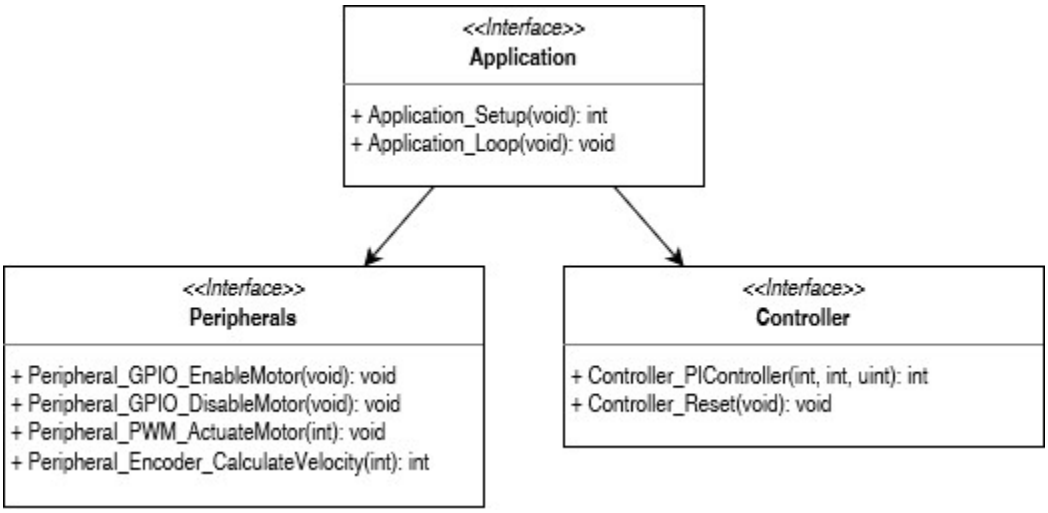
# Programming Task 1: Digital I/O and control

Start Assignment

- Due 4 Feb by 18:00
- Points 1
- Submitting a file upload
- File types c
- Available after 19 Jan at 10:00

# Summary

The purpose of this task is to develop modules for the **controller** and **peripherals**. This means that you will develop all the features specified in the contracts (header files), including those that may not be used in the current state of the application. Additionally, you should not have any cross-module dependencies. For example, the controller should not be dependent on any information contained in the peripherals, application, or main files.




*Application, Peripherals and Controller modules*

# 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. 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 one.

All the peripheral **initialization** has been done for you in Task 0. This includes enabling and configuring the necessary control registers. Timer 1 is configured to count the encoder increments, Timer 3 is configured to generate two PWM signals, and GPIO Port A has two pins ready for digital output. You only need to read/write to the data registers for the necessary peripherals. It is strongly recommended that you revisit Tutorial 2 before beginning this part.

## Reading the encoder

Remember, the configuration is already done; you just want to read the register containing the counter value. To help you with this task, start by answering the following questions:

1. *Which* register is it that you are interested in reading? Check the reference manual **[STM-RefMan]**, Section 30.3.22!
2. *Where* is it located? Check its address! Recall from Tutorial 2, (Base address + Offset) to get the memory location...
   - Check **[STM-RefMan]**, Table 2 for the base address, and **[STM-RefMan]** Section 30.4 for the particular register offset.
3. *What* is the value stored at that location? Add the register address to the Memory window in Keil, and see if it changes by rotating the motor by hand.
   - You can right-click in the Memory window and choose "Decimal" and "Signed->Short" for a more human-readable format.
4. *How* can you instead access that address with C-code, using the CMSIS-style macros (see Tutorial 2)?
5. *Why* can you not read the entire 32-bit register? What could happen if you do not mask away other the bit fields?
6. *Who* decides if the number in the counter should be interpreted as signed or unsigned?
   - On a bit-level, what is the actual difference between -32,768 and 65,535?

> Without connecting the power, try spinning the motor by hand in both directions. You may notice that the number of counts is larger than the encoder resolution specified in the datasheet. To understand how the encoder mode works, see the reference manual **[STM-RefMan]** , section 30.3.22.

## Actuating the motor

Next, you can focus on driving the motor. Before you continue, you should read the user manual for the BTN8982 DC Motor Control Shield **[Infineon-Shield]** . Make sure to read sections 1.1, 1.2, 2.4 and 2.5.

Based on the content in the user manual, you should be able to determine that to drive the motor, you will need to control four pins in total.

> The user manual labels the pins by their name on the Arduino Uno. The microcontroller we are using has Arduino-compatible headers but does not use the same labeling. Check **[STM-Pinout]** to see what they correspond to.

You will need to use GPIO to enable the half-bridges, and Timer3 to generate the corresponding control signals.

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

You will have noticed that the function prototype `Peripheral_PWM_ActuateMotor` has been specified to utilize 31 bits of the integer, and as such, expects your implementation to read an input of 1,073,741,823 as 100% duty cycle clockwise, and -1,073,741,824 as 100% duty cycle counter-clockwise. You should apply the necessary scaling inside the appropriate function. Using the number 1,000,000,000 is a decent approximation, but there is a much more efficient solution using bit-shifting...

