

Project Work - Embedded Systems for Mechatronics, MF2103

1 Introduction

This project is divided into three tasks, which correspond to the respective modules of the course MF2103. They are as follows

- 1) Digital I/O and control
- 2) Real-time operating systems
- 3) Distributed systems using TCP/IP

You are to develop and submit each task separately, over the progress of the course. The three tasks to be submitted are found under sections

- 1) 8
- 2) 9.2
- 3) 10.8

The project however starts with Task 0 (section 7), where we guide you through the steps of creating the project structure, and configuring the microcontroller. In between your tasks, we will also guide you through some further configuration steps to make sure you focus on the course content, and avoid many microcontroller-specific issues.

2 Prerequisites

It is assumed that you have completed all activities of a module before you can start working on the task for that particular module.

3 Intended outcome

After completing this project, you will be able to:

- Develop modular, interface-based programs for embedded systems
- Use professional toolchains for the ARM Cortex-M microprocessor
- Utilise standard hardware abstraction layers to access memory-mapped hardware
- Read the application programming interface (API) for external libraries
- Develop an embedded application, running on a real-time operating system (RTOS)
- Design a distributed, embedded control system using TCP/IP over Ethernet

4 Reporting & Deadlines

Submit your code for each task to the appropriate assignment on Canvas.

You will also need to present your results to a member of the teaching team.

Submission guidelines:

- 1) You need only submit the C-code and no additional documents.
- 2) In section 7.5, you will be given an initial set of skeleton files to start with. You are allowed to modify the C-files, but you cannot modify the header H-files.
- 3) Only submit the C-files and H-files you have created or modified. Do not submit the whole project.
- 4) Your code should be well documented, with explanatory comments.

Remember the possibility to obtain bonus points! See Course Memo!

The tasks for the bonus points can be found in sections 9.3.1 and 10.8.1.

5 References

See the Course Literature page on Canvas for the list of documents referred to in this report. ([Wolf-CasC], [Martin-DG], etc)

<https://kth.instructure.com/courses/22101/pages/course-literature-and-material>

6 Project Description

In this project, you are to develop an embedded control system on the STM32 target hardware. The software is expected to control the speed of a motor, using a PID control algorithm. Ultimately, you are expected to develop this software over a distributed system, where one processor is used to calculate the control signal, while another is used to handle the sensors and actuators.

Luckily, we have already hired an architect that have already broken down the software into 4 modules. You will be even provided with a code skeleton of each module. Each module consists of an interface (header-file), and implementation (C source code). It will be your job to implement these modules, without altering the interfaces.

The Figure below shows the project modules.

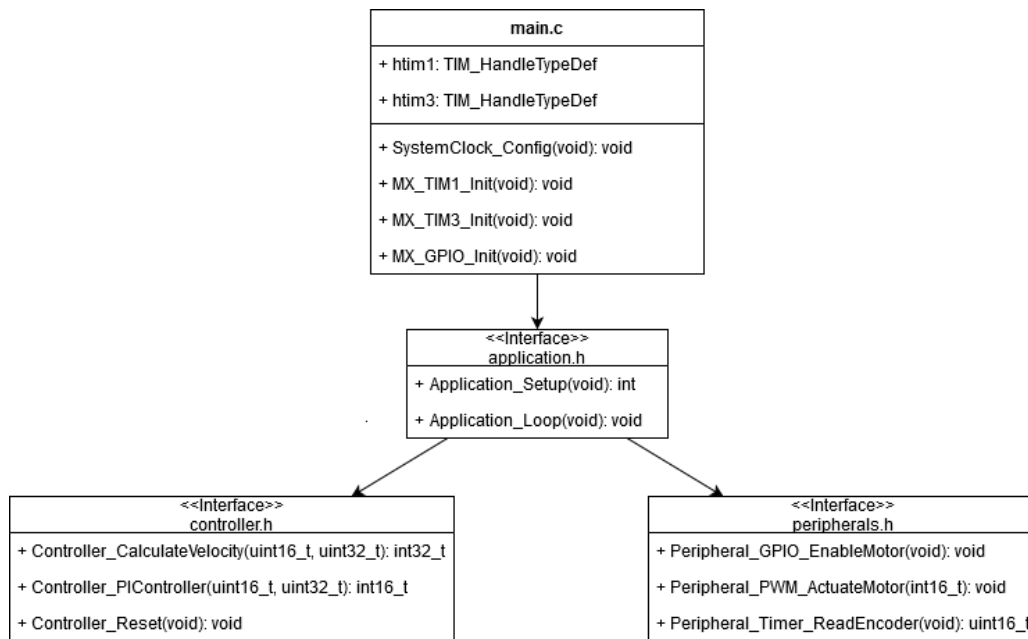


Figure 1. Project modules

One of the ideas behind this project is to give you a feel for working on a large-scale project, with development done by several software teams in parallel. You can imagine that there are four teams working on this project. Each team working on different layers of the software, roughly translating to the different areas of mechatronics. Your task will be to assume the roles of the different teams, detailed below.

- 1) The **core** team is responsible for configuring the microcontroller, initialising the necessary clocks, peripherals, and setting up the ethernet interface. This is implemented in the main module.
- 2) The **hardware** team is responsible for working with the sensors and actuators. It assumes that the core layer works as it expected but is only dependent on CMSIS Core. This is implemented in the peripherals module.
- 3) The **control** team is responsible for developing the control algorithm for the motor. It assumes that the core and hardware layers work as they should. This is implemented in the controller module.
- 4) The **application** team is responsible for the top-level functionality. It assumes all the lower layers work as they should. This is implemented in the application module.

