RC Car (KART)

Lecture Summerschool 1 (SS1)

Hes·so VALAIS WALLIS

School of Engineering

Orientation: Systems Engineering (Synd)
Specialisation: Infotronics (IT)
Course: Summerschool 1 (SS1)
Authors: Silvan Zahno, Axel Amand, François Corthay, Charles Praplan
Date: 14.03.2025
Version: v3.0.1

Contents

1 Security Guide 5
1.1 Consequences 6
2 Introduction 7
2.1 Objective 7
2.2 Evaluation 7
2.3 Files 7
2.4 Tools 8
2.5 Cabling 9
3 System Architecture 10
3.1 Block diagram 11
3.2 Functions 12
4 Hardware Components 13
4.1 Good Practices 14
4.2 Motherboard (MB) 14
4.2.1 Power 14
4.2.2 SODIMM Daughterboard Connector 15
4.2.3 I/Os 15
4.2.4 FPGA Reset 16
4.2.5 UART Sniffer 16
4.2.6 BLE Socket 16
4.3 Dautherboard (DB) 17
4.3.1 Power 17
4.3.2 Programming 17
4.3.3 Connection with Motherboard 17
4.3.4 I/O 18
4.4 Motors 19
4.4.1 DC-Motor 19
4.4.2 Stepper-Motor 19
4.4.3 PMOD DC-Stepper Control board 20
4.5 End of turn switch 21
4.6 Hall Sensor 21
4.7 Bluetooth Dongle NRF52840 22
4.8 Sensors & I/Os 22
4.8.1 Servo Motor 22
4.8.2 Custom modules 22
5 FPGA Design 23
5.1 Toplevel 24
5.1.1 Packages 24
5.1.2 Custom blocks 25
5.1.3 Testbenches 25
5.1.4 Embedded LEDs 25
5.2 Direct Current (DC) Motor Controller 26
5.2.1 Overview 26
5.2.2 PWM Generation 27

5.2.3 Hardware orientation 27
5.2.4 Bluetooth connection 27
5.2.5 Restart 28
5.2.6 Tests 28
5.3 Stepper Motor Controller 29
5.3.1 Overview 29
5.3.2 Driving coils 30
5.3.3 Initialisation of the Kart 31
5.3.4 Tasks Summary 32
5.3.5 Tests 32
5.4 Sensors Controller 33
5.4.1 Overview 33
5.4.2 Hardware setup 34
5.4.3 Hall counter 35
5.4.4 Ultrasound Ranger *(Optional)* 36
5.4.5 Servomotors controller *(Optional)* 37
5.4.6 User functionalities *(Optional)* 38
5.5 Optional features 39
5.5.1 DCMotor - Acceleration ramp 39
5.5.2 StepperMotor - Dynamic steering frequency 39
5.5.3 Sensors 39
5.5.4 Other 40
6 Testing 41
6.1 Per module 42
6.1.1 DC Motor testing 42
6.1.2 Stepper Motor testing 43
6.1.3 Sensors Controller testing 45
6.2 Whole circuit 48
6.2.1 Modules Simulation 48
6.2.2 Full-board 50
6.3 Setting up the board 51
6.3.1 I/Os configuration 51
6.3.2 Pining setup 51
6.3.3 Onboard LEDs 52
6.4 Programming the board 52
6.5 USB commands emulation 53
6.5.1 Quick Test 54
6.5.2 Registers R/W 54
7 Communication 55
7.1 General Principle 56
7.1.1 Serial Port Configuration 56
7.1.2 Message format 56
7.2 Registers 57
7.3 Initialisation Sequence 59
Appendices 60
A Tools 61
I HDL Designer 61

II Modelsim 62
III Microchip Libero 63
B PMod boards 69
I Inputs 69
II Outputs 70
C Inspiration 73
Glossary 79

1 Security Guide

In the pursuit of creating a small remote-controlled car, it is imperative to consider the security-aspect of the project. Ensuring the integrity of the hardware, preventing damage to components and personal safety are vital. This chapter outlines the security measures and protocols that must be adhered to throughout the course of the project. These measures have been put in place to protect both the project's hardware and the participants involved.

Some security rules are also mentioned again in the followoing chapters, where they are relevant.

Think before doing
Certainly, thinking before taking action is a fundamental principle of effective project management and security.

No Hardware to Leave the Premises
To safeguard the project's hardware components, it is strictly forbidden for any team member to take project-related hardware home.

All hardware must be properly stored in the designated laboratory cabinets.

Behaviour in the Labors
To safeguard the project's hardware components, it is strictly forbidden to have any food and drinks close to the project-related hardware.

In addition protection for other equipments such as tables, instruments must be used.

The Labors needs to be kept clean and tidy.

Mechanical Precautions
The motors used in the remote-controlled car can pose a physical risk, and special attention should be given to prevent accidents and injuries. Team members must exercise caution when handling the motors, ensuring that they are properly secured, and that all moving parts are well-guarded to prevent accidental contact.

Personal Protective Equipment (PPE)
When working with mechanical tools and machines, it's imperative to prioritize the safety of all team members. PPE, such as safety goggles, gloves, ear protection, and dust masks, should be worn as appropriate when operating machinery. Always remember, safety first.

Mechanic and Electronic

Use either:

- Nylon screws and risers to secure the boards to your Kart.
- Plastic washer between the risers and/or screws and any electronic board.

Power Disconnection Protocol

Completely disconnect any power source before making changes or modifications on the hardware.

Battery Precautions

Before connecting batteries to the remote-controlled car, comprehensive functional testing is mandatory. The following precautions must be taken:

- Prior to connecting the batteries, all functionalities of the car must be tested using a laboratory power supply limited to $0.15A$, excluding the motors. When attaching the motors and other custom equipment, the power supply limit must be increased to $1.2A$. This precaution prevents sudden surges of current from damaging the circuitry during the initial testing phase.
- **Batteries are recharged by the electronics laboratory (23N219) ONLY.**

Secure SODIMM Connector

The FPGA board is a critical component of the remote-controlled car project. To ensure its proper functioning, the SODIMM connector must be inspected at each test to verify that the FPGA board is securely connected. Loose connections can result in system malfunctions, data corruption, and potential damage to the FPGA.

Help is available

In case of any questions or concerns regarding the security of the project, the team members are encouraged to contact assistants and professors.

1.1 Consequences

It is crucial to emphasize that any deviation from the security measures outlined in this chapter will result in consequences. It will involve the **deduction of points from the final project grade**. This penalty serves as a reminder of the importance of adhering to the security protocols and maintaining the integrity of the project.

In conclusion, security is a paramount consideration in the development of the remote-controlled car project. By following these protocols and cooperating with professors, the team can create a safe and secure environment for project development while minimizing the risk of damage.

2 Introduction

The Kart module (SS1) is a Summer School module for students between the 2nd and the 3rd semester. It is a home-made model car remotely controlled by a smartphone. The Appendix C - Inspiration gives an overview of previous years karts.

The work of the students can be summarized in four main tasks:
- design and assembly of the chassis and the body
- analysis of the DC (DC) motor [1]
- configuration of the controlling **F**ield **P**rogrammable **G**ate **A**rray (FPGA) (FPGA) [2]
- completion and extension of the control Graphical User Interface (GUI) (GUI) on the smartphone

> This document only covers the configuration and programming of the control electronics.

2.1 Objective

For the control electronics part there are three mandatory objectives:
- Control block for the DC Motor, see Section 5.2 - **P**ulse **W**idth **M**odulation (PWM) Modulator
- Control block for the stepper motor, see Section 5.3 - 4 coils sequence generator
- Hall-Sensor counter, see Section 5.4

2.2 Evaluation

Fullfilling all mandatory objectives mentioned in Section 2.1 will result in a grade of 4.0. The students are free to implement additional features. For every added feature, the grade will be increased. Depending on the complexity of the feature between 0.1 and 1.0. until the maximum grade of 6.0 is reached.

Optional features are described in Optional Features - Section 5.5.

2.3 Files

All necessary files can be collected via two methods:
1. Either downloaded via a zip file at the following link https://github.com/hei-synd-ss1/ss1-vhdl/archive/refs/heads/main.zip [3] directly from github and extract it into your preferred location.
2. Alternatively a group specific repository can be created via github classroom by using the following invitation link: https://classroom.github.com/a/fI0MXIU3. You repository can then be cloned with `git`.

```
git clone https://github.com/hei-synd-ss1-stud/2025-ss1-vhdl-<groupname>.git
```

> Make sure there is no space characters in the full projects path. HDL may hang while booting or files may not be loaded/saved correctly.

2.4 Tools

The design environment for the control electronic consists of several tools.

- Mentor HDL designer for graphical design entry [4]

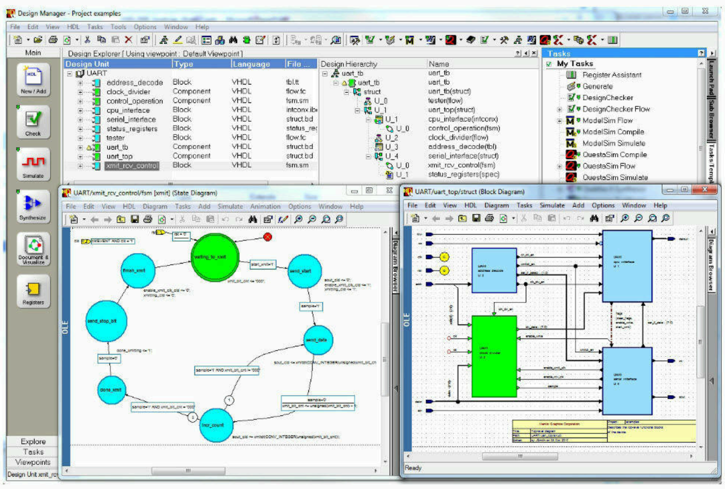


Figure 1 - Mentor HDL Designer

- Mentor ModelSim for simulation [5]

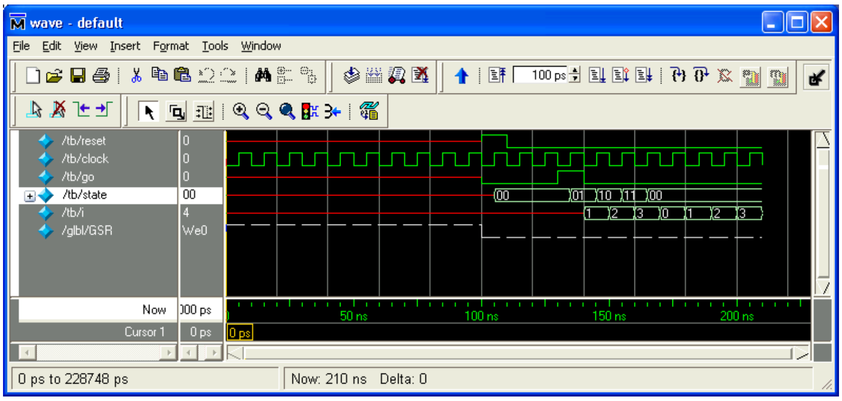


Figure 2 - Mentor Modelsim

- Microchip Libero IDE for synthesis and programming[6]

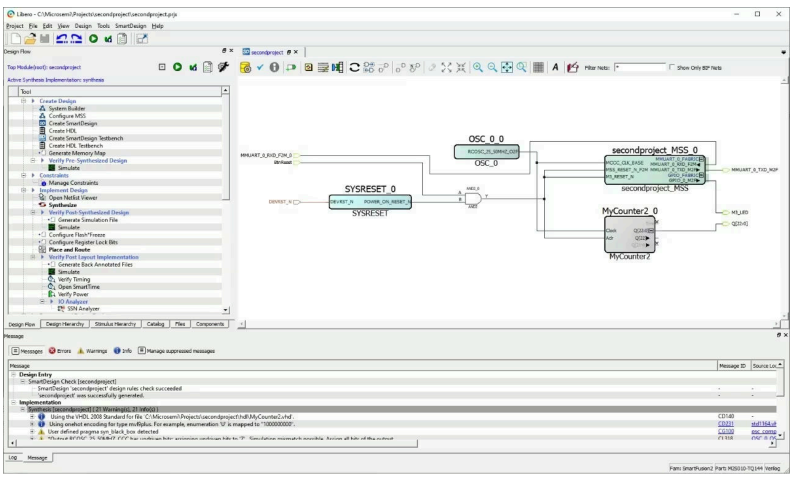
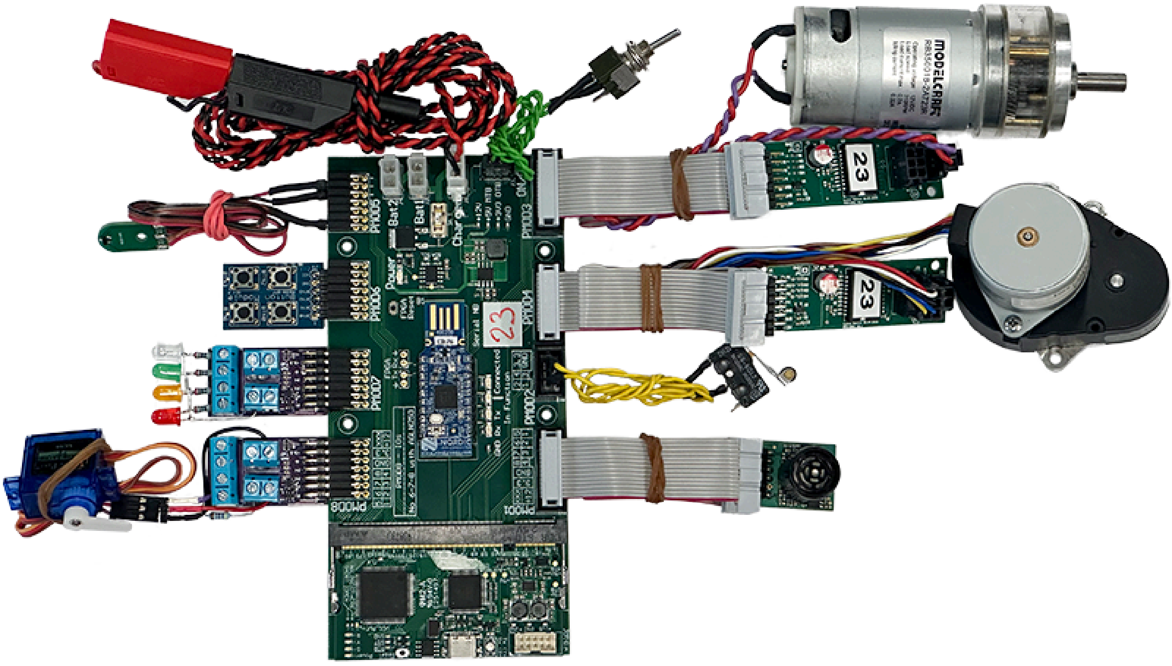


Figure 3 - Microchip Libero

2.5 Cabling

The default solution uses the following connection scheme:



Module	PMOD	Pins
Range Sensor	PM1	8
Stepper End	PM2	1-3-GND
DC Motor	PM3	Through PMOD-MotorDriver board
Stepper Motor	PM4	Through PMOD-MotorDriver board
Hall Sensor	PM5	1
Buttons inputs	PM6	5-6-7-8
Leds outputs	PM7	Through PMOD-OD2 board
Servomotors outputs	PM8	Through PMOD-OD2 board

Table 1 - Default hardware Cabling (**`Kart.pdc`**)

It is possible to modify:
- Where each module is placed
- How many inputs and outputs exist
- If custom pins are used

to connect the correct signals from the Hardware the toe FPGA you will have to modify the **`Board/concat/Kart.pdc`** file during the Synthesis - Appendix III process.

3 System Architecture

The architecture is essentially split between two parts: the embedded electronic which drives the kart and reads the various sensors, communicating with a smartphone to be controlled remotely.

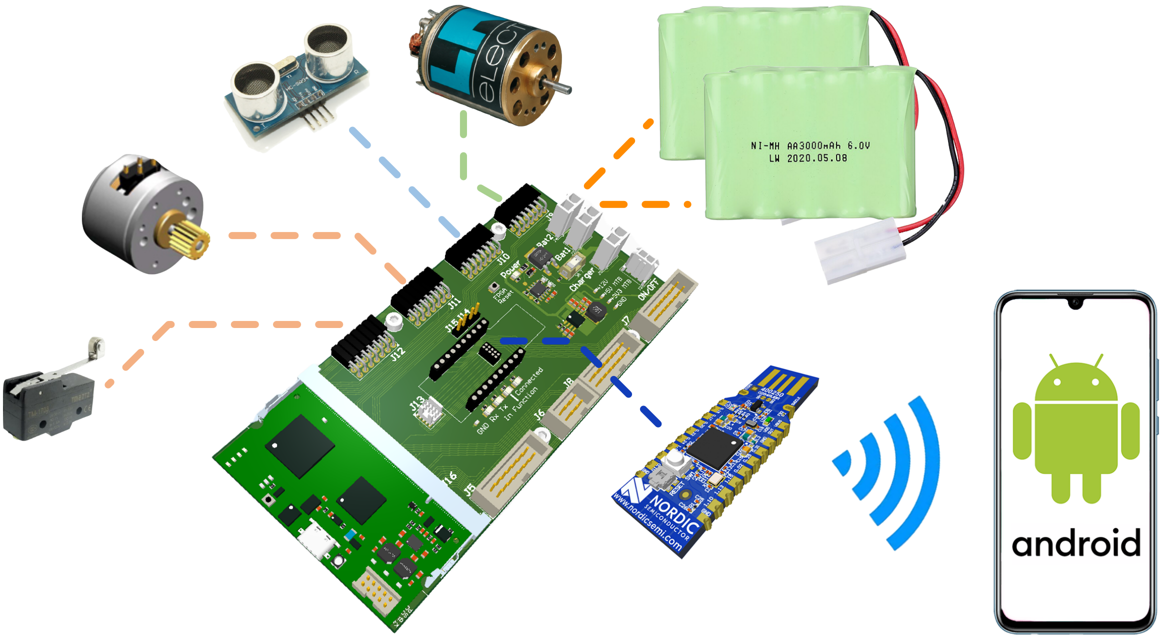


Figure 4 - System Architecture Overview
(The shown config is custom and does not correspond to Table 1)

Contents

3.1 Block diagram 11
3.2 Functions 12

3.1 Block diagram

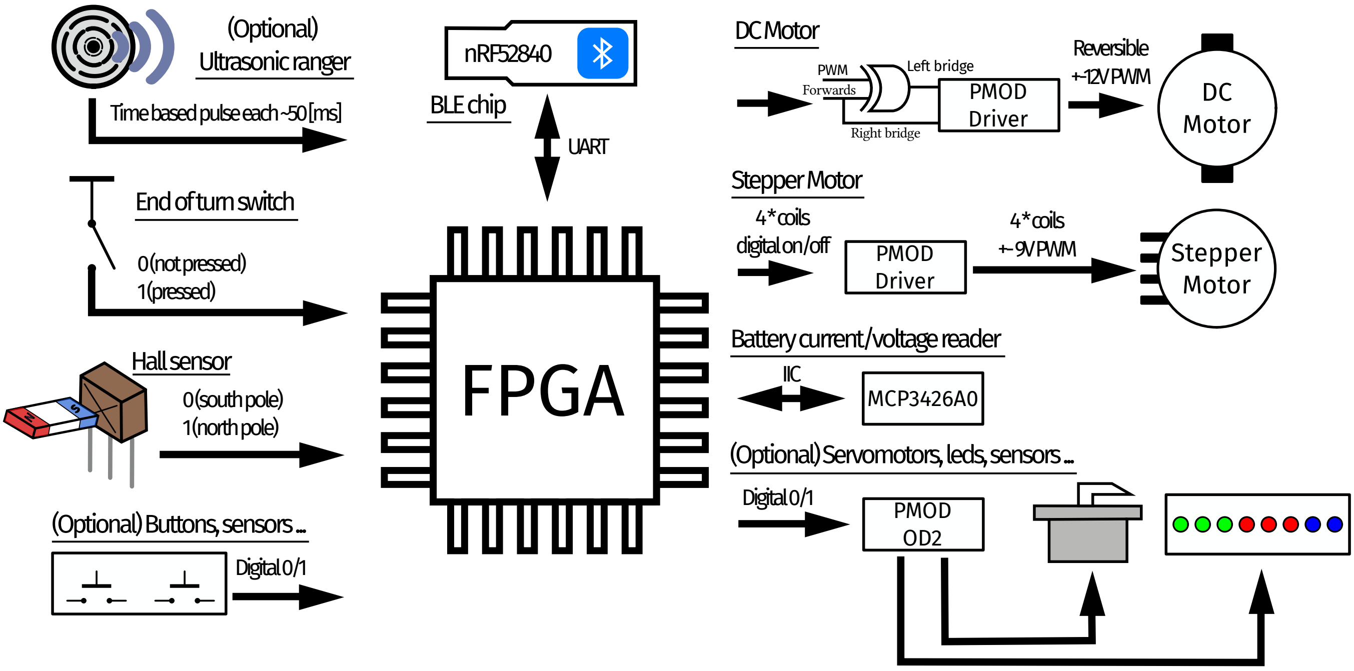


Figure 5 - Electronic Architecture

The system is centered around the FPGA which gets the following inputs:

- **End of turn switch**: used to initialise the angle of the wheels
- **Hall sensor**: combined with magnets, allows to calculate the speed of the car
- **Ultrasonic ranger (optional)**: permits to detect distances and brake in case of an obstacle
- **Buttons, sensors … (optional)**: any extra hardware which creates 0-3.3V digital inputs can be wired to the FPGA and used internally for extra features

The followings are available on the board itself:

- **BLE chip**: the FPGA communicates through UART with a nRF52849 BLE chip to link with the smartphone
- **Battery reader**: a MCP3426A0 chip allows to read through I2C both the current and voltage from the system
- **User leds (optional)**: 3 user leds can be freely controlled for debug purposes

Finally, outputs are:

- **DC motor**: the system is propulsed by a 12V brushed DC motor
- **Stepper motor**: the steering of the car is carried out by a 4-coils stepper motor
- **Servomotors, leds, sensors … (optional)**: any digital output can be wired to the FPGA through the PMOD-OD2 board. This board can translate outputs into higher voltages and currents. The selectable output voltages of this board are +3.3, +5 or +12 V

3.2 Functions

The minimal system allows to communicate with the smartphone as well as operate all required sensors and actuators. The system needs to:

- Propel the kart forward and backward with the help of the DC motor - Section 4.4.1 - and motor driver PMOD - Section 4.4.3
- Steer the kart with the help of the stepper motor - Section 4.4.2, a motor driver PMOD board - Section 4.4.3 - and the end of turn switch - Section 4.5
- Count the hall sensor pulses - Section 4.6 - to measure the speed
- Set registers correctly to communicate through the UART serial link with a custom communication protocol - Section 7 - with the smartphone over bluetooth - Section 4.7.

