LIÈGE université
School of Engineering

Université de Liège
Faculté des Sciences Appliquées

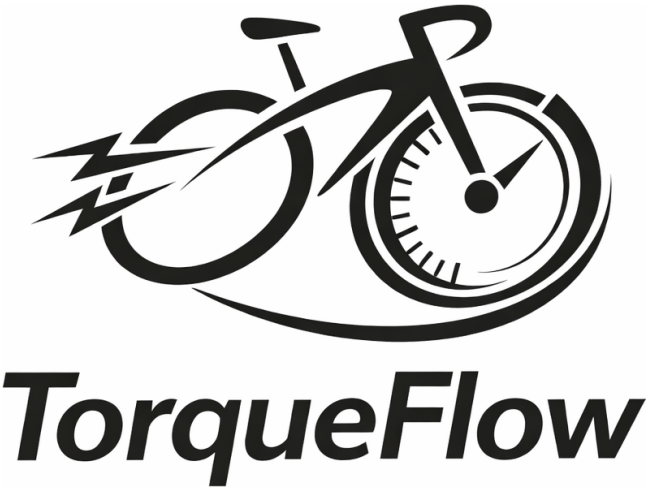
APRI0007

Major Project in Electrical Engineering

Final Report

Automatic Bicycle Gear Shifting System

Academic year 2025–2026



Authors
BANDIN Thomas (s2500726)
BODEUX Lucien (s202552)
RUTTEN Maxence (s224381)
SCHUMACHER Maxim (s224693)

Contents

1 Introduction **1**

2 Project Development and Management **1**

2.1 Project Charter . . . 1

2.2 Project Execution Plan . . . 2

2.3 Risk Assessment and Mitigation . . . 2

3 Hardware Implementation **3**

3.1 Servo Motor . . . 3

3.1.1 Servomotor support . . . 4

3.1.2 Validation . . . 5

3.1.3 Servomotor feedback . . . 6

3.2 Bafang CAN Sensor (Torque and Pedalling Cadence) . . . 6

3.2.1 Brief overview of the CAN protocol . . . 6

3.2.2 Implementation . . . 7

3.2.3 Validation . . . 8

3.3 Hall Sensor (Rear Wheel Speed) . . . 8

3.3.1 Implementation . . . 9

3.3.2 Validation . . . 9

3.4 LCD screen . . . 10

3.4.1 Implementation . . . 10

3.4.2 Validation . . . 10

3.5 Buttons . . . 11

3.5.1 Implementation . . . 11

3.6 PCB and power supply . . . 11

3.6.1 Power converters . . . 14

3.6.2 Power Supply . . . 14

3.7 Overall package . . . 14

4 Control Strategy **15**

4.1 Robustness mechanisms and protections . . . 15

4.2 General control algorithm . . . 16

4.3 Practical implementation parameters . . . 17

5 Software Implementation **18**

5.1 Main software modules . . . 19

5.1.1 `can_torque.inc` . . . 19

5.1.2 `hall.inc` . . . 20

5.1.3 debounce.inc . . . 21
5.1.4 servo_hw.inc . . . 22
5.1.5 Control.inc . . . 22
5.1.6 wait.inc . . . 23
5.1.7 lcd_direct.inc . . . 23
5.1.8 pinconfig.inc . . . 24
5.2 Main loop organisation and timing strategy . . . 25
5.2.1 main.asm . . . 26

6 Lessons Learned and Possible Improvements **27**

7 Annexes **28**

1 Introduction

This project was carried out in the scope of the major project in Electrical Engineering. Its objective is to develop a working embedded system able to shift the gears of a bicycle automatically, while still allowing the rider to interact with the drivetrain when needed.

The prototype is built around a real bicycle setup. It measures crank torque, pedalling cadence and rear-wheel speed, and uses these quantities to evaluate the current riding conditions. Based on this information, the microcontroller decides whether a gear change is required. The mechanical action is then performed by a servomotor, which pulls or releases the derailleur cable in order to change the engaged gear.

The system supports two main operating modes: an automatic mode and a manual mode. In automatic mode, gear shifts are selected by the controller according to the measured riding conditions and the cadence target set by the user. In manual mode, the rider can directly request gear changes using push-buttons.

The user interacts with the prototype through a simple interface composed of push-buttons, status LEDs and an LCD screen. The buttons are used to switch between operating modes, request manual shifts, change the displayed page and adjust the cadence target used by the automatic controller. The LCD gives direct feedback on the state of the bicycle and of the controller by displaying the main useful quantities, such as torque, cadence, power, speed and target-setting information. The LEDs provide an additional visual indication of some operating states.

The development followed an incremental approach. The main building blocks were first tested separately, including the sensors, the servomotor and the user interface, before being integrated into a complete firmware architecture. Several design choices were refined during this process in order to obtain a solution that was robust enough for the real mechanical setup.

This report will first begin by describing the different components of the system on their physical level. Then the control strategy will be displayed and finally, the software will be depicted.

2 Project Development and Management

After the general objective of the prototype had been defined, the project was structured around three elements: the project charter, the project execution plan, and the risk assessment. These tools were not only used to organize the work, but also to clarify which technical choices were the most relevant for a real bicycle setup. Although the project was initially intended as a test-bench validation, the final prototype was integrated on a real bicycle and can be used under practical riding conditions.

2.1 Project Charter

The project charter defined the functional target of the system: an embedded prototype able to shift bicycle gears automatically, while keeping manual control available to the rider. The initial concept was based on rider power and road slope, since both quantities are related to the effort required during cycling.

During development, cadence was selected as the main variable for gear selection. This choice comes from the role of a bicycle transmission itself. Gears allow the rider to produce comparable power levels with different torque–cadence combinations. For example, the same power can correspond to a hard gear with low cadence and high torque, or to an easier gear with higher cadence and lower torque. These two situations do not imply the same shift direction, although the power value is similar. Power is therefore useful to estimate the effort level, but it is not sufficient to decide whether the system should shift up or down. In the final architecture, cadence is used as the main control variable, while power is used to adapt the cadence target.

Road slope was considered in the initial concept, but it was not retained as an explicit input in the final controller. Its effect is already visible through the rider's response: on an uphill section, an unsuitable gear naturally leads to lower cadence and higher torque. Since both quantities are already measured, adding a slope sensor would mostly provide redundant information. It would also introduce additional calibration issues and sensitivity to bicycle orientation and vibrations. The final controller therefore focuses on the quantities that directly describe the interaction between the rider and the drivetrain.

2.2 Project Execution Plan

The project execution plan was built around the progressive validation of the critical subsystems. The actuation mechanism, the torque and cadence sensing, the rear-wheel speed measurement, the buttons and the LCD feedback were first tested independently before being integrated into the complete firmware. This approach made debugging more controlled and reduced the risk of combining several unvalidated blocks at once.

The sensing architecture was refined during this validation phