Dept. of Radio Electronics

Faculty of Electrical Engineering and Communication Technologies

Brno University of Technology

Microcontrollers and Embedded Systems
Laboratory Exercises

doc. Ing. Aleš Povalač, Ph.D.

October 2024

Table of Contents

1 STM32CubeIDE, Git, basic functions 3

2 ISR, bouncing buttons, SysTick timer 7

3 LED display on a shift register 11

4 A/D converter and bargraph 15

5 UART communication with DMA, EEPROM on I2C 18

6 DS18B20 and NTC temperature sensors 23

7 FreeRTOS and accelerometer operation 26

8 Matrix keypad and code lock 30

9 HID USB mouse 37

10 Ethernet, TCP client and server 41

1 STM32CubeIDE, Git, basic functions

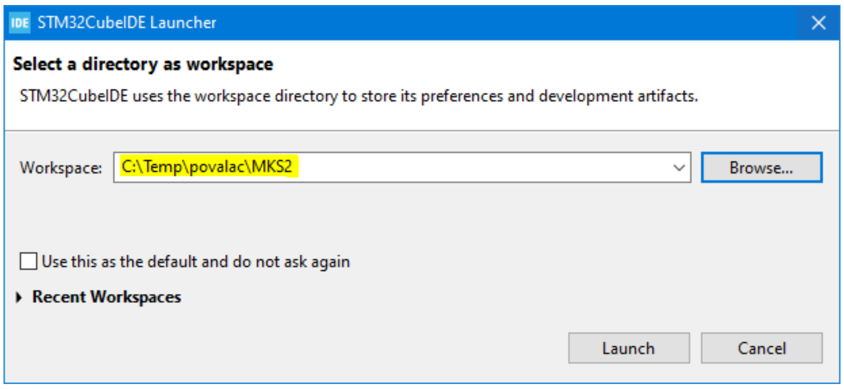
1.1 Assignment

- Get to know TortoiseGit. Create a repository on Github, make a working copy of the directory (Git Clone).
- Learn about the NUCLEO-F030R8 development board, the STMrel module and the STM32CubeIDE environment.
- Create a simple application that will flash the LD2 diode. Test on the development board, verify the debugging by stepping. Save the created application to the repository (Git Commit).
- Create an application that will implement the blinking of the LD2 diode according to a predefined string in the Morse code. Commit to the repository. Then modify the code so that the Morse code sequence is written in a single 32-bit constant, commit to the repository again.
- Save the local repository to the server (Git Push).

1.2 Instructions

1.2.1 <u>Basic acquaintance</u>

- Create a Github account, create a new repository called e.g. MKS.
- Create your working directory in **C:\Temp** and use Git Clone to create a working copy of your personal repository. To sign in to Github, use the Sign in with your browser option.
- Switch the STM32CubeIDE workspace to this working copy (at startup or via File / Switch Workspace).

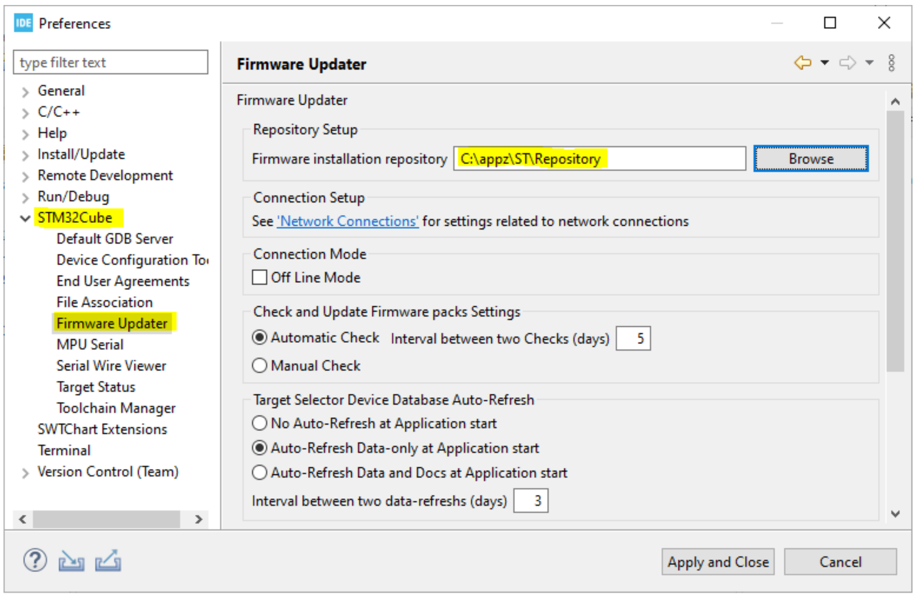

STM32CubeIDE Launcher

Select a directory as workspace

STM32CubeIDE uses the workspace directory to store its preferences and development artifacts.

Workspace: C:\Temp\povalac\MKS2

- From now on, work only on the working copy.
- The necessary schematics for the development board can be found in eLearning. We will not use the STMrel module in this exercise.
- The STM32CubeIDE environment stores a lot of data locally, especially library repositories. To avoid having to download everything again, it is advisable to set the path to the central library repository (valid for classrooms, not private PCs) in Window / Preferences / STM32Cube / Firmware Updater:



Firmware Updater

Firmware Updater

Repository Setup

Firmware installation repository C:\appz\ST\Repository

Connection Setup

See 'Network Connections' for settings related to network connections

Connection Mode

☐ Off Line Mode

Check and Update Firmware packs Settings

● Automatic Check Interval between two Checks (days) 5

○ Manual Check

Target Selector Device Database Auto-Refresh

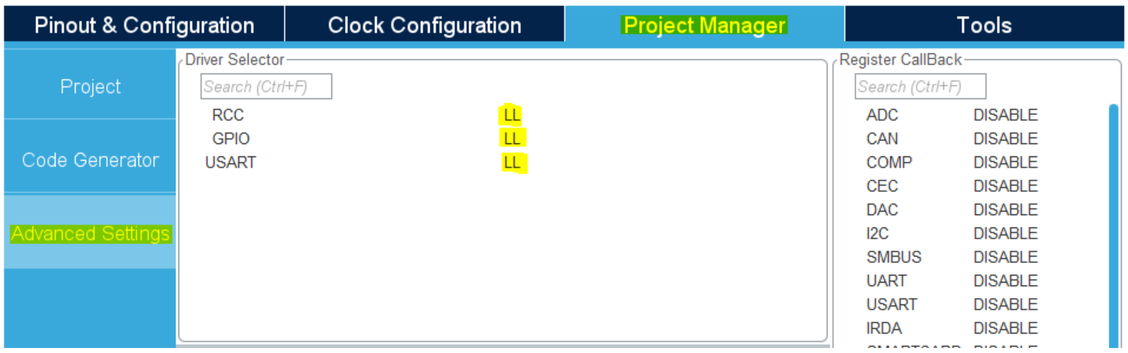
○ No Auto-Refresh at Application start

● Auto-Refresh Data-only at Application start

○ Auto-Refresh Data and Docs at Application start

Interval between two data-refreshs (days) 3

- A new project with libraries is created via File / New / STM32 Project / **Board** Selector / NUCLEO-F030R8. For the first few exercises we will **use the LL libraries**. Confirm the initialization of all peripherals to the default settings. To switch libraries from the complex HAL (which we will use later) to the more compact LL, go to Project Manager / Advanced Settings / Driver Selector, and select LL for all drivers:



Driver Selector

RCC	LL
GPIO	LL
USART	LL

Register CallBack

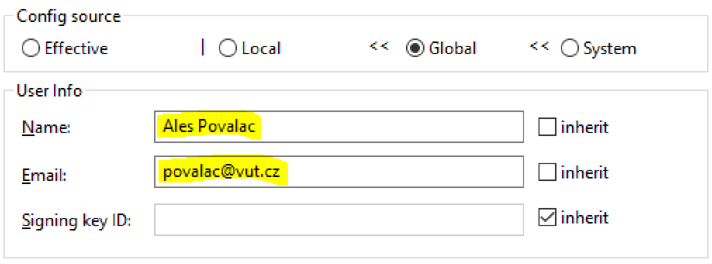
ADC	DISABLE
CAN	DISABLE
COMP	DISABLE
CEC	DISABLE
DAC	DISABLE
I2C	DISABLE
SMBUS	DISABLE
UART	DISABLE
USART	DISABLE
IRDA	DISABLE

- After saving the IOC file, the code is automatically generated, or you can start the generation manually using Project / Generate Code (Alt+K).
- The compilation (Build all) can be started using the shortcut Ctrl+B, debugging the project (i.e. application start, Debug) using F11.
- Confirm the default debugger configuration when you start it for the first time. If the debugger firmware on the connected development board is out of date, an upgrade may be requested.
- You can use keyboard shortcuts during debugging steps, see the Run menu.

1.2.2 Versioning and TortoiseGit

- Always commit only the source code (c, h, Makefile, etc.), not the results of the compilation (o, d, elf, hex, workspace files, etc.). An ignore list can be used, a `.gitignore` file can be created manually or using Git / Add to ignore list. It is preferable to ignore the Debug folders in individual projects (i.e. `/*/Debug`) and the `.metadata` folder.

- Commit the working copy to the repository using Git Commit. Git will first prompt for a name and email.



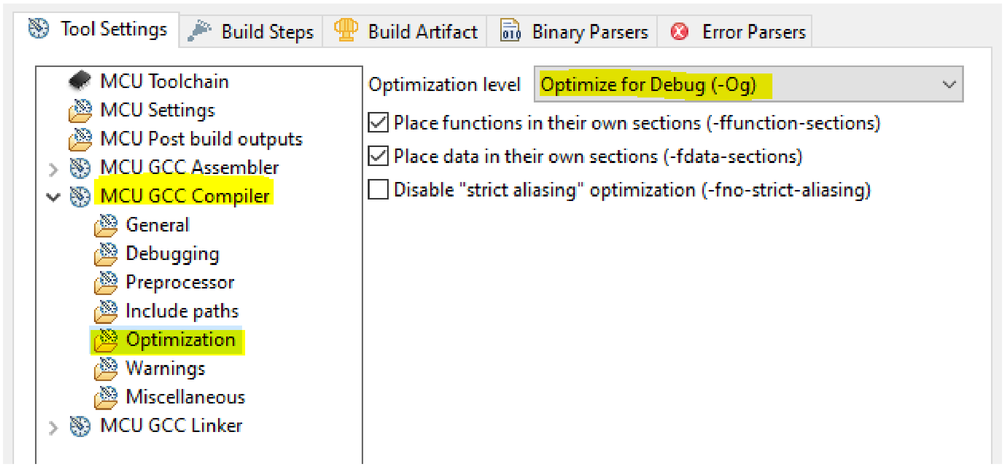
Config source
( ) Effective | ( ) Local << (●) Global << ( ) System

User Info
Name: Ales Povalac ☐ inherit
Email: povalac@vut.cz ☐ inherit
Signing key ID: ☑ inherit

- Save the local repository to the server (Git Push). If it requires a Github login, use the Sign in with your browser option. See the result on Github.

1.2.3 LED flashing

- If you have confirmed the initialization of the peripherals to the default settings, the LD2 LED on the development kit is correctly configured. Remember to switch the libraries from HAL to LL.
- After saving the IOC file, the code is automatically generated, or you can start the generation manually using Project / Generate Code (Alt+K).
- A good practice for larger projects is to enable compiler optimization as soon as the project is created. The item is fairly deep in the settings: Project / Properties / C/C++ **Build** / Settings / MCU GCC **Compiler** / Optimization / Optimization Level, the recommended setting is Optimize for Debug (-Og).



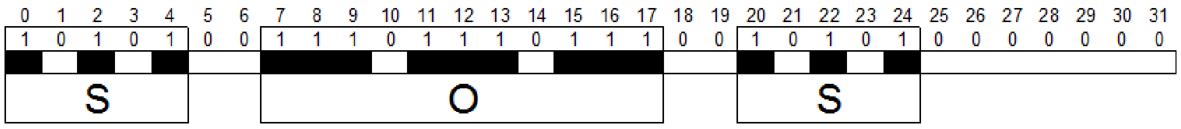
Optimization level: Optimize for Debug (-Og)
☑ Place functions in their own sections (-ffunction-sections)
☑ Place data in their own sections (-fdata-sections)
☐ Disable "strict aliasing" optimization (-fno-strict-aliasing)

- In the main.c file, always add your own code between the `USER CODE BEGIN` and `USER CODE END` comments. If you break this rule, your code will be deleted when the project is regenerated from CubeMX.
- The peripheral initializations are automatically generated, so it is necessary to only add the code for the LED flashing. This will be located in the `while (1)` loop, add it before the comment `/* USER CODE END WHILE */`.
- Output pin level changes can be made using LL functions:
  - `LL_GPIO_SetOutputPin(LD2_GPIO_Port, LD2_Pin);`
  - `LL_GPIO_ResetOutputPin(LD2_GPIO_Port, LD2_Pin);`

- These functions use the generated definitions from the main.h file:
  - `#define LD2_GPIO_Port GPIOA`
  - `#define LD2_Pin        LL_GPIO_PIN_5`
- Similarly, direct access to the corresponding GPIO peripheral can be used using the BSRR and BRR registers:
  - `GPIOA->BSRR = (1<<5); // set`
  - `GPIOA->BRR = (1<<5);  // reset`
- Waiting can be carried out using **`LL_mDelay(200)`**, the function takes the delay time in milliseconds as a parameter.

1.2.4 Flashing in Morse code

- The development board will use LD2 to flash the code "SOS" in Morse code.
- You can store the sequence as ones (LED on) and zeros (LED off) e.g. in an array of type **`uint8_t array[32]`** and initialize it directly in the code.



- In addition to the array itself, you will need an iteration variable to point to this array. If the field at the index according to this variable is 0, we will set the output for LD2 to log. 0, if it is non-zero to log. 1.
- Commit the working copy to Git.
- Memory usage optimization: the sequence will be stored in the **`uint32_t`** type, use binary notation of elements (e.g. `0b0100`) to achieve the same functionality. You will need bitwise operations and bit shifts.
- Commit the working copy to Git.
- Finally, don't forget to push the local repository to the server (Git Push).

2 ISR, bouncing buttons, SysTick timer

2.1 Assignment

- Create a new project, configure LED1 and LED2 as output, S1 and S2 as input with internal pullup. Enable the falling edge triggered interrupt on EXTI0 (button S2), create an ISR that will invert LED2 state.
- Create a code that will flash LED1 with a period of 300ms. Use timing based on the SysTick timer. Write the LED1 handler to the blink() function, call this function from main().
- Create a non-blocking button handler that will read them with a period of 40 ms. The status of LED2 will be based on the button event – after pressing S2 it will be on light up for 100ms, after pressing S1 for 1s. Write the handler in the button() function, add its call to main().
- Modify the S1 operation so that the button is sampled with a period of 5 ms. Shift the state of the S1 button using the bit shift left to a static 16-bit variable, decide to light LED2 based on the state 0x7FFF, which indicates the button is released without further bouncing.
- Source code formatting will also be evaluated, especially block indentation. There must be no global variables in the main.c module except Tick, the main() code may only call the blink() and button() tasks in the infinite loop.

2.2 Instructions

2.2.1 Basic acquaintance

- Create a working copy of your repository from Github (Git Clone) or update your repository from the server (Git Pull).
- Create the basic structure using the procedure from the first exercise, i.e. via File / New / STM32 Project / **Board** Selector / NUCLEO-F030R8. Confirm the initialization of all peripherals to the default settings. **Do not forget to change the libraries to LL.** Commit each completed point.
- We will use the STMrel extension module with the following IOC settings:
  - LED1 (left) = GPIO_Output @ PA4
  - LED2 (right) = GPIO_Output @ PB0
  - S1 (right) = GPIO_Input @ PC1 **+ Pull-up**
  - S2 (left) = GPIO_EXTI0 @ PC0 **+ Pull-up**

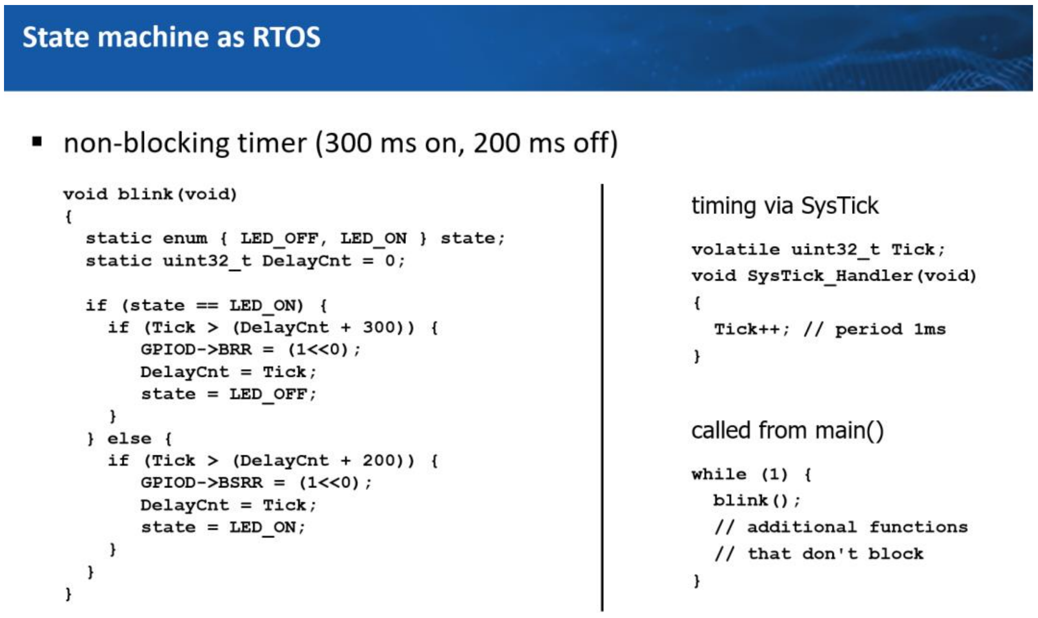
2.2.2 External interrupts

- Set the appropriate pins (left mouse button on the pin, select GPIO_Output, GPIO_Input or GPIO_EXTI0 as required) and name them (right mouse button, Enter User Label, insert name). For the button pins, enable pull-ups via System Core / GPIO / GPIO / PC0 (then PC1) / GPIO Pull-up/Pull-down, set to Pull-up.
- In addition to the GPIO configuration, you must enable interrupts from EXTI0, i.e. check System Core / NVIC / EXTI line 0 and 1 interrupts.

- The interrupt event is handled by the **`EXTI0_1_IRQHandler()`** in the stm32f0xx_it.c file, which after checking the interrupt source clears the pending bit and performs the action, in our case negating the LED2 state. In the **`USER CODE BEGIN LL_EXTI_LINE_0`** section, we need to add the LED toggle using **`LL_GPIO_TogglePin()`**.
- The infinite loop in the main() function will be empty. Commit the completed point.

2.2.3 <u>SysTick timer and non-blocking waiting</u>

- The SysTick timer will be used for the time base, all tasks (functions) in the main() superloop will be non-blocking.



State machine as RTOS

- non-blocking timer (300 ms on, 200 ms off)

```
void blink(void)
{
  static enum { LED_OFF, LED_ON } state;
  static uint32_t DelayCnt = 0;

  if (state == LED_ON) {
    if (Tick > (DelayCnt + 300)) {
       GPIOD->BRR = (1<<0);
       DelayCnt = Tick;
       state = LED_OFF;
    }
  } else {
    if (Tick > (DelayCnt + 200)) {
       GPIOD->BSRR = (1<<0);
       DelayCnt = Tick;
       state = LED_ON;
    }
  }
}
```

timing via SysTick

```
volatile uint32_t Tick;
void SysTick_Handler(void)
{
  Tick++; // period 1ms
}
```

called from main()

```
while (1) {
  blink();
  // additional functions
  // that don't block
}
```

- At the main.c module level, in the appropriate **`USER CODE`** section, create a global **`uint32_t`** variable Tick, marking it as volatile due to access from the ISR and the main code:

  ```
  volatile uint32_t Tick;
  ```

- The variable will also be used from another module, so it must be added to the main.h header file:

  ```
  extern volatile uint32_t Tick;
  ```

- In the stm32f0xx_it.c module we add its incrementation in the **`SysTick_Handler()`** interrupt handler. Since this module includes the main.h header file, it can work with the global variable:

  ```
  Tick++;
  ```

- Since LL libraries do not enable SysTick interrupt handling by default, it must be enabled as part of the **`USER CODE BEGIN 2`** initialization:

  ```
  LL_SYSTICK_EnableIT();
  ```

- The **`blink()`** task will simply handle the operation of LED1 in a non-blocking way, i.e. its toggling. Define the **`LED_TIME_BLINK`** symbol (using **`#define`** in the **`USER CODE BEGIN PD`** section) to the value 300.

```
void blink(void)
{
  static uint32_t delay;

  if (Tick > delay + LED_TIME_BLINK) {
    LL_GPIO_TogglePin(...);
    delay = Tick;
  }
}
```

2.2.4 Handling the buttons

- Switch the configuration of the S2 button from EXTI0 to GPIO_Input (with pull-up – it must be **switched on again** and the pin renamed again), generate the code.
- Create a function **button()**. The function will use non-blocking waiting similar to the blink task, every 40ms it will sample the pins connected to the buttons and if they are pressed it will light LED2 for a defined time. Example of S2 read, which will light LED2 for the duration of **LED_TIME_SHORT**, i.e. 100ms:

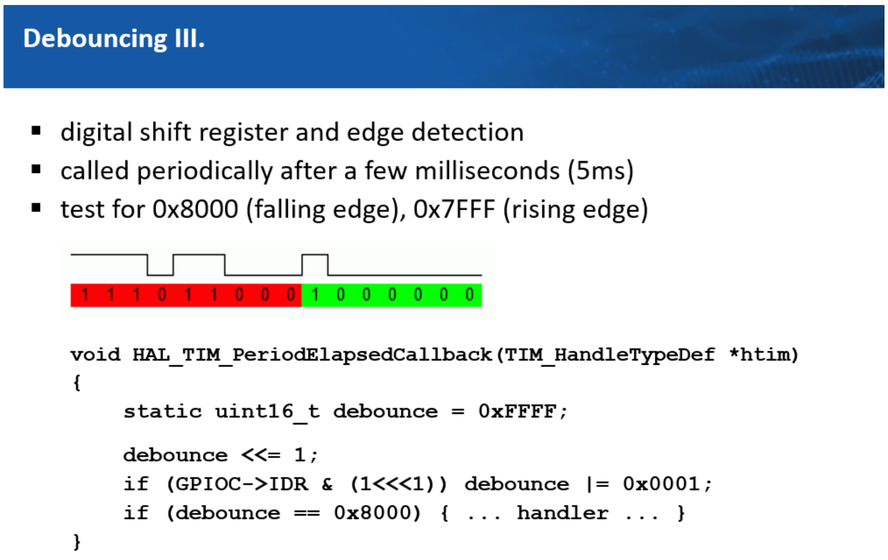
```
static uint32_t old_s2;
uint32_t new_s2 = LL_GPIO_IsInputPinSet(S2_GPIO_Port, S2_Pin);

if (old_s2 && !new_s2) { // falling edge
  off_time = Tick + LED_TIME_SHORT;
  LL_GPIO_SetOutputPin(LED2_GPIO_Port, LED2_Pin);
}
old_s2 = new_s2;
```

- The LED is turned off by a condition that tests whether the defined time has elapsed:

```
static uint32_t off_time;

if (Tick > off_time) {
  LL_GPIO_ResetOutputPin(LED2_GPIO_Port, LED2_Pin);
}
```

- Fulfill the code with the S1 button operation, which lights LED2 for the **LED_TIME_LONG** time, i.e. 1000ms.
- Make a commit working copy to Git.
- The basic debouncing with simple reading after a defined time (here 40 ms) is used for low-demanding applications, typically for user interface inputs where exceptional bounces are not critical.

2.2.5 <u>Advanced debouncing</u>



Debouncing III.

- digital shift register and edge detection
- called periodically after a few milliseconds (5ms)
- test for 0x8000 (falling edge), 0x7FFF (rising edge)

1 1 1 0 1 1 0 0 0 1 0 0 0 0 0 0

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    static uint16_t debounce = 0xFFFF;

    debounce <<= 1;
    if (GPIOC->IDR & (1<<1)) debounce |= 0x0001;
    if (debounce == 0x8000) { ... handler ... }
}
```

- Modify the button() function to include a part called with a period of 5ms. Define a static variable:

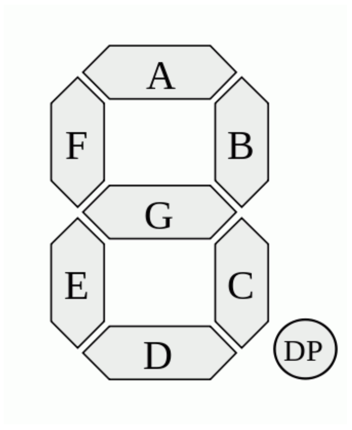
```
static uint16_t debounce = 0xFFFF;
```

- Perform a bit rotation to the left every 5ms and set the lowest bit (LSB) to 1 if the corresponding GPIO pin of the S1 button is in log. 1.
- Test the variable to the value 0x7FFF, light LED2 for the defined time in case of a match. The value 0x7FFF indicates that the button was switched on (MSB is zero) and then was off for a minimum of 15x 5ms = 75ms without bounces. This method of debouncing is used for significantly bouncing buttons, or for inputs where the bouncing causes a major problem that cannot be easily corrected by user feedback.
- Commit the working copy to Git, save the repository using Git Push.

3 LED display on a shift register

3.1 Assignment

- Create a driver for a pair of SCT2024 shift registers on the extension board. The driver will be in an independent module for separate compilation. It will contain at least the global functions **`sct_init()`** for hardware initialization and **`sct_led(uint32_t value)`** for setting the light of each segment and LED according to the bit positions of the **`value`**. Test the driver.
- Create seven-segment characters for digits 0 to 9 at each position. Make the function **`sct_value(uint16_t value)`** to display a number in the range 000-999. The display will show values from 0 to 999 in increments of 111.



- Add a rotary encoder to change the value displayed on the seven-segment display. The minimum encoder value on the display will be 0, the maximum 150.
- Source code formatting will also be evaluated, especially block indentation.

3.2 Instructions

3.2.1 Basic acquaintance

- Create a working copy of your repository from Github (Git Clone) or update your repository from the server (Git Pull).
- Create the basic structure using the procedure from the first exercise, i.e. via File / New / STM32 Project / **Board** Selector / NUCLEO-F030R8. Confirm the initialization of all peripherals to the default settings. Since we will be using HAL libraries, **keep** the driver selection settings **at the default HAL**.
- **Commit each completed item.**

3.2.2 Driver for shift register

- Set the corresponding pins connected to the SCT register as output (left button on the pin, select GPIO_Output) and name them (right button, Enter User Label, insert name). Use the following naming convention:
    - SCT_NLA @ PB5
    - SCT_SDI @ PB4
    - SCT_CLK @ PB3
    - SCT_NOE @ PB10
- Generate the code (just save the CubeMX file or Project / Generation Code).
- The driver will consist of sct.c and sct.h files, which you create using File / New / Source/Header File.

- All initialization is provided by the code generated by CubeMX. Therefore, only the **`sct_led(0)`** call will be in the **`sct_init()`** function. Add main.h to the included files.
- The function **`void sct_led(uint32_t value)`** writes to the shift registers so that bits 0 to 31 of the **`value`** correspond to the individual seven-segment LEDs on the board. Steps:
    - shift bits from **`value`** to SDI (to the right, i.e. start with the LSB)
    - then generate a pulse on CLK
    - after sending all 32 bits, write to the output registers by a pulse on /LA
- Details can be found in the datasheet SCT2024, especially on page 5. The function will use HAL calls to access GPIOs with symbolic names defined in main.h, e.g.:

```
HAL_GPIO_WritePin(SCT_CLK_GPIO_Port, SCT_CLK_Pin, 1);
```

- Always add your own code in the main.c file between the **`USER CODE BEGIN`** and **`USER CODE END`** comments. If you break this rule, your code will be deleted when the project is regenerated from CubeMX.
- Use the constant 0x7A5C36DE for testing. The display will show "bYE" and LEDs will alternately light up. Put the code in the **`USER CODE 2`** section, we will use the **`HAL_Delay()`** function instead of an empty loop:

```
sct_init();
sct_led(0x7A5C36DE);
HAL_Delay(1000);
```

3.2.3 Display of digits and values

- Create a function **`void sct_value(uint16_t value)`** that writes the **`value`** to the display.
- To translate the digits to the corresponding segments connected to the shift registers, it is convenient to use a table that can look like this:

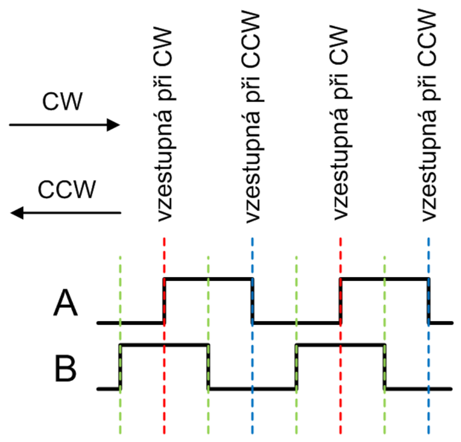
```
static const uint32_t reg_values[3][10] = {
	{
		//PCDE--------GFAB @ DIS1
		0b0111000000000111 << 16,
		0b0100000000000001 << 16,
		0b0011000000001011 << 16,
		0b0110000000001011 << 16,
		0b0100000000001101 << 16,
		0b0110000000001110 << 16,
		0b0111000000001110 << 16,
		0b0100000000000011 << 16,
		0b0111000000001111 << 16,
		0b0110000000001111 << 16,
	},
	{
		//----PCDEGFAB---- @ DIS2
		0b0000011101110000 << 0,
		0b0000010000010000 << 0,
		0b0000001110110000 << 0,
		0b0000011010110000 << 0,
		0b0000010011010000 << 0,
		0b0000011011100000 << 0,
		0b0000011111100000 << 0,
                0b0000010000110000 << 0,
                0b0000011111110000 << 0,
                0b0000011011110000 << 0,
        },
        {
                //PCDE--------GFAB @ DIS3
                0b0111000000000111 << 0,
                0b0100000000000001 << 0,
                0b0011000000001011 << 0,
                0b0110000000001011 << 0,
                0b0100000000001101 << 0,
                0b0110000000001110 << 0,
                0b0111000000001110 << 0,
                0b0100000000000011 << 0,
                0b0111000000001111 << 0,
                0b0110000000001111 << 0,
        },
};
```

- Add individual digits with OR to a 32-bit variable, e.g. for the hundreds position use:

  ```
  reg |= reg_values[0][value / 100 % 10];
  ```

- Test in the main loop (i.e. in **USER CODE 3**) by counting from 0 to 999 in increments of 111. Commit the working copy to Git.

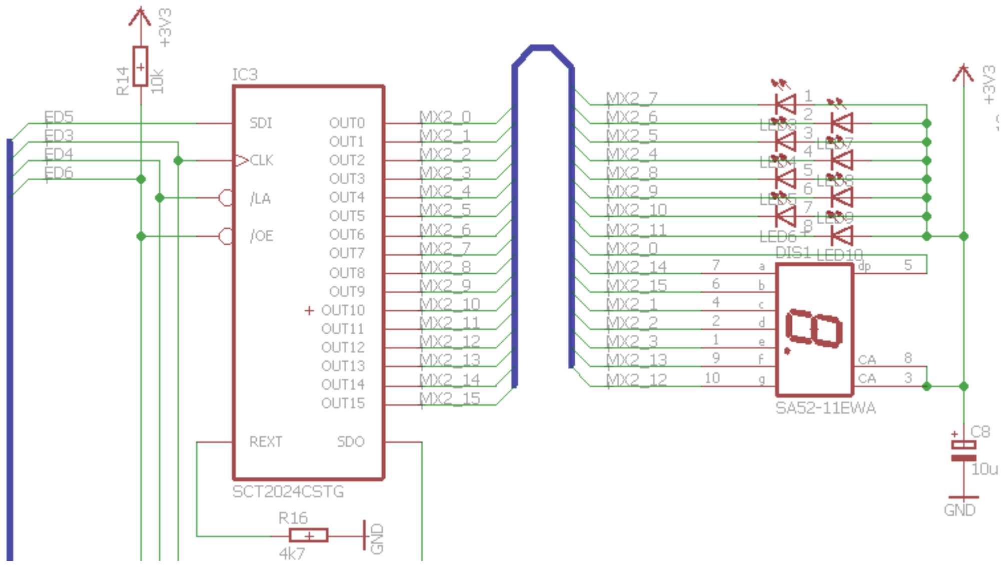
3.2.4 Rotary encoder

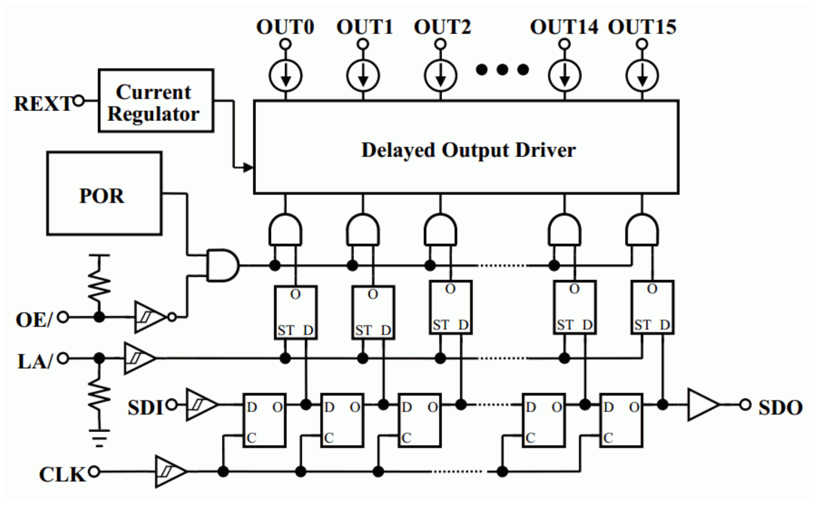


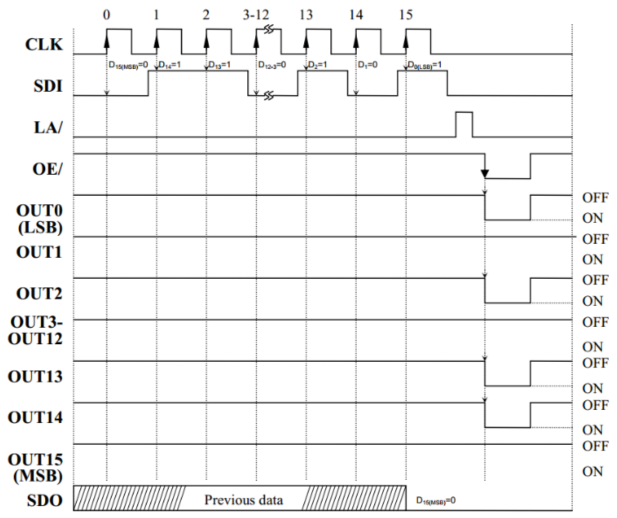
- The hardware will take care of reading the pulses from the rotary encoder. In CubeMX, select the TIM1 timer and switch Combined Channels to Encoder Mode. This will activate pins PA8 and PA9 to which the encoder is connected.
- To make the encoder count in the right direction, you need to invert one of the signals – for channel 1 set Polarity to Falling Edge. Since we want to count up to 150, set this limitation – Counter Period (AutoReload) to 150.
- Generate the code.
- The counter in encoder mode needs to be started. Add a call to the **USER CODE 2** section:

  ```
  HAL_TIM_Encoder_Start(&htim1, htim1.Channel);
  ```

- In the main loop, comment out the display test and insert code to update the display with the current counter value, which can be obtained by calling **__HAL_TIM_GET_COUNTER(&htim1)**. Complete the infinite loop with a short wait (about 50ms).
- Commit the working copy to Git, save the repository using Git Push.


+3V3
R14
10k
IC3
ED5
ED3
ED4
ED6
SDI
CLK
/LA
/OE
OUT0
OUT1
OUT2
OUT3
OUT4
OUT5
OUT6
OUT7
OUT8
OUT9
OUT10
OUT11
OUT12
OUT13
OUT14
OUT15
REXT
SDO
SCT2024CSTG
R16
4k7
GND
MX2_0
MX2_1
MX2_2
MX2_3
MX2_4
MX2_5
MX2_6
MX2_7
MX2_8
MX2_9
MX2_10
MX2_11
MX2_12
MX2_13
MX2_14
MX2_15
LED3
LED4
LED5
LED6
LED7
LED8
LED9
LED10
DIS1
SA52-11EWA
a
b
c
d
e
f
g
dp
CA
C8
10u
+3V3
GND


OUT0
OUT1
OUT2
OUT14
OUT15
REXT
Current Regulator
Delayed Output Driver
POR
OE/
LA/
SDI
CLK
SDO
O
ST D
D O
C


0
1
2
3-12
13
14
15
CLK
SDI
LA/
OE/
OUT0 (LSB)
OUT1
OUT2
OUT3-OUT12
OUT13
OUT14
OUT15 (MSB)
SDO
OFF
ON
Previous data
$D_{15(MSB)}=0$
$D_{14}=1$
$D_{13}=1$
$D_{12-3}=0$
$D_{2}=1$
$D_{1}=0$
$D_{0(LSB)}=1$

4 A/D converter and bargraph

4.1 Assignment

- Configure the ADC on the development board to read data from trimmer R2. Using the seven-segment display driver, show the trimmer position converted to the 0-500 range on the display. Modify the driver to make the LED above the display an eight-segment bargraph. The progression of the bargraph will be linear, it must support values from minimum to maximum.
- Use exponential smoothing for the ADC value. Choose a Q factor of about 12.
- Add an internal reference and an internal temperature sensor to the ADC channels. Press S1 to display the calculated AVCC supply voltage with two decimal places. Press S2 to display the microcontroller temperature in °C. When the button is released, the reading remains displayed for 1 second, then it returns to the value of trimmer R2.
- Source code formatting will also be evaluated, especially block indentation.

4.2 Instructions

4.2.1 Basic acquaintance

- Create a working copy of your repository from Github (Git Clone) or update your repository from the server (Git Pull).
- Create a new project via File / New / STM32 Project / Board Selector / NUCLEO-F030R8. We will be using the HAL libraries, so leave the Targeted Project Type at STM32Cube. Confirm initialization of all peripherals to default settings.

4.2.2 ADC configuration and readout

- Activate the corresponding ADC input (IN0) in CubeMX. Select the Continuous Conversion Mode and the maximum possible sampling time (Sampling Time = 239.5 Cycles). It is also advisable to select Overrun data overwritten (occurs during debugging). Enable ADC global interrupt.
- Next, set the output GPIO pins for the LED driver (PB3 = SCT_CLK, PB4 = SCT_SDI, PB5 = SCT_NLA, PB10 = SCT_NOE). Don't forget to initialize the LED driver with **`sct_init()`**.
- Before starting the AD converter, it is very useful to run its autocalibration once to compensate for manufacturing tolerances. We will then use ADC mode with an interrupt call after the conversion is complete:

```
HAL_ADCEx_Calibration_Start(&hadc);
HAL_ADC_Start_IT(&hadc);
```

- After each conversion is completed, a callback is called to store the value from the ADC in a static volatile variable:

```
static volatile uint32_t raw_pot;

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    raw_pot = HAL_ADC_GetValue(hadc);
}
```

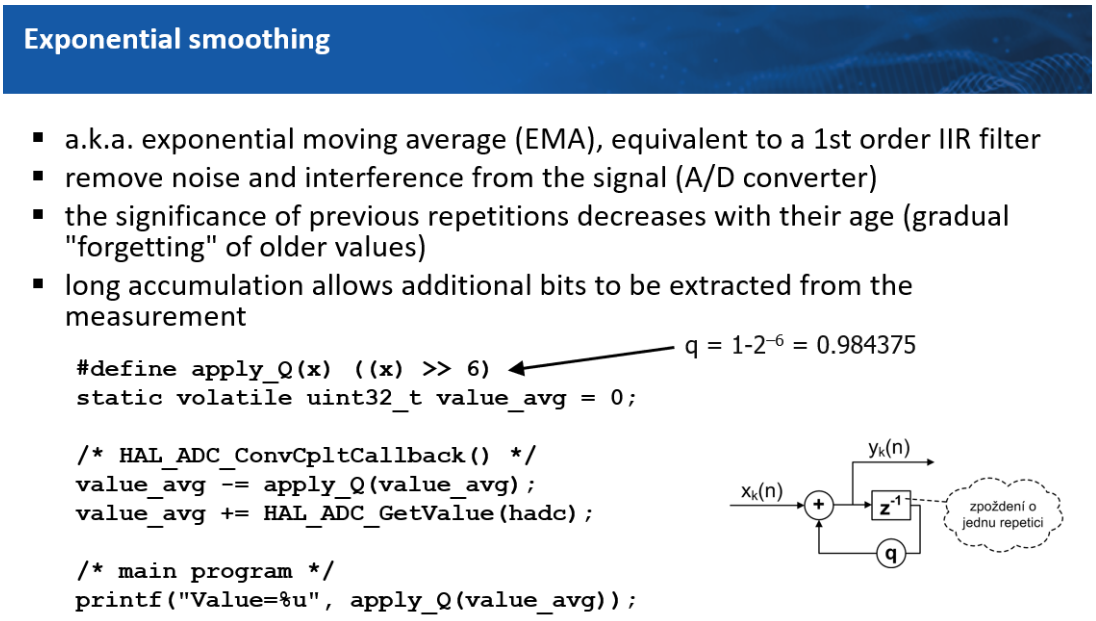
- In the infinite loop inside the main code, rescale the obtained 12-bit value to the range 0 to 500, display it on the screen (**`sct_value()`** function from the last exercise) and add a short wait (50ms).
- Extend the **`sct_value()`** function with the **`uint8_t led`** parameter, which will indicate the number of lit bargraph LEDs in the range 0-8. Extending the **`reg_values[]`** table with an additional position to represent the LEDs, and producing the result in binary OR along with the other segments, is recommended:

  ```
  reg |= reg_values[3][led];
  ```

- LED bargraph wiring hint: `//----43215678---- @ LED`

4.2.3 Exponential smoothing of ADC value

- Implement exponential smoothing in the ADC callback. Use the bit offset (ADC_Q) of about 12 (i.e., division by 2^{12}).

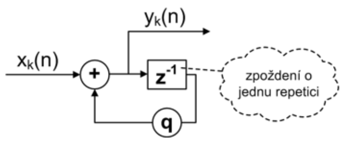


Exponential smoothing

- a.k.a. exponential moving average (EMA), equivalent to a 1st order IIR filter
- remove noise and interference from the signal (A/D converter)
- the significance of previous repetitions decreases with their age (gradual "forgetting" of older values)
- long accumulation allows additional bits to be extracted from the measurement

$q = 1\text{-}2^{-6} = 0.984375$

```
#define apply_Q(x)  ((x) >> 6)
static volatile uint32_t value_avg = 0;

/* HAL_ADC_ConvCpltCallback() */
value_avg -= apply_Q(value_avg);
value_avg += HAL_ADC_GetValue(hadc);

/* main program */
printf("Value=%u", apply_Q(value_avg));
```



- The callback code will contain:

  ```
  static uint32_t avg_pot;

  raw_pot = avg_pot >> ADC_Q;
  avg_pot -= raw_pot;
  avg_pot += HAL_ADC_GetValue(hadc);
  ```

- Test the filtering – the values on the display will be "lazy", changing with a slight delay after the trimmer is updated.

4.2.4 Multichannel ADC, internal reference and temperature sensor

- Activate the Temperature Sensor Channel and Vrefint Channel inputs in the ADC block in CubeMX. Also select the GPIO inputs for the buttons (PC0 = S2, PC1 = S1). Activate pull-up for both button inputs.
- The ADC callback is called after each conversion is complete, all three channels are continuously cycled. To distinguish the results, define a static variable **`channel`** in the callback. For channel 0, the

code keeps an exponential smoothing of the trimmer value. Channel 1 represents the raw temperature value (**raw_temp**), channel 2 the raw voltage reference value (**raw_volt**). Define the appropriate static volatile variables at module level for these new values.

- Completion of the conversion sequence of all three channels can be detected by testing the **EOSEQ** flag. If set, reset the **channel** variable, otherwise increment it:

```
if (__HAL_ADC_GET_FLAG(hadc, ADC_FLAG_EOS)) channel = 0;
else channel++;
```

- Procedures for calculating the actual AVCC voltage (i.e. AREF) and internal temperature in °C can be found in STM32SnippetsF0:

```
/* Temperature sensor calibration value address */
#define TEMP110_CAL_ADDR ((uint16_t*) ((uint32_t) 0x1FFFF7C2))
#define TEMP30_CAL_ADDR  ((uint16_t*) ((uint32_t) 0x1FFFF7B8))

/* Internal voltage reference calibration value address */
#define VREFINT_CAL_ADDR ((uint16_t*) ((uint32_t) 0x1FFFF7BA))

uint32_t voltage = 330 * (*VREFINT_CAL_ADDR) / raw_volt;

int32_t temperature = (raw_temp - (int32_t)(*TEMP30_CAL_ADDR));
temperature = temperature * (int32_t)(110 - 30);
temperature = temperature / (int32_t)(*TEMP110_CAL_ADDR - *TEMP30_CAL_ADDR);
temperature = temperature + 30;
```

- Consider the infinite loop code in main() to be a simple state machine, e.g. with states:

```
static enum { SHOW_POT, SHOW_VOLT, SHOW_TEMP } state = SHOW_POT;
```

- Display the appropriate value depending on **state**. If the button is pressed (read using **HAL_GPIO_ReadPin()**), change the state and set a static variable indicating the current time. Return to the default state after 1s has elapsed from the saved time.

- The procedure is similar to the *ISR, bouncing buttons, SysTick timer* exercise. SysTick does not need to be explicitly initialized (done in HAL libraries, default period 1ms), its value is available by calling the **HAL_GetTick()** function.

5 UART communication with DMA, EEPROM on I2C

5.1 Assignment

- Create a project with the USART2 interface, implement the loopback and verify its function.
- Extend the USART operation by using DMA for reception in circular mode. Test by echoing the received commands back.
- Create a parser for a simple text protocol. Use this protocol to implement the control of LED1 and LED2 on the development kit.
- Add support for external I2C EEPROM memory. Extend the protocol with the function of reading and writing an 8-bit number to a defined address in the EEPROM memory. Implement a dump command to show the first 16 bytes of external memory in hexadecimal, including eight byte line alignment.

<u>Text protocol commands:</u>

- HELLO
- LED1 ON|OFF
- LED2 ON|OFF
- STATUS
- READ address
- WRITE address value
- DUMP

5.2 Instructions

5.2.1 <u>Basic acquaintance</u>

- Create a working copy of your repository from Github (Git Clone) or update your repository from the server (Git Pull).
- Create a new project via File / New / STM32 Project / Board Selector / NUCLEO-F030R8. We will be using the HAL libraries, so leave the Targeted Project Type at STM32Cube. Confirm initialization of all peripherals to default settings.

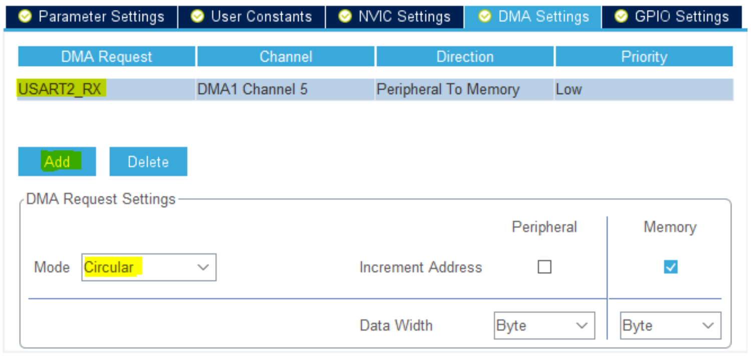
5.2.2 <u>UART loopback</u>

- The loopback feature sends each received byte back to the sender immediately. Termite is the optimal program for testing serial communication, the default port speed is 38400 Bd.
- The function is implemented using HAL libraries in an infinite loop of main():

```
uint8_t c;
HAL_UART_Receive(&huart2, &c, 1, HAL_MAX_DELAY);
HAL_UART_Transmit(&huart2, &c, 1, HAL_MAX_DELAY);
```

5.2.3 Using a circular buffer with DMA

- For the USART2 interface, set the DMA Request to receive, use the circular mode.

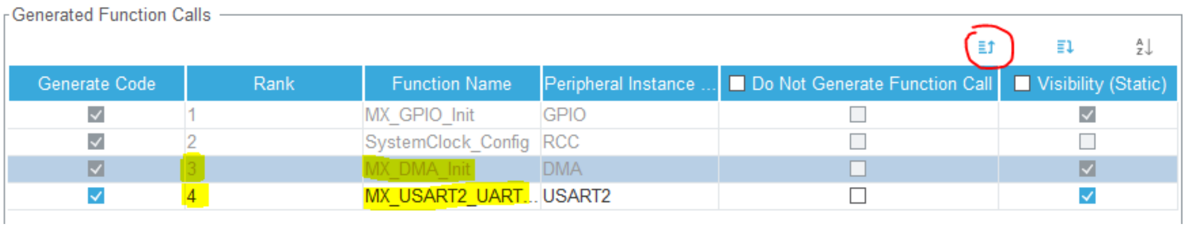


DMA Request	Channel	Direction	Priority
USART2_RX	DMA1 Channel 5	Peripheral To Memory	Low

DMA Request Settings

		Peripheral	Memory
Mode: Circular	Increment Address	☐	✓
	Data Width	Byte	Byte

- The reception is implemented using a circular buffer. Individual received characters are stored successively in the buffer, after reaching the end of the buffer it automatically continues again from the beginning. The write index can therefore be calculated based on the buffer length and the DMA transfer registers, the read index is handled by software.
- Former versions of STM32CubeIDE generated the initialization code in a wrong order. Verify the initialization order in IOC / Project Manager / Advanced Settings / Generated Function Calls. Use the button to move MX_DMA_Init over MX_USART2_UART_Init if necessary.


Generated Function Calls

Generate Code	Rank	Function Name	Peripheral Instance ...	Do Not Generate Function Call	Visibility (Static)
✓	1	MX_GPIO_Init	GPIO	☐	✓
✓	2	SystemClock_Config	RCC	☐	☐
✓	3	MX_DMA_Init	DMA	☐	✓
✓	4	MX_USART2_UART...	USART2	☐	✓

- Let’s declare the buffer and the read and write indices:

```
#define RX_BUFFER_LEN 64
static uint8_t uart_rx_buf[RX_BUFFER_LEN];
static volatile uint16_t uart_rx_read_ptr = 0;
#define uart_rx_write_ptr (RX_BUFFER_LEN - hdma_usart2_rx.Instance->CNDTR)
```

- First, we need to activate the DMA transfer from the UART:

```
HAL_UART_Receive_DMA(&huart2, uart_rx_buf, RX_BUFFER_LEN);
```

- In the main loop, we then process individual bytes from the buffer one by one. If the code is blocked for a certain period of time (e.g. due to an ISR processing), the data is accumulated in the buffer and processed as soon as CPU time is available:

```
while (uart_rx_read_ptr != uart_rx_write_ptr) {
    uint8_t b = uart_rx_buf[uart_rx_read_ptr];

    // increase read pointer
    if (++uart_rx_read_ptr >= RX_BUFFER_LEN) uart_rx_read_ptr = 0;

    // process every received byte with the RX state machine
    uart_byte_available(b);
}
```

- The **uart_byte_available()** function stores the printable characters (ASCII codes 32 to 126) in another buffer that already contains the completed text statement. When the end of the line is found, it calls a handler to execute the command:

```
static void uart_byte_available(uint8_t c)
{
    static uint16_t cnt;
    static char data[CMD_BUFFER_LEN];

    if (cnt < CMD_BUFFER_LEN && c >= 32 && c <= 126) data[cnt++] = c;
    if ((c == '\n' || c == '\r') && cnt > 0) {
        data[cnt] = '\0';
        uart_process_command(data);
        cnt = 0;
    }
}
```

- The appropriate buffer size for the **CMD_BUFFER_LEN** text command must be defined (e.g. 256B).
- To test the **uart_process_command()** function, it is useful to add a print of the received command:

```
printf("received: '%s'\n", cmd);
```

- For **printf()** to work, we need to include **<stdio.h>** and create a system function **_write()**:

```
int _write(int file, char const *buf, int n)
{
    /* stdout redirection to UART2 */
    HAL_UART_Transmit(&huart2, (uint8_t*)(buf), n, HAL_MAX_DELAY);
    return n;
}
```

- Commit the working copy.

5.2.4 Text protocol handling

- Implement the HELLO, LED1, LED2, and STATUS commands.
  - The HELLO command prints the welcome text.
  - The LED1 and LED2 commands turn the individual LEDs on or off according to the ON or OFF parameter.
  - The STATUS command returns the current status of both LEDs.
- Use the **strtok()** parsing function http://www.cplusplus.com/reference/cstring/strtok/, the separator will be the space character. Include **<string.h>** header file. Use the **strcasecmp()** http://www.cplusplus.com/reference/cstring/strcmp/ function to distinguish between commands, ignoring character case.
- Repeated calls to **strtok()** with the **NULL** first parameter return additional tokens in the form of C strings. Complex statements with many parameters can be parsed in this sequential manner.

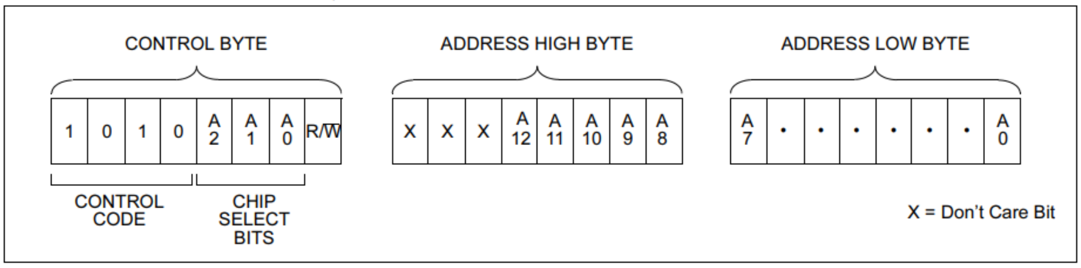
```
char *token;
token = strtok(cmd, " ");

if (strcasecmp(token, "HELLO") == 0) {
    printf("Communication OK\n");
}
else if (strcasecmp(token, "LED1") == 0) {
    token = strtok(NULL, " ");
    if (strcasecmp(token, "ON") == 0) {
        ...
    } else if (strcasecmp(token, "OFF") == 0) {
        ...
    }
    printf("OK\n");
}
else if (strcasecmp...
```

- Use **HAL_GPIO_WritePin()**, **HAL_GPIO_ReadPin()** and formatted output. Setup the output pins connected to LEDs (PA4 and PB0) in CubeMX.
- Commit the working copy.

5.2.5 External EEPROM

- The EEPROM memory is connected to I2C bus, pins PB9 (SDA) and PB8 (SCL) – pins must be remapped (in CubeMX pinout press Ctrl + grab pin with left mouse button, the tool will show possible mappings, then drag to target). Enable the I2C peripheral mode I2C1 in the CubeMX configuration and then use the functions from the HAL libraries **HAL_I2C_Mem_Read()**, **HAL_I2C_Mem_Write()** and **HAL_I2C_IsDeviceReady()**.
- The external EEPROM is Microchip 24LC64-I/SN. You can find information about the address (1-0-1-0-A2-A1-A0-RW) in the datasheet. Pins A0 to A2 are connected to GND, so the address of **EEPROM_ADDR** is 0b10100000 = 0xA0. Next, it is necessary to find information about the addressing method – the address inside the EEPROM consists of a high and a low byte, so it is a 16-bit addressing (constant **I2C_MEMADD_SIZE_16BIT**).

FIGURE 3-3: ADDRESS SEQUENCE BIT ASSIGNMENTS



- Reading one **value** byte from the **addr** address, or writing one **value** byte to the **addr** address:

```
HAL_I2C_Mem_Read(&hi2c1, EEPROM_ADDR, addr, I2C_MEMADD_SIZE_16BIT, &value, 1, 1000);
HAL_I2C_Mem_Write(&hi2c1, EEPROM_ADDR, addr, I2C_MEMADD_SIZE_16BIT, &value, 1, 1000);
```

- It is necessary to wait for the operation to complete after each write to EEPROM, e.g.:

```
/* Check if the EEPROM is ready for a new operation */
while (HAL_I2C_IsDeviceReady(&hi2c1, EEPROM_ADDR, 300, 1000) == HAL_TIMEOUT) {}
```

- Implement commands for working with EEPROM to the communication protocol:
    - The READ command reads the specified address from memory and prints its contents.
    - The WRITE command writes the specified value to the specified memory address. It waits for the operation to complete and confirms execution.
    - The DUMP command hexadecimally prints the values at addresses 0x0000 to 0x000F.
- Use the **`atoi()`** function from the **`<stdlib.h>`** header file to convert the strings produced by tokenization to numbers.
- Example of function verification in the terminal:

```
read 2
Address 0x0002 = 0xFF
write 2 50
OK
read 2
Address 0x0002 = 0x32
dump
FF FF 32 FF FF FF FF FF
FF FF FF FF FF FF FF FF
```

- Commit the working copy to Git, save the repository using Git Push.

6 DS18B20 and NTC temperature sensors

6.1 Assignment

- Initialize the 1-wire bus and the DS18B20 temperature sensor with the supplied library. Initialize the seven-segment LED display. Periodically start the temperature conversion and show its result on the display.
- Configure the ADC to read from the NTCC-10K type NTC temperature sensor. Based on the data in the datasheet of this sensor, create a MATLAB or Python script to generate a lookup table that will be indexed by the ADC value and will contain the recalculated actual temperature.
- Add a function to change the displayed data. Button S2 lights LED1 and displays the NTC reading, button S1 lights LED2 and displays the DS18B20 reading.

6.2 Instructions

6.2.1 Basic acquaintance

- Create a working copy of your repository from Github (Git Clone) or update your repository from the server (Git Pull).
- Create a new project via File / New / STM32 Project / Board Selector / NUCLEO-F030R8. We will be using the HAL libraries, so leave the Targeted Project Type at STM32Cube. Confirm initialization of all peripherals to default settings.
- Use the driver for the seven-segment LED display from the previous exercises, add a decimal point separator. Use LEDs (LED1 @ PA4, LED2 @ PB0, push-pull output) and buttons (S1 @ PC1, S2 @ PC0, input **with active pull-up**).

6.2.2 DS18B20 temperature sensor and 1-wire bus

- Initialize the communication by calling **`OWInit()`**. The main() infinite loop will contain the **`OWConvertAll()`** function call, followed by **`HAL_Delay()`** waiting for the duration of the conversion (**`CONVERT_T_DELAY`** = 750ms) and reading the result using **`OWReadTemperature()`**.
- *The 1wire.h file contains an inline _delay_us() function that is tuned for the default configuration of the development kits used. For a universal solution, it is preferable to rewrite this function using a regular timer with a granularity of 1µs. Due to the short intervals, direct register accesses and busy-waiting should be used. The DWT_CYCCNT cycle counter can be used for higher cores (M3 and above).*
- For proper operation, we need to define pin DQ @ PA10, output mode **open-drain**, output level high.
- The **`OWReadTemperature()`** function returns 0 in case of error, non-zero value in case of success. It also passes the reference to the measured value that represents the temperature in hundredths of a degree Celsius, e.g. a value of 1208 corresponds to 12.08°C and will be displayed as "12.1" on the LED display. The function declaration can be found in the 1wire.h header:

```
extern uint8_t OWReadTemperature(int16_t *temperature);
```

- So using the function will look like this:

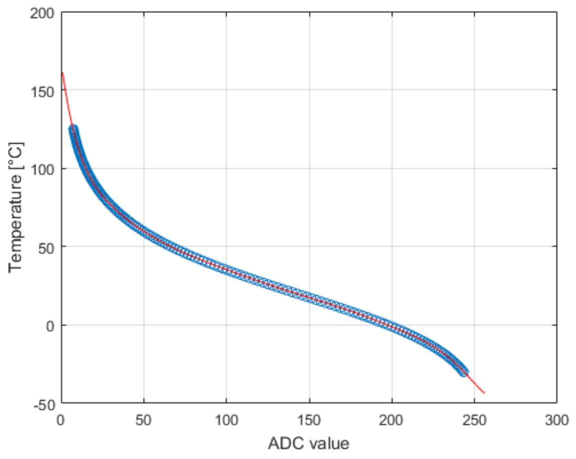
```
int16_t temp_18b20;
OWReadTemperature(&temp_18b20);
```

- Commit the working copy.

6.2.3 NTC temperature sensor and lookup table

- The temperature dependence of the NTCC-10K temperature sensor resistance is not linear. The NTC is specified by a nominal resistance of 10kΩ at 25°C and a constant B=4050 K. The datasheet indicates the resistance of the sensor for selected temperatures. For use in a microcontroller, it is best to have a table stored in program memory that will be indexed by the ADC value and will directly provide the actual sensor temperature. Such a table is easiest to prepare in a higher scripting language, such as MATLAB or Python.
- Resistance values for temperatures from -30 to +125°C can be found in the NTCC-10K datasheet. These data are prepared in the **`ntc.csv`** file in eLearning, ready to be read by the **`csvread()`** function in MATLAB.
- The reading on the AD converter must be calculated from the resistance value *r* at temperature *t*. This is simple math, the NTC sensor is wired as a resistive divider with resistor R5 (10k), the 3.3V supply is also the ADC reference, so it will cancel out. The range of the ADC is at most 2^{12} = 4096, but the resolution is unnecessarily high (table size, measurement tolerance), so we will work with a range of 2^{10} = 1024. This calculation gives a vector of *ad* values.
- Use the **`plot()`** function to plot the values to a graph (the dependence of *t* on *ad*) and then interleave them with a polynomial of sufficiently high order:

```
p = polyfit(ad, t, 10);
```



- Then create a vector of ADC values *ad2* (it will represent the indices of the array) and calculate the *t2* values of the identified polynomial (it will represent the temperature rounded to tenths of a degree for these indices). Plot the result on a graph:

```
ad2 = 0:1023;
t2 = round(polyval(p, ad2), 1);
hold on, plot(ad2, t2, 'r');
```

- The last step is to store the array values in a suitable format, i.e. comma-separated values for the C language, formatted in tenths of a degree:

```
dlmwrite('data.dlm', t2*10, ',');
```

- Implementation in C is straightforward. It is necessary to define a static constant array of the **`int16_t`** type, which will contain values copied from the **`data.dlm`** file. The value read from the ADC will be used for indexing this array.

- The ADC input of the NTC thermistor is ADC_IN1 @ PA1, converter resolution **10 bits**, continuous conversion, overwrite on overflow. For simplicity, no interrupt or DMA is required. The initialization contains calibration and continuous conversion start:

  ```
  HAL_ADCEx_Calibration_Start(&hadc);
  HAL_ADC_Start(&hadc);
  ```

- The AD value is read with **`HAL_ADC_GetValue(&hadc)`**.
- Commit the working copy, **including the `.m` file** used to generate the lookup table.

6.2.4 Changing display data

- Extend the code as specified in the assignment, use a state machine:
  - button S2 lights up LED1 and displays the NTC reading
  - button S1 lights up LED2 and displays the data from DS18B20
- The DS18B20 conversion must not be triggered more frequently than after 750 ms.
- After prolonged operation, both sensors will show a temperature about 3°C higher than the room temperature due to heating from the environment.
- Commit the working copy to Git, save the repository using Git Push.