7 Task0 – Project Setup and microcontroller Configuration

For this first task, you will assume the role of the Core team, and your task is to configure the microcontroller, initialising the necessary clocks, peripherals, and setting up the internet protocol. Based on the overall architecture, you are told that the 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 analysing 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 tutorial2.

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

7.1 Install STM32CubeMX

Just one more application to install. Download and install STM32CubeMX from <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/>.

7.2 Create a new project using Keil MDK

Refer to instructions from Tutorial1 if necessary!

- 1) First, create a new project in the Keil MDK (Microcontroller Development Kit).
- 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:
 - a) Compiler → I/O
 - i) STDERR: ✓ Variant: ITM
 - ii) STDIN: ✓ Variant: ITM
 - iii) STDOUT ✓ Variant: ITM
 - iv) TTY: ✓ Variant: ITM

- b) Device → Startup: ✓
- c) Device → STM32Cube HAL
 - i) GPIO: ✓
 - ii) SPI: ✓
 - iii) TIM: ✓
- 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, Keil MDK will prompt you to launch the STM32CubeMX application. Do it now. (If you get prompted for a missing Java JRE, see instructions in section 7.1)

7.3 Configure Device using STM32CubeMX

In the following subsections, we will use STM32CubeMX to configure the Nucleo-L476RG board as follows:

- 1) Core clock frequency to be set to 40 MHz.
- 2) CoreSight debugging to be enabled through the ST-Link debug adaptor.
- 3) TIM1 timer to 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 to be configured for PWM generation mode. The timer will produce a PWM signal with a frequency of 20 kHz.
- 5) SPI3 to be configured to full duplex master mode. It is used to connect to the ethernet shield.

The application consists of several windows and pages. To help you navigate, please refer to the figure below that highlights the main sections referred to in the instructions below.

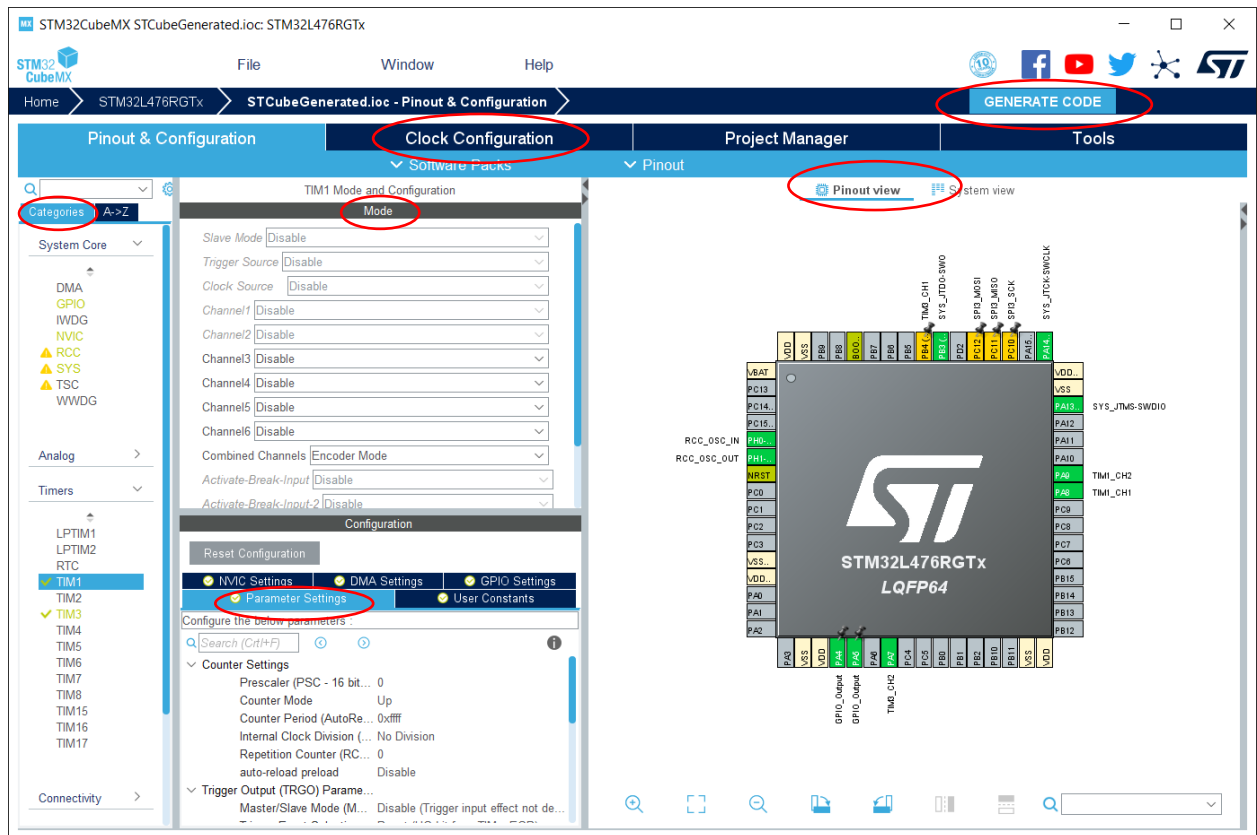


Figure 2. The STM32CubeMX application

- 1) Configure the Reset and Clock Controller (RCC)
 - a) Under the “Pinout & Configuration” page, select System Core → RCC (under Categories)
 - b) In the middle window “RCC Mode and Configuration”
 - i) Set High Speed Clock to Crystal/Ceramic Resonator
 - ii) Set Low Speed Clock to Disable
- 2) Configure the System (SYS) to the desired debugging mode
 - a) Under the “Pinout & Configuration” page, select System Core → SYS (under Categories)
 - i) In the middle window “SYS Mode and Configuration”, set Debug to Trace Asynchronous SW
- 3) Configure the Advanced-control timer (TIM1) to work in Encoder mode
 - a) Under the “Pinout & Configuration” page, select Timers → TIM1 (under Categories)
 - i) In the middle window “TIM1 Mode and Configuration”, set Combined Channels to Encoder Mode.
 - b) In the middle bottom window “Parameter Settings”
 - i) Set Counter Period to 0xFFFF (press gear to set to Hexadecimal)
 - ii) Set Encoder Mode to Encoder Mode TI1 and TI2

Think about the period of the encoder counter timer. Why is it set to 0xFFFF? Keep this in mind till you proceed to §7.6 where the timers are initialised in the code.

- 4) Configure the General purpose timer (TIM3) to work in PWM Generation mode
 - a) Under the “Pinout & Configuration” page, select Timers → TIM3 (under Categories).
 - b) In the middle window “TIM3 Mode and Configuration”
 - i) Set Channel 1 to PWM Generation CH1
 - ii) Set Channel 2 to PWM Generation CH2
 - iii) In the middle bottom window “Parameter Settings”, set Counter Period to 2000 (press gear to set to Decimal)

Think about the period of the PWM timer. Why is it set to 2000?

- 5) In the Pinout view, select each of the following “pins”, to set the pin functionality according to the following table: (Note that in some cases, the pin might already be set to as desired)

Pin	Selection
PB4	TIM3_CH1
PA5	GPIO_Output
PA6	GPIO_Output
PB3	SYS_JTDO-SWO
PC12	SPI3_MOSI
PC11	SPI3_MISO
PC10	SPI3_SCK
PA14	SYS_LTCK-SWCLK
PA13	SYS_JTMS-SWDIO
PA9	TIM1_CH2
PA8	TIM1_CH1
PA7	TIM3_CH2
PH0	RCC_OSC_IN
PH1	RCC_OSC_OUT

Table 1. Pin Configuration

- 6) Configure the main clock to the desired frequency
 - a) Under the “Clock Configuration” page, set HCLK to 40 Mhz.

- b) You will get a prompt that “No Solution found using current selected Sources”. Click OK to use other sources.
- c) If successful, all clocks on the right will show 40 MHz.
- 7) 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) Keil MDK might also prompt 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 Keil MDK.
 - 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!

7.4 Configure Target Options

- 1) Back in Keil MDK, select the magic wand to open Options for Target... window
- 2) Under the Debug tab, choose ST-Link Debugger, then click Settings.
- 3) In the new window that appear, select the Trace tab
 - a) Set the Core Clock to 40 MHz
 - b) Trace Enable: ✓
- 4) select the Flash Download tab
 - a) Select Erase Full Chip
 - b) Program: ✓
 - c) Verify: ✓
 - d) Reset and Run: ✓
- 5) Select OK twice, to close all dialog windows.

7.5 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 to your project from <https://kth.instructure.com/courses/22101/modules/items/298564>
 - a) Open a Windows File Explorer
 - b) Go to the folder where your project files exist.
 - c) 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 Keil MDK
 - a) Go to Keil MDK.
 - b) Inside the Project window on the left, right-click Target 1 and choose Manage Project Items...

- c) Click Add Files... and add the following files to the empty "Source Group 1".
 - i) source/peripherals.c
 - ii) source/controller.c
 - iii) source/application.c
- d) 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".
 - a) Right-click Target 1, and select Options for Target 'Target1'.
 - b) Under C/C++, click the "..." next to Include Paths,
 - c) In the new window that appears, click the "new" button, and then the "..." that appears.
 - d) In the File Dialog that appears, select the Include folder containing the header files.
 - e) select OK to close all dialogs.

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 compiler 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 Keil MDK 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 Keil MDK 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 "STM32L476RG". (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

7.6 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.

1) Include header files

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

2) 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?

3) 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?

4) 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?

5) 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);  
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3);  
/* 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?

6) Set the error handler to avoid undefined behaviour

```
/* USER CODE BEGIN Error_Handler_Debug */  
printf("Initialization error!\n\r");  
for(;;);  
/* USER CODE END Error_Handler_Debug */
```

Note: `for(;;)` is simply another way to define an endless loop.

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 the peripheral.

