Learning Objectives:

- Gain experience with microprocessor-based closed loop control
- Gain experience with multi-tasking, synchronization, and interprocess communication (IPC) using FreeRTOS, an open source, real-time kernel for embedded systems
- Apply the concepts learned in Project #1
- Flex your hardware-building muscles w/ “real” circuits

Project 2 demos will be conducted on Monday 26-Feb-2018. Project 2 Deliverables are due to D2L by 10:00 PM on Friday 02-Mar-2018. This project will be done in teams of two.

Project: Microprocessor-based Closed-loop Motor Control

This project is designed to give you practical experience with a common, and often essential, embedded system function – closed loop control. Closed-loop control involves using feedback from one or more sensors to adjust the input parameters that control the output of the circuit the application is controlling. An example of a control system is automotive cruise control. The driver sets a target speed, engages the cruise control and takes his or her foot off the accelerator. The cruise control does its best to keep the car moving at the target speed (called the *setpoint*) even as the car goes up and down hills. The control circuit for this project is much simpler than cruise control but the same general principles apply.

You will be provided with a schematic and Bill of Materials (BOM) for a simple motor/generator circuit. You will mount two small DC motors on a board of your own choosing. You will also mount a Hall sensor on the same board as the motors to count the number of times a small magnet mounted to the motor shaft of the first motor (called the drive motor) passes by the sensor when the motor is turning. A Digilent PmodHB3 (2A H- bridge circuit) will be used to drive the first motor and to wire the output of the Hall sensor back to the Microblaze via one of the PMOD connectors on the Nexys4 DDR.

The control circuit should make use of PWM to drive the PmodHB3 EN pin, thus controlling the speed of the motor. You can adapt the PWM generation circuit from Project #1 or make use of an AXI Timer configured for PWM using the PWM driver code provided by the instructor and included in the project release. You will also need to route the output of the slide switch on the PmodENC to the DIR input of the PmodHB3 to control the direction of the motor.

The configuration of a single magnet and single Hall sensor can be used to detect the speed of the motor but not the rotation direction. Determining which way the motor is turning would require a second magnet mounted 90° away from the first magnet and a second Hall sensor (which would be a nifty addition for extra credit if you can get it working).

You can purchase the electronic and electromechanical components you need for this project from the EPL (FAB 84-20).

Motor System

The electromechanical part of this project consists of two small 5VDC "hobbyist" motors. The first motor (called the drive motor in this document) is connected to a Digilent PmodHB3, a 2A H-Bridge capable of driving up to a 12V motor. The PmodHB3 inputs are EN (Enable) and DIR (direction). EN is driven by a PWM signal (0 – 3.3V) to enable/disable the bridge and thus control the speed of the motor. The larger the duty cycle, the higher the average voltage to the motor windings and the faster the motor turns. The motor windings are best powered from an external power source. You could use one of the bench supplies in the lab or you could do what others have done and sacrifice one of your old phone chargers by cutting off the connector and wiring the +5V and GND leads to the PmodHB3. The ideal phone charger would provide 5VDC @ 1500 ma but the current needs of the motor should be less, except maybe for startup.

The second motor (called the load motor in this document) is used as a variable load on the drive motor. It should be mounted on the same board as the first motor with the drive shafts of both motors connected; neoprene tubing or heatshrink tubing can be used for this. The mechanical connection should be tight enough to avoid slippage, without compromising the placement of the magnet (for the Hall sensor) on the drive motor shaft. The load motor operates as a generator by wiring a resistor and a switch between the M+ and M- tabs on the load motor. Closing the switch allows current to flow through the circuit so the resistor adds load.

The speed of the drive motor is detected with a Hall Effect sensor mounted close to the shaft on the drive motor. A small magnet (included in the kit from the EPL) should be securely mounted to the drive motor shaft, perhaps with a bit of superglue. The output of the Hall Effect sensor will pulse every time the magnet passes near it. You may have to play with positioning the sensor to get a reliable reading. The speed (RPM) of the motor can be calculated by counting the number of pulses in a known interval; for example, counting the number of pulses in 1 second will give you revolutions per second. Multiply by 60 and you have revolutions per minute. You may find it advantageous to do some simple software filtering to get a "true" reading. For example, you could average several readings together and/or you could toss out "crazy" readings (ex: the motor speed suddenly increased from, let's say 500 RPMs to 10000 RPMs in a second or two).