Use the datasheets for the motor and encoder to validate the maximum RPM, and make sure your algorithm gives a smooth signal, handling counter overflows using proper integer arithmetic!

You will likely want to be able to test your code, even though the controller is not yet ready. For this purpose, there is a global variable defined in controller.c, which lets you mock-up the controller to return some hard-coded, non-zero number—by default 25% duty cycle. Similarly, there is a variable in peripherals.c defined in the global scope, allowing it to easily be observed in the Watch window when debugging.

# Controller module

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

You do not have to use a model-based design approach, such as discretization and pole-placement! For this course, manual tuning by “trial-and-error” is enough. Consider the following steps to incrementally develop the controller:

1. When you have validated the motor/encoder in open-loop control, change it to a simple P controller.
    - A good starting value can be achieved by choosing the controller gain equal to the inverse of the steady-state gain (the ratio between steady-state input and output). You know the range of input already, from designing the peripherals module. The range of the motor output can be found in the datasheet.
    - Manually tune the proportional gain, without introducing too strong oscillations.

2. Next, add integral action to eliminate the steady-state error.
    - A good starting value is to let the integrating time constant $T_i$ be equal to the time constant $T$ of the uncontrolled system. Use the Logic Analyzer to estimate this.
    - For the integral part, it may be easier to use the parallel form; $K_i = K_p / T_i$.
    - When implementing the integral part, pay attention to the units. Depending on what you "measure" the time constant in, you will need to scale the integral proportionally—since system time is counted in milliseconds.
    - You will still need to manually fine-tune the integral gain to achieve the desired closed-loop performance.
    - Finally, it is always good practice to implement some basic anti-windup. For this case, it is sufficient to apply clamping, preventing the integral from growing too large (and overflowing). Test it by holding the flywheel still with your fingers while monitoring the control signal in the Analyzer.

Some debugging recommendations:

- Use the Logic Analyzer to observe your controller's performance.
- You can use the Watch feature to tune your controller online, without re-building!

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!
3. Also note that the controller interface uses pass-by-reference instead of pass-by-value. You will need to de-reference to get the value. The keyword `const` specifies that the values are read-only.

## Continuous operation

The purpose of `Controller_PIController` is normally to be called continuously over long periods of time. However, there may be cases where the application would disable the motor and suspend control. When control is resumed, it is undesired for the controller to retain old values for the internal states. For this reason, `Controller_Reset` can be called by the application in between.

If you have implemented the controller already, you will likely also have noticed that during the first pulse of the reference, your motor overshoots—this is a related issue. Depending on your implementation, the first segment of the integral is calculated from the time when the microprocessor is powered on, until the first time the controller is called! This could be potentially catastrophic…

One way to address this is to inhibit the controller during the first call in a sequence of control actions. For this call, the controller can return zero. You can assume that the application would call `Controller_Reset` once before the start of every such sequence.

Your final goal is to improve your code and implement the function `Controller_Reset` such that the desired performance is achieved.

# 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

Clockwise can be defined by positioning the board such that you are facing the flywheel.

The controller implementation must 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.

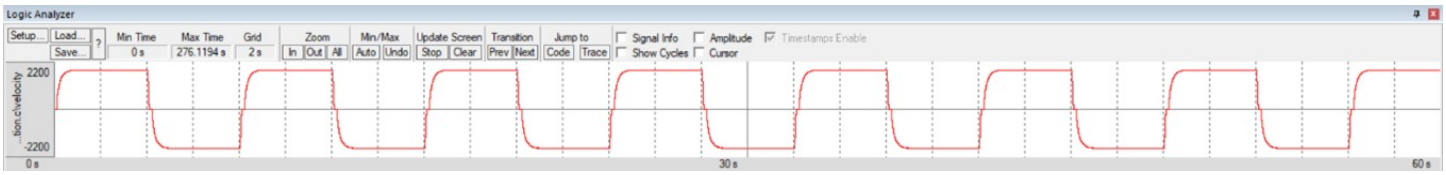
Furthermore, the code should satisfy the following specifications:

- Neither controller nor peripherals should use any floating-point instructions—only integer datatypes will be permitted.
- All interface “contracts” must be satisfied—including the requirement on the resolution on the duty cycle—as specified in the peripherals header-file.
- The controller must return zero the first time it is called after a controller reset has been performed.
- The controller must implement some anti-windup measure, such as clamping, in order to prevent integer overflows.
    - You can easily test this by disconnecting the power to the motor and studying the control signal using the Logic Analyzer.

You must also "prove" (convincingly enough) that your controller is guaranteed safe, i.e., under normal operation, no arithmetic operations in your code should result in unexpected behavior, such as implicit type conversions or integer overflows. **Hint**: Look into saturation intrinsics (see **this Lecture (https://canvas.kth.se/courses/59527/files/9577710?wrap=1)** ↓ **(https://canvas.kth.se/courses/59527/files/9577710/download?download_frd=1)** )!

Finally, this is not a course in control theory, and so the only **performance** requirement on the controller is that the velocity settles at the reference value within one period of the square wave reference; some oscillations are permissible. However, you should be able to explain how you went about the process of controller design and tuning, and how the control algorithm works. The resulting

real-time plot could look something like below.




**Remember!** As instructed, 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!