a bit of a register
#define bit_clr(reg, bit_val)     reg &= (~(1 << bit_val))    //For clearing a bit of a register
#define bit_tgl(reg, bit_val)     reg ^= (1 << bit_val)       //For toggling a bit of a register
#define get_bit(reg, bit_val)     (reg & (1 << bit_val))      //For extracting the bit state of a register
#define get_reg(reg, msk)         (reg & msk)                 //For extracting the states of masked bits of a register
```

Unlocking a Locked STM8 Chip

If you have accidentally locked a STM8 chip by setting the ***Readout Protection*** configuration bit and no longer able to use it, you can unlock it easily.

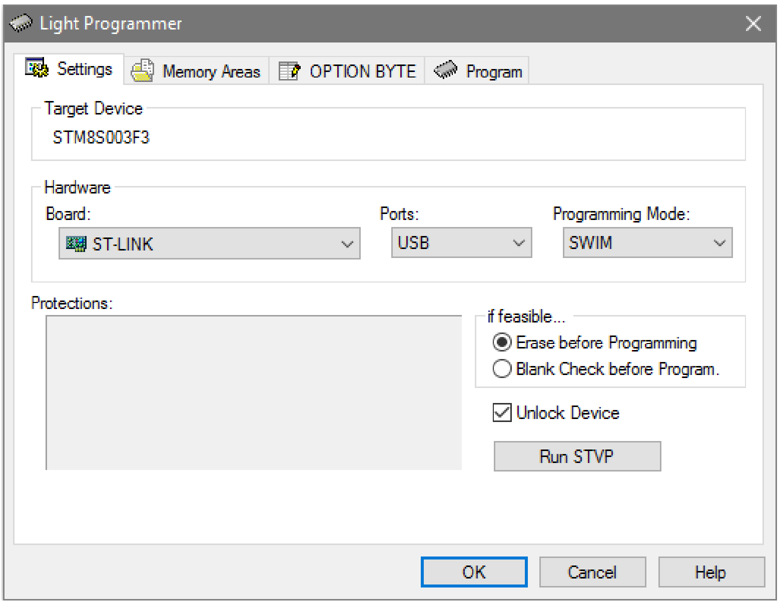
When you lock a chip, the programmer interface will give you a warning notification. If you retry to reprogram/erase a locked chip you'll get an error like this:



```
 Erase (if feasible)
 Program
 Verify

Initialization...
 > PROGRAM MEMORY <[redacted]
   done

Programming...
 PROGRAM MEMORY:
   > Erase not feasible
   > Programming
     Cut Version and Revision of device: 1.2
     Error : The device is protected
     Warning : Operation aborted
```

No matter what you do, you won't be able to use it.

To unlock, go to the light programmer interface of STVD and check the ***Unlock Device*** checkbox as shown below:



Light Programmer

Target Device
STM8S003F3

Hardware
Board: ST-LINK
Ports: USB
Programming Mode: SWIM

Protections:

if feasible...
● Erase before Programming
○ Blank Check before Program.

☑ Unlock Device

Also select ***Erase before Programming*** radio button because it is highly likely that your target chip is not empty. Now once you retry to reprogram, it will get unlocked.

Mastering C Language

You need not to be a C whiz to work with microcontrollers but certain things will surely help you to resolve some critical problems with simple codes. You must check supported data types whenever you begin working in a new development environment and should always use unsigned-signed designations to avoid unnecessary mistakes. Likewise, variable size is also important. Pointer, structures, unions and arrays are helpful features of C-language. You must learn how to use and apply them successfully. Without these you can still work but things will look really dirty. When coding for a new work, you must try to settle what you wish from your system and how should it behave. There should be an organized workflow and thereby your code will automatically be formulated in a state-of-machine algorithm or as a real-time system. You must try to avoid delays and loops wherever possible. Try to avoid polling and use interrupt-based systems. This will make your device behave in real-time with zero latency. However, you must be careful in handling interrupts because interrupts within interrupts will cause your system to crash miserably. Functions make things modular and thus easy to modify or debug. Repeated tasks should be placed in functions. A blinking LED code may look simple and stupid but sometimes very useful for testing stuffs. Some basic knowledge on mathematics and algorithms are also requirements for becoming a good embedded-system specialist.

Epilogue

In the end, I would like to share that my tiny raw-level knowledge and experiences with STM32s (http://embedded-lab.com/blog/stm32-tutorials/) earlier paid off handsomely. Due to that I was able to compiler this article decently and quickly. Personally, I feel that whosoever knows STM8 micros well will master STM32s and vice-versa because except the cores all hardware in both architectures are not just similar but same sometimes. This is for the first time I have admired STM's SPL. My experiences with STM32 SPL was not well as so I decided to go on my own. However, this time things were different. Things were joyful and less difficult.

I would like to thank a few people who influenced me in composing this article:

- Firstly, my friends and acquaintances. I wanted to help them out and that drove me to dig things deep.

- **Mr. Ben Ryves** (benryves.com/tutorials/stm8s-discovery/).
  Though his methods are different than mine, his article guided me a lot in the beginning. It is perhaps the most popular site for tutorials on STM8s and well organized. He used STM8S105 Discovery.

- **Mark Stevens** (http://blog.mark-stevens.co.uk)
  His tutorials on STM8 are not based on SPL. He showed stuffs with raw-level register access and with IAR compiler. Still his blog is informative.

- http://www.emcu.it/. This Italian site was helpful in getting some early info.

- STMicroelectronics team for releasing the STM8CubeMX during my writeup.

- Cosmic team for freeing up their C compiler.

I spent nearly three months straight putting together all these things and at present, I must say that I have great expectations from STM8 microcontrollers.

Happy coding.

Author: Shawon M. Shahryiar
https://www.facebook.com/groups/microarena
https://www.facebook.com/MicroArena *24.04.2017*