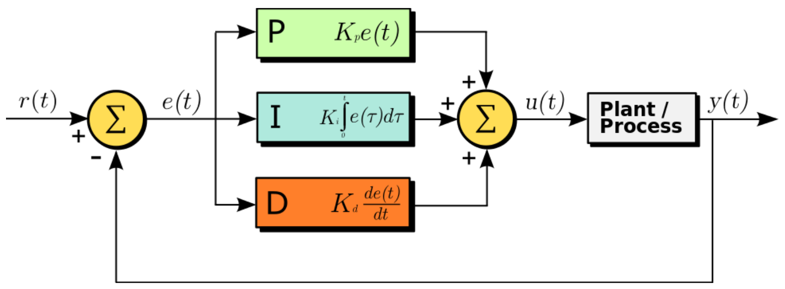
Application

You will develop an application (C program) that demonstrates closed loop control. The application will display information about the system on the PmodOLEDrgb and the LEDs on the Nexys4 DDR.

The application should implement a PID controller to provide closed loop control of the motor. The user should be able to tune the control circuit by using the btnU (top button) and btnD (bottom button) to slowly increase (btnU) or decrease (btnD) the control parameters. We recommend that you use switches 3 and 2 to select which parameter (KP, KI, or KD) is being modified by the buttons. You can use 3 LEDs, one for each constant, to indicate the constant you are modifying.

Consider using fixed point arithmetic or floating point to provide one or two decimal points of precision. For example, say you want to vary KP from 0 to 100. Instead of incrementing by 1 between 0 and 100, increment between 0 and 1000. Divide the count by 10 and you have an implied decimal point – this is fixed point arithmetic. If adding a single decimal point to KP does not give you fine enough resolution to tune your control loop than consider implementing two decimal points by counting between 0 and 10,000 and dividing by 100.

The desired control signal should go directly to the H-bridge driving the motor. An "error" signal is generated if there is a difference between the desired and actual speed; that "error" *e* in the figure) signal is used by the PID algorithm to adjust the control parameters (K_p, K_i, *and* K_d in the figure) to adjust the PWM value to the PmodHB3. Use floating point to ensure the best results.



Source: http://commons.wikimedia.org/wiki/

An important design decision is how to represent the motor speed or rotational velocity (both desired and actual) and the control constants K_p, K_i, K_d. The motor speed will be determined by an 8-bit unsigned (0-255) quantity because we are implementing 8-bit PWM. That 8-bit PWM value should scale over the entire motor speed range as much as possible. Also, you will need convert the Hall sensor pulses into a comparable speed measure that corresponds to the PWM value. For example, if the maximum motor speed is 6000 RPM (PWM = 255) then the half-speed (PWM = 127) should be 3000 RPM. The results should be mostly linear.

The control constants, say K_p, can also represented by an 8-bit quantity. What happens when you multiply the "error" signal (the difference between desired and actual speed) by K_p? Multiplying two 8-bit numbers gives you a 16-bit number. To convert back to 8-bit, you have to decide how many bits to throw out to get a resulting signal that is appropriate.

Since the error signal can be positive (motor is turning slower than the setpoint) or negative (motor is turning faster than the setpoint) it is possible for the sum of the desired speed and (K_p * error) to overflow or underflow. Since the command to the motor can only be positive (i.e. you can't set PWM to a value < 0) you should "clamp" the PWM value to 0 (min) or 255 (max). You also need to be careful of how to *handle maximum and minimum speeds with respect to the error signal. Ultimately, the magnitude of the* error signal will determine how you scale your control and speed representation. You will have to experiment with what is the best magnitude of the correction signal, error, and KP.

Nexys4 DDR Device Mapping

- *PmodOLEDrgb:* The OLED display is used to display the K_p, K_i, *and* K_d values and Motor Speed. Additional innovations are welcome.

- *PmodENC:*
  - Rotary Encoder – The rotary encoder knob is used to select the motor speed. Twisting the knob the clockwise increases the motor speed (like a volume control). Twisting the knob counter clockwise decreases motor speed.
  - Switch – The switch on the PmodENC changes the direction of the motor. Looking at the PmodENC with the switch on the top, with the switch in the left position, the motor turns counterclockwise, with the switch in the right position, the motor turns clockwise. This command only takes effect when the rotor is stopped (0 RPM). Make sure that the direction is not changed while the motor is running.