1) Define a new function in `main.c`

```
/* USER CODE BEGIN 0 */  
uint32_t SysTick_ms(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.

2) In `main.h`, add a declaration to this function.

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

Tips: To open `main.h` file, right click on the `#include "main.h"` statement at the start of the file, and select "open document `main.h`"

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?

7.7 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.

7.7.1 The Watch Window

- 1) In the Project window, navigate to the file `application.c` and open it.
- 2) Right click the variable `encoder` and choose Add 'encoder' 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`, `control` and `millisec`.

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.

7.7.2 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.

7.8 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.

8 Task1 - Digital I/O and control

Your task now is to design a PI Controller that controls the motor speed. You are to develop the Peripherals and Controller modules.

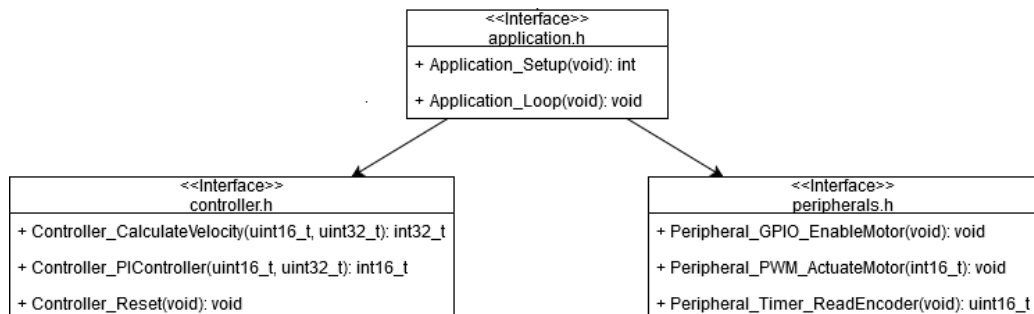


Figure 3. Peripherals and Controller modules

8.1 Peripherals module

First, you should implement the peripherals module, by completing the body of the empty functions in `peripherals.c`. To understand the context on when these functions are used, you can study `application.c`.

How can you define the duty cycle of a 16-bit Timer in PWM mode, using memory-mapped registers?

Remember that after creating the project, the datasheet can be found in the same window as the Project View, in the Books tab, under Device Data Books.

You want to be able to test your code, even though the controller is not yet ready, and hence have not the complete logic to control and run the motor. So, when developing this Peripherals module, you should mock-up the controller that just returns some hardcoded control signal (see tips on how to do that in section 8.2).

Try spinning the motor by hand in both directions. Can you figure out the relation between the observed encoder counter values and the Timer 1 period?

You should use the macros defined in the CMSIS Core to access the peripherals, check [STM-RefMan] for the names of the registers. It is strongly recommended that you use the debugger and Watch functionality to convince yourself that each function behaves correctly, before moving on to the next.

8.2 Controller

Next, you will assume the role of the controls team, whose job it is to design a PI-controller to drive the motor, by taking advantage of the peripheral functionality you just developed.

Consider the following steps to incrementally develop the controller:

- 1) First, hard-code it to return some constant, non-zero number.

- 2) Then change it to a simple P controller.
- 3) Finally, add integral action.

Since the sample rate is fixed (100 Hz), what can happen if you make the controller too aggressive? You should be able to experimentally determine the threshold value.

Some debugging recommendations:

- Use the Logic Analyzer to observe your controller's performance
- You can use the Watch feature to tune your control gains online!

It is up to you to decide on which units and scaling to use internally, but you should try to utilise the capacity of the given datatype in a clever way. Recall that the `Peripheral_PWM_ActuateMotor` function accepts a signed 32-bit integer, for example. Additionally, when performing mathematics in C, keep in mind which datatypes you are using;

1. You need to understand how they behave when you exceed the maximum number that it can carry.
2. The order of operators, and choice of literals also becomes very important – for instance, recall that dividing an integer by a larger integer always yields zero.

8.3 To be approved on this task ...

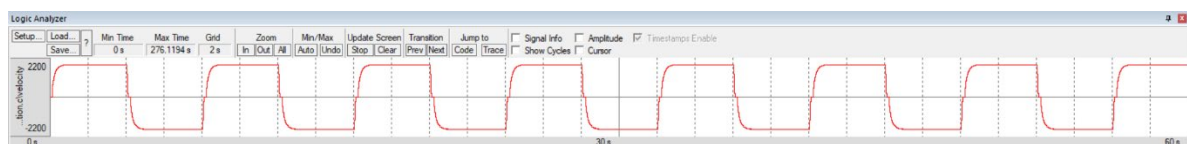
Each student in the group should be able to explain how the code works.

You also need to be able to drive the motor in both directions, by using the two PWM signals with duty cycles between 0 to 100 percent:

- 1) Motor stationary if value is 0
- 2) Motor clockwise if value is > 0
- 3) Motor anti-clockwise if value is < 0

The controller implementation should be *independent* of the sampling period. Instead, use the fact that the time is provided when invoking the function. You should be able to change `PERIOD_CTRL` to 50 ms for example, and still achieve similar performance.

This is not a course in control theory, and so the only requirements on the controller is that it does not result in an unstable system, and that the velocity settles at the reference value within one period of the square wave reference – some oscillations are also permissible! The resulting real-time plot could look something like below.



Remember! As instructed under section 4, you cannot modify the header files you received in the skeleton. You will only submit the controller and peripherals C-files, if they do not work with the given application code you will not pass the assignment!

9 Task2 - Real-time operating systems

In this task, you are to modify your controller application to run under the real-time operating system Keil RTX – An implementation of the CMSIS-RTOS architecture.

9.1 Enable CMSIS-RTOS

You first need to add Keil RTX (The CMSIS-RTOS implementation) to your project.

- 1) Add Keil RTX component to your project
 - a) Select “Manage Run-Time Environment”
 - b) Select the following software component CMSIS → RTOS (API) → Keil RTX:
✓
- 2) If you try to Build the project now, you will notice that several interrupt handlers are defined multiple times, including the SysTick_Handler. This will naturally cause problems.
- 3) To remedy, you need to prevent a conflict in how SysTick is incremented, given that you have the STM32Cube hardware abstraction layer enabled as well.
- 4) Select “Manage Run-Time Environment” again
- 5) Under “Device → STM32Cube Framework (API) → STM32CubeMX”, select the Play button to start the STM32CubeMX application.
- 6) Configure the Nested Vector Interrupt Controller (NVIC)
 - a) Select System Core → NVIC (under Categories)
 - i) In the middle window “NVIC Mode and Configuration”, set Priority Group to 4 bits for pre-emption ...
 - b) Under the NVIC Interrupt Table, set the following pre-emption priorities;
 - i) System service call via SWI instruction: 14
 - ii) Pendable request for system service: 15
 - iii) Time base: System tick timer: 15
 - c) Select the “Code generation” tab (Still the middle window “NVIC Mode and Configuration”, next to NVIC)
 - d) For the column “Generate IRQ handler”, untick the following entries
 - i) System service call via SWI instruction
 - ii) Pendable request for system service
 - iii) Time base: System tick timer
- 8) Finally, generate the new configuration code.
 - 7) Select GENERATE CODE in the top right.
 - 8) You may be prompted to download some software packages. Accept.
 - 9) You will eventually be prompted that the code was successfully generated. Press Close.
- 7) You have now excluded the HAL interrupt definitions, and resolved the earlier conflicts.
- 8) Now, open the RTX_Conf_CM.c in the Project View under ‘CMSIS’.
- 9) Open the Configuration Wizard view, and set the following:
 - a) Default Thread stack size: 1024
 - b) Main Thread stack size: 1024

- c) RTOS Kernel Timer input clock frequency: 40000000 (same as the target core clock)
 - d) Round-Robin thread switching: OFF
- 10) Switch back to the Text Editor view, and place a breakpoint on the switch (error_code) inside the os_error function. This will let you know when the RTOS has encountered an error.

9.2 Initialize the application

You will now change the way the software is initialized to start using the operating system in our application.

- 1) In application.c, include the header file "cmsis_os.h".
- 2) In application.c, change the infinite loop to simply be as follow (the current code in the loop will be moved into the threads):

```
void Application_Loop() {  
    // Do nothing  
    osSignalWait(0x01, osWaitForever);  
}
```

- 3) In main.c, make the following changes to the code – within the appropriate user block. You want to change the timer so that it is based on the OS time, by changing the implementation of the SysTick_ms function.

```
/* USER CODE BEGIN 0 */  
extern uint32_t os_time;  
uint32_t SysTick_ms(void) {return os_time;}  
/* USER CODE END 0 */
```

Where does this os_time come from?

9.3 Your Task - Develop the code

Your task now is to break up the code that is currently in the infinite loop into two separate threads;

- Thread1: Sample the encoder, calculate the control signal and apply it
- Thread2: Toggle the direction of the reference

Which thread should be assigned with a higher priority?

Do this using Delays – where each thread should *begin* with an appropriate osDelay function call, for the correct delay value.

Review what we have learnt in Tutorial3, is there another way to set the periods for threads, which one is better to achieve accurate control?

9.3.1 To be approved on this task ...

Each student in a group should be able to explain how the code works.

You also need to:

- 1) Have a reasonable speed control performance with the RTOS setting, with no performance loss compared to the previous task.
- 2) Make sure the threads satisfy the same timing requirements as with the previous task. Use the “Event Viewer” window to show that the desired execution order is satisfied.

9.3.2 Bonus point

Do this same task using timers and signals, where

- a) You replace the instances of `osDelay` with `osSignalWait`.
- b) Add virtual timers which activate the corresponding signals. You should create a new static callback function that will signal the correct thread based on the function parameter.

10 Task3 - Distributed systems using TCP/IP

This task gives you experience with socket programming in C. You will also learn to work with conditional compilation and advanced features for versioning inside Keil MDK.

The aim of this task is to distribute the functionality of the control system across two microcontrollers. For communication, we will be using TCP/IP, and the client-server architecture.

Arguably, it might not be “realistic” to distribute such a simple application. A case can also be made for other communication protocols that are more suitable for real-time control. However, the purpose is to familiarise yourself with using a high-level communication protocol, the standard socket API. You should also be able to observe how timing and task ordering has an impact on the close-loop performance of an embedded control system. To complete this task, you will need to consider things like handshaking, scheduling, fault handling, and other relevant design choices.

You will divide the code you developed so far to run on two separate microcontrollers:

- 1) **Client** software will run on one microcontroller. Its main functionality is to read the speed sensor and write to the motor.
- 2) **Server** software will run on another microcontroller. Its main functionality is to produce the reference and PID controller signals.

Luckily the code is written in a modular way. So, it should be relatively easy to distribute the modules between the two microcontrollers.

You will essentially need to develop two separate programs (think 2 `main()` functions), yet these programs need to share some code. To facilitate this, you will learn how to work with a Keil MDK project that contains two targets.

All the necessary code to configure your microcontroller, and initialise communication will be provided to you in the subsections that follow. Your task is to write the application-layer code for the distributed system, as specified in section 10.8.

10.1 Configure the ethernet shield

The Ethernet shield has its own microprocessor, buffer, and connectivity capacity. The SPI protocol will be used to connect the microcontroller to the shield. To do this, you should now enable the SPI3 peripheral, and slave-selector pin in STM32CubeMX.

- 1) Select “Manage Run-Time Environment”.
- 2) Under “Device → STM32Cube Framework (API) → STM32CubeMX”, select the Play button to start the STM32CubeMX application.
- 3) Select the “Pinout & Configuration” page
- 4) In the Pinout view, set the pin functionality of PB6 to be GPIO_Output.
- 5) Configure PB6
 - a) Select System Core → GPIO (under Categories)
 - i) In the middle window “GPIO Mode and Configuration”, set PB6’s GPIO Pull-up/Pull-down to Pull-up
- 6) Configure SPI3

- a) Under the “Pinout & Configuration” page, select Connectivity → SPI3 (under Categories)
- b) In the middle window “SPI3 Mode and Configuration”
 - i) Set Mode to Full-Duplex Master
 - ii) Set Hardware NSS Signal to Disable
 - iii) In the middle bottom window “Configuration”, Set Data Size to 8 Bits
- 9) Finally, generate the new configuration code.
- 10) Select GENERATE CODE in the top right.
- 11) You may be prompted to download some software packages. Accept.
- 12) You will eventually be prompted that the code was successfully generated. Press Close.
- 13) Keil MDK might also be prompted that “for the current project new generated code is available for import”. Press YES to import changes.
- 14) You may close the STM32CubeMX application and return to Keil MDK.
- 15) Your Keil project now contains all the necessary code to initialize the device as graphically specified in STM32CubeMX.

Note: There are two types of boards. You should identify which type you have in your equipment in order to understand the behaviour of the reset button.

- 1) The “red” Wiznet boards correctly reset the shield and the microcontroller
- 2) “black” Seeed Studio ones only reset the shield.

10.2 Add an external Ethernet library


You will now include a driver (ioLibrary) that implements the Ethernet API, making it easier for you to develop your application. You can refer to section 7.5 to recall how to import C- and H-files into your projects.

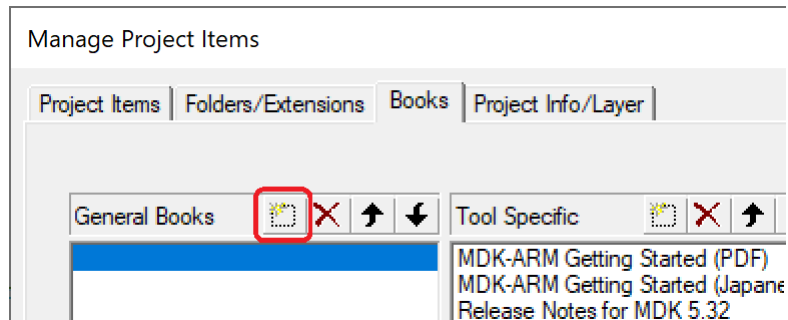
- 1) Download the Wiznet’s GitHub repository
 - a) Go to https://github.com/Wiznet/ioLibrary_Driver
 - b) Select Code → Download ZIP
 - c) Unzip the downloaded zip file to the base directory of your Keil project.
- 2) In Keil MDK, import the following C-files into your project. It is good practice to add them to a new “Source Group”, that can – for example – be called Ethernet.
 - a) Ethernet/socket.c
 - b) Ethernet/wizchip_conf.c
 - c) Ethernet/W5500/w5500.c
- 3) Include the following folders that contain the necessary header h-files in the “include path”.
 - a) Ethernet
 - b) Ethernet/W5500

Note: The socket APIs of this library are based on Berkeley socket APIs. So, the APIs have very similar names and interfaces. But there are some small differences. For your task, you should be able to design your application using only standard socket calls, such as connect, send/recv, getsockopt, etc.

10.3 Import the Socket API documentation