4 Hardware Components

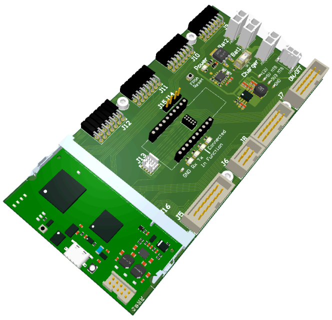


Figure 6 - Kart PCB

Mechanic and Electronic

Use either:

- Nylon screws and risers to secure the boards to your Kart.
- Plastic washer between the risers and/or screws and any electronic board.

Contents

4.1 Good Practices 14
4.2 Motherboard (MB) 14
4.2.1 Power 14
4.2.2 SODIMM Daughterboard Connector 15
4.2.3 I/Os 15
4.2.4 FPGA Reset 16
4.2.5 UART Sniffer 16
4.2.6 BLE Socket 16
4.3 Dautherboard (DB) 17
4.3.1 Power 17
4.3.2 Programming 17
4.3.3 Connection with Motherboard 17
4.3.4 I/O 18
4.4 Motors 19
4.4.1 DC-Motor 19
4.4.2 Stepper-Motor 19
4.4.3 PMOD DC-Stepper Control board 20
4.5 End of turn switch 21
4.6 Hall Sensor 21
4.7 Bluetooth Dongle NRF52840 22
4.8 Sensors & I/Os 22
4.8.1 Servo Motor 22
4.8.2 Custom modules 22

4.1 Good Practices

In order not to damage the hardware, strictly follow the Section 1 - Security Guidelines.

4.2 Motherboard (MB)

The Kart motherboard can receive any compatible FPGA daughterboard - Section 4.3 such as the AGLN250 [7] one used during the Kart project. The Motherboard connects all pheripherals to the I/Os and powers the system at the same time [2].

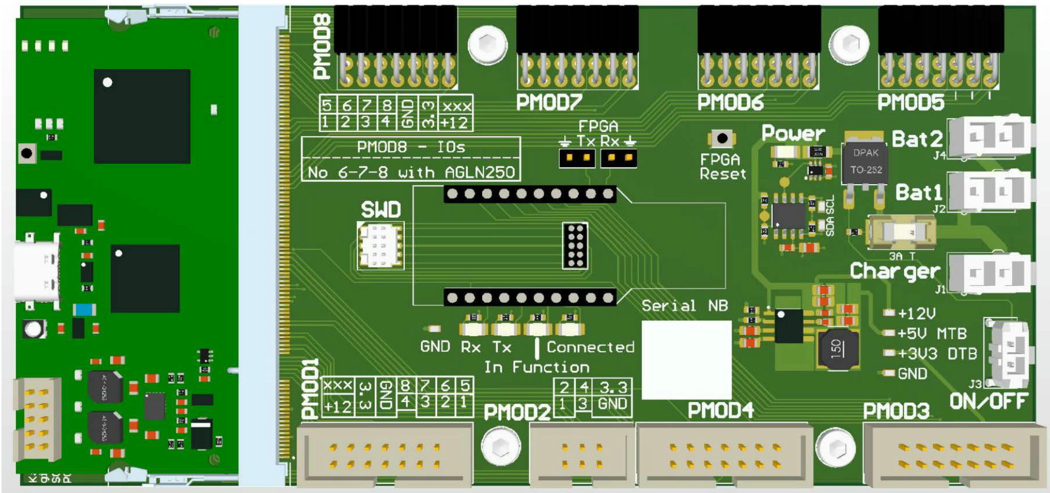


Figure 7 - Motherboard PCB

4.2.1 Power

The main power entry point of the system is through the motherboard, either by using:

1. the two battery connectors with two +6V / 2400mAh packs put in series
2. the charger port with a +12V input from a regulated DC supply.

The +12V is then reduced to a +5V rail through a buck converter. Finally, the daughterboard is fed with the +5V to provide a +3.3V rail.

4.2.1.1 Charging

> Do not try to manually charge batteries while mounted on the motherboard. Charging batteries is handled by the electronic lab directly. Simply ask and hand them your packs.

4.2.1.2 Power-on

A switch must be connected to the corresponding port to power the board. The +12V is then transported to the PMODs and the buck converter through a 1.25A-T fuse. A green LED shows the board power status.

> Follow the tests guideline given under Section 6 before powering anything.
> If the fuse breaks, check for any short-circuit before replacing it and power-cycling the circuit.

4.2.1.3 Power State

An I2C dual-inputs ADC converter (MCP3426A0) [8] is present on the board to read both the battery voltage and the current consumption.

Access the data

The chip is read from the FPGA each second through dedicated I2C lines. The information can be read from the smartphone at will by accessing the corresponding registers - Section 7.2.

4.2.2 SODIMM Daughterboard Connector

The daughterboard - Section 4.3 is connected to the motherboard through a SODIMM-200 (DDR2 RAM) connector.

4.2.3 I/Os

The board allows for multiple I/Os to be plugged following the PMOD wiring [9], slightly modified to add a +12V rail, under the following form:
- 4 dual connectors for direct plug (PMODs 5 to 8)
- 3 dual connectors for flat-cables (PMODs 1, 3, 4)
- 1 single connector for flat-cable (PMOD 2)

> PMOD8 signals P6 to P8 cannot be used with the AGLN250 FPGA. Use only the upper row (PM8_1 to PM8_4).

The pins are described on the board itself and correspond to the following:

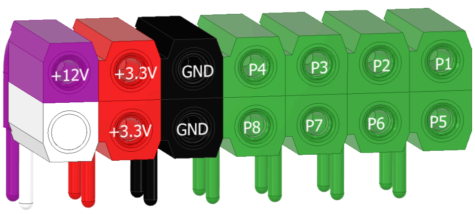


Figure 8 - PMOD pinning (header)

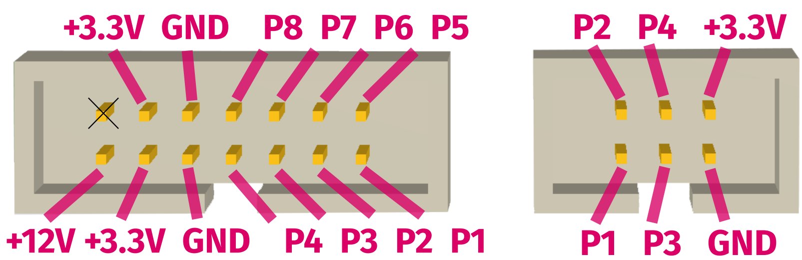


Figure 9 - PMOD pinning (flat)

> At all time, ensure there is **NO VOLTAGE FEEDBACK** from anything else than 3.3V on the I/Os. Use a dedicated PMOD board if needed. They are presented in the Appendix B.

4.2.4 FPGA Reset

A small button allows the user to reset the FPGA from the motherboard.

4.2.5 UART Sniffer

Both Tx from the FPGA and the BLE module can be sniffed by wiring a dedicated UART-USB chip to the provided headers or directly with an oscilloscope.

4.2.6 BLE Socket

The Bluetooth ⟺ USB dongle - Section 4.7 [10] can be inserted in its dedicated socket to control the Kart with a smartphone, or easily removed to be plugged in a PC directly. One can emulate the BLE module with the help of a custom serial interpreter - Section 6.5 by simply plugging the daughterboard in a PC through the USB-C. The communication is merged between both the PC and the BLE module.

> ⚠ Trying to communicate simultaneously from the BLE module and the PC will result in undefined behavior (surely scrambled and wrong data read by the FPGA).

One can listen to what the FPGA communicates to the BLE module by opening a serial terminal on the USB-C COM port, but not write on it simultaneously.

To listen to what the BLE module communicates, it must be plugged through a USB extension cable to the PC and another serial terminal opened.

4.3 Dautherboard (DB)

The FPGA daughterboard embeds an Igloo **`AGLN250`** chip [7] in a **`VQ100`** package, driven by a **`10MHz`** clock.

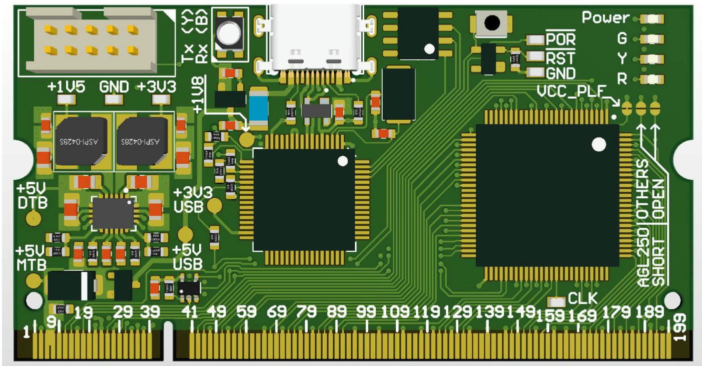


Figure 10 - Daughterboard PCB

4.3.1 Power

The board is powered either through a USB-C connector (+5V, providing a JTAG access along an USB-UART converter [11]) or through the motherboard - Section 4.2 via an external +5V rail.
It will automatically resolve the path if both supplies are wired simultaneously, choosing the motherboard rail in priority.
Internally, it creates a +3.3V rail used by both the FPGA and the motherboard, plus another +1.5V rail for the FPGA core supply.

4.3.2 Programming

The board can be programmed by using Libero IDE - Appendix III and plugging a Microsemi FlashPro 4 dongle on the dedicated 10 pins header.
It is also possible to use the USB-C connector with the help of OpenOCD and custom scripts.
Both the USB and the FlashPro can be plugged at the same time. The FlashPro gains priority over the USB JTAG signals.

> ⚠ While using the FlashPro without the motherboard powered, it is necessary to plug both the USB-C and the FlashPro to be able to program the card.

4.3.3 Connection with Motherboard

The board is linked to the motherboard - Section 4.2 through an SODIMM-200 connector (the 200 gold fingers on the FPGA board) [2]. By design, it cannot be inserted the wrong way.
The connector pining is shown in the Kart Pinning Datasheet [12].

4.3.4 I/O

Reset

A button is found on the board to reset the FPGA.

> Be careful when pushing it while it is inserted in a motherboard. Do not apply force on the SODIMM connector.

LEDs

A blue LED indicates that the board is powered (top right of the board), while a second found near the USB connector shows in and out transaction over UART.
The red led indicates if the stepper end switch is pressed.
The yellow led toggles on and off when a magnet is rotated in front of the hall sensor 1.
The green led indicates if the smartphone is connected to the BLE module:

- It blinks when connected in the **solution** version
- It stays on when connected in the **student** version

PoR

The board also features a Power on Reset (PoR) circuitry that will detect low FPGA voltages and reset it (discharged batteries, too high current consumption ...).

4.4 Motors

4.4.1 DC-Motor

The DC Motor is used to propel the kart forward. It is a brushed DC motor, running on +12V and drawing a current of $I_{\text{max}} = 0.7A$ and $I_{\text{idle}} = 0.32A$ [1]. The PMOD DC-Stepper Motor Driver allows to control the motor via a PWM Signal [13] through a H-Bridge [14].

Color	Pin
Red	+
Violet	-

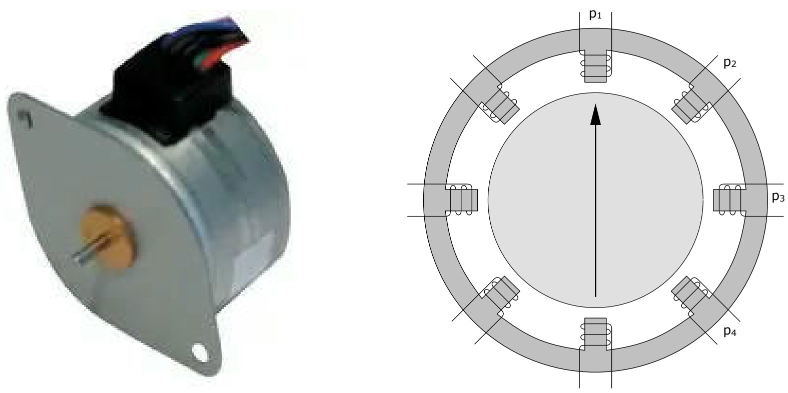
Figure 11 - Modelcraft RB350018-2A723R DC Motor and its pining

4.4.2 Stepper-Motor

The Stepper Motor is used for steering the kart. It is a bipolar stepper motor with a step angle of 7.5° - 48 steps per rotation and a nominal current of $I = 0.86A@5V$ with a $R = 5.8\Omega$ [15].
It is attached to a 100:1 reductor gear which leads to an output axis with a step angle 0.075° - 4800 steps per rotation.

The motor is controlled with the PMOD DC-Stepper Motor Driver - Section 4.4.3 hosting a dual full H-Bridge to control the 4 coils of the stepper motor.

The calibration can be performed using the End-of-Turn switch - Section 4.5.



Color	Pin
White	Coil 1
Blue	Coil 2
Black	GND
Brown	GND
Red	Coil 3
Yellow	Coil 4

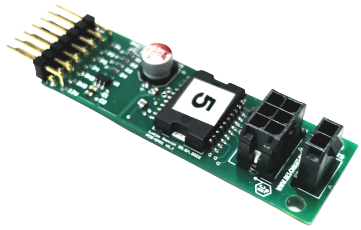
Figure 12 - Nanotec SP3575M0906-A Steppermotor, coils and pining

4.4.3 PMOD DC-Stepper Control board

The control board hosts a dual full H-Bridge [16].

The 6-pins connector is used to connect a stepper motor and the 2-pins one the DC Motor.

> ⚠ Only one motor can be connected at any given time.



Row	Pin	Descr.	Row	Pin	Descr.
	2	P1: NC		1	P5: Left Bridge A
	4	P2: NC		3	P6: Right Bridge A
	6	P3: NC		5	P7: Left Bridge B
Top	8	P4: NC	Bottom	7	P8: Right Bridge B
	10	GND		9	GND
	12	3.3V		11	3.3V
	14	12V		13	NC

Figure 13 - PMOD DC-Stepper Control board and pining

The Figure 14 shows the block diagram of the main component, the L298P:

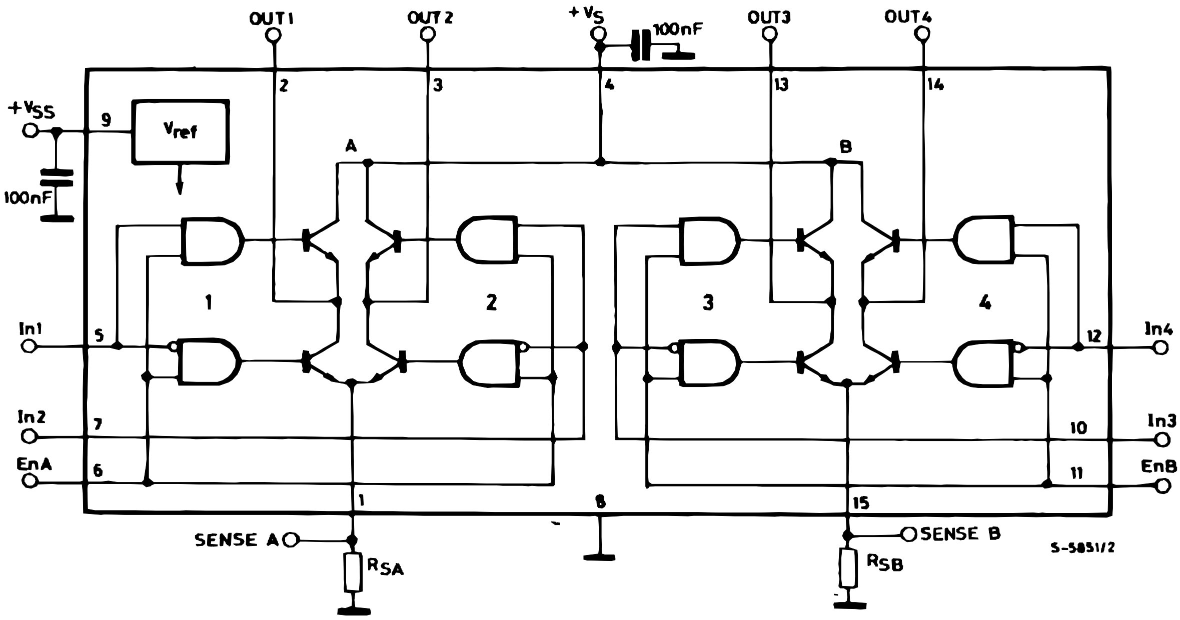
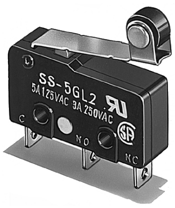
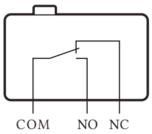


Figure 14 - PMOD DC-Stepper Control board

4.5 End of turn switch

The end of turn switch is used to identify the steering “zero” position. The used switch is a Omron miniature high reliability and security switch [17].





Color	Pin
Yellow	PMOD I/O
Yellow	GND

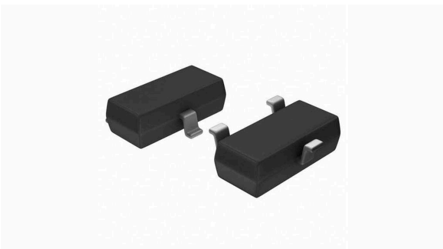
Figure 15 - Omron SS-5T Miniature High Reliability and Security Switch and its pining

An internal pull-up must be enabled on the FPGA side.

4.6 Hall Sensor

One or two Hall sensors are used to track the distance driven by the kart. The SS311PT/SS411P digital Hall-effect sensors [18] are operated by a magnetic field and designed to respond to ***alternating*** North and South poles with their Schmitt-trigger [19] output.

They can be powered between `2.7Vdc` to `7Vdc` with an open collector output integrating a 10kohm pull-up resistor already.



Color	Pin
Red	3.3 V power supply
Brown	Hall output
Black	GND

Figure 16 - Hall Sensor Honeywell SS311PT and its wiring

No internal pull resistor should be enabled on the FPGA side.

4.7 Bluetooth Dongle NRF52840

The nRF52840 Dongle is a small, low-cost USB dongle that supports Bluetooth 5.4, Bluetooth mesh, Thread, Zigbee, 802.15.4, ANT and 2.4 GHz proprietary protocols [10], [20]. In this project it is used to communicate with the smartphone. The output of the dongle is an UART serial link that is connected to the FPGA. The communication protocol is defined in Section 7.

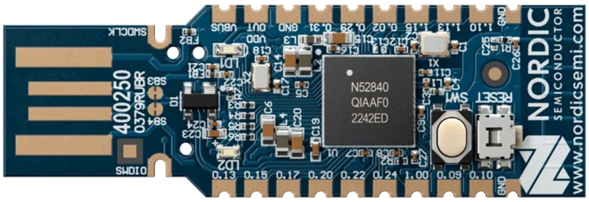


Figure 17 - NRF52840 Bluetooth Dongle

4.8 Sensors & I/Os

Various sensors can be mounted on the motherboard - Section 4.2 through the exposed PMOD - Appendix B connectors.
Only +3.3V I/Os can be connected on PMOD connectors (either by directly plugging them into the pin headers or through the IDC cables).
Pins 6, 7 and 8 of the PMOD8 CANNOT be used with the current FPGA.

4.8.1 Servo Motor

Servos are a great choice for robotics projects, automation, RC models and so on. They can be used to drive custom mechanical parts of your RC-Car. The angle is controlled with the control pin. The pulse width of a $f = 50\text{Hz}$ signal defines the angle, see Figure 18.

Color	Pin
Yellow	Control
Red	5V
Black	GND

Figure 18 - Servo Motor and pining

Avilable ones are the Reely S-7361. Use boards like the PMOD-OD2 - Appendix iii to control such devices.

4.8.2 Custom modules

Feeling adventurous?

You can wire other/custom boards to interface your Kart with the world. Feel free to propose your ideas and discuss the fesability with a professor.

5 FPGA Design

At least the three different modules must be completed:

- The DC motor controller (Section 5.2) receives a **`prescaler`** and a **`speed value`** to build the corresponding PWM and **`direction`** signals.
- The stepper motor controller (Section 5.3) receives a **`prescaler`** and the desired **`angle`** and builds the **`coil`** controls signals.
- The sensor controller (Section 5.4) manages I/O comprising the **`hall sensors`** to retrieve the driving speed and the **`range finder`** to get the distance from an obstacle (optional).

Contents

5.1 Toplevel ... 24
5.1.1 Packages ... 24
5.1.2 Custom blocks ... 25
5.1.3 Testbenches ... 25
5.1.4 Embedded LEDs ... 25
5.2 DC Motor Controller ... 26
5.2.1 Overview ... 26
5.2.2 PWM Generation ... 27
5.2.3 Hardware orientation ... 27
5.2.4 Bluetooth connection ... 27
5.2.5 Restart ... 28
5.2.6 Tests ... 28
5.3 Stepper Motor Controller ... 29
5.3.1 Overview ... 29
5.3.2 Driving coils ... 30
5.3.3 Initialisation of the Kart ... 31
5.3.4 Tasks Summary ... 32
5.3.5 Tests ... 32
5.4 Sensors Controller ... 33
5.4.1 Overview ... 33
5.4.2 Hardware setup ... 34
5.4.3 Hall counter ... 35
5.4.4 Ultrasound Ranger *(Optional)* ... 36
5.4.5 Servomotors controller *(Optional)* ... 37
5.4.6 User functionalities *(Optional)* ... 38
5.5 Optional features ... 39
5.5.1 DCMotor - Acceleration ramp ... 39
5.5.2 StepperMotor - Dynamic steering frequency ... 39
5.5.3 Sensors ... 39
5.5.4 Other ... 40

5.1 Toplevel

The toplevel contains all elements. Except for special, custom functionalities, there is no need to change anything. The left part is dedicated to the serial communication and the right one comprises:

- Stepper-Motor Controller
- DC-Motor Controller
- Sensor Controller
- Control Register Controller

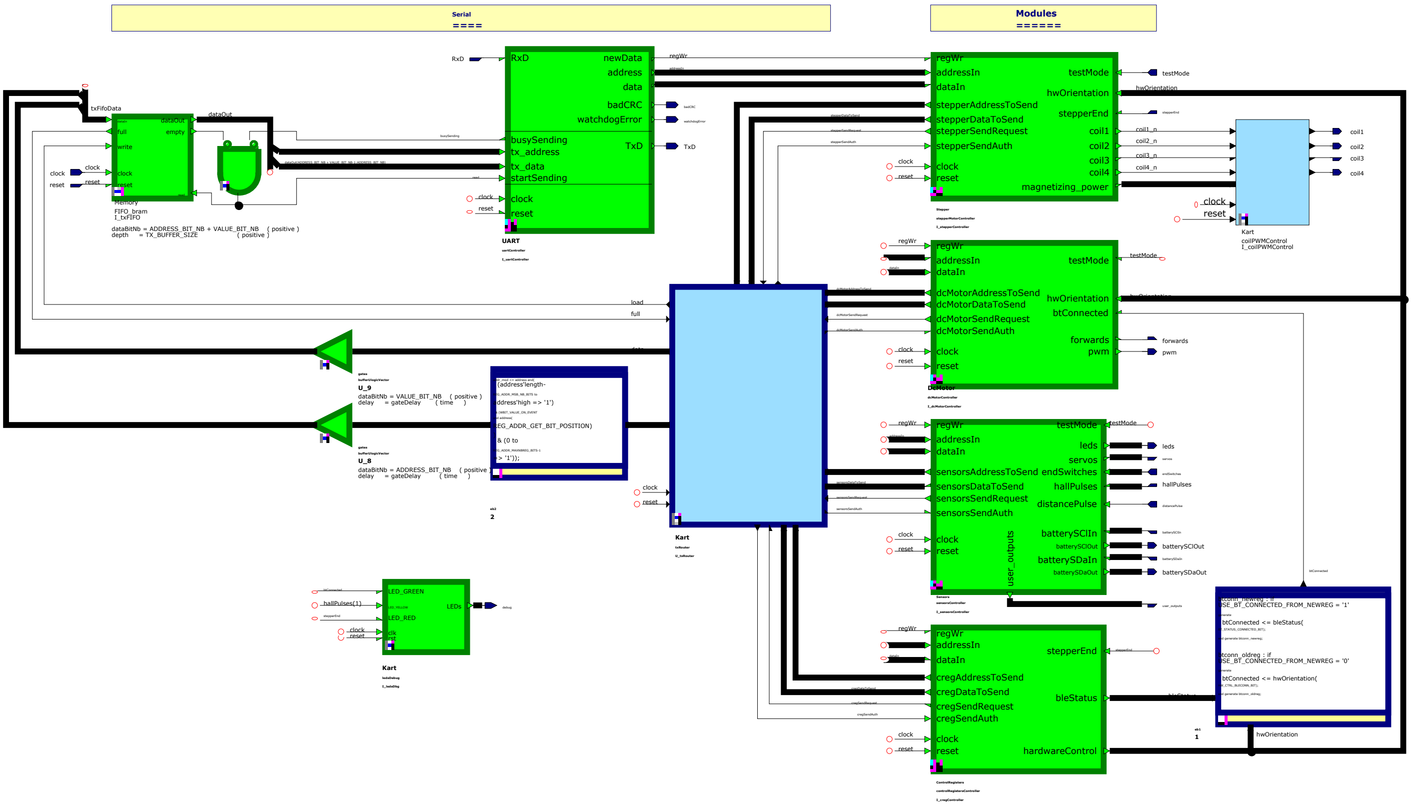


Figure 19 - RC-Car top-level

5.1.1 Packages

The system being more complex than those seen during laboratories, a lot of the constant, types, sizes ... are stored in **packages**.
Those can be found in the **Kart** library, named **Kart.pkg** and **Kart_Student.pkg**.

The **Kart_Student** package is the only one that may be modified to adapt the system to your needs.

Required modifications are explained under the impacted sections.

5.1.2 Custom blocks

In the various parts presented below, blocks will have to be created as seen during laboratories. There is one important rule:

> ⚠ All blocks must have a unique name across ALL project libraries.

This is important as the synthesis tool will not be able to differentiate between two blocks with the same name, leading to errors.

Also, think ahead when designing blocks: well-formed and fully functional blocks can be reused in other parts of the project, which is a great way to save time.

Finally, it is possible to create different blocks content to test different functionalities without losing your previous work. Right click the block ⇨ Open As ⇨ New View … ⇨ Graphical View ⇨ Block Diagram. Then click on the block again ⇨ Change Default View ⇨ select your view.

Only one view can be active in the whole project at the same time. Different functionalities require different blocks.

5.1.3 Testbenches

Some testbenches are made available in the corresponding ***_test** libraries.

Some are already filled with tests, while others are partially empty where only the **clock** and **reset** signals are generated. It is up to you to fill them with the necessary tests to validate your blocks.

You are free to create as many extra testbenches you need to debug your system.

5.1.4 Embedded LEDs

The user can control three LEDs present on the Daughterboard - Section 4.3 from within the FPGA.

By default, those LEDs correspond to the following:
- A blue LED indicates that the board is powered (top right of the board), while a second found near the USB connector shows in and out transaction over UART.
- The red led indicates if the stepper end switch is pressed.
- The yellow led toggles on and off when a magnet is rotated in front of the hall sensor 1.
- The green led indicates if the smartphone is connected to the BLE module:
  - It blinks when connected in the **solution** version
  - It stays on when connected in the **student** version

You can change the signals connected to the **`ledsDebug`** block in **`Kart/KartController`** to wire your own and help debug your system.

5.2 DC Motor Controller

The DC motor controller is in charge of the Kart's propulsion, both in forward and reverse.

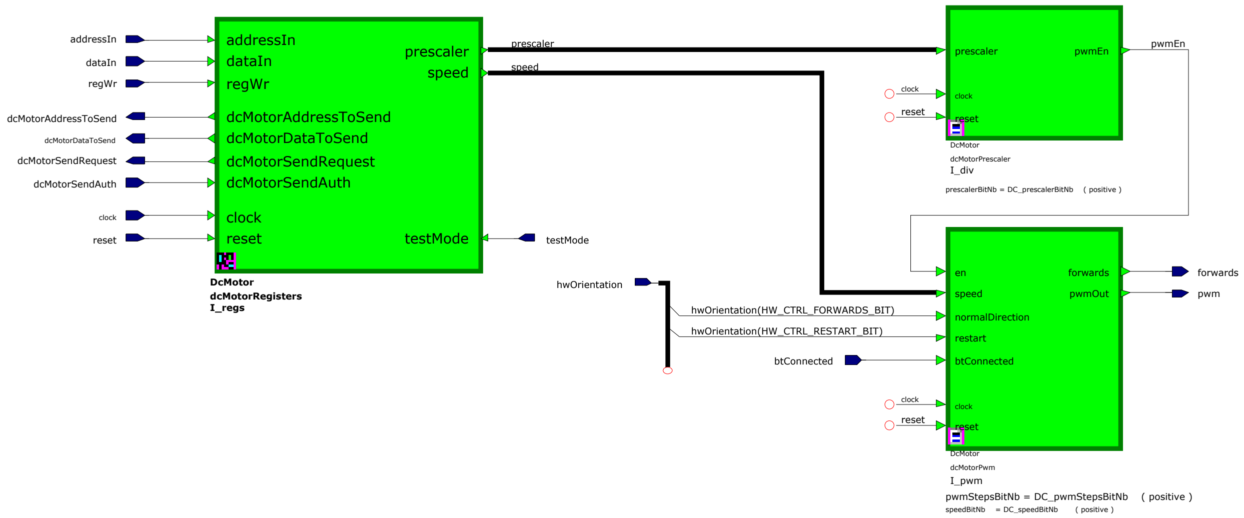


Figure 20 - DC module top-level

For this, two signals are generated :

- A **pwm** with a settable frequency whose duty-cycle is modified to control the speed
- A **forwards** signal to drive either forward or backward.

5.2.1 Overview

Since a PWM is used to drive the motor and the power transistors cannot switch at too high frequencies the period must be controlled. For this, the block **dcMotorPrescaler** generates an **en** signal to divide the clock frequency based on the DC motor prescaler register - Section 7.2 following the formula:

$$f_{\mathrm{PWM\_DC}} = \frac{f_{\mathrm{clk}}}{\mathrm{PWM}_{\mathrm{steps}} * \mathrm{prescaler}} = \frac{10\mathrm{MHz}}{16 * \mathrm{prescaler}} \quad (1)$$

The minimal value of the PWM signal is studied in another part of the project.

The block must then act on the duty-cycle of the generated PWM according to the **speed** signal. It is set in the DC motor speed register - Section 7.2 and ranges from:

$$-15 = 0\mathrm{b}1111'1111'1111'0001 \text{ to } 15 = 0\mathrm{b}0000'0000'0000'1111 \quad (2)$$

The two output signals are then used to drive a H-Bridge [14], powering the DC motor.

5.2.2 PWM Generation

Create a structure which is able to:
- Extract the absolute value of the **`speed[MSB-1 : 0]`** signal
- Generate the PWM on 16 steps, cadenced by the **`en`** signal
- Set the **`forwards`** signal to **`'1'`** if the Kart should drive forward, **`'0'`** otherwise

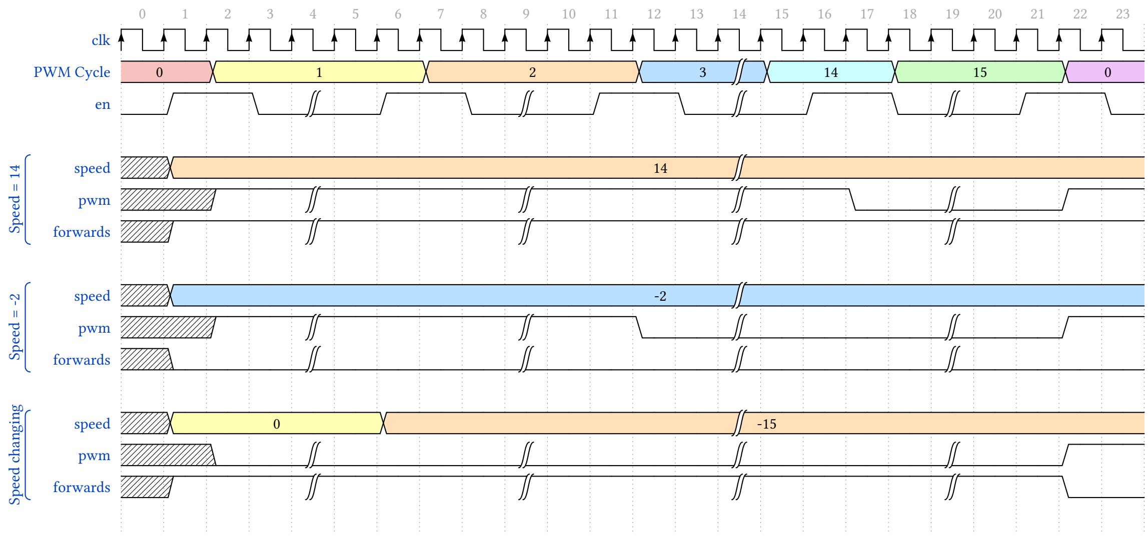


Figure 21 - DC Motor timings diagram

> Draw the circuit of the **`dcMotorPwm`** block.

5.2.3 Hardware orientation

The mechanical design can either lead the Kart to drive forward or backward when a positive voltage is applied to the DC motor.

In order to cope with this, a setup signal, **`normalDirection`**, is provided to the block. **`normalDirection = '1'`** means that a positive voltage applied to the DC motor lets the kart drive forwards.

> Update the circuit in order to cope for the different mechanical design possibilities.

The setup bit is configured in the hardware control register - Section 7.2.

5.2.4 Bluetooth connection

When the Bluetooth connection is lost, the DC motor should not turn to prevent any damage.

A control signal, **`btConnected`**, is provided to the block. When **`btConnected = '0'`**, the DC motor must stop.

If the **`btConnected`** signal rises back to **`'1'`**, the motor must not move until the **`speed`** register is modified. Else the Kart would dangerously resume moving without the user being in control.

The BT connection bit is given by the hardware control register - Section 7.2.

The Figure 22 shows a timing diagram corresponding to the behavior of the DC motor controller when the Bluetooth connection is lost and then re-established:

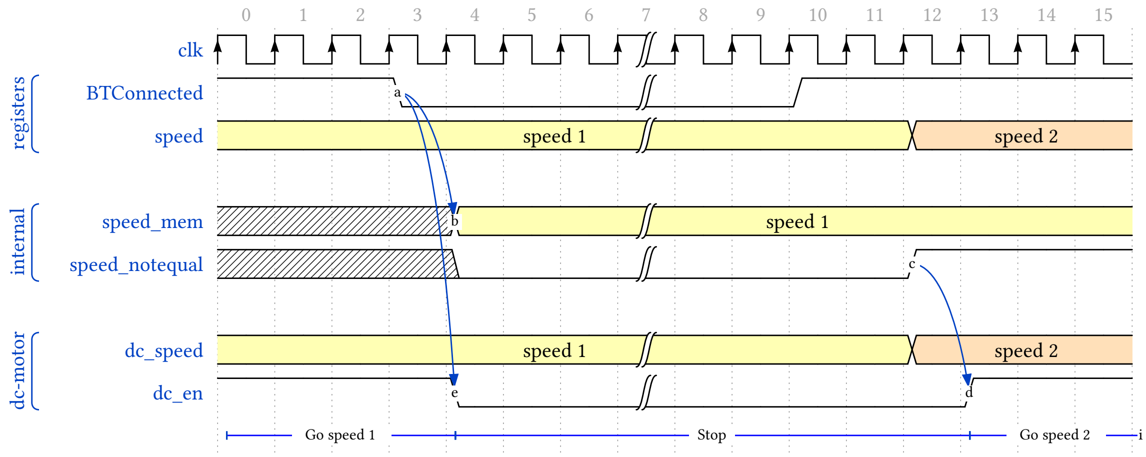


Figure 22 - Bluetooth disconnection timing diagram

Update the circuit in order to stop the motor on connection loss and resume only after the connection is resumed AND the speed modified.

5.2.5 Restart

When the Android application connects or the user triggers the signal manually, it sends a restart command to the Kart. To reflect this, the signal **`restart`** rises to **`1`** and stays on as long as the stepper motor did not hit the end switch.

Since the user has during this time no control, the DC motor must not move at all.

When the signal falls, the motor must not move until a new **`speed`** has been sent, mimicking the behavior as when the bluetooth disconnects - see Figure 22.

Update the circuit in order to stop the motor when **`restart`** = **`1`** and resume only after the connection is resumed AND the speed modified.

5.2.6 Tests

Refer to Section 6.1.1 - DC Motor testing to test your block fully before deploying it on the FPGA.

5.3 Stepper Motor Controller

The stepper motor controller is in charge of the Kart's steering to reach the desired angle.

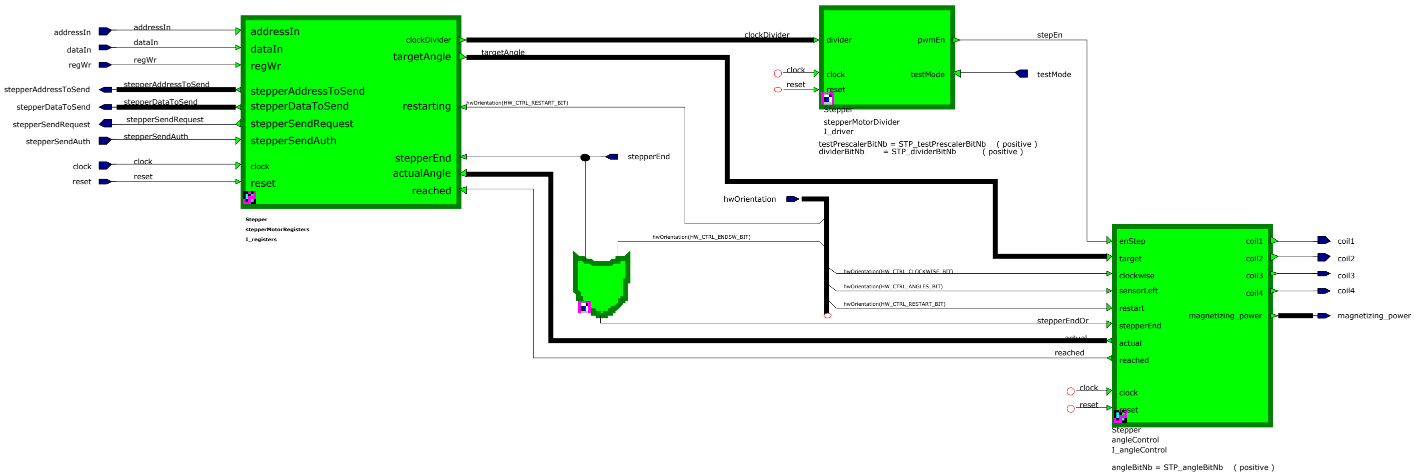


Figure 23 - Stepper module top-level

For this, five signals are generated to control the hardware and 2 signals to give feedback to the software:

- **coil1**, **coil2**, **coil3** and **coil4** power the different coils of the stepper
- **magnetizing_power** defines the mean voltage applied to active coils
- **reached** is set to `'1'` when the target angle is reached
- **actualAngle** gives the current angle of the stepper motor

5.3.1 Overview

The coils must be powered in sequence to allow the stepper to rotate, but due to the nature of the wiring, switching coils too fast may result in it slipping, i.e. the motor does not have the time to join the magnetized coil before another one pulls it back.

For this the block **`stepperMotorDivider`** creates an **`enStep`** signal which pulses at a frequency given by the **`prescaler`** register following the formula:

$$f_{\text{PWM\_step}} = \frac{f_{\text{base}}}{\text{prescaler}} = \frac{100\text{kHz}}{\text{prescaler}} \quad (3)$$

The user must then, based on this signal and the **`targetAngle`** set in the stepper motor target angle register - Section 7.2, actuate the coils in the right order to join the given position.

It must keep track of the current angle itself since no external sensor gives this information, and reflect it in the **`actualAngle`** output to be sent to the smartphone, setting the signal **`reached`** to **`'1'`** when the angle is reached.

Finally, magnetizing coils for too long will heat the motor up. When the kart is not turning, the consumption can be minimized by reducing the value of the **`magnetizing_power`** signal.

> The angle values represent the number of steps of the steppermotor. The maximum angle is 4096 (12 bits), which corresponds to 307° on the current 4800 steps motors. The stepper end switch represents the zero position - see Section 4.4.2

5.3.2 Driving coils

To control the stepper, one coil at a time should be magnetized, known as wave drive.

Other driving methods such as full-step and half-step are also allowed. In such case, knowing that multiple coils may be active at the same time, you **NEED** to request for a new fuse to be installed on your board. The default one is not able to handle such current draw.

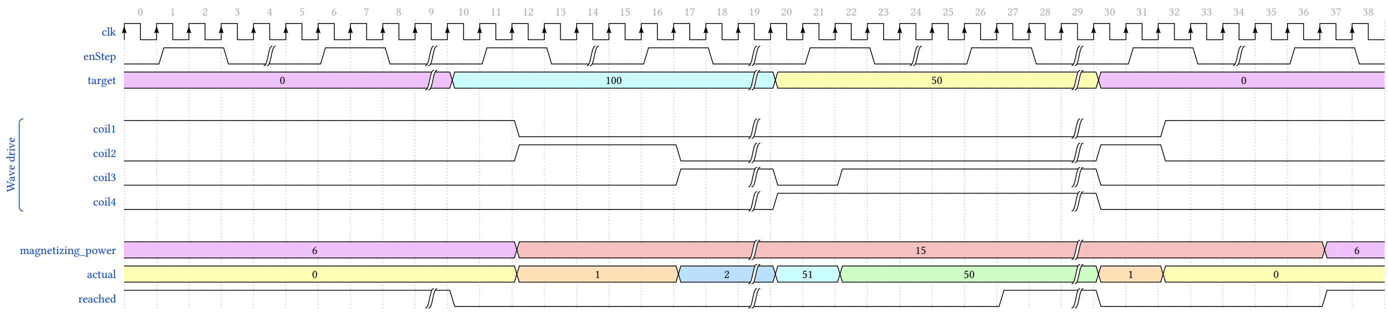


Figure 24 - DC Motor timings diagram

Each time the signal **`enStep`** pulses, the coils must change state in the correct direction to reach the target. For that, a counter must remember how many steps have been taken until now and write it in the vector **`actual`**. This vector is always positive and given on 12 bits.

The flag reached should:
- fall to **`'0'`** when **`target`** is not the same as **`actual`**
- rise to **`'1'`** when **`target`** is the same as **`actual`** **AND** the last step has been fully taken

*Coils must always be magnetized enough, i.e. at least one **`enStep`** duration long to avoid stacking up drift.*

The **`magnetizing_power`** signal must be set to a value between 0 and 15, 0 being no power and 15 the highest. It should change when the motor is in idle compared to when in movement.

The vector **`actual`** and the flag **`reached`** are both used internally to trigger events which will transmit their values to the smartphone. A mishandling of those signals will result in a flood of the communication system, which may in turn imply loss of data with the smartphone, commands not updating and the GUI freezing because of too many interrupts.

Draw the circuit of the **`angleControl`** block.

5.3.3 Initialisation of the Kart

After startup, the system does not know where the wheels are. The Kart cannot be controlled until a restart phase is performed. During the initialisation the wheels are turned in the direction of the **`stepperEnd`** switch until it is reached. The angle is then set to **`zero`** or **`target*2`** - see Table 2. The **`target`** vector being already set, the wheels will then position themselves in the middle of the Kart. The **`target`** value depends on the hardware setup of the steering made on the smartphone.

5.3.3.1 Restart signal

The initialisation phase is activated by the **`restart`** signal coming from the hardware control register - Section 7.2. This bit is set by the remote control smartphone after a successful Bluetooth pairing, together with the appropriate **`sensorLeft`** and **`clockwise`** setup bits as well as the stepper motor period register, in order for the FPGA hardware to discover the zero angle position. A restart could also be issued anytime from the smartphone to recover from events like excessive drift of the wheels.

The **`restart`** signal remains in a high state until the **`stepperEnd`** switch is pressed.

5.3.3.2 Hardware orientation

The mechanical design allows for the following variations:

- the **`stepperEnd`** switch can be placed at the maximum steering angle on the left or right side, which can be setup with the **`sensorLeft`** signal.
- **`clockwise`** changes the coil sequence 1 ⇨ 2 ⇨ 3 ⇨ 4 ⇨ … (**`clockwise = '1'`**) to 4 ⇨ 3 ⇨ 2 ⇨ 1 ⇨ … (**`clockwise = '0'`**. Depending on how the motor is mounted, the steering will turn to the left or right.

The corresponding setup bits are configured in the hardware control register - Section 7.2.

The following 4 combination of those signals changes the **`restart`** behaviour:

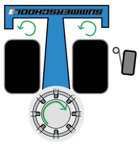
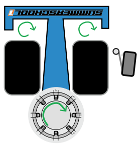
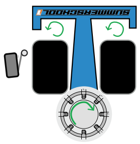
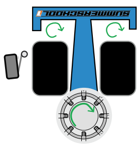
sensorLeft	clockwise	Situation	Reset phase
0	0		Reverse coils order during reset Reset **`actual = 0`** Normal coils order to reach target
0	1		Normal coils order during reset Reset **`actual = target*2`** Reverse coils order to reach target
1	0		Normal coils order during reset Reset **`actual = target*2`** Reverse coils order to reach target
1	1		Reverse coils order during reset Reset **`actual = 0`** Normal coils order to reach target

Table 2 - Stepper motor hardware configurations

The **actual** vector can only be positive, so there are two cases where it is reset to 0, and two others where the reset value is target $* 2$. This is due to the fact that the **stepperEnd** switch is once on the minimum steering position and once on the maximum steering position. The minimum steering position is **0** and the maximum steering position is **target*2**. The **target** value during restart represents the central position of the kart, which means **target*2** is the maximum steering position.

A mux is already provided generating the signal **zeroingValue** to load the correct value on reset.

5.3.4 Tasks Summary

1. After startup, do not move the wheels until a **restart** signal is received
2. Upon detecting a **restart**, the system needs to transistion into a restart state
3. Move the wheels according to the **sensorLeft** and **clockwise** signals to hit the **stepperEnd** switch
   - Once it is hit, the **restart** signal falls. If the switch is already pressed when **restart** rises, it stays on for only one clock period, letting the **actual** vector to be reset.
4. When hitting it, reset the **actual** vector to the value given by the **zeroingValue** signal depending on **sensorLeft** and **clockwise**
5. Move according to the **target** while counting and outputting the **actual** value
   - When idle, set the **magnetizing_power** to a lower value
   - When moving, set the **magnetizing_power** to a higher value
   - When a new position is requested, set the **reached** signal to **'0'** until reaching it
   - When the position is reached and the last step is fully taken, set the **reached** signal to **'1'**

During the restart phase, the steering motor will not try to turn further than what the kart's mechanical structure allows because of the **stepperEnd** switch. In the other direction, it is the programmer's task not to request a too large **target** angle.

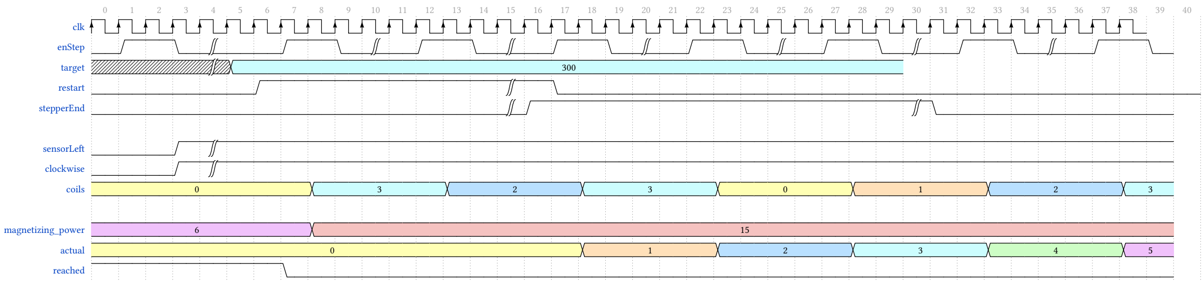


Figure 25 - Stepper restart sequence diagram

Update the circuit to integrate the restart phase.

5.3.5 Tests

Refer to Section 6.1.2 - Stepper Motor testing to test your block fully before deploying it on the FPGA.

5.4 Sensors Controller

The sensors controller handles the other I/Os of the system:

- Generic, 0-3.3V digital inputs. *Up to 16 inputs.*
- Generic, 0-3.3V digital outputs. *8 available outputs, shared with servomotors control outputs.*
- Servomotors control signals outputs. *8 available outputs, shared with generic outputs.*
- User dedicated outputs. *8 available 16 bits registers.*
- Dedicated I2C battery voltage and current reader.
- Dedicated hall sensors reader.
- Dedicated supersonic range finder pulse reader.

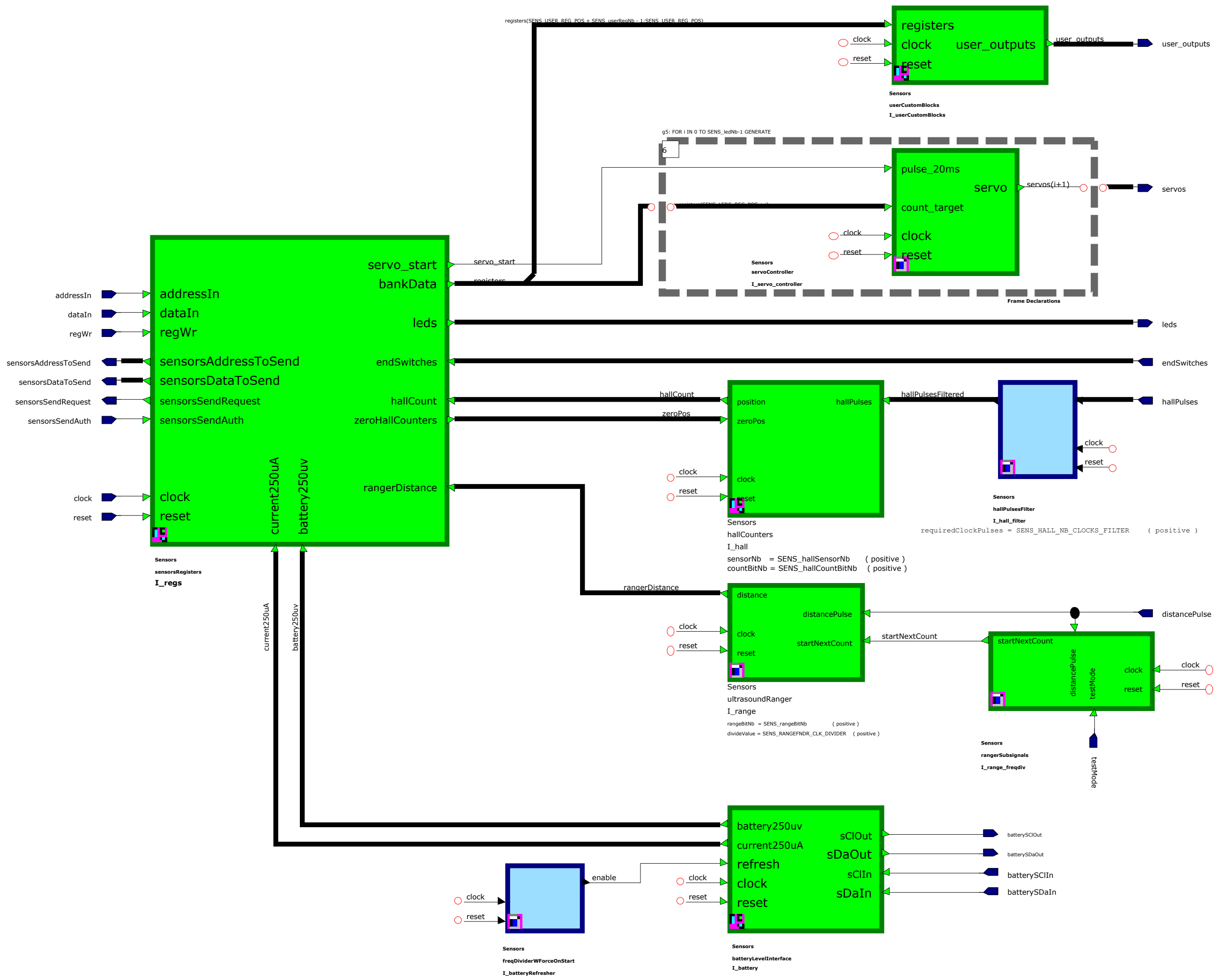


Figure 26 - Sensors module top-level

5.4.1 Overview

5.4.1.1 Outputs

The **`leds`** vector is controlled by the smartphone and used for generic outputs at will. It can simply set those on or off, but also handle toggling them at predefined frequencies. See the LEDx registers - Section 7.2.

The **servos** vector is controlled by the smartphone and used for servomotors outputs at will. It recreates the required signal pulses to control servomotors. See the SERVOx registers - Section 7.2.

The **user_outputs** vector may contain any output pins for user-specific features not related to generic outputs, e.g. controlling an SPI leds strip, send UART commands to a module …

5.4.1.2 Inputs

The **endSwitches** vector is used as generic inputs. Each transition of their value is forwarded to the smartphone.

> As the state changes of the **endSwitches** are transmitted to the smartphone, these inputs are not designed for rapid fluctuations. Swift changes could potentially overwhelm the communication system, leading to issues such as data loss with the smartphone, commands failing to update, and the graphical user interface (GUI) freezing due to an excessive number of interrupts.

The **freqDividerWForceOnStart** along the **batteryLevelInterface** blocks handle communication with the motherboard's I2C voltage and current reader - see Section 4.2.1.3. Both values are read periodically, typically each second, and forwarded to the smartphone when they change from at least a predetermined value.

The **hallPulses** vector reflect the state of the hall sensors. They are used to determine the Kart's speed / slipping.

The **distancePulse** signal is used to read the distance from the ultrasound ranger. It can detect obstacles in front or back of the Kart while giving a range estimation of the obstacle.

5.4.2 Hardware setup

Based on which sensors you intend to use or not, modify the **Kart_Student** package in the **Kart** library:

- Set **STD_HALL_NUMBER** to the desired number of hall sensors to use *(1-2)*
- Set **STD_ENDSW_NUMBER** to the desired number of digital inputs to use *(0-16)*
- Set **STD_LEDS_NUMBER** to the desired number of digital outputs to use *(0-8)*
- Set **STD_SERVOS_NUMBER** to the desired number of servomotors to use *(0-8)*

> - When using both LEDs and servos, the sum of both must not exceed 8.
> - Registers will be stacked in the order LEDs then servos. E.g.: 3 LEDs and 2 servos ⇨ LEDs will use registers LEDS_1 to 3, and servos registers SERVOS_4 and 5.

> Setup the **Kart_Student** package.

5.4.3 Hall counter

The hall sensors presented in Section 4.6 create pulses based on the speed of the Kart. They are first passing through the **`hallPulsesFilter`** block which is implemented and does the following:
- Get the raw pulses from the hall sensors
- Debounces the input to avoid false transitions
- Transmits the filtered pulses to the **`hallCounters`** block

The **`hallPulses`** vector contains one bit per hall sensor. Using one or two sensors is up to the user. The second one can be used to detect slipping, calculate the covered path …

Even if you only use one hall sensor, the second exists, just held to **`'0'`** *at all time.*

The following behavior must be implemented:
- The **`hallCounters`** block receives the **`hallPulses`** vector and must count both the rising and falling edges of the pulses for each sensor separately.
- If the vector **`zeroPos`** is set to **`'1'`** for a particular sensor, the corresponding counter must reset to 0.

The two hall counters values must be concatenated to the **`hallCount`** vector. The second counter needs to be shifted by 16bit: $\mathrm{hallCount} = (\mathrm{hallCount}_2 \ll 16) + \mathrm{hallCount}_1$. It must **AT ALL TIME** reflect the current counters values.

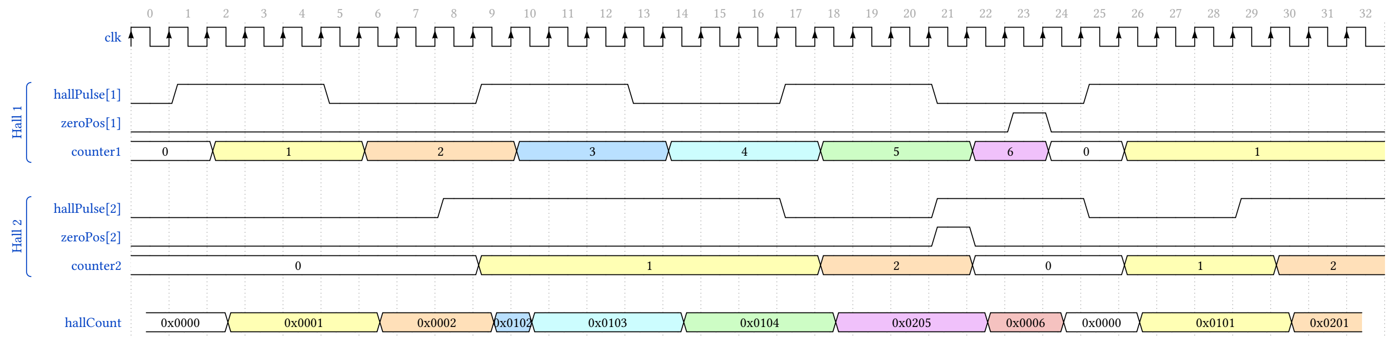


Figure 27 - Hall sensors pulses

Draw the circuit of the **`hallCounters`** block.

5.4.3.1 Tests

Refer to Section 6.1.3 - Sensors testing to test your block fully before deploying it on the FPGA.

5.4.4 Ultrasound Ranger *(Optional)*

An ultrasound ranger is useful to detect obstacles in the front or back of the Kart. Based on the PMOD-MAXSONAR board from Digilent, it can be plugged into any one-row PMOD connector.

The ranger outputs a pulse named **PW** whose length is to be counted. The distance to an object is then determined following the rule $147\frac{\mu s}{\text{inch}} = 57.84\frac{\mu s}{\text{cm}}$.

There is no start/stop indication: the sensor continuously outputs pulses between 0.88 and 37.5 ms long, each 49 ms.

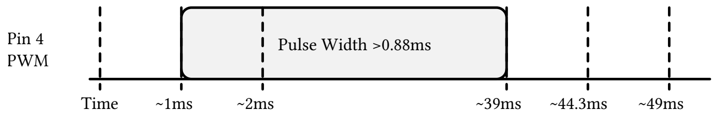


Figure 28 - MaxSonar PW pulse

It must implement the following behavior:
- Wait for the signal **`startNextCount`** to be **`'1'`** (around each 333 [ms]), indicating that the next pulse must be registered. This signal intends to slow down updates of the **`distance`** signal to avoid flooding the communication system.
- Wait for the next incoming pulse from **`distancePulse`**
- Count the pulse length in **microseconds**. *Counting clock periods would overflow the 16-bit register.*
- Update the **`distance`** vector with the new value

It is up to the user to decide wheter to send invalid pulses, i.e. calculated time > 37.5 [ms] or < 0.88 [ms] by sending the calculated value nevertheless, a predefined value instead, nothing …

Make sure to update the **`distance`** vector only after **`startNextCount`** has been detected and the whole pulse has been counted to avoid flooding the communication.

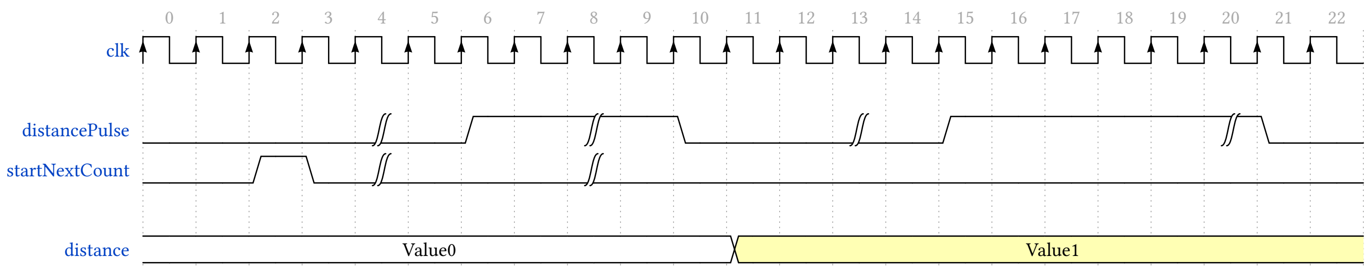


Figure 29 - Distance sensor timing diagram

Draw the circuit of the **`ultrasoundRanger`** block.

5.4.4.1 Tests

Refer to Section 6.1.3 - Sensors testing to test your block fully before deploying it on the FPGA.

5.4.5 Servomotors controller *(Optional)*

Servomotors are easy to control and allow for many applications requiring circular motions. Those can also be transformed easily into linear ones with the help of a bit of mechanic.

A typical servomotor requires a pulse each 20 [ms] whose duration ranges from 1 [ms] (-90°) to 2 [ms] (90°) as shown in the following figure:

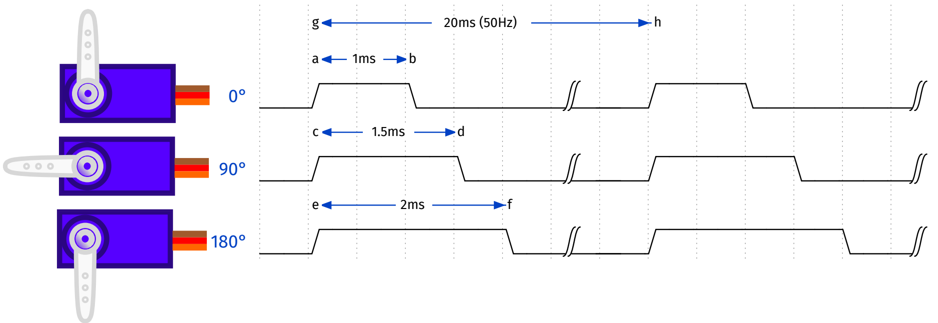


Figure 30 - Servo Motor Control pulse

Each model of servomotor may diverge from the standard, having wider or narrower angles ranges. It is up to you to check that the model you intend to use fits your expectations. Also, each one has a specific maximal torque you must not exceed.

> Do not try to send pulses outside the standard range, as the servo may overheat and break.

The block **`servoController`** must implement the following behavior:

- Set the output **`servo`** to **`'0'`**
- Wait for the signal **`pulse_20ms`** to rise
- Set the output **`servo`** to **`'1'`** and begin counting
- Once the count value corresponds to the one given in **`countTarget`**, set the output back to **`'0'`**
  - If the **`countTarget`** changes during the pulse generation, wait for the next **`pulse_20ms`** to take it into account
- Wait for the next **`pulse_20ms`** to start again

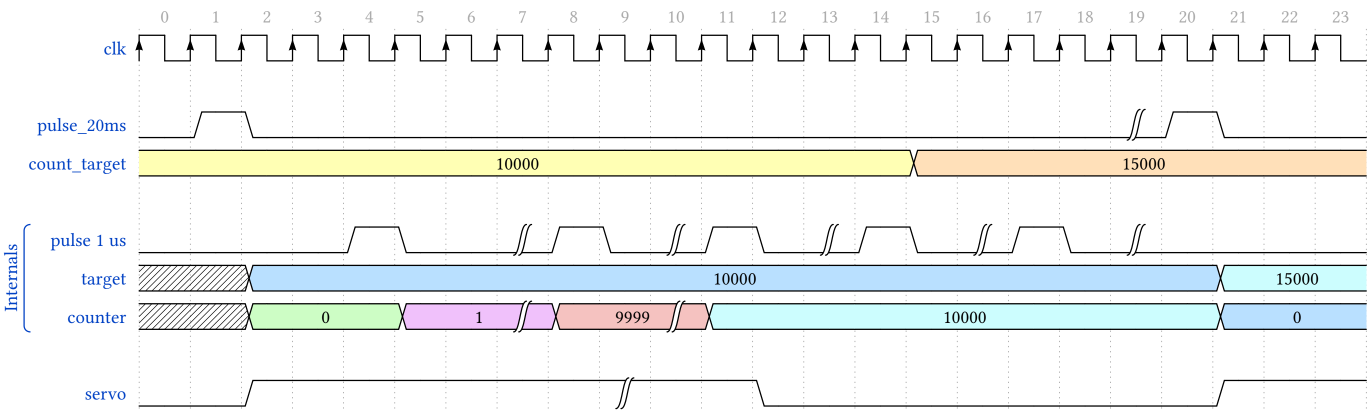


Figure 31 - Servomotors timing diagram

Draw the circuit of the **`servoController`** block.

5.4.5.1 Tests

Refer to Section 6.1.3 - Sensors testing to test your block fully before deploying it on the FPGA.

5.4.6 User functionalities *(Optional)*

The **`userCustomBlocks`** can be used to create specific user features.

As input, the **`registers`** signal is an array of 8 times 16 bits registers that can set by the smartphone through the USERx registers - Section 7.2.

As output, the **`user_outputs`** signal is a vector of **`STD_USER_OUTPUTS_NUMBER`**. It is already wired up to the top level, making it possible to link your signals to physical pins of the FPGA.

Modify the **`Kart_Student`** package by setting the **`STD_USER_OUTPUTS_NUMBER`** constant to how many signals you intend to output from the **`userCustomBlocks`** block.

Internally, simply wire you signals one next to the other in the **`user_outputs`** vector.

You may use those to:

- Control an SPI LED strip
- Send UART commands to an external module
- Control seven segments displays
- Create specific tones with a buzzer
- …

5.5 Optional features

Fullfilling all mandatory objectives mentioned in Section 2.1 will result in a grade of 4.0. The students are free to implement additional features. For every added feature, the grade can be increased depending on the quality of the feature, until the maximum grade of 6.0 is reached. Hereafter some ideas, but feel free to imagine your own.

5.5.1 DCMotor - Acceleration ramp

The DC motor controller can be improved by adding an acceleration / braking ramp. This feature allows the motor to smoothly modify its speed until the desired one is reached, avoiding sudden movements and protecting it from high current peaks and is done FPGA-side only

Supporting it gives you up to 1.0 on your grade if you can show a correct behavior by:
- Showing a correct acceleration ramp in simulation when the **`speed`** register is modified, even if it is modified **while** already ramping to another speed
- ***Creating a DC ramp through the use of the smartphone does not count as an FPGA feature***

5.5.2 StepperMotor - Dynamic steering frequency

The stepper motor makes use of a fixed frequency to move the wheels, but it cannot be too high since the motor would slip due to sudden acceleration. But if the motor begins moving with a slow frequency, it can then be increased with the motor already being in rotation since the inertia helps reduce the acceleration needed.

Supporting it gives you up to 1.0 on your grade if you can show a correct behavior by:
- Showing a correct simulation where the frequency changes when a **`target`** is set, and how it reacts if the **`target`** is modified while the motor is already moving
- ***Modifying it through the use of the smartphone does not count as an FPGA feature***

5.5.3 Sensors

5.5.3.1 Ultrasound ranger

Implementing the ultrasound ranger - section 5.4.4 can be done FPGA-side only, *but a smartphone use case is recommended.*

Supporting it gives you up to 0.5 on your grade if you can show a correct behavior by:
- Using the USB tester - section 6.5 to show that the distance is correctly transmitted when moving an object in front of the ranger
- Demonstrating how it works through a smartphone implementation
  - By using it as a parking sensor, audibly or visually, showing that the distance from an object impacts the feedback on the smartphone
  - Displaying the live distance in a label / animation
  - Other functionality proving that the ranger is correctly implemented

5.5.3.2 Servomotors

Implementing the servomotors - section 5.4.5 can be done FPGA-side only, *but a smartphone use case is recommended.*

Supporting it gives you up to 0.5 on your grade if you can show a correct behavior by:
- Using the USB tester - section 6.5 to move it to various positions on demand
- Demonstrating how it works through a smartphone implementation
  - By moving it to various positions on demand: slider, buttons, …

- Other functionality proving that the servomotor is correctly implemented **through non-continuous, various positioning**

5.5.4 Other

You may propose any other feature to implement which add "something" to your system:
- A special way to control part of the Kart (0.5-1.0)
- SPI leds strip control (2.0)
- Led blinking (0.1-0.5)
- New sensors support requiring VHDL implementation
- …

Discuss those with a supervisor to establish the feasibility and the grading for the task.

6 Testing

Three types of testers are available to fully validate the design before flashing the FPGA .
Per module simulation - specific functionalities of the circuit:
- DC Motor Module - Section 6.1.1
- Stepper Motor Module - Section 6.1.2
- Sensors Module - Section 6.1.3

Circuit simulation - overall circuit behavior:
- Overall circuit (modules only) - Section 6.2.1
- Full circuit (with COM emulation) - Section 6.2.2

Finally, a USB tester - Section 6.5 allows to test and control the Kart by using a PC to emulate the smartphone by connecting it directly via USB.

> 🔥 Always complete simulations tests before any wiring and programming of the board. Always use a stabilised DC power supply while developing.

Contents

6.1 Per module ... 42
- 6.1.1 DC Motor testing ... 42
- 6.1.2 Stepper Motor testing ... 43
- 6.1.3 Sensors Controller testing ... 45

6.2 Whole circuit ... 48
- 6.2.1 Modules Simulation ... 48
- 6.2.2 Full-board ... 50

6.3 Setting up the board ... 51
- 6.3.1 I/Os configuration ... 51
- 6.3.2 Pining setup ... 51
- 6.3.3 Onboard LEDs ... 52

6.4 Programming the board ... 52

6.5 USB commands emulation ... 53
- 6.5.1 Quick Test ... 54
- 6.5.2 Registers R/W ... 54

6.1 Per module

Each module can be tested individually.

6.1.1 DC Motor testing

The DC motor functionality can be tested through the **`DCMotor_test`** ⇨ **`dcMotor_tb`** block.

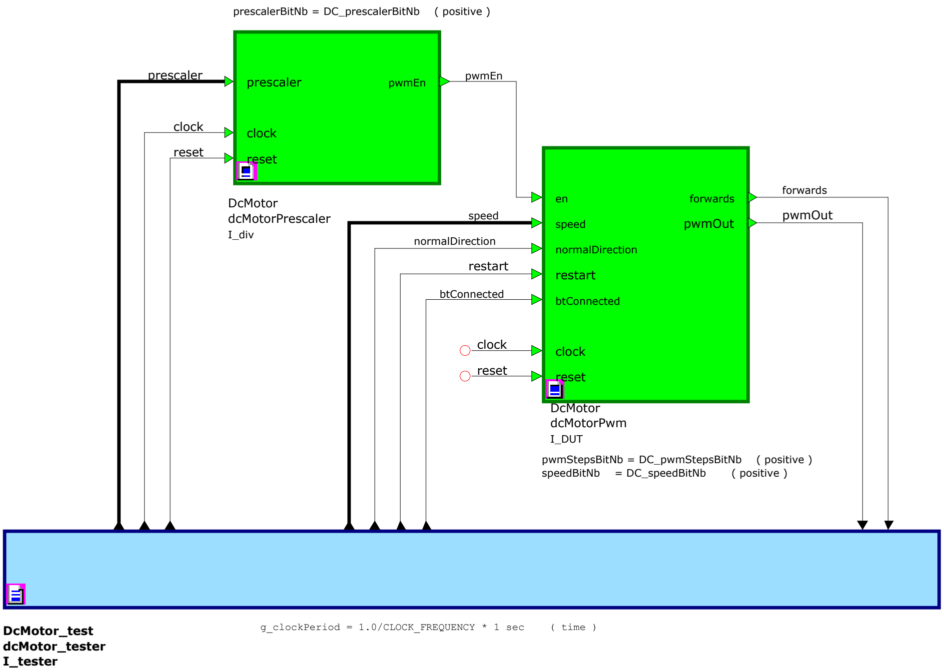


Figure 32 - DC Module Testbench

The corresponding simulation layout file for Modelsim is available under **`$SIMULATION_DIR/DCMotor/dcMotor.do`**. Predefined signals are color coded:

- The blue header shows which test is performed
- The yellow signals are those generated by the tester
- The purple signals are the one generated by your implementation
- The green signals are internal ones

The tester **`DCMotor_test`** ⇨ **`dcMotor_tester`** only generates **clock** and **reset**. It must be filled by yourself - you should notably test:

- Setting a correct **`prescaler`** value and the **`btConnected`** signal
- Setting a few positive speeds values
- Setting a few negative speeds values
- Redo the tests with the **`normalDirection`** inverted
- Holding the **`restart`** signal at **`'1'`** and ensure the motor stops
- Releasing the **`restart`** signal and ensure the motor does not move until a new speed is sent
- Losing the **`btConnected`** signal and ensure the motor stops
- Retrieve the **`btConnected`** signal and ensure the motor does not move until a new speed is sent

6.1.2 Stepper Motor testing

The stepper motor functionality can be tested through the **`StepperMotor_test`** ⇨ **`stepperMotorController_tb`** block.

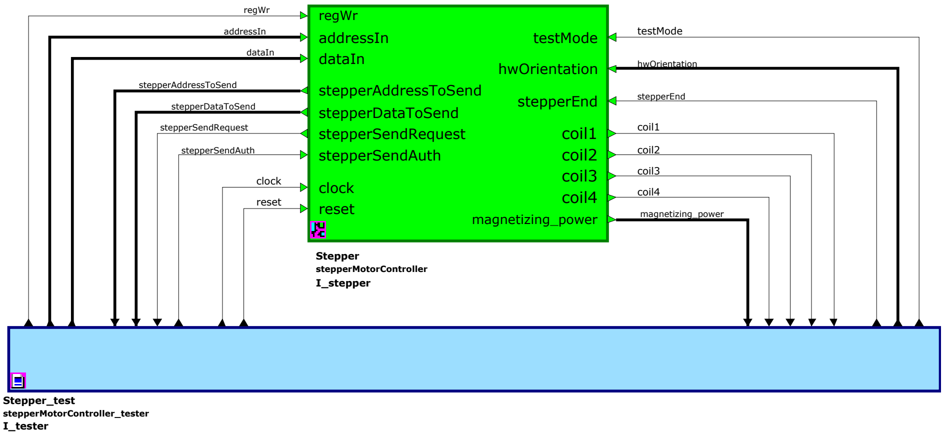


Figure 33 - Stepper Module Testbench

The corresponding simulation layout file for Modelsim is available under **`$SIMULATION_DIR/Stepper/stepperMotorController.do`**.

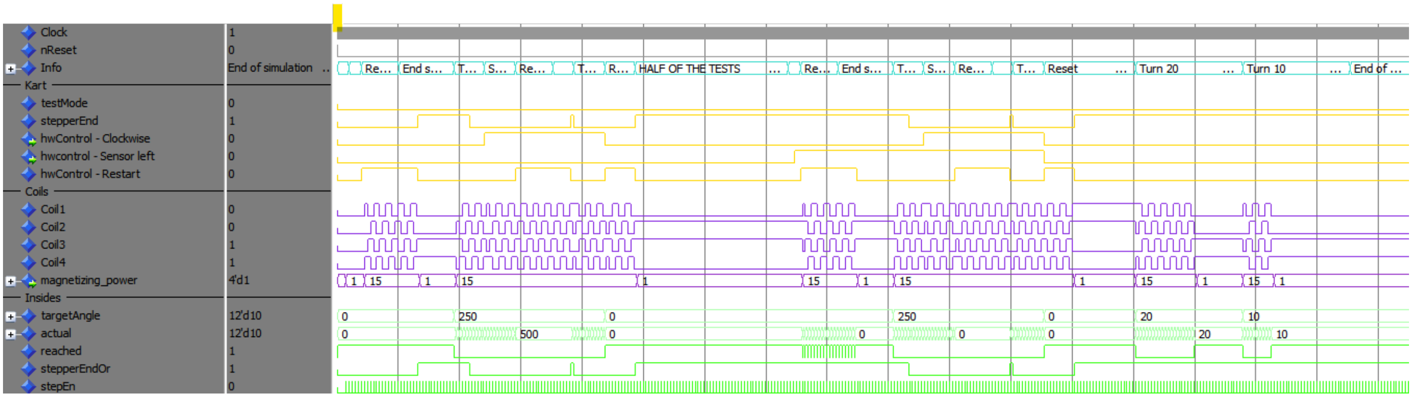


Figure 34 - Stepper Module Simulation

- The blue header shows which test is performed (correspond to signal **`testInfo`** in the tester)
- The yellow signals are those generated by the tester
- The purple signals are the one generated by your implementation
- The green signals are internal ones

6.1.2.1 Testing

The tester is already pre-filled and performs the following tests:

- The **`prescaler`** is set, outputting pulses on **`stepEn`**
- Waits a bit, expecting the coils to not move and the **`magnetizing_power`** to be reduced
- The **`restart`** signal is set to **`'1'`**, expecting the coils to turn as 4 ⇨ 3 ⇨ 2 ⇨ 1 ⇨ 4… and the **`magnetizing_power`** to be increased
- The **`stepperEnd`** is pressed, releasing the **`restart`** signal, expecting the coils to stop moving and the **`magnetizing_power`** to be reduced
- A **`target`** of **`250`** is set, expecting the coils to move in order 1 ⇨ 2 ⇨ 3 ⇨ 4 ⇨ 1… and the **`magnetizing_power`** to be increased

- **`clockwise`** is toggled, expecting the coils to move in order 4 ⇨ 3 ⇨ 2 ⇨ 1 ⇨ 4… and the **`magnetizing_power`** to be increased
- The **`restart`** signal is set to **`'1'`**, expecting the coils to turn as 4 ⇨ 3 ⇨ 2 ⇨ 1 ⇨ 4… and the **`magnetizing_power`**
- The **`stepperEnd`** is pressed, releasing the **`restart`** signal, expecting the coils to continue turning as 1 ⇨ 2 ⇨ 3 ⇨ 4 ⇨ 1… and the **`magnetizing_power`** to be increased
- *A reset is performed*
- The **`sensorLeft`** signal is set to **`'1'`**, expecting the coils to stop moving and the **`magnetizing_power`** to be reduced
- The **`restart`** signal is set to **`'1'`**, expecting the coils to turn as 1 ⇨ 2 ⇨ 3 ⇨ 4 ⇨ 1… and the **`magnetizing_power`** to be increased
- The **`stepperEnd`** is pressed, releasing the **`restart`** signal, expecting the coils to stop moving and the **`magnetizing_power`** to be reduced
- A **`target`** of **`250`** is set, expecting the coils to move in order 1 ⇨ 2 ⇨ 3 ⇨ 4 ⇨ 1… and the **`magnetizing_power`** to be increased
- **`clockwise`** is toggled, expecting the coils to move in order 4 ⇨ 3 ⇨ 2 ⇨ 1 ⇨ 4… and the **`magnetizing_power`** to be increased
- The **`restart`** signal is set to **`'1'`**, expecting the coils to turn as 1 ⇨ 2 ⇨ 3 ⇨ 4 ⇨ 1… and the **`magnetizing_power`** to be increased
- The **`stepperEnd`** is pressed, releasing the **`restart`** signal, expecting the coils to continue turning as 4 ⇨ 3 ⇨ 2 ⇨ 1 ⇨ 4… and the **`magnetizing_power`** to be increased
- *A reset is performed*
- With **`clockwise`** set to **`0`** and **`sensorLeft`** set to **`0`**, the **`target`** is set to **`20`**, expecting the coils to move in order 1 ⇨ 2 ⇨ 3 ⇨ 4 ⇨ 1… and the **`magnetizing_power`** to be increased. When the count is 20, the **`reached`** signal should be set to **`'1'`** and the **`magnetizing_power`** should be reduced
- The **`target`** is set to **`15`**, expecting the coils to move in order 4 ⇨ 3 ⇨ 2 ⇨ 1 ⇨ 4… and the **`magnetizing_power`** to be increased, the **`reached`** signal falling. When the position is reached, **`teached`** should be set to **`'1'`** and the **`magnetizing_power`** reduced

Transcript

The transcript window gives you details on if automated tests passed or not:

- For the **`Coil1...4`** signals, you get **`Coil direction OK`** or **`Coil direction error`**.
- For the **`reached`** signal, you get **`Reached flag OK`** or **`Reached flag error`**.
- For the **`actual`** signal, you get **`Position readback OK`** or **`Position readback error`**.

> **Automated tests** are here to help you debug your design. Following your implementation choices, they may give wrong logs.
> **In any case, you CANNOT count on automated tests only. Always validate your design by checking your signals.**

6.1.3 Sensors Controller testing

The Hall Sensor, Ultrasound Ranger and servomotors outputs each have their own tester in the **`Sensors_test`** library.

> **Hardware setup**
> Remember to have set constants based on which sensors you use. For this, modify the **`Kart_Student`** package in the **`Kart`** library:
> - Set **`STD_HALL_NUMBER`** to the desired number of hall sensors to use *(1-2)*
> - Set **`STD_SERVOS_NUMBER`** to the desired number of servomotors to use *(0-8), at least 1 to test the block*

6.1.3.1 Hall Sensor

The Hall Sensor functionality can be tested through the **`hall_tb`** block.

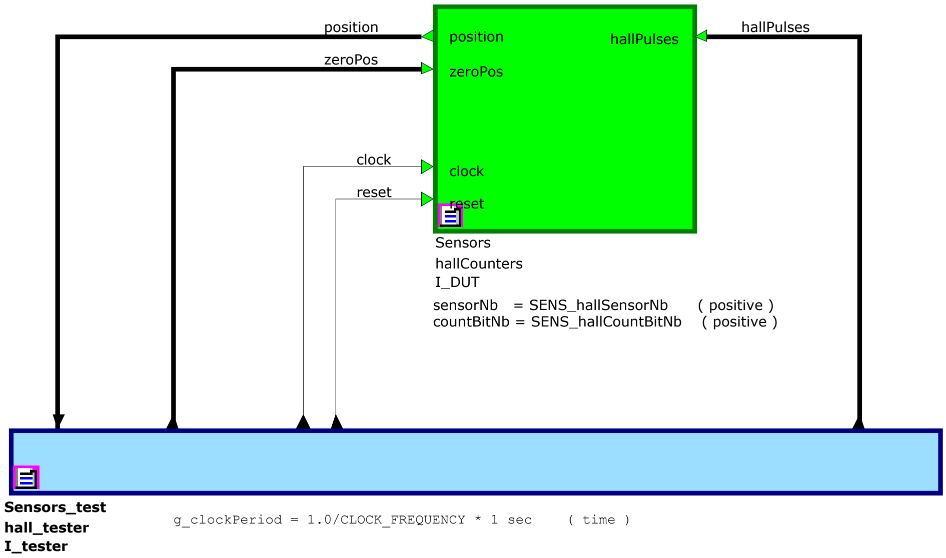


Figure 35 - Hall sensor Testbench

The corresponding simulation layout file for Modelsim is available under **`$SIMULATION_DIR/Sensors/hall.do`**. Predefined signals are color coded:
- The blue header shows which test is performed
- The yellow signals are those generated by the tester
- The purple signals are the one generated by your implementation
- The green signals are internal ones

The tester **`Sensors_test`** ⇨ **`hall_tester`** only generates **clock** and **reset**. It must be filled by yourself - you should notably test:
- Create pulses mimicing the hall sensor for all your sensors by writing to **`hallPulses`**
- Count all the rising and falling edges of those pulses
- Ensure the **`position`** signal always reflect the concatenation of your counters like **`position(31 downto 16) = hall1`** & **`position(15 downto 0) = hall2`**
- Ensure the counters are reset when the signal **`zeroPos`** is **`'1'`** for a sensor

6.1.3.2 Ultrasound Ranger (Optional)

The Ultrasound ranger functionality can be tested through the **`ultrasound_tb`** block.

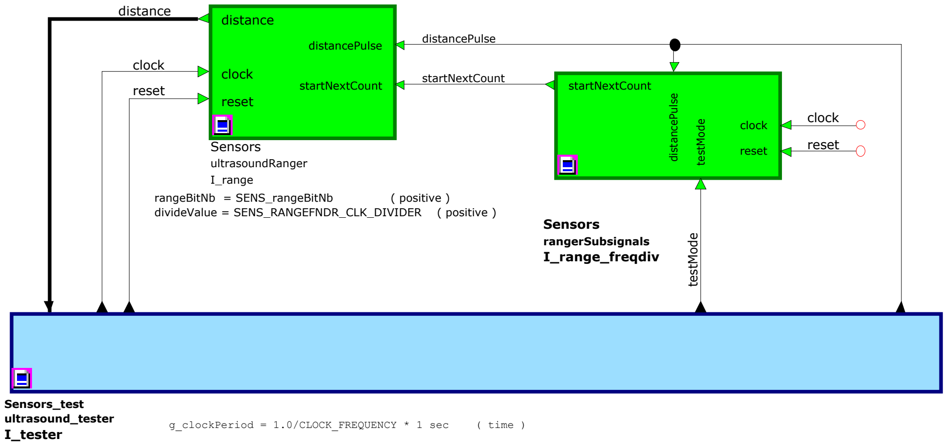


Figure 36 - Ultrasound ranger Testbench

The corresponding simulation layout file for Modelsim is available under **`$SIMULATION_DIR/Sensors/ultrasound.do`**. Predefined signals are color coded:

- The blue header shows which test is performed
- The yellow signals are those generated by the tester
- The purple signals are the one generated by your implementation
- The green signals are internal ones

The tester **`Sensors_test`** ⇨ **`ultrasound_tester`** only generates **clock** and **reset**.
The block **`rangerSubsignals`** generates a pulse each 100 clocks to simulate when the user should be updating the **`distance`** vector or not.
It must be filled by yourself - you should notably test:

- Generate a known length **`distancePulse`** signal, *smaller than the 100 clocks of the* ***`rangerSubsignals`*** *block*, after **`startNextCount`** has created a pulse
- Calculate pulse length and ensure it is correctly stored in the **`distance`** vector afterwards
- Generate two consecutive pulses, one right after **`startNextCount`** pulses and the second following close. Only the first pulse value should be stored in the **`distance`** vector

6.1.3.3 Servomotors (Optional)

The servomotor functionality can be tested through the **`servo_tb`** block.

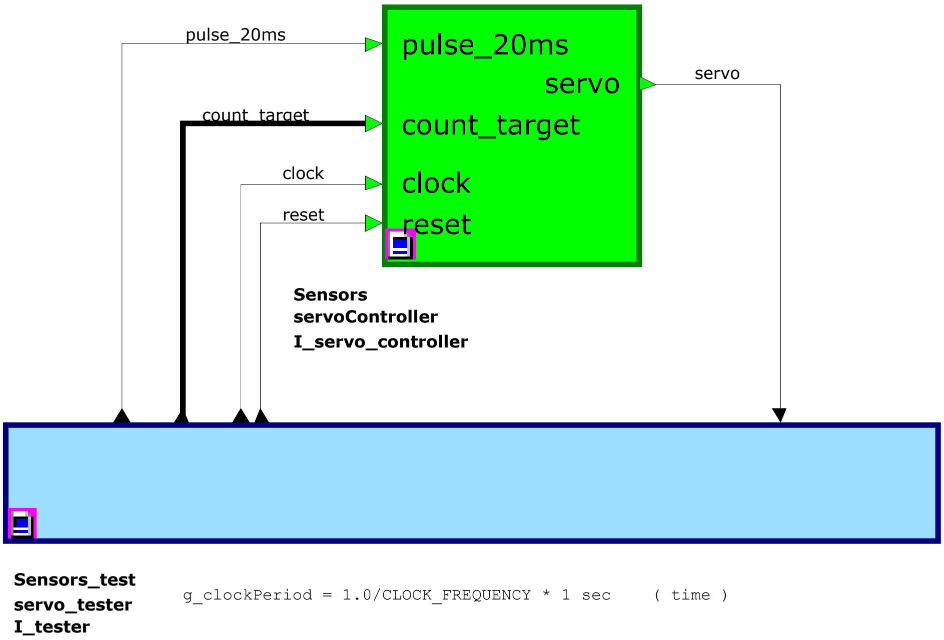


Figure 37 - Servomotor Testbench

The corresponding simulation layout file for Modelsim is available under **`$SIMULATION_DIR/Sensors/servo.do`**. Predefined signals are color coded:

- The blue header shows which test is performed
- The yellow signals are those generated by the tester
- The purple signals are the one generated by your implementation
- The green signals are internal ones

The tester **`Sensors_test`** ⇨ **`servo_tester`** generates **`clock`**, **`reset`**, and **`pulse_20ms`** each 30′000 clock cycles for the simulation.
It must be filled by yourself - you should notably test:

- Set a **`target`** and wait for **`pulse_20ms`** - the **`servo`** signal should stay low
- Once the **`pulse_20ms`** arrives, ensure that the **`servo`** signal goes high for a certain amount of time, then goes low again according to the **`count_target`** vector value
- Wait for **`pulse_20ms`** to arrive and try changing the **`count_target`** during counting - the **`servo`** signal should change according to the first **`count_target`** value and only on the next **`pulse_20ms`** take the new value into account

6.2 Whole circuit

6.2.1 Modules Simulation

In addition to the dedicated modules testers, the overall behavior can be tested through the **`Kart_test`** ⇨ **`kartController_tb`** block.

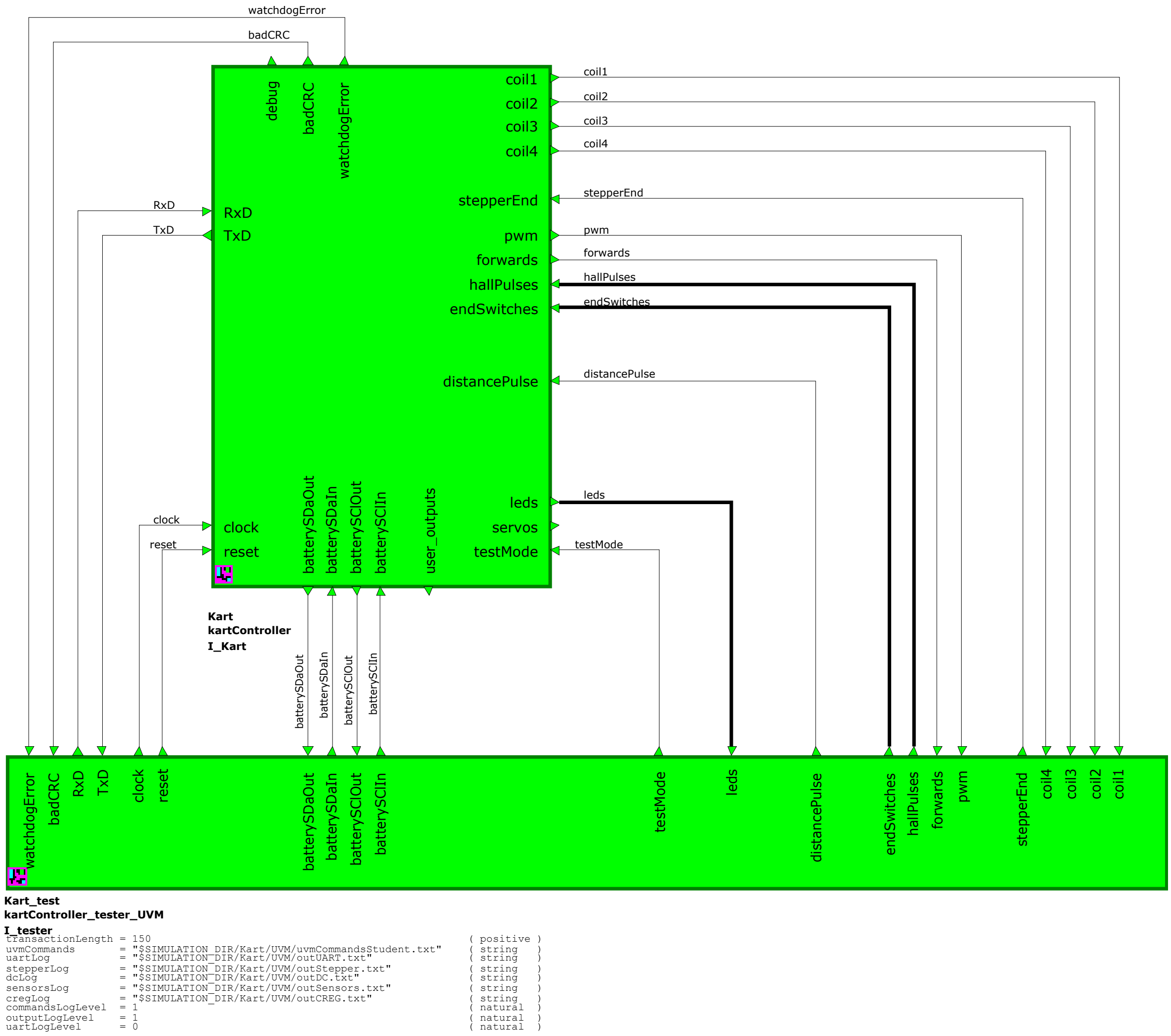


Figure 38 - Kart Toplevel Testbench

The tester internal layout differs because it makes use of the UVM technology - see Figure 39.

The tester reads the commands given in **`$SIMULATION_DIR/Kart/UVM/uvmCommandsStudent.txt`** and creates different logs under **`$SIMULATION_DIR/Kart/UVM/outXXX.txt`**. The file can be modified without the circuit being recompiled. The corresponding simulation layout file for Modelsim is available under **`$SIMULATION_DIR/Kart/kartStudent.do`**.

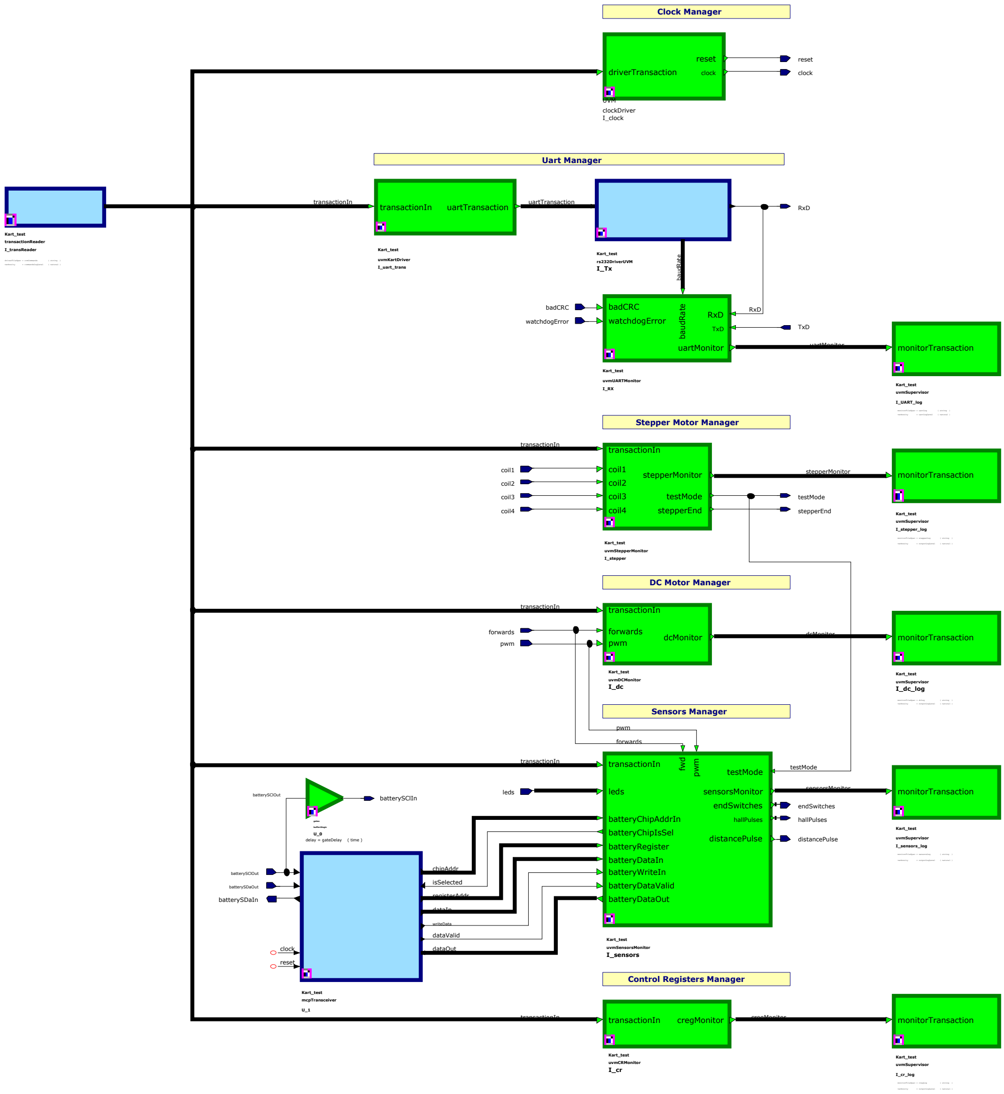


Figure 39 - Kart Toplevel Tester

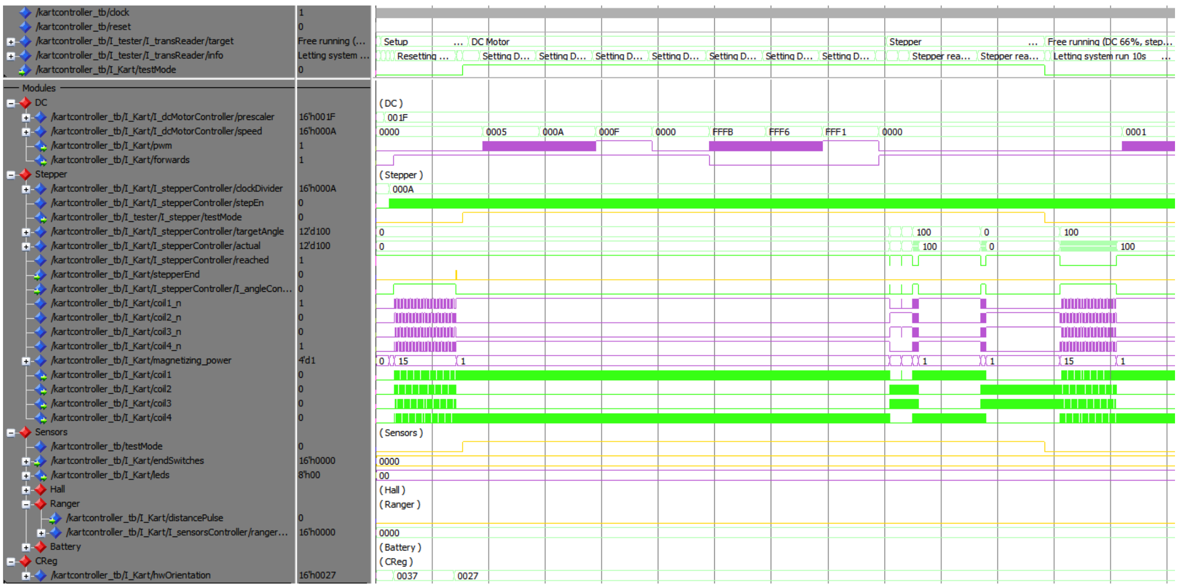


Figure 40 - Kart Toplevel Simulation

6.2.1.1 Tests

All the student-designed blocks are tested (check both **`target`** and **`info`** signals on the simulation). There is no direct error logging. One must check the functionality "by hand" in the simulation window (by correlating signals with the info from the transcript window or the log files).

6.2.2 Full-board

Another tester, checking the whole system (including Rx/Tx frames, registers managers …) can be loaded through the **`Kart_test`** ⇒ **`kartController_full_tb`** block and the **`$SIMULATION_DIR/Kart/kartStudent.do`** file.

It is mostly intended for people developing the full circuit, but left there for curious people:

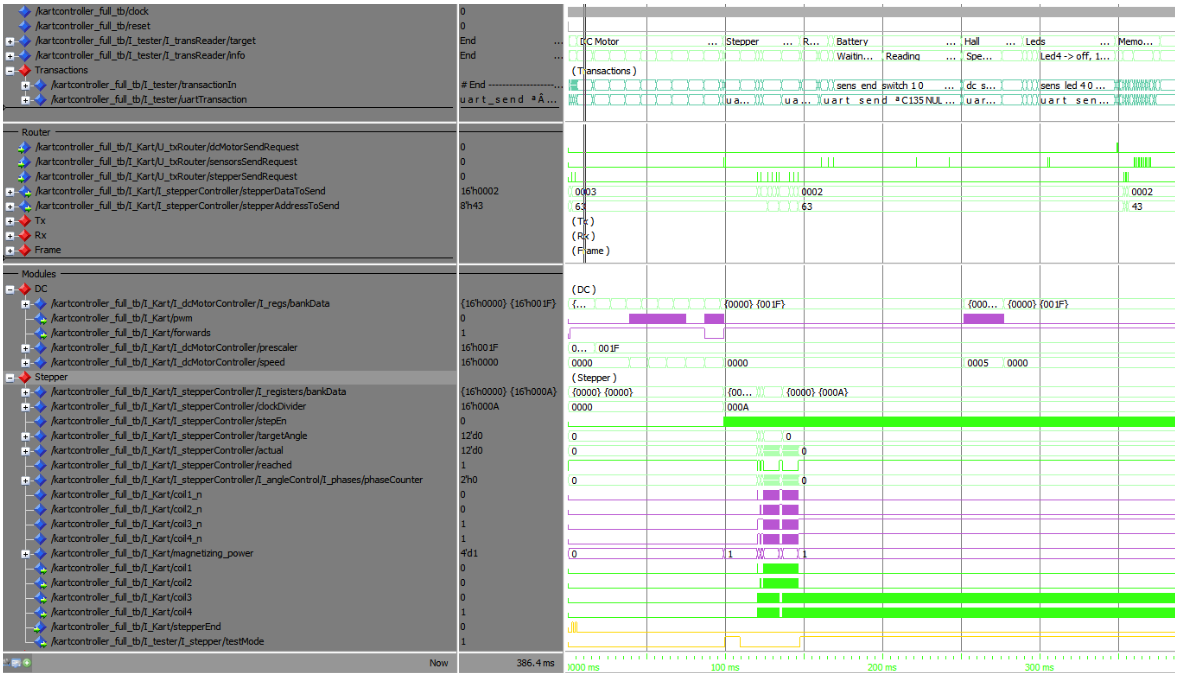


Figure 41 - Kart Full Simulation

6.3 Setting up the board

Once your whole circuit is working in simulation, the last step is to tell the FPGA which physical pin corresponds to which signal and setup how many inputs and outputs you really use.

6.3.1 I/Os configuration

Based on which sensors you intend to use or not, modify the **`Kart_Student`** package in the **`Kart`** library:

- Set **`STD_HALL_NUMBER`** to the desired number of hall sensors to use *(1-2)*
- Set **`STD_ENDSW_NUMBER`** to the desired number of digital inputs to use *(0-16)*
- Set **`STD_LEDS_NUMBER`** to the desired number of digital outputs to use *(0-8)*
- Set **`STD_SERVOS_NUMBER`** to the desired number of servomotors to use *(0-8)*
  - When using both LEDs and servos, the sum of both must not exceed 8.
  - Registers will be stacked in the order LEDs then servos. E.g.: 3 LEDs and 2 servos => LEDs will use registers LEDS_1 to 3, and servos registers SERVOS_4 and 5.

LEDs and Servos

LEDs and SERVOs registers are shared among the system and stacked next to the others (LEDs first). Here is an example:

- I have two ‘leds’ type outputs, and three servomotors.
- I set **`STD_LEDS_NUMBER`** to **`2`**.
- I set **`STD_SERVOS_NUMBER`** to **`3`**.
- In the pining (see next chapter), I assign **`{leds[1]}`** and **`{leds[2]}`** to my two outputs, and **`{servos[1]}`**, **`{servos[2]}`** and **`{servos[3]}`** to the servomotor outputs.
- From my application, I set registers **`LED1`** and **`LED2`** as defined in Table 10 for leds, and **`LED3`**, **`LED4`** and **`LED5`** as defined for servos.

6.3.2 Pining setup

This is done by altering the constraints file found under **`Board/concat/Kart.pdc`**.

All PMODs are listed, along with other signals such as the clock, I2C, UART …

A simple signal is defined with its name like **`stepperEnd`**, and signals from a vector are written such as **`{leds[1]}`**.

> **Warning**
> Do not modify the signals which are not linked to PMODs I/Os.

Inputs
To wire an input, set the IO to the correct signal name such as:
`set_io myVHDLSignalName -pinname 97 -fixed yes -DIRECTION Input`
You can also append `-RES_PULL Up` and `-RES_PULL Down` to the end of the line to enable a pull-up or pull-down resistor.

Outputs
To wire an output, set the IO to the correct signal name such as:
`set_io myVHDLSignalName -pinname 97 -fixed yes -DIRECTION Output`
Refer to the PMOD - Appendix B pages to know how boards should be used, with or without pull resistors …

Valid signals
Only signals found in the **Board/Kart_Board** VHDL block can be used:

Inputs
- **stepperEnd**: where the stepper end switch is connected
- **distancePulse**, *opt*: where the ultrasound ranger PWM pin is connected
- **{halls[x]}**, from 1 to STD_HALL_NUMBER: where the hall sensors are connected
- **dc_A**, **dc_B**: where the DC motor control pins are connected
- **{endSwitches[x]}**, from 1 to STD_ENDSW_NUMBER, *opt*: where digital, 3.3V inputs are connected

Outputs
- **coil1** to **coil4**: where the coils of the stepper motor are connected
- **{leds[x]}** from 1 to STD_LEDS_NUMBER, *opt*: where the digital outputs are connected
- **{servos[x]}** from 1 to STD_SERVOS_NUMBER, *opt*: where the servomotors outputs are connected

6.3.3 Onboard LEDs

A blue LED indicates that the board is powered (top right of the board), while a second found near the USB connector shows in and out transaction over UART.
The red led indicates if the stepper end switch is pressed.
The yellow led toggles on and off when a magnet is rotated in front of the hall sensor 1.
The green led indicates if the smartphone is connected to the BLE module:
- It blinks when connected in the **solution** version
- It stays on when connected in the **student** version

6.4 Programming the board

Professor validation
You are sole responsible to determine when your circuit is ready to be flashed and tested on hardware, but feel free to discard any remaining doubt with your professors before programming your system.
They will also show you once how the deployment toolchain works.

Refer to Libero - Appendix III to use and deploy your design thanks to the Libero IDE.

6.5 USB commands emulation

Power Precautions
Use a laboratory power supply limited from $0.15A$ (no motors) up to $1.2A$ (with motors).

To test the communication directly from a PC, the tool **EBS3 UART Interpreter** is available in the project folder under **CommandInterpreter** either as a Windows executable, Linux executable or Python script.

Hereafter the steps to follow in order to communicate with the Kart:

- Power the circuit off
- Remove the BLE module - Section 4.7 from the motherboard - Section 4.2
- Power the motherboard - Section 4.2 with a regulated DC voltage supply with +12V
- Wire the USB-C present on the daughterboard - Section 4.3 to your PC
  - Two new UART COM ports should be detected
- Download and/or open the Kart Command Interpreter utility (available in the VHDL project in the folder `CommandInterpreter/`)
  - Linux
  - Windows
  - Source code
- In the top menu `Serial` ⇨ `Port`, select the correct COM port
  - Should be the biggest

To test the connection, click the `Read` button. The `Tx` and `Rx` values should change, with `Rx` becoming green (frame correctly received), and a text added to the text area.

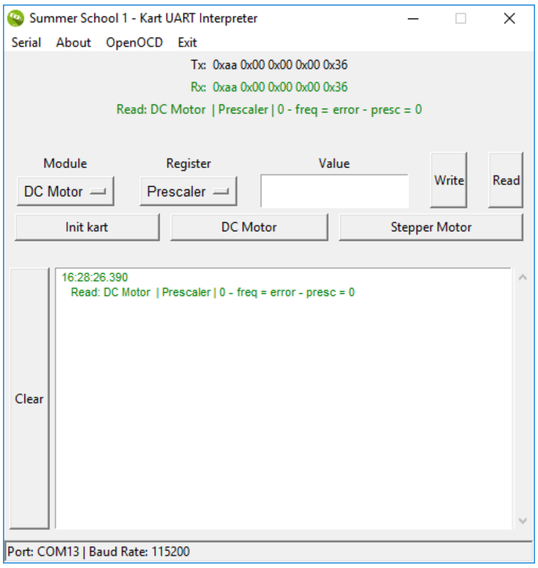


Figure 42 - Kart EBS3 UART Interpreter

6.5.1 Quick Test

The simplest way to test both motors are the following three buttons **Init Kart**, **DC Motor**, and **Stepper Motor**:

Button	Effect	Output
Init Kart	• Set the DC prescaler, stepper prescaler, and execute the restart sequence. **Must be pressed first.** • Answer the 4 prompts following your hardware configuration.	Init Kart DC Prescaler to 31 Stepper Prescaler to 500 BT as connected CReg to reset w. stepper end (0b111111) CReg to normal mode (0b100111) Init done
DC Motor	• Set the speed to full for 2s • Set it to 0 for 2s • Set it to full in reverse for 2s *if hall sensors are mounted, extra messages will tell the speed.*	DC test DC speed tp 15 DC speed tp 0 DC speed t0 −15 DC speed to 0 DC test done
Stepper Motor	• Set the stepper to **`400`** (**`30°`**) • Detect angle reached • Set the stepper to **`0`** • Detect angle reached	Stepper test Stepper tp 400 (30°) 10:46:33.169 Read: Stepper Motor | Actual Angle | 28 ... 10:46:34.804 Read: Stepper Motor | Actual Angle | 364 10:46:34.999 Read: Stepper Motor | Stepper HW | stepper open - position reached Stepper to 0 10:46:36.207 Read: Stepper Motor | Actual Angle | 364 ... Read: Stepper Motor | Actual Angle | 28 10:46:38.029 Read: Stepper Motor | stepper open - position reached Stepper test done

Table 3 - Quick Test buttons

6.5.2 Registers R/W

Each register can be read and/or written by hand following their data description - Section 7.2. For this, select the **`Module`** first, then which **`Register`** to access.

Read

To read, simply click the **`Read`** button. Successful read will be shown in green (CRC is ok) and logged, with extra computed informations. For example the DC prescaler logs the motor frequency.

Write

To write, enter a value in the value box such as:
- Direct integer *(only DC speed may be negative)*
- **`0bxxxx`** binary values
- **`0xxxxx`** hexadecimal values

Then click on the **`Write`** button.

> **Reset**
> Simply power-cycle the FPGA board to reset all registers.

7 Communication

This section defines the serial link protocol [21] used to communicate between the Kart and the BLE Module - Section 4.7 [10] to the PC or Android Smartphone.

Contents

7.1 General Principle 56
7.1.1 Serial Port Configuration 56
7.1.2 Message format 56
7.2 Registers 57
7.3 Initialisation Sequence 59

7.1 General Principle

The system incorporates a BLE module - see Section 4.7. To prevent line congestion, data transmission from the Kart to the User occurs exclusively during specific events or upon the User's request. This strategy helps optimize communication by minimizing unnecessary data transfer and enhancing overall system efficiency.

7.1.1 Serial Port Configuration

The module communicates with the FPGA through UART with the following settings:

Reading State	Data bits	Parity	Stop bits	Handshake	Baudrate
`HIGH`	`8`	`NONE`	`1`	`NONE`	`115'200`

Table 4 - Serial Port Configuration

7.1.2 Message format

SoF (1byte)	Address (1byte)	Data (2bytes)	EoF (1byte)
`0xAA`	`UINT8`	`UINT16 / INT16 / VECTOR16` (MSB First)	`CRC8 / ITU`

Table 5 - Message Format

The address is decomposed as follows: `0bMMWRRRRR`

- `MM` : targeted module
  - `0b00` : DC Motor
  - `0b01` : Stepper Motor
  - `0b10` : Sensors
  - `0b11` : Control Registers
- `W` : defines if the data is saved to (`'1'`) or read from (`'0'`) the FPGA
  - The FPGA will respond to a request with the exact same address when `W = '0'`
  - The FPGA will save incoming data in the targeted register when `W = '1'`
  - The FPGA will send data on predefined events with the `W` bit set to `'0'`
- `RRRRR` : targeted register

7.1.2.1 Frame example

For the BLE module to light LED1 with it changing each 500 ms, the following frame is sent:

SoF (1byte)	Address (1byte)	Data High (1byte)	Data Low (1byte)	EoF (1byte)
`0xAA`	`0b10100001`	`0b10000001`	`0b11110100`	`0x74`

Table 6 - Frame example LED+ 500ms

7.2 Registers

Device	Access	From	To
DC Motor	**Read**	`0x00`	`0x1F`
	Write	`0x20`	`0x3F`
Stepper Motor	**Read**	`0x40`	`0x5F`
	Write	`0x60`	`0x7F`
Sensors	**Read**	`0x80`	`0x9F`
	Write	`0xA0`	`0xBF`
Control	**Read**	`0xC0`	`0xDF`
	Write	`0xE0`	`0xFF`

Table 7 - Memory Map

DC Motor					
Addr	**Name**	**Type**	**Description**	**Direction**	**Event**
`0x00`	Prescaler	UINT16	DC PWM frequency f_{PWM} $\frac{f_{\mathrm{clk}}}{\mathrm{PWM}_{\mathrm{steps}} * \text{ prescaler}} = \frac{10\mathrm{MHz}}{16 * \text{ prescaler}}$	⇨	
`0x01`	Speed	INT5	Desired speed from **`-15`** (**`0xFFF1`**) to **`15`** (**`0x000f`**) negative = backwards	⇨	

Table 8 - DC Motor Registers

Stepper Motor					
Addr	**Name**	**Type**	**Description**	**Direction**	**Event**
`0x00`	Prescaler	UINT16	Stepper switching frquency $f_{\mathrm{step}} = 100 \frac{\mathrm{kHz}}{\mathrm{prescaler}}$	⇨	
`0x01`	Target angle	UINT16	Desired steering angle in motor steps 0 = end switch	⇨	
`0x02`	Actual angle	UINT16	Actual steering angle in motor steps 0 = end switch	⇨	When a delta of at least **`STP_ANGLE_DELTA_DEG`** (2°) from the last registered value happens
`0x03`	Stepper HW	UINT14 + Vector2	Bit[0] : stepper end Bit[1]: position reached Bits[15:2]: actual steering angle	⇨	Sent when stepper end is pressed (rising edge) or position reached (rising edge)

Table 9 - Stepper Motor Registers

Sensors					
Addr	**Name**	**Type**	**Description**	**Direction**	**Event**
`0x00`	LED1	BIT + UINT15	Bit[15]: on - off Bits[14:0]: half-period in ms if 0, led status = bit 15	⇨	
	LEDx	...	...	...	
`0x07`	LED8	BIT + UINT15	Bit[15]: on - off Bits[14:0]: half-period in ms if 0, led status = bit 15	⇨	
`0x00`	SERVO1	UINT16	Servo target pulse duration in clock pulses (10′000 = 1 ms)	⇨	
	SERVOx	...	...	...	
`0x07`	SERVO8	UINT16	Servo target pulse duration in clock pulses (10′000 = 1 ms)	⇨	
`0x08`	USER1	UINT16	User register for custom data	⇨	
	USERx	...	...	...	
`0x0F`	USER8	UINT16	User register for custom data	⇨	
`0x10`	Voltage	UINT16	Battery Voltage $U = \text{register} * 250e^{-6} * 7.8V$	⇨	When a delta of at least `SENS_BATT_DELTA_MV` (50) from the last registered value happens
`0x11`	Current	UINT16	Consumed current $I = \text{register} * \frac{250e^{-6}}{100*5e^{-3}}$	⇨	When a delta of at least `SENS_CURR_DELTA_MA` (50) from the last registered value happens
`0x12`	Range finder	UINT16	Distance to sensor $D = \text{register} * \frac{25.4}{147e^{-6}*(\frac{10M}{10})}$ Register zeroed if less than 152mm (sensor min distance) or greater than 1500mm (arbitary max distance) Event not sent in such case	⇨	When a delta of at least `SENS_RANGEFNDR_MM` (60) from the last registered value happens
`0x13`	End Switches	Vector16	Sensors current values Right justified (sensor 1 is bit 0)	⇨	On any edge change of any sensor
`0x14`	Hall 1	UINT16	Hall pulses count Zeroed on overflow of the register	⇨	Each `SENS_HALL_OLD_SEND_TIMEOUT_MS` (100ms) if value changed from last time
`0x15`	Hall 2	UINT16	Hall pulses count Zeroed on overflow of the register	⇨	Each `SENS_HALL_OLD_SEND_TIMEOUT_MS` (100ms) if value changed from last time

Table 10 - Sensors Registers

> The **LEDx** and **SERVOx** registers are shared. Use either of the register format according to the set output type. See Board Setup - Section 6.3 for more information.

Control					
Addr	**Name**	**Type**	**Description**	**Direction**	**Comment**
`0x00`	Hardware Control	Vector6	Bit[0] **`Forwards`**: when `'0'` the Kart goes backwards, when `'1'` the dc-motor turns forward Bit[1] **`Clockwise`**: when `'1'`, the Kart turns to the right as the stepper coils go from 1 to 4 Bit[2] **`sensorLeft`**: when `'1'` the stepperEnd switch is located on the left otherwise the right Bit[3] **`stepperEnd`**: emulates the end switch contact for the stepper motor Bit[4] **`Restart`**: restart the stepper-Motor module and stop the DC motor when `'1'` Bit[5] **`Stepper end emulation`**: *for tests only*, simulate the **`stepperEnd`** signal	⇨	The end sensor always defines angle 0. Angles are always positive numbers in registers. The stepper motor phases sequences have to be switched according to bits 1 and 2.
`0x01`	BT Status	Vector1	Bit[0] **`btConnected`**: when `'0'`, the smartphone is disconnected	NRF⇨	The register is set by the NRF itself, since it is not possible to foresee the disconnection. If the bluetooth connection is lost, the kart must stop.

Table 11 - Control Registers

The **`Hardware Control[4]`** ⇨ **`restart bit`** is automatically reset back to `'0'` when the **`stepper_end`** input is activated.

7.3 Initialisation Sequence

Multiple registers must be set before the Kart can drive. The following sequence is used by the EBS3 serial interpreter - Section 6.5 :

- Write **`DC Motor`** ⇨ **`Prescaler`** to **`31`** (around 21 kHz PWM frequency)
- Write **`Stepper Motor`** ⇨ **`Prescaler`** to **`400`** (250 Hz coil switching frequency)
- Tell the smartphone it is connected by writing **`Control Registers`** ⇨ **`BT status`** to **`1`**
- Write the value of the kart center position in the **`Stepper Motor`** ⇨ **`Target angle`** register
- Write **`Control Registers`**⇨ **`Hardware Control`** to **`0b10xxx`** to restart the system
  - The stepper should turn until hitting the end switch, except if already on it
- Read **`Stepper Motor`**⇨ **`Stepper HW`** and check the last bit
  - If is `'1'`, it means we are already zeroed
  - If not, wait for an event from this register to tell the reset is complete

The Kart is now ready to function !

Appendices

Contents

A Tools ... 61
  I HDL Designer ... 61
  II Modelsim ... 62
  III Microchip Libero ... 63
    i Overview ... 63
    ii Synthesis ... 64
    iii Flashing ... 67
B PMod boards ... 69
  I Inputs ... 69
    i Ultrasound Ranger ... 69
    ii Buttons / Digital Inputs ... 70
  II Outputs ... 70
    i Digital Signals ... 70
    ii Breadboard ... 70
    iii PMOD-OD2 board ... 71
C Inspiration ... 73

A Tools

I HDL Designer

Mentor HDL designer is the tool for graphical design entry as used during laboratories [4]

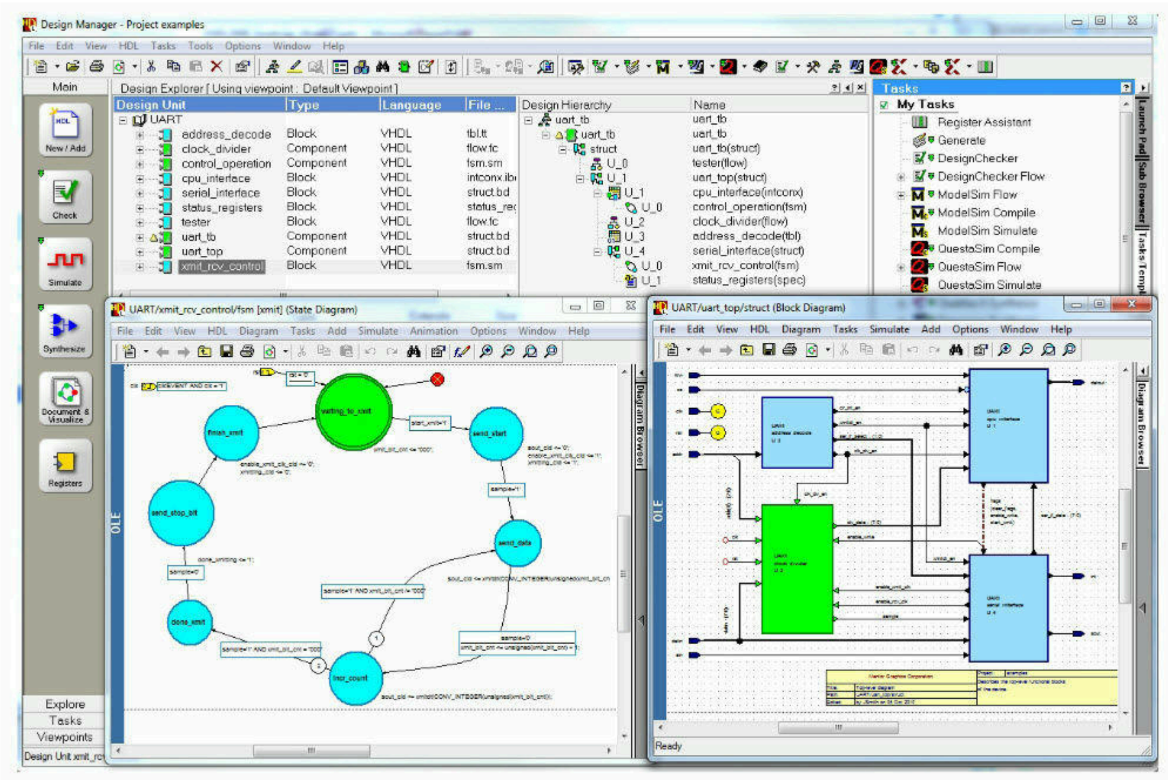
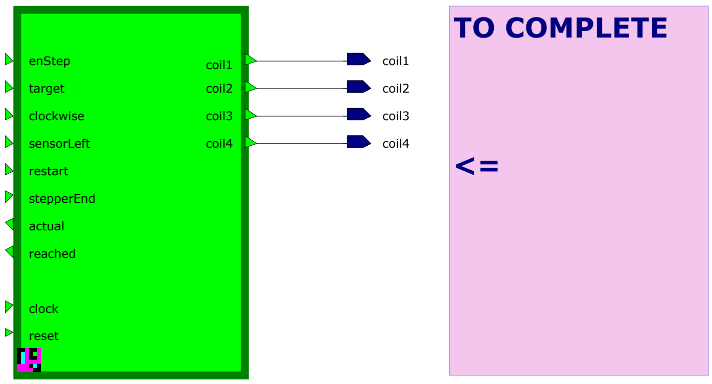


Figure 43 - Mentor HDL Designer

Always run the **`kart.bat`** file to launch the project.

Parts you must complete are pointed by purple text blocks:



A cheatsheet is available online under https://github.com/hei-synd-ss1/ss1-docs/blob/main/control-electronics/EDA_Tools_Cheatsheet.pdf.

II Modelsim

Mentor ModelSim for simulation [5]

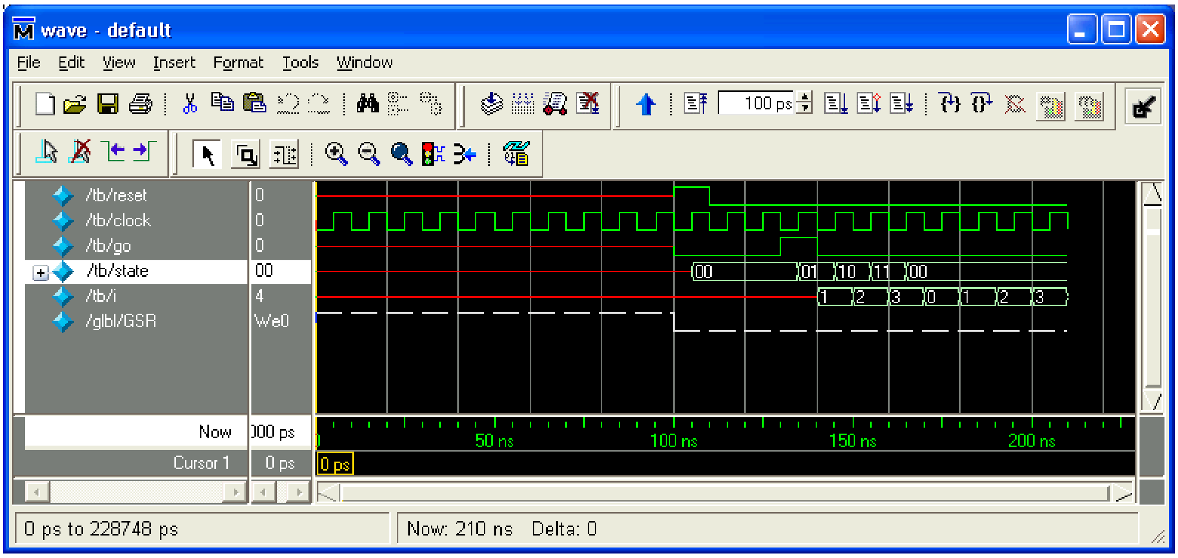


Figure 44 - Mentor Modelsim

The simulations are explained under Section 6. They allow to test on the module level itself or the complete circuit in operation by simulating commands received from the smartphone.

The simulations files are available under the **`Simulation`** folder and contains among others the **`.do`** waveforms files related to all VHDL tester.

III Microchip Libero

Microchip Libero IDE for synthesis and programming[6]

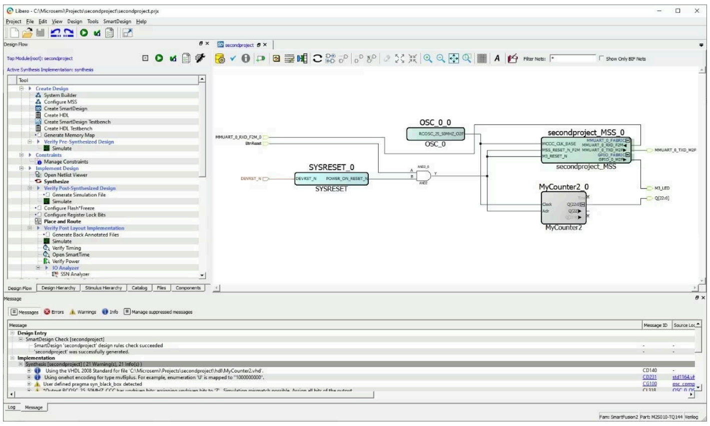


Figure 45 - Microchip Libero

Libero SoC is a design software from Microchip (former Actel) for FPGA .

i Overview

1 Synthesis

Libero SoC can be launched as a standalone or from one of its ***.prjx** project file to complete the synthesis process.

One can launch Libero directly from the Kart project by running the correct task.
- On HDL Designer, open the **`Board`** library
- Highlight the top-level block **`Kart_Board`**
- On the tasks list on the right, first run **`Prepare for synthesis`** then **`Libero Project Navigator`**

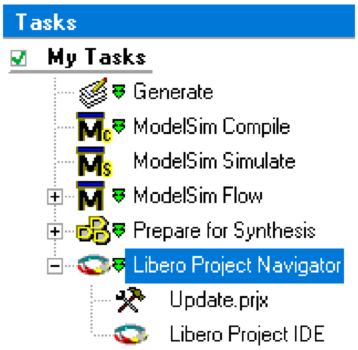


Figure 46 - HDL Designer - Tasks

2 Flash

The FPGA flash is done through the FlashPro software included with Libero through the generated ***.pdb** bitfile thanks to a dedicated programmer such as the FlashPro4.

It can be launched as a standalone or directly from within Libero.

FlashPro can also be used to generate ***.svf** files which can then be used with OpenOCD to flash the FPGA by its USB port.

ii Synthesis

1 Prepare project

After running the HDL task, the project window opens. On the list located on the left, locate the **Compile** ⇨ **Constraints** ⇨ **yourConstraintsFile.pdc** ⇨ right-click ⇨ **Mark as used**:

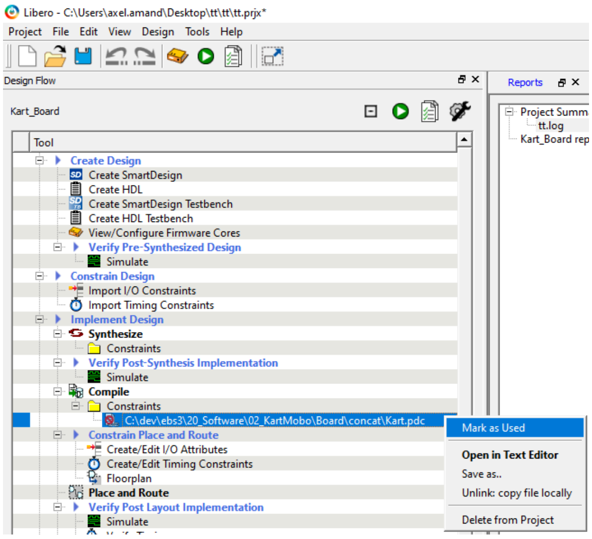


Figure 47 - Libero - Use constraints

Right click on **Synthesize** ⇨ **Open Interactively**. In the newly opened window, on the left, set the correct clock frequency and exit while saving:

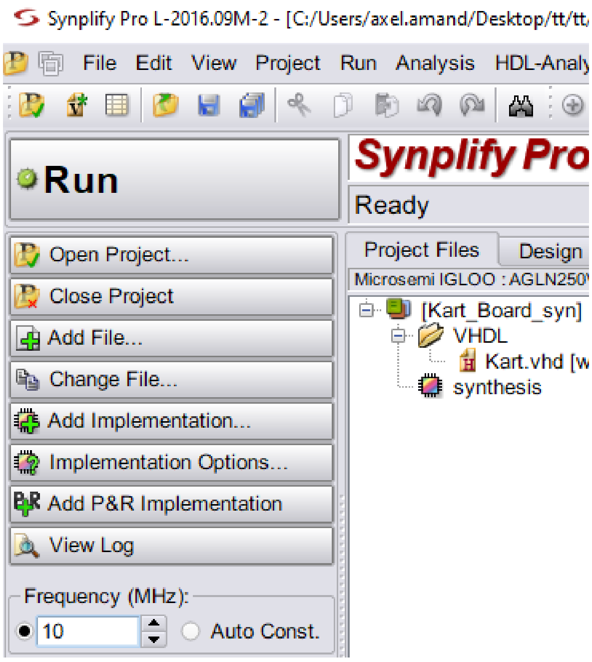


Figure 48 - Libero - Clock setup

This step is required for the program to estimate if the implementation reaches the correct timings.

Right click on **Compile** ⇨ **Open Interactively** ⇨ **I/O attribute editor** and check that pins are correctly linked to the internal signals with correct settings:

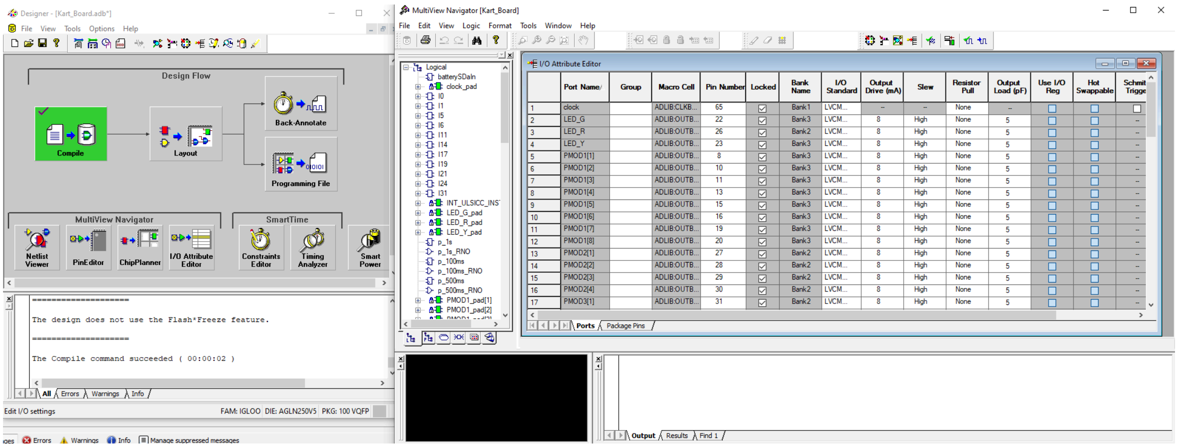


Figure 49 - Libero - Constraints check

2 Synthesize

When the project is ready, the VHDL file must be synthesized, compiled and rooted on the chip.

Either click on **Layout** in the previously opened window (**Compile** ⇨ **Open Interactively** ⇨ **Layout** ⇨ **Ok**) or double click on **Place and Route** in the task list.

The **Message** window on the bottom of Libero can be used to check for errors and warning (both should always be checked). Some parts of the circuit may be pruned, clocks inferred unintentionally, unused signals found …

In addition, reports can be browsed in the **Reports** tab, notably:

- **Synthesize - prjName.srr**:

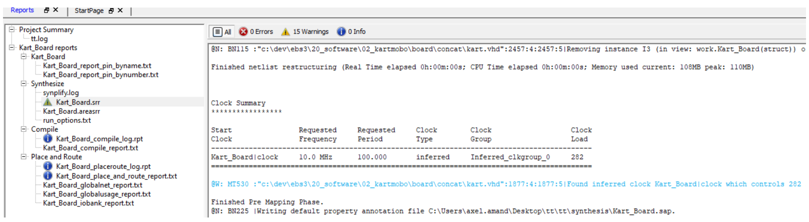


Figure 50 - Libero - Logfile

  which contains informations on:
  - **Clock summary**: should only show the actual clocks - if some inferred ones are found, there is a design problem in the VHDL code
  - **Performance summary**: shows the worst slacks, if the timings are met and the potential fastest clock usable
  - **Core Cells** and **RAM/ROM** usages: how full the FPGA is
- **Compile - prjName_compile_log.rpt** shows the following:
  - **Compile Report**: more detailed view of used cells, BRAM block, I/Os …
  - **I/O Technology**: ensure the standard is set to **LVCMOS33**
  - **I/O Placement**: ensure I/Os are all locked (Placed and UnPlaced ones may indicate errors or that the compilation used a pad to root a signal more easily because said pad was not locked even if not used ⇨ user must ensure the pad is not rooted to anything on the board or lock it beforehand)
- **Place and Route - prjName_globalnet_report.txt** shows global clocks and reset signals found under **Nets Sharing Loads**. In most cases, only a clock and a reset should be shown.

Inferred clocks
Incorrect designs may lead the synthesizer to infer some clocks (e.g. may appear on state machines depending on a signal to trigger each state).

While clock inference is only a warning, the consequences are for part or all of the design to be clocked by a random signal and thus not work at all when flashed.
Stop and correct the problems.

Here is an example of inferred clock:

```
Clock Summary
****************

Start                                    Requested     Requested     Clock        Clock                  Clock
Clock                                    Frequency     Period        Type         Group                  Load
------------------------------------------------------------------------------------------------------------------
Kart_Board|clock                         10.0 MHz      100.000       inferred     Inferred_clkgroup_0    1328
coilControl|stepdelayed_inferred_clock   10.0 MHz      100.000       inferred     Inferred_clkgroup_1    4
==================================================================================================================
```

Figure 51 - Libero - PDC warning

Locked pins
All I/Os referenced in the VHDL file must be linked to a pad of the chip.

If at the end of the compilation the following window appears:

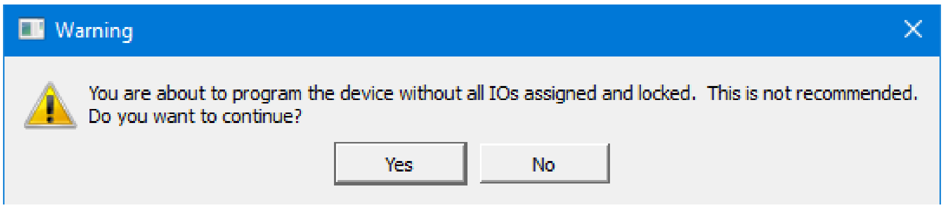


Figure 52 - Libero - PDC warning

the *.pdc constraints file is missing some I/Os.

> **This is a critical error.**
> **STOP and correct the issues before continuing.**
> **Failing to do so may result in destruction of the electronic.**

3 Bitfile

If the compilation is successfull, double click on **Generate Programming Data** to generate a ***.pdb** bitfile.

FlashPro can be launched directly with a right click on **Program Device ⇒ Open Interactively**.

iii Flashing

1 FlashPro

Overview
FlashPro is the official tool supported for the Karts FPGA .

Usage Launch FlashPro directly or from within the Libero project with a right click on **Program Device ⇨ Open Interactively**.

Wire a compatible programmer (such as a FlashPro4) and click on **Refresh/Rescan for Programmers** which should show found devices:

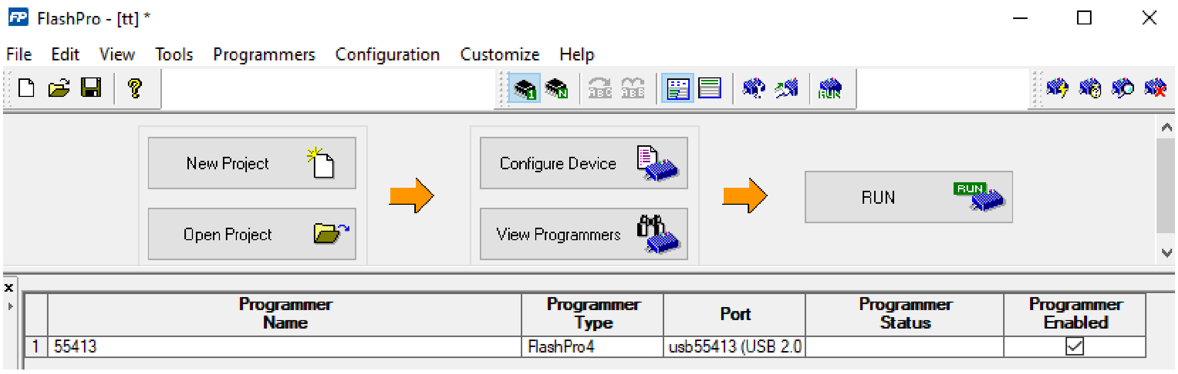


Figure 53 - Flashpro - Programmers

Under **Configuration ⇨ Load Programming File** select the previsouly created ***.pdb** file.

Wire the board, connect the programmer, then click on **PROGRAM**. The advancement is shown in column **Programmer Status**.

> ⚠ The programmer does not supply the daughterboard with power. An external power source (motherboard or USB-C) is needed.

> 🔥 I/Os states normally remain in high-impedance state (with potential pull resistors) while the chip is being programmed. For sensitive applications, disconnect it from the motherboard beforehand.

2 OpenOCD

> ⚠ This method is not currently supported by all your teachers. You have no guarantee to receive any help from them. Ask first before going down this route.

The board can be programmed without any third-parties hardware through OpenOCD thanks to the embedded FT2232HL chip on the daughterboard which offers both an UART and a JTAG interface.

If interested, refer to the **`doc/Kart_AGLN250.pdf`** - section **OpenOCD**.

To not run flash commands by hand, once OpenOCD is installed correctly with its extension files and added to the path, you can run the **EBS3 UART Interpreter** - Section 6.5 and click on **OpenOCD** in the toolbar. Select any of your **.svf** file for the tool to locate all required files. It will then launch the programming and logs are output in the textbox.

B PMod boards

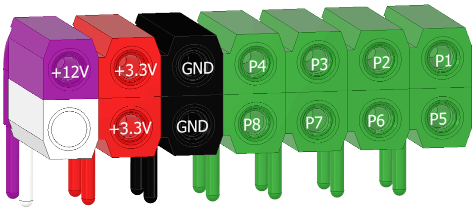


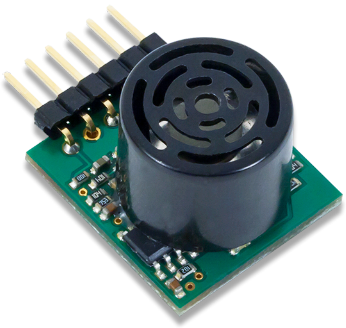
Figure 54 - PMOD Pinning [9], [22]

I Inputs

The sink per pin cannot be higher than 8 mA.
Never ever input a voltage different than +3.3V.
Internal pull-up/down can be enabled at will on the FPGA.

i Ultrasound Ranger

An ultrasound ranger can detect if there is an obstacle at the front or back of the kart. It is based on the PMOD-MAXSONAR board from Digilent [23], and can be plugged into any one-row PMOD connector.



Pin	Descr.
1	*AN (Unused)*
2	*RX (Unused)*
3	*TX (Unused)*
4	PWM
5	GND
6	3.3V

Figure 55 - PMOD MAXSONAR

Use th **`PWM`** pin with no internal pull resistor.
Beware not to wire it on the +12V pin !

See Section 5.4.4 for more informations on the generated pulse.

ii Buttons / Digital Inputs

The **PMOD-CON1** - *Wire terminal connectors* - [24] and the **PMOD-TPH** - *Pin headers* - [25] can both be used to interface digital inputs.

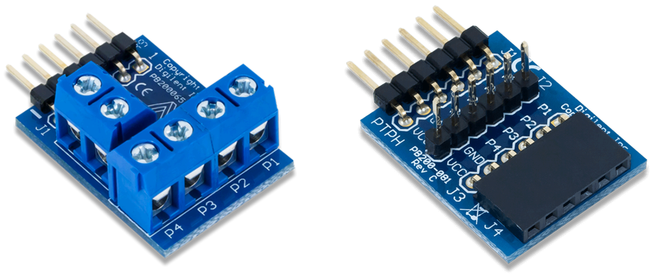


Table 12 - PMOD-CON1 and PMOD-TPH boards

The first connects inputs thanks to screw terminals, the second with pin headers.

> Enable pull resistors on the FPGA side based on your input type (push-pull, open-drain ...).

II Outputs

> The source or sink per pin cannot be higher than 8 mA.
> The outputs must always be in the +3.3V range and there are no voltage feedback protection !

i Digital Signals

Direct drive from the FPGA is only possible for signals attacking high-resistance circuits like MOSFETs gates.
Such elements may be wired directly, or by using the PMOD-CON1 / PMOD-TPH - chapter ii boards.

You must at all time respect the previous warning.

ii Breadboard

The **PMOD-BB** board is intended for tests. It is a small breadboard which allows to plug components in and test a small circuit before designing a custom circuit.

> Use the given breadboard and do not solder on the holes directly.

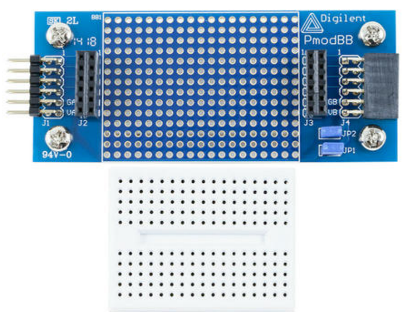
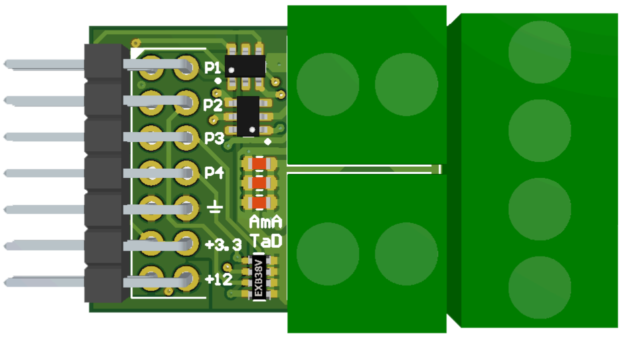


Figure 56 - PMOD Breadboard

iii PMOD-OD2 board

The **PMOD-OD2** ia a custom board allowing to control up to 4 outputs with a selectable voltage level thanks to open-drain outputs.

The double-rows PMOD connector is used to avoid confusion when plugging the board thanks to its keying pin. The lower row is unused.

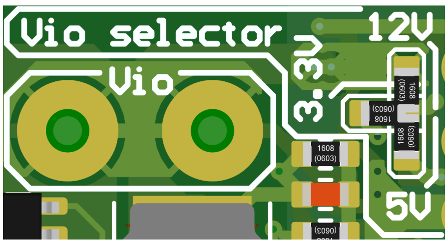


		Pins	
Descr.		1	2
X	P1	3	4
X	P2	5	6
X	P3	7	8
X	P4	9	10
X	GND	11	12
X	3.3V	13	14
X	12V		

Table 13 - PMOD-OD2 board and pining

Output Voltage

The voltage is selected by soldering one of the three following resistors on the backside of the board to switch between **+3.3V**, **+5V** or **+12V**:



Only one resistor must be soldered at any given time.

Table 14 - Vio selection

Terminal

There are three screw terminals:

- A double terminal for Vio
- A double terminal for GND
- A quad terminal for the four outputs

They are all indicated on the back of the board.

Always use the terminals of the same board for **Vio**, **Px** and **GND**. Do not root either of those from another source to avoid destroying the protections in place.

The board uses negative logic.
When setting up your constraint file, use the signals `{servos[x]}` to use the PMOD-OD2 board. The signal is already inverted FPGA-side.

Wiring loads

Since outputs are open-drains, two wiring methods can be used:

- **Leds, relays, small DC motors …**:
  - If needed (e.g. LEDs), put a resistor in series with the load.
    - For inductive loads, the circuit is already protected with flyback diodes.
  - Set the Vio jumper to the desired voltage
  - Wire the positive side of the load on the Vio terminal (J1) and the negative one to Px (J3)
  - Ensure no resistor from R2 to R5 is soldered.
  - When Cx is `'0'`, the output is left floating and there is no conduction.
  - When Cx is `'1'`, the transistor is driven and the output conducts.
- **Pseudo push-pull, servos control …**:
  - Basically, it is only possible to either close the transistor (output a '0') or left it open (output a 'Z'). Some loads require a well-defined '0' or '1'.
  - Add a resistor either:
    - Between the Vio terminal (J1) and the Px terminal (J3) - *external resistor*
    - Solder one on the R2 to R5 pads - *soldered resistor*
  - Set the Vio jumper to the desired voltage.
  - When Cx is `'0'`, the output is `'1'`.
  - When Cx is `'1'`, the output is `'0'`.

> **DO NOT TRY TO DRIVE CURRENT THROUGH THE Px PIN.**
> All the current would flow through the resistor, creating a voltage drop and power loss, leading to a fire hazard.

Both methods are shown in Figure 57 :

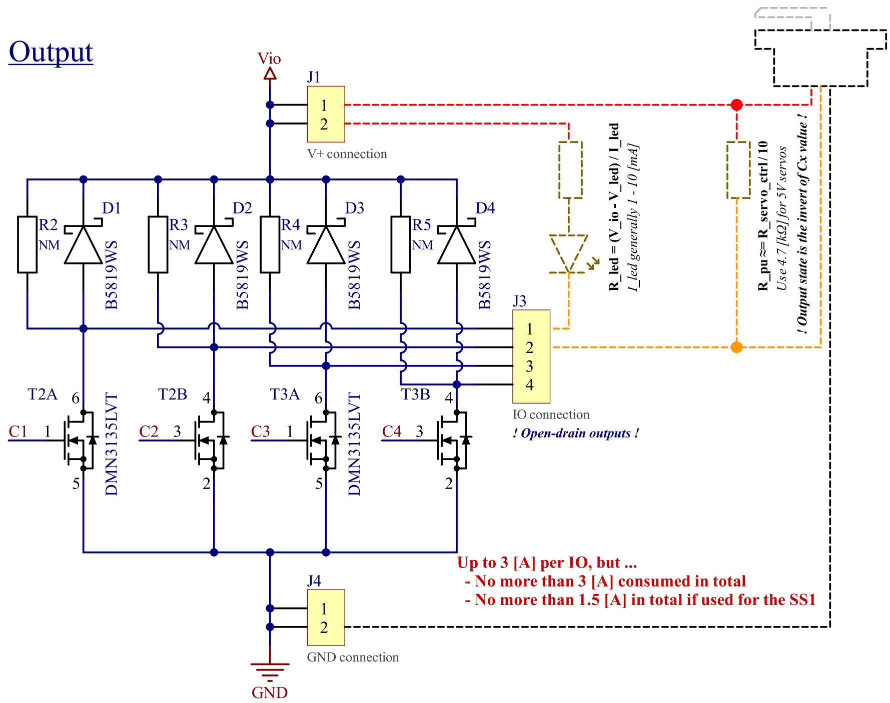


Figure 57 - Driving loads with PMOD-OD2

C Inspiration

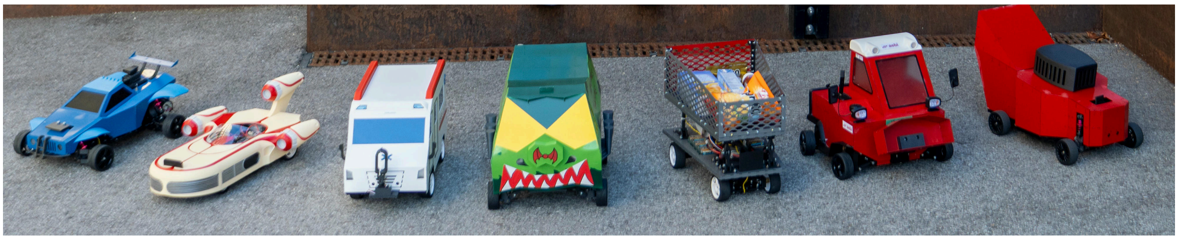


Figure 58 - Summerschool 2024

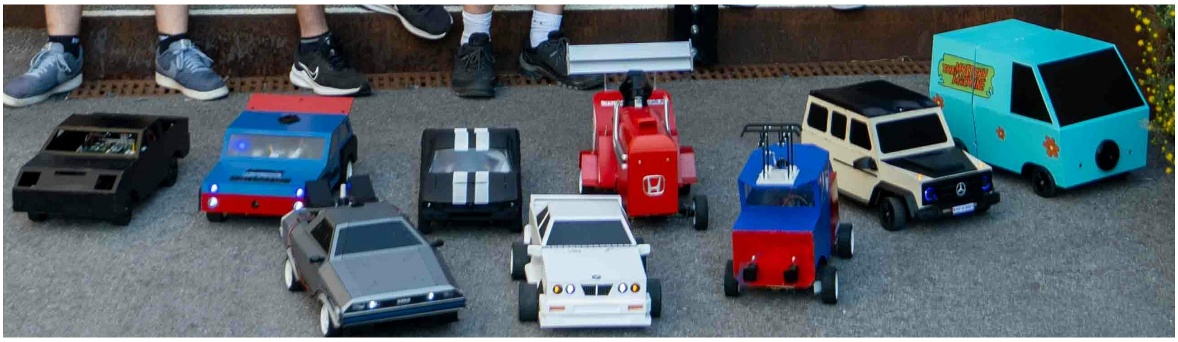


Figure 59 - Summerschool 2023

Figure 60 - Summerschool 2022

Figure 61 - Summerschool 2020

Figure 62 - Summerschool 2018

Figure 63 - Summerschool 2017

Figure 64 - Summerschool 2015

Figure 65 - Summerschool 2013

Figure 66 - Summerschool 2012

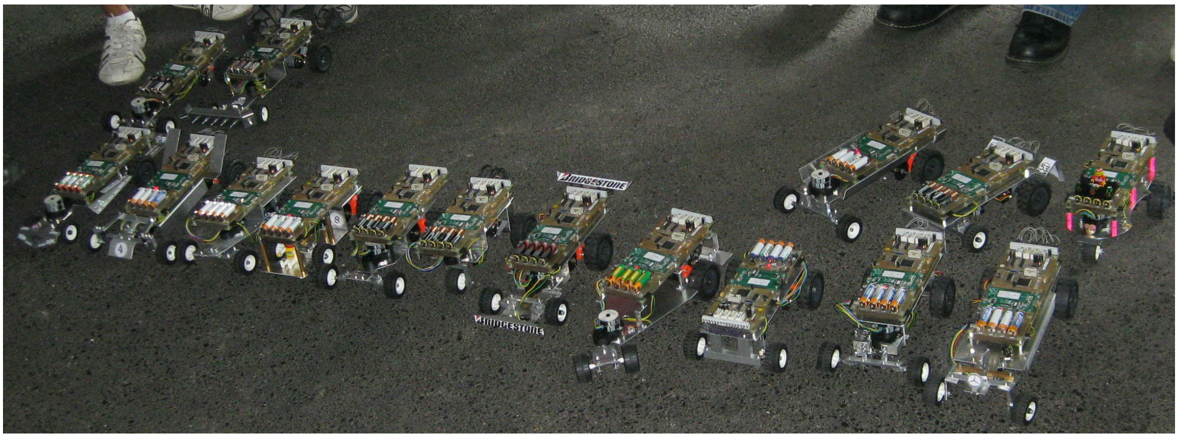


Figure 67 - Summerschool 2009

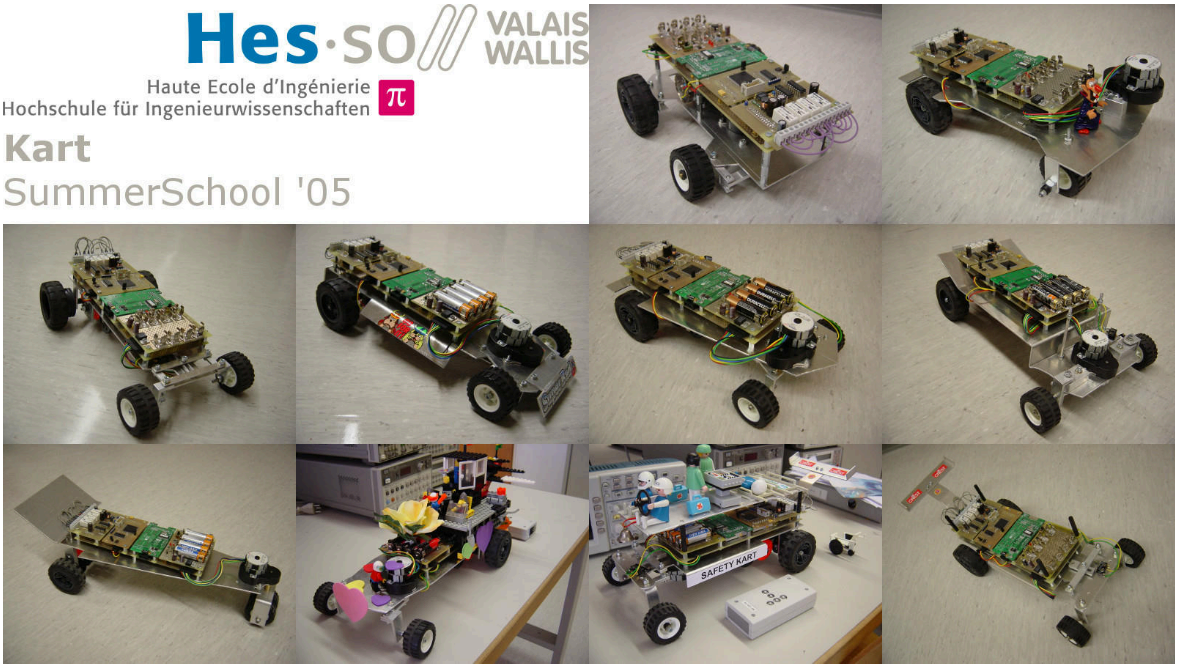


Figure 68 - Summerschool 2005

Figure 69 - Summerschool 2004

Glossary

DC **– Direct Current** 2, 7, 23, 26

FPGA **– Field Programmable Gate Array** 7, 11, 41, 56, 63, 65, 67

GUI **– Graphical User Interface** 7

PWM **– Pulse Width Modulation** 7, 23

Bibliography

[1] S. ye industrial, "DC Motor Modelcraft RB350018-2A723R 12V Technical Datasheet." 2006.

[2] A. Amand and S. Zahno, "FPGA-EBS3 Electornic Technical Documentation." 2022.

[3] "Hei-Synd-SS1/ss1-vhdl: HEVS SS1 Kart Summerschool Project Based on the EBS3 Igloo Board.." Accessed: Sep. 01, 2023. [Online]. Available: https://github.com/hei-synd-ss1/ss1-vhdl

[4] "HDL Designer Visualizing Complex RTL Designs." Accessed: Sep. 01, 2023. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/hdl-designer/

[5] "ModelSim HDL Simulator." Accessed: Sep. 01, 2023. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/modelsim/

[6] "Libero® IDE | Microchip Technology." Accessed: Sep. 01, 2023. [Online]. Available: https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/fpga/libero-ide

[7] "IGLOO Nano Low Power Flash FPGAs with Flash*Freeze Technology Datasheet." 2019.

[8] M. Technologies, "Microchip MCP3426/7/8, 16-Bit, Multi-Channel ΔΣ Analog-to-Digital Converter with I2CTM Interface and On-Board Reference." 2009.

[9] D. Inc, "Digilent Pmod Inteface Specification v1.2.0." 2017.

[10] "nRF52840 Dongle." Accessed: Sep. 01, 2023. [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/nRF52840-Dongle

[11] F. INC, "FTDI FT2232H Dual High Speed USB to Multiporpose UART/FIFO IC." 2019.

[12] A. Amand and S. Zahno, "Kart SODIMM-200 Pinning." 2022.

[13] "Pulse-Width Modulation." Aug. 29, 2023. Accessed: Sep. 15, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Pulse-width_modulation&oldid=1172794179

[14] "H-Bridge." Aug. 07, 2023. Accessed: Sep. 15, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=H-bridge&oldid=1169162530

[15] N. Inc, "Nanotec SP3575M0906-A Stepper Motor Datasheet." 2006.

[16] D. Inc, "Digilent Pmod DC Stepper Schematic." 2022.

[17] Omron, "Datasheet Omron Subminiature Basic Switch Offers High Reliability and Security." 2005.

[18] Honeywell, "Honeywell SS311PT/SS411P Bipolar Hall-effect Digital Position Sensors with Build-in Pull-up Resistor." 2009.

[19] "Schmitt Trigger." Aug. 28, 2023. Accessed: Sep. 15, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Schmitt_trigger&oldid=1172685652

[20] N. Semiconductor, "Datasheet Nordic NRF52840 Dongle." 2020.

[21] "Universal Asynchronous Receiver-Transmitter." Aug. 28, 2023. Accessed: Sep. 01, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver-transmitter&oldid=1172648211

[22] "Pmod™ - Digilent Reference." Accessed: Sep. 01, 2023. [Online]. Available: https://digilent.com/reference/pmod/start

[23] D. Inc, "Pmod MAXSONAR Reference Manual." 2014.

[24] D. Inc, "Digilent Pmod CON3 Schematic." 2015.

[25] D. Inc, "Pmod CON3 Reference Manual." 2016.