- *Nexys4 DDR Slide Switches:*
  - Switches[15:6] – not assigned
  - Switches[5:4] – controls the amount the selected control constant is incremented or decremented on each press of the btnU or btnD buttons. These switches are only needed if you implement one or two decimal fractions for the control constants. Pick whatever constants you need for your algorithm but one suggestion is to implement the following:
    - 00: If both switches are open, the Rotary Encoder changes the motor speed by ±1
    - 01: If Switch[4] is closed, change the K_p or K_i, or K_d value by ±5 on each button press
    - 1x: If Switch[5] is closed, irregardless of Switch[4], change the K_p or K_i, or K_d value by ±10 on each button press.
  - Switches[3:2] – Used to Select one of the three PID parameters (KP, KI, KD) to control.
  - Switches[1:0]
    - 00: If both switches are open, the Rotary Encoder changes the motor speed by ±1 on every "click"
    - 01: If Switch[0] is closed, the Rotary Encoder changes the speed by ±5
    - 1x: If Switch[1] is closed, regardless of Switch[0], the Rotary Encoder changes the speed by ±10

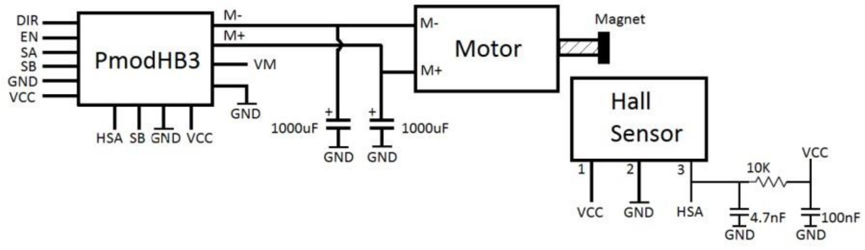
- *Nexys4 DDR Pushbuttons:*
  - btnC – sets desired motor speed to 0 and turns off the PWM signal to the motor and resets the control parameters to a default value of your choosing.
  - btnU – Increment the Proportion control constant
  - btnD – Decrement the Proportion control constant

- *Nexys4 LEDs:*
  - LED[15:3] – not assigned
  - LED[2:0] - Display the selected PID constant

Hardware

Control Circuit

The Project 2 release package contains a schematic, a PmodHB3 Reference Manual, and Bill of Materials (the BOM) for the suggested control circuit. The motors needs to be mounted securely on a solid platform, proto board or small piece of plastic. The motor may be changing speeds quickly so it should be mounted securely. The Hall sensor and its circuit needs to be positioned close to the drive motor shaft. The magnet needs to be mounted on the drive motor shaft so that when the shaft rotates the magnet passes near the Hall sensor which will detect the magnet's presence.



VM - Connect it to VCC
VCC - FPGA 3.3V output from Pmod Connector.
GND - FPGA Pmod Connector ground.

The driver motor output and Hall sensor input need to be connected to the PmodHB3 as shown in the schematic. HSA (SA_IN on the PmodHB3 schematic) is pin 1 of the 4 pin header on the PmodHB3. The Hall Effect sensor has an open drain output meaning it needs to be connected to a pullup resistor. According to the PmodHB3 schematic SA_IN has a 10K pullup resistor and passes through a Schmitt trigger inverting buffer before being routed to pin 3 on the Pmod connector. You may be able to leave out the 10K pullup on this circuit schematic because it is built into the PmodHB3, but I would include the 4.7nf and 100nf capacitors to GND near the Hall sensor.

The Load motor shaft is mechanically connected to the drive motor shaft. The load motor operates as a generator and it should be connected to a resistor and a switch. Closing the switch allows current to flow through the circuit and the resistor adding additional load to the generator.

The control circuit and hardware can be built in whatever manner you are comfortable with. You can use a proto strip (the white plastic boards with rows of holes for components and wires) or you can build something a bit more lasting on a piece of perf board (the board containing rows of solder plated holes) or you can design, fabricate, and assemble a printed circuit board in the EPL.

The Tachometer

The magnet mounted to the drive motor shaft and the Hall Effect sensor can be used to determine the speed of the motor. Although you can repurpose your hardware pulse-width detection from Project 1 as a tachometer, we recommend an alternate approach that is more suitable for calculating RPM (Revolutions Per Minute).

The Hall sensor produces a pulse every time the magnet mounted to the motor shaft passes by the sensor. The higher the rotational speed of the motor, the closer together the pulses will be. In effect you are determining an unknown frequency for a known interval (seconds or minutes). As we discussed earlier in the term, a way to do that is to count pulses for a known amount of time.

With that in mind it should be fairly easy to implement a Verilog module that detects a rising edge on the sensor and counts the number of rising edges in a 1 second interval. Multiply that count by 60 and you have RPM's. Your implementation can either produce a count of rising edges in 1 second that can be read by the Microblaze via a register and converted to RPM's in your application or, it occurs to us, that you could do the multiply by 60 directly in the implementation and return RPM's to the Microblaze through a register. The Series 7 FPGA on the Nexys4 DDR contains a number of hardware multiply blocks that can perform fast multiplications. Clocking the edge detection logic with the 100MHz AXI clock should provide plenty of margin for detecting a rising edge on an under 50 KHz signal from the Hall sensor. You may also be able to use an AXI timer in capture mode to perform a similar function.

Embedded System Configuration

We will be using the Nexys4DDR board file to create the system instead of starting with a nexys4fpga.v and constraints file like we did in Project #1. Include the external DDR memory, switches, LEDs, and buttons in your configuration. You will want the GPIO that is created for the buttons and switches to be capable of generating an interrupt whenever any of the buttons are pressed or the value of the switches changed.

The resulting embedded system should have this minimum configuration. You can add additional hardware as you see fit:

- Microblaze, mdm, etc. Configure hardware floating point in the Microblaze. That should keep the program size reasonable and provide fast computation for the control loop.
- Local (BRAM) memory of 128KB for program/data memory
- External DDR SDRAM, Caches are used, since the hardware is targeted to run with FreeRTOS
    - Cache - Enable, 32KB
    - DDR SDRAM 128Mb (On-Board).
- FIT timer set to generate 5 KHz interrupts: The interrupt can be used to debounce the pushbutton input, generate delays used in the logic, and to capture and filter the tachometer readings.

- GPIO, one port, 4 bits wide: This is an output port that you can place debug information on.

- Digilent PmodEnc and PmodOLEDrgb IP: Provide access to the PmodENC and PmodOLEDrgb. You can use the same drivers that you used in Project #1.

- UartLite peripheral: Used to send the signals to PC to be plotted on a graph. Configure the UART Baud Rate to 15200, N-8-1.

- AXI Timer/counter 0: Used to generate the "systick" for FreeRTOS.

- AXI Timer/counter 1 (optional) May be used to generate the PWM signal to the PmodHB3 or to generate the clock for your PWM generation logic.

- One AXI Watchdog Timer used to provide a Watchdog timer for the system. Configure the Watchdog timer interval to about 2 or 3 seconds lest you get tired of waiting for disaster to strike.

- One AXI Interrupt Controller with the Watchdog timer interrupt, Timer 0 interrupt, and AXI GPIO interrupt from the pushbuttons as interrupt sources. You may want to add a 4th interrupt that generates an interrupt at a fixed interval to set the rate for the PID control update. This fixed interval interrupt could be generated directly in Verilog or by using a FIT Timer.

You may want to change the port/pin assignments in the constraints file to move the PmodOLEDrgb from the JA connector to the JB connector. Doing so will free the JA connector for your PmodHB3 control circuit interface. The embedded system "wrapper" file can be created with the IP Integrator in Vivado.

Embedded System Drivers and Link Map

Your Project 2 application will use the drivers and standalone OS and FreeRTOS provided by Xilinx. The following additional drivers and files should be included in your software platform:

- PmodEnc and PmodOLEDrgb drivers as used in Project 1.

- The AXI Timer PWM library included in the release if you are planning to generate the PWM signal to the driver motor controller.

- Your driver for the Tachometer.