To check how function should be called, you will need to check the API documentation regularly. To facilitate this, you can import it into Keil, to be made available under Books.

Navigate to Project → Manage → Project Items... (or click the  button), and bring up the “Books” tab. In the General Books column, click the New Book icon



Then add the documentation file, found in the ioLibrary repository:

Book title: Wiznet Socket API (W5500)

Book Path: `.\ioLibrary\iolibrary.chm`

10.4 Create two targets

Now that you have laid the groundwork, let's create two targets for the server and client applications.

- 1) In Windows Explorer, go to the Source folder, and create two new copies of the `application.c` file
 - a) `app-client.c`
 - b) `app-server.c`
- 2) Create new Client and Server Targets
 - a) In Keil MDK, right-click on your target (Target 1) and select Manage Project Items...
 - b) In the same Group that contains the `application.c` file, add the two newly created files.
 - c) Under Project Targets, add two *new* targets, named “Client” and “Server”.
 - d) press OK.
 - e) Next to “Options for Target” (the magic wand), you will find a dropdown list from which one can choose among the targets in the project.
 - f) Select the Client target. Then select the Server target.
 - i) Notice how the Project window adjust accordingly. At the moment there are no differences in the files between the targets. So you won't notice much differences. But once we make changes with one target and not the other, you will notice the differences.
 - ii) If you try to Build the client now, you will get a similar error to when you added RTOS for the first time; namely that the Application functions are multiply defined.

- iii) Of course, you need to include *only* one of the implementations of the application interface for each target. We fix this in the next steps.
- 3) Configure the Client target
 - a) Make sure the Client target is selected in the dropdown list.
 - b) In the Source Group that contains the application.c files, right-click on application.c, and select Options for File.
 - c) untick Include in target Build, to remove this file from the build process.
 - d) Similarly, remove app-server.c from the build.
 - e) The result is that Keil will only compile the app-client.c when the client target is chosen.
 - f) Right-click on main.c, and select Options for File.
 - g) select Always Build (the option may be grey. Select it once to unselect, and then select it again to get a bold selection)
- 4) Configure the Server target
 - a) Make sure the Server target is selected in the dropdown list.
 - b) Repeat the same steps performed earlier for the Client target. Of course, you should remove app-client.c instead of app-server.c from the build.
- 5) Notice that excluded files have a red symbol on them, while main.c has an asterix. Switch between the Client and Server targets, and notice the differences now.

Note: An alternative to creating two targets would be to create two MDK projects for each of the Server and Client applications. With such an approach, you will then need to deal with the following issues:

- 1) How to share common code that is needed by both the Client and Server? Create a third common library?
- 2) How to switch between the two applications? Keil allows only one project to be open at a time.

10.5 Configure targets for conditional compilation

In section 10.3, you created two targets, and configured each to include/exclude specific files. But what if you want to use the same file in both targets, with only small changes in the logic for each target? There are scenarios where you would want to include/exclude small pieces of code within the same file, while keeping most code in common. For this, conditional compilation can be useful. You will see how this is used in the section that follows. For now, let's define some terms that differentiate the two targets.

- 1) Select the Client target.
- 2) Open **Options for Target...**
- 3) Select **C/C++**
- 4) Under Preprocessor Symbols, enter the following two terms under Define.
 - a) `_ETHERNET_ENABLED _CLIENT_CONFIG`
- 5) Select Ok.
- 6) Select the Server target.
- 7) Repeat the steps above, with the following terms under Define instead:
 - a) `_ETHERNET_ENABLED _SERVER_CONFIG`

10.6 Connect the microcontrollers

During development, you would want to have two microcontrollers connected to your computer. One microcontroller is dedicated for the Client target, while the other is for the Server target.

Since the Client & Server targets were copied from the same original Target, they are currently both configured to use the same ST-Link Debugger. This is not desirable, since each microcontroller has its own ST-Link Debugger.

Just to confirm that both targets share the same ST-Link configuration:

- 1) Select Client target
- 2) Select “Options for Target”
- 3) Select Debug page
- 4) Select Settings, next to the ST-Link Debugger
- 5) Note the Serial Number.
- 6) Now select Server target, and note the Serial Number.
- 7) Unsurprisingly, both targets are using the same Debugger.

You need to change one of the targets, by reconfiguring its ST-Link Debugger. Let's decide that the Client target will use the microcontroller you have used so far (we call it Client Microcontroller from now on). So, we only need to reconfigure the Server target.

- 1) Disconnect the Client microcontroller, to avoid confusion when selecting Debuggers.
- 2) Connect the new Server microcontroller to your machine.
- 3) Select the Server Target.
- 4) Select “Options for Target”
- 5) Select Debug page
- 6) Select Settings, next to the ST-Link Debugger
- 7) Since there is only one debugger connected, the target Debugger will be set correctly.
- 8) Now, connect the Client microcontroller (Keep the Server connected).
- 9) You should now be able to observe 2 different Serial numbers when you check the ST-Link Debugger settings for each of the Client and Server targets.
 - a) Just above the Serial Number, note also that the drop-down list of Debug Adaptor Units now contains 2 entries. Selecting a different entry will change the ST-Link Debugger used for that particular Target.

10.7 Initialize the ethernet shield

You will now add the necessary functions needed to interface with the shield. Essentially, this consists of letting the Wizchip library know which SPI peripheral to use. Additionally, it will set the information necessary for the two microcontrollers to find each other over the IP protocol.

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. Try to instead understand what is being done.

- 1) Include the wizchip drivers:

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

What does `#ifdef _ETHERNET_ENABLED` mean? Read through the previous sections and see if you can recognise the term `_ETHERNET_ENABLED`

- 2) Add initialisation *callback function* (your implementations of those functions will be called by the driver later) declarations:

```
/* USER CODE BEGIN PFP */
#ifdef _ETHERNET_ENABLED
static void Ethernet_CS_SEL(void);
static void Ethernet_CS_DESEL(void);
static uint8_t Ethernet_RB(void);
static void Ethernet_WB(uint8_t b);
static void Ethernet_Config(void);
#endif
/* USER CODE END PFP */
```

- 3) Right after all peripherals are initialized, initialize the Ethernet driver.

```
/* USER CODE BEGIN 2 */
#ifdef _ETHERNET_ENABLED
Ethernet_Config();
#endif
Application_Setup();
/* USER CODE END 2 */
```

- 4) Implement callback functions and the `Ethernet_Config` initialization function. Callback functions will be called by the driver for SPI communication with the shield.

```
/* USER CODE BEGIN 4 */
#ifdef _ETHERNET_ENABLED
// Callback to select slave for SPI
static void Ethernet_CS_SEL(void)
{
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
}

// Callback to deselect slave for SPI
static void Ethernet_CS_DESEL(void)
{
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
}

// Callback to receive over SPI
static uint8_t Ethernet_RB(void)
```



```
{
    uint8_t rbuf;
    HAL_SPI_Receive(&hspi3, &rbuf, 1, 0xFFFFFFFF);
    return rbuf;
}

// Callback to transmit over SPI
static void Ethernet_WB(uint8_t b)
{
    HAL_SPI_Transmit(&hspi3, &b, 1, 0xFFFFFFFF);
}

// Configure the ethernet shield over SPI
static void Ethernet_Config(void)
{
    reg_wizchip_cs_cbfunc(Ethernet_CS_SEL, Ethernet_CS_DESEL);
    reg_wizchip_spi_cbfunc(Ethernet_RB, Ethernet_WB);

    wizchip_init(NULL, NULL);

