Programming Task 0: Getting started with CubeMX

Note: This is not an assignment that you need to submit, or be graded on, but it is a prerequisite for completing the remainder of the programming project!

For this "task", you will assume the role of the Core team, and your task is to configure the microcontroller, initializing the necessary clocks, peripherals, and setting up the internet protocol. Based on the overall architecture, you are told that that following features are needed:

1. A PWM signal to drive the DC-motor to the desired speed.
2. An encoder to calculate the actual motor speed

So far in Tutorial 2, you have learnt to configure your microcontroller by analyzing the reference manual, and writing program code to configure the desired peripheral, through its memory-mapped registers. You have also taken advantage of the CMSIS Core, that included all the necessary variables and macros to configure and access peripherals, with a higher level of abstraction.

In this task, you will take advantage of an even higher-level approach. STM32CubeMX is an application that provides a convenient graphical user interface for configuring your specific microcontroller, its pins, peripherals, clocks, etc. Once configured, you can then use the STM32CubeMX code generation tool to produce C-code that initializes the device. The generated code is still CMSIS-compliant. This approach essentially replaces the effort of reading the reference manuals, and programming the registers of the peripherals, as detailed in Tutorial 2.

STM32CubeMX is well integrated and works seamlessly with Keil µVision. So let's start by creating a new development project in µVision, after which we will trigger STM32CubeMX to configure the device, and generate the necessary configuration code. Finally, we will come back to µVision to develop the remaining manual code.

Install STM32CubeMX

Just one more application to install. Download and install STM32CubeMX from **https://www.st.com/en/development-tools/stm32cubemx.html (https://www.st.com/en/development-tools/stm32cubemx.html)** .

Note that STM32CubeMX needs a Java run-time environment (JRE). So, if you don't have it

installed, do it from **https://www.java.com/en/download/ (https://www.java.com/en/download/)** .

Create a new project using Keil MDK

First, make sure you have the correct versions of the packages installed:

Pack	Version	Comment
ARM::CMSIS	6.1.0	Core CMSIS framework.
ARM::CMSIS-RTX	5.9.0	Contains an implementation of CMSIS-RTOS, which will be used in Task 2.
ARM::CMSIS-View	1.2.0	Contains features for the Event Recorder, which will be used in Task 2.
Keil::ARM_Compiler	1.7.2	Allows re-targeting STDOUT to the ITM (CoreSight).
Keil::STM32L4xx_DFP	2.7.0	Provides CubeMX support.

Refer to instructions form Tutorial 1 if necessary!

1. First, create a new project in Keil µVision.
   - Keep in mind: you should not need to create multiple copies of this project, and you will not need to make a new project for the different tasks. Development will be incremental, and you will learn techniques for handling code variants within the same project structure. Name your project accordingly—don't call it "Task0"!
   - If you are interested in versioning while developing code, you should consider Git. KTH provides all students with **GitHub Enterprise (https://gits-15.sys.kth.se/)** for free!
2. When prompted for the Device for Target, select STM32L476RG.
3. When prompted to "Manage Run-Time Environment", choose the following software components that you want included in your new project:
   - Compiler → I/O
     - STDOUT ✓, Variant: ITM
   - Device → Startup: ✓
   - Device → STM32Cube Framework (API)
     - STM32CubeMX
4. You will notice that some of the selections are orange in color. This means that the selections require additional components. Click Resolve, to automatically fill in the missing dependencies.
5. Click OK.
6. Having identified that the selected device can be supported by STM32CubeMX, µVision will prompt you to launch the STM32CubeMX application. Do it now. If you do not, you will likely get an `Error #545: Required input file from generator CubeMX`. You can start CubeMX from the RTE manager by clicking the green "play" button next to CubeMX.

Configure Device using STM32CubeMX

In the following subsections, you will be instructed on how to configure the Nucleo-L476RG board as follows:

1. Core clock frequency will be set to 40 MHz.
2. CoreSight debugging will be enabled through the ST-Link debug adapter.
3. TIM1 timer will be configured for encoder mode. This timer will behave more like a counter, which will count the motor encoder pulses without the need for any interrupts.
4. TIM3 timer will be configured for PWM generation mode. The timer will produce a PWM signal with a frequency of 20 kHz.

The application consists of several windows and pages. For each peripheral, there is a Mode (upper half) and Configuration (lower half) window, which helps organize the setup. The appearance of the Configuration window will be dependent on the choices made in the Mode window, so make sure you follow the instructions in the correct order. For now, you will only need the “Parameter Settings” tab in the Configuration window.

To help you navigate, please refer to the figure below that highlights some of the main sections referred to in the instructions below.

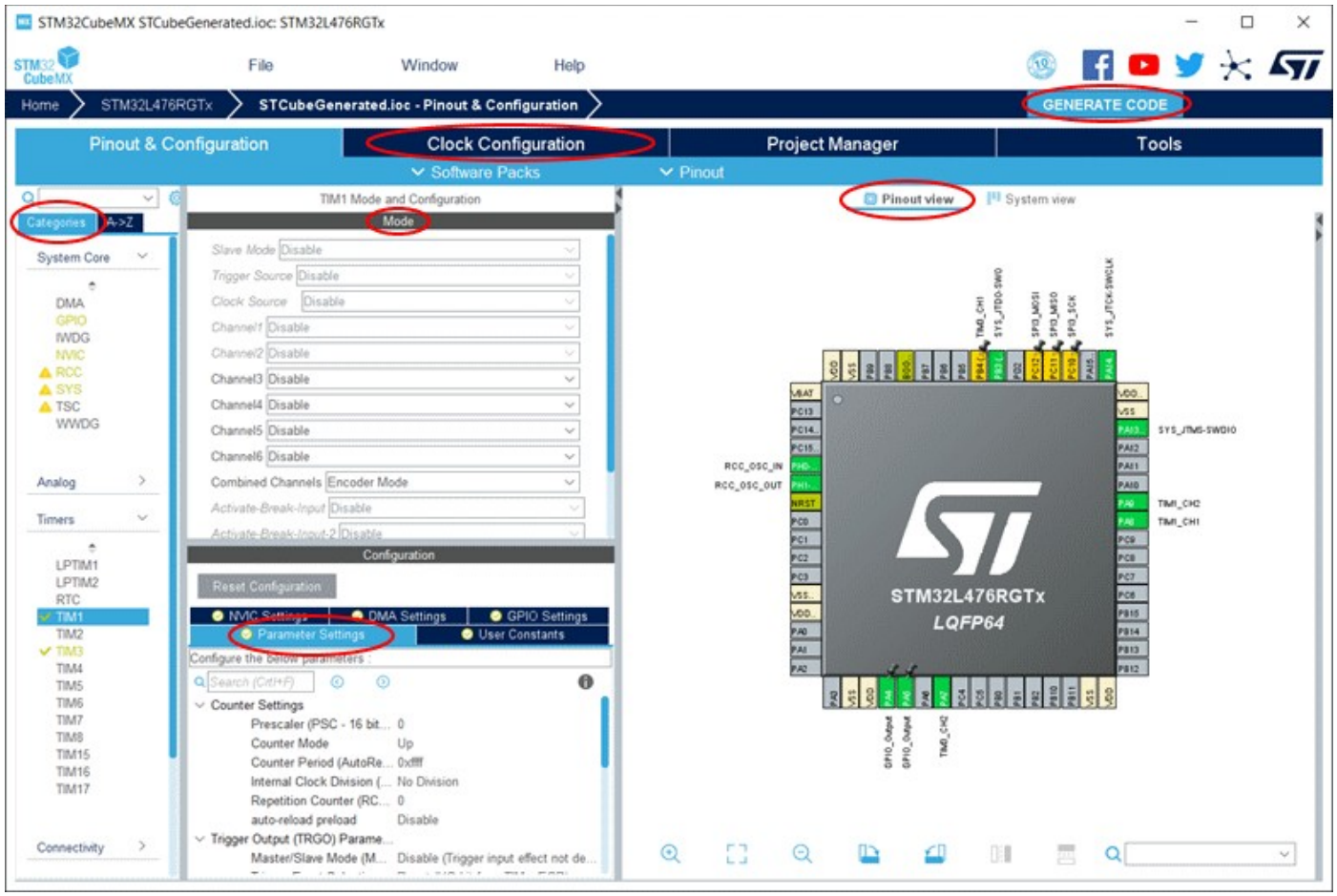

Figure 2. The STM32CubeMX application

Under the Pinout & Configuration page, in the leftmost panel (under Categories), configure the

following peripherals:

System Core

1. RCC (Reset and Clock Controller)
   - Mode
     - High Speed Clock (HSE) → Crystal/Ceramic Resonator
     - Low Speed Clock (LSE) → Disable
2. SYS (System)
   - Mode
     - Debug → Trace Asynchronous Sw

Timers

1. TIM1 (Advanced-control timer)
   - Mode
     - Combined Channels → Encoder Mode
   - Configuration [Parameter Settings]
     - Prescaler (PSC) → 0
     - Counter Period (ARR) → 65535 (or 0xFFFF in hexadecimal)
     - Encoder Mode → Encoder Mode TI1 and TI2

     > Think about the period of the encoder counter timer. Why is it set to 0xFFFF? Keep this in mind for when you will initialize the timers.

2. TIM3 (General-purpose timer)
   - Mode
     - Channel 1 → PWM Generation CH1
     - Channel 2 → PWM Generation CH2
   - Configuration [Parameter Settings]
     - Prescaler (PSC) → 0
     - Counter Period (ARR) → 2047 (decimal)
     - Mode → PWM mode 1

     > Think about the period of the PWM timer. Why is it set to 2047? What is the resulting PWM frequency?

Next, you will configure the functionality of some of the pins. Recall that a given pin can be used for general purpose I/O, as well as several alternate functions, which are determined by the hardware. By selecting the pin function here, all other functions are disabled. In the Pinout view, set the pin functionality according to the following table: (Note that in some cases, the pin might already be set to as desired)

Table 1. Pin Configuration

Pin	Selection
PB4	TIM3_CH1
PB3	SYS_JTDO-SWO
PA14	SYS_JTCK-SWCLK
PA13	SYS_JTMS-SWDIO
PA9	TIM1_CH2
PA8	TIM1_CH1
PA5	GPIO_Output
PA6	GPIO_Output
PA7	TIM3_CH2
PH0	RCC_OSC_IN
PH1	RCC_OSC_OUT

After that, we will configure the main clocks to the desired frequency.

- Under the “Clock Configuration” page, set HCLK to 40 Mhz.
- You may get a prompt that “No Solution found using current selected Sources”. Click OK to use other sources.
- If successful, all clocks on the right will show 40 MHz.

Finally, it is time to generate the configuration code.

1. Under the “Project Manager” page, click GENERATE CODE in the top right.
2. You may be prompted to download some software packages. Accept.
3. You will eventually be prompted that the code was successfully generated. Press Close.
4. µVision might also prompted that “for the current project new generated code is available for import”. Press YES to import changes.
5. You may close the STM32CubeMX application and return to µVision.
6. Your Keil project now contains all the necessary code to initialize the device as graphically specified in STM32CubeMX.

Note: It may warn that the project generation encountered a problem, but the code should still be OK!

Configure Compiler Options

Now, you will set some of the compiler settings for the Project—you will use settings which have been chosen to achieve a balance between code size, debug view, and compilation-time feedback. If interested, you can read more about it here.

Back in µVision, select the magic wand to open Options for Target… window:

1. Under the Target tab, in the Code Generation section
   - ARM Compiler: Use default compiler version 6
   - Use MicroLIB: ✓
2. Under the C/C++ tab
   - Optimization: Level 0 (-O0)
   - One ELF Section per Function: ✓
   - Short enums/wchar: ✓
   - Warning: AC5-like warnings
   - Language C: gnu99
3. Under the Asm tab:
   - Assembler Option: armclang (Auto Select)

Configure Debugger Options

1. Under the Debug tab, choose ST-Link Debugger, then click Settings.
2. In the new window that appear, select the Trace tab
   - Set the Core Clock to 40 MHz
   - Trace Enable: ✓
   - Untick EXCTRC: Exception tracing under Trace Events
   - Untick all ITM Stimulus ports except Port 31 and 0.

   > Check Tutorial 1: "ST-Link/V2 and Keil", section 19 again. What features are you actually interested in enabling?

3. Select the Flash Download tab
   - Select Erase Full Chip
   - Program: ✓
   - Verify: ✓
   - Reset and Run: ✓
4. Select OK twice, to close all dialog windows.

Add Skeleton C- & H-files to the project

As promised, in this project, you are to be given a skeleton code that you are expected to complement. It is now time to copy this skeleton code into your project.

1. Download and copy the **skeleton code files (https://canvas.kth.se/courses/59527/files/9577780?wrap=1)** ⬇ **(https://canvas.kth.se/courses/59527/files/9577780/download?download_frd=1)** to you project. **(https://canvas.kth.se/courses/31938/files/5132405)**

Open a Windows File Explorer
- Go to the folder where your project files exist.
- Download, unzip and save the folders Source and Include from the course material to the base directory of your Keil project. (That is, the folder where the *.uvprojx file exists. and not for example the MDK-ARM folder).

2. Import C-files into your project in µVision
   - Go to µVision.
   - Inside the Project window on the left, right-click Target 1 and choose Manage Project Items…
   - Click Add Files… and add the following files to the empty “Source Group 1”.
   - source/peripherals.c
   - source/controller.c
   - source/application.c
   - You have now made these C-files part of the project, which means they will be compiled and linked when you build the project.
3. Next, we need to include the folder that contains header h-files in the “include path”.
   - Right-click Target 1 and select Options for Target ‘Target1’.
   - Under the tab C/C++ (AC6), click the “…” next to Include Paths,
   - In the new window that appears, click the “new” button, and then the “…” that appears.
   - In the File Dialog that appears, select the Include folder containing the header files.
   - select OK to close all dialogs.
4. Finally, we will add some additional error-checking to your user-code.
   - Right click Source Group 1 and select Options for Group 'Source Group 1'
   - Under the tab C/C++ (AC6), add the following additional compiler flags under Misc Controls:

     ```
     -Werror=sign-conversion -Werror=implicit-int-conversion -Werror=implicit-function-declaration
     ```

You noticed from the instructions above that c- & h-files are handled differently. Why is that? A project consists of C-files that get compiled and linked into an executable. H-files are not directly part of this process. However, H-files are included in C-files (or other H-files), when a file includes a directive such as #include “controller.h”. Note here that the C-file does not specify the complete path to the controller.h file. So, for the complier to know where to find such h-files, we need to configure the Include Paths, to include the set of folders where the compiler can find them.

Note here that the Project window in µVision is not a file manager and does not necessarily reflect the actual structure inside the folder containing your project. Rather, it is organized in Source Groups, which are the files that will be compiled by the toolchain. So, you need to be careful when you move files around in µVision and/or Windows File Explorer, since the two are not in synch.

To make your project structure more comprehensible, you can consider the following:

1. Rename “Target 1” to something more appropriate, like “Default”. (to rename, click on the item, then click on the item again after 1-2 seconds)
2. Rename “Source Group 1” to “Modules” or something similar

Develop the code

The code generated by STM32CubeMX is meant to be modified manually by a programmer. One should also be able to reconfigure the device, and regenerate code to reflect these changes. With such re-generation, it would not be desired if any manual code is lost.

To solve this conflict, the generated code includes USER CODE blocks within which the programmer can write any manual code. Code within such blocks is safely preserved across regenerations. Any code outside the USER CODE blocks will be overwritten upon new code generations.

The main.c file is the main (as the name suggests 😊) entry point, which contains the main function.

> Analyse the content of the main function and try to understand what it tries to do. Try to also navigate to the functions called from this main function, to understand what they do in turn.
>
> Tips: To jump to a specific function, right-click on that line, and select “Go to Definition ...”. This will jump to that function, which can be either inside the same file, or open a new file that contains the function.

The main file is used to perform initial setup of the board after startup. It contains an endless loop that is meant to invoke all application-specific code. The architect wants to minimize adding any application-specific code to this main.c file. To do so, you are asked to insert function calls to the Application module instead, where the application-specific code is written.

So, in main.c, add the following pieces of code within the appropriate user block. Do not simply copy/paste this code into the user blocks but retype it (and let the Keil IDE help you autocomplete as you type). To better understand:

- Read through the existing code from the top-down and try to understand the existing statements, as well as the overall structure.
- When adding the specific code blocks below, make sure you understand what you are adding, and the context within which you are adding this code.
- Include header files

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include "application.h"
/* USER CODE END Includes */
```

- Just after all peripherals are initialized, make a call to the `Application_Setup()` function, in order to do any necessary other configurations.

```
/* USER CODE BEGIN 2 */
Application_Setup();
/* USER CODE END 2 */
```

> What does this function do? Why is this function called after the peripherals are initialized? Could it be ok to call it before?

- Make a call to the `Application_Loop()` function.

```
/* USER CODE BEGIN WHILE */
while (1)
{
  Application_Loop();
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

> Where in the main function is this function called? What is the purpose of this function call?

- Start the Timer1 peripheral channels

```
/* USER CODE BEGIN TIM1_Init 2 */
HAL_TIM_Encoder_Start(&htim1,TIM_CHANNEL_1);
HAL_TIM_Encoder_Start(&htim1,TIM_CHANNEL_2);
/* USER CODE END TIM1_Init 2 */
```

> Where is this located in the code structure? When will this initialization occur during runtime? Put a breakpoint, and make sure you understand when this is called. What is the period of the Timer 1? Why was it set this way?

- Start the Timer3 peripheral channels

```
/* USER CODE BEGIN TIM3_Init 2 */
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_2);
/* USER CODE END TIM3_Init 2 */
```

> Where is this located in the code structure? When will this initialization occur during runtime? Put a breakpoint, and make sure you understand when this is called. What is the period of the Timer 3? Why was it set this way?

The architect also desires to provide a function that returns the clock tick in milliseconds. This function is available by calling `HAL_GetTick()`. However, the architect wants to wrap this hardware-specific function call to avoid other modules directly calling HAL functions. Remember: even if the underlying hardware is changed, the other modules should not have to be altered.

1. In main.h, declare a new function.

```
/* USER CODE BEGIN EFP */
uint32_t Main_GetTickMillisec(void);
/* USER CODE END EFP */
```

   Tips: To open main.h, right click anywhere in main.c and click Toggle Header/Code File.

   > What does this declaration do? Why is it added to the h-file and not the c-file? What happens if you remove this declaration and try to build the code?

2. Define the function in main.c

```
/* USER CODE BEGIN 0 */
uint32_t Main_GetTickMillisec(void) {return HAL_GetTick();}
/* USER CODE END 0 */
```

   > What does this function do? Who calls this function? Use the debugger to understand who calls this function, and what it does when it gets called.

Build & run the project

At this point, you should already have done the suggested tutorials on debugging with CoreSight, breakpoints, the Watch window, Logic Analyzer etc. If you feel uncertain, it is suggested that you re-visit that material.

Build the project and run it in debug mode.

The Watch Window

1. In the Project window, navigate to the file application.c and open it.
2. Right click the variable `millisec` and choose Add millisec to… → Watch 1. The Watch 1 window should appear next to the command window. If it doesn't, you can bring it out under View → Watch Windows.
3. Do the same thing for the variables reference, velocity, and control.

As you'd expect from the code you have written so far, you should notice that the variables `reference` and `millisec` change over time. `velocity` and `control` remain at 0.

The Logic Analyzer

1. Right click the variable velocity and choose Add to… → Logic Analyzer.
2. Right click the y-axis and enable Adaptive Min/Max.
3. The Logic Analyzer allows you graphically monitor continuous signals over time.

Understand the project code

Now is a good time to view the provided code templates and try to understand their content. They are mostly empty, and it is your task to complete them with the necessary functionality. But the structure and names of the functions should give you a good indication of their purpose.