Project 2 Tasks Summary

Select a partner

You will work in teams of two for Projects 2. We have set D2L up for self-enrollment. We will assign teams for any students who have not indicated a team within a few days of project assignment. To self-enroll in a group:

- One member of your team should claim an unused group (Group #) and add himself/herself to the group
- Same person should email, text, or whatever the other member(s) of the team with the group number
- Other member(s) of the team should self-enroll in the same group
- When a group is full I will rename the group to match the team members

While there are many ways to split the work on this project one suggestion is that one partner creates the embedded system and builds the control circuit hardware while the other partner creates the control application and peripheral. Both partners could (and should) team up on the control system design, system integration and data gathering and reporting. While you may collaborate with other teams, all of the work you submit must be your own and those you collaborated with should be named in your report.

Download the Project 2 release package

Download the Project 2 release package from the course website.

Mount the motors, build the control circuit and connect the circuit to your FPGA development board.

Suggestion: At this point in the project do not connect the shafts of the two motors together. Doing this could make it easier to implement your application and test the control loop.

A former student suggested that you may be able to use the PWM and pulse-width detection circuitry from your Project #1 to test your control circuit. Connect the PWM output from your Project 1 to the PmodHB3. This allows you to start operating the motor without control feedback. Make sure that the direction is set to a constant logic level before the PWM output is applied to the EN signal of the PmodHB3 since you should never switch directions while the motor is turning. This is because doing so will enable short circuits from VCC to GND, potentially damaging the components in the H-Bridge.

You can (and should) characterize the control circuit by sweeping the PWM value from 0 to 255 and observing the output of the Hall sensor on an oscilloscope or logic analyzer. Doing so will provide the minimum and maximum speed of the motor and the offset (smallest duty cycle) needed to start the motor rotating. Adding some delay between PWM values should give the circuit a chance to settle.

Implement your motor control and Tachometer drivers

The driver package should most likely include these functions:

- Initialize the peripheral being used into the correct mode.
- Drive a PWM signal from your hardware IP to the PmodHB3.
- Capture Hall sensor frequency (RPM) readings

Build the embedded system and top level module for the project

Build your embedded system using the IP Integrator and Nexys4 DDR board support. Generate a “wrapper” for your embedded system. Your embedded system should include all of the peripherals your application needs, including your IP block.

Don’t forget to configure the FIT timer and connect it to the system clock, peripheral reset and the interrupt controller. The FIT timer interrupt should be the highest priority interrupt in your application.

Synthesize, implement, generate bitstream and export your hardware.

Note: You may modify the wrapper to include any additional hardware your design. If that hardware generates ports at the top level you will have to create a constraints file that maps the ports to FPGA pins and include that in the project. If you only add devices through the Nexys4 board support no additional constraint files are needed…they are generated by the board package.

Integrate the hardware and software and tune your control system response

This application is more complex than the application in the first project. You may implement the user interface as described in this write-up or you can innovate and develop a user interface of your own design. The important thing is to implement the closed loop control algorithm in a way it can be quickly and effectively demonstrated. It would be to your advantage to do this development under the Standalone OS. Even though FreeRTOS is the target OS, targeting your initial design to it adds complexity to the integration task. If you have experience with FreeRTOS or another Real-time Kernel than feel free to go for it.

Test your closed loop control system

Once you have built the control circuit and your hardware system, you are ready to integrate the system and run your application. As part of your experimentation you should determine the values of $K_p/K_i/K_d$ that gets the motor speed up to the setpoint quickly and then stabilize on, or as near to, the set point as is possible.

Refactor your application from the Standalone OS to Free RTOS

The target OS for this project is FreeRTOS. The FreeRTOS “Hello World” example can be added to your SDK project when you create the FreeRTOS BSP. We will be releasing Example code of a tasking model (An LED blink application using GPIO interrupts) The example code uses two Tasks and one Queue, one interrupt generated GPIO switch, and one GPIO LED.

We have included a diagram of a possible threading model for your application. You do not have to use this model since your tasking model should be something that your existing standalone OS code “slides” into, but at a minimum, it’s food for thought.

Note: We have not implemented this threading model yet but from a block diagram perspective it seems reasonable. We may provide more FreeRTOS implementation details as the project proceeds.

Implement the Watchdog Timer Functionality

The system described above includes a Xilinx AXI Watchdog timer peripheral. This peripheral serves as both a time-base (a simple counter like the FIT timer) and a watchdog timer. The next task in the project is to include watchdog timer functionality in the application. The application and watchdog timer should operate with the following characteristics:

- The application should provide some kind of recovery action. This could be as simple as displaying a message and waiting in an infinite loop for a reset or a power cycle, or it could include provision to restart the application. Your application can use the `XWdtTb_IsWdtExpired()` API function to determine whether a reset was caused by a WDT timeout.

- The application should have the ability to force a WDT crash. This is most easily accomplished by preventing the calls to the `XWdtTb_RestartWdt()` from occurring when the “Force Crash” switch is on. You can use the leftmost slide switch (SW[15]) for the “Force Crash” switch.

- The application should check that it is, in fact, running. A simple way to do this is to periodically set a “system running” flag in the master thread. The WDT interrupt handler can read and clear this flag and restart the WDT when the handler is triggered on the first WDT overflow.

- It would be straightforward to provide better coverage by using individual flags for each of the threads. The flags could be set periodically by each of the tasks and then checked/cleared by the WDT functionality in the master thread. While this would work quite nicely for most of the threads it could prove to be problematic in the Control inputs thread, which is waiting for a change in the button or switch state (signaled by the semaphore). Changes may not occur often enough to avoid a WDT timeout.

- The WDT should force a CPU reset when it expires the second time (the first generates an interrupt – this is how the WDT operates). To do this the WDT_Reset output from AXI Watchdog Timer peripheral should be connected to the Aux_Reset_In input on the proc_sys_reset block.

Demonstrate your project and submit the deliverables

Once you have your project working be prepared to demonstrate it to the instructor or T/A. Please try to demonstrate this project on or before the deadline. Submit your deliverables to the D2L Dropbox in a single .zip or .rar file of the form <yournames>_proj2.zip. Only one submission per team is necessary. We will grade the submission with the latest timestamp if there are more than one.

Deliverables

- A demonstration of your project to the instructor or TA. You must bring your own control and motor system.
- A five to seven (5 to 7) page project report explaining the operation of your design, most notably your control algorithm and user interface. Please include at least two "interesting" graphs showing the results from the control algorithm. List the work done by each team member. Be sure to note any work that you borrowed from somebody else.
- Source code for your C application(s). Please take ownership of your application. We want to see your program structure and comments, not ours.
- All files regarding your motor control and speed measurement system (IP), including the Verilog hardware, and driver code.
- Your constraint and top level Verilog files if you have added any.
- A schematic for your embedded system. You can generate this from your block design by right-clicking in the diagram pane and selecting *Save as PDF File…*

Grading

You will be graded on the following:

- The functionality of your demo. Part of this will be how well your demo works, it must be functional. We will look at other "quality" of design issues such as unnecessary display flicker, slow response of the system to buttons and control changes. (60 pts)
- The quality of your design expressed in your C and Verilog source code. Please comment your code to help us understand how it works. The better we understand it, the better grade we can give you. (20 pts)
- The quality of your project report. (20 pts)

Extra Credit Opportunities

Project 2 offers several opportunities to earn extra credit points. Here are some suggestions but we are willing to be amazed and amused:

- Innovate on the user interface –you want to be able to easily "tinker" with the control loop
- Enhance the tachometer functionality to be able to detect direction of rotation in addition to the speed.

References

[1] *Digilent Nexys4™ DDR Board Reference Manual*. Copyright Digilent, Inc.
[2] *Digilent PmodHB3 H-Bridge Motor Control Reference Manual*. Copyright Digilent, Inc.
[3] *Digilent PmodENC™ Reference Manual*. Copyright Digilent, Inc.
[4] *Getting Started in ECE 544 (Vivado/Nexys4 DDR)* by Roy Kravitz, et. al.
[5] *Digilent PmodOLEDrgb* User Manual

Revision History

Rev 1.0	20-Feb-2017 DH	Major Revision to Project #2. This project does closed loop Proportional control to maintain the speed of a small electric motor. Many of the concepts (closed loop control, create/package custom IP, etc.) are borrowed from the previous project concept but the application is different.
Rev 1.1	28-Apr-2017 RK	Changed pulse width detect from high/low count to counting rising edges for a one second interval. This algorithm is better for using a tach pulse from a Hall Effect sensor to calculate RPM. Also changed KP factor selection to provide greater precision using fixed point. Organization, typographical and writing style changes.
Rev 2.0	08-Feb-2018 SB/RK	Major revision. Combined Project #2 and #3. Moved from Xilkernel to FreeRTOS. Eliminated step to develop custom peripheral (there are enough changes)