#ifdef _SERVER_CONFIG
    wiz_NetInfo netInfo = {
        .mac = {0x52,0x08,0xDC,0x12,0x34,0x57},    // Physical (MAC) address
        .ip  = {192, 168, 0, 10},                  // Logical (IP) address
        .sn  = {255, 255, 255, 0},                 // Subnet mask
        .gw  = {192, 168, 0, 1}},                 // Gateway address
#else
    wiz_NetInfo netInfo = {
        .mac = {0x52,0x08,0xDC,0x12,0x34,0x58},    // Physical (MAC) address
        .ip  = {192, 168, 0, 11},                  // Logical (IP) address
        .sn  = {255, 255, 255, 0},                 // Subnet mask
        .gw  = {192, 168, 0, 1}},                 // Gateway address
#endif
    wizchip_setnetinfo(&netInfo);

    wiz_NetTimeout timeout = {
        .retry_cnt = 2,                            // Retry count (RCR)
        .time_100us = 2000},                      // Retry time value (RTR)
    wizchip_settimeout(&timeout);
}
#endif // _ETHERNET_ENABLED
/* USER CODE END 4 */
```

Callback functions are nontrivial as they make code execution flow non-obvious. Why did Wizchip decide to use them in their driver?

The Ethernet driver is configured and initialized. You can now make TCP/IP socket function calls directly, without any need for any additional setup of the lower layers.

10.8 Conditional compilation - Explained

In the code of the previous section, hopefully you have noticed the code `#ifdef ... #endif`. What does this mean? This is a directive to the Preprocessor (a step before the compiler compiles the code in the file) to include the C-code inside the `#ifdef ... #endif` block, if the term (for example `_SERVER_CONFIG`) is defined. Otherwise, the compiler won't even see this C-code block. The effect of using this directive is to make the same file contain different C-code for the compiler, depending on target settings.

And where did `_SERVER_CONFIG` come from? See section 10.4, and hopefully you can connect the dots yourself.

When you now navigate to the `main.c` file, you will notice that some of the Ethernet code just entered is greyed out depending on whether Client or Server target is selected. This means that the grey code will not be compiled. This is called *conditional compilation*, and is used throughout both the CubeMX and CMSIS code.

In this particular case, what is the effect of this conditional compilation? How will the code differ between the Client and Server targets?

Note: It is important to tick **Always Build** for the `main.c` for *all targets* to make sure that the conditional compilation works as intended when switching between targets.

10.9 Internet protocol (IP) configuration

The `netInfo` struct contains configuration information used for the internet protocol. Below is a brief explanation of each property.

Physical (MAC) address	Address used by the data-link (ethernet) layer to distinguish interfaces on a network. These must be unique on the network, but will not be used after setup. The Address Resolution Protocol takes care of the rest.
Logical (IP) address	Address used by the network (IP) layer to distinguish hosts on a network. These must be unique on the network, and you will use the IP address when connecting to a socket.
Subnet mask	Defines the size of the subnet. {255, 255, 255, 0} means that all IP addresses with prefix 192.168.0 belong to the subnet. Any data addressed outside the subnet will be sent to the default gateway, which is usually a router.
Gateway address	Physical address of the default gateway. This will not be used, since there is no inter-networking in this project.

10.10 Your Task - Develop the code

You are now ready to complete your task, by programming the socket-based communication. Recall that you are to distribute the functionality of the control system according to the following architecture:

- 1) The Client shall
 - a) Read the encoder and calculate the RPM
 - b) Transmit the velocity to the server
 - c) Receive the control signal from the server
 - d) Actuate the motor with PWM
- 2) The Server shall
 - a) Generate the alternating reference square-wave
 - b) Receive the velocity from the client
 - c) Calculate the control signal from the error
 - d) Transmit the control signal to the client

Tips

- 1) You should re-use the threads from the previous task, with some minor changes.
- 2) Make sure you read and understand the WIZnet socket APIs
- 3) The TCP handshake should take place in the infinite loop.
- 4) The functions for sending and receiving are *blocking*, so they will halt their thread execution until they complete.
- 5) You will need to use some inter-thread communication (signals, mailboxes etc.) to synchronise threads.
- 6) To avoid the two microprocessors drifting out of sync, only one should dictate the application frequency (e.g. with a virtual timer). The corresponding thread on the other device could be activated by new incoming messages.
- 7) You will likely need to decrease the sample frequency (larger sample period), to give enough time for the communication to take place within one sample.

10.10.1 To be approved on this task ...

Each student in a group should be able to explain how the code works.

You also need to:

- 1) Have a reasonable speed control performance with the RTOS setting, with no significant performance loss compared to the previous two tasks.
- 2) The system must work no matter which board is powered up first or when the cable is connected.
- 3) The client and server must be able to recover when the connection is temporarily broken. Connection can fail in two scenarios (a) one of the nodes loses power, or (b) the cable is temporarily disconnected.
- 4) Draw a timing diagram to explain how the control loop works. A UML sequence diagram [Wolf-CasC, p.31] can be used, for example.
- 5) You should be prepared to answer questions along the following lines:
 - a) You should be able to explain how the connection is established, how the communication takes place and what information is being sent over the network.
 - b) How did you design the system to minimise the end-to-end delay of the control signal?

10.10.2 Bonus Points

Bonus points are awarded for a successful completion of the following task:

- 1) The motor must be signalled to stop **within one sample** of when the connection is broken, or the server is reset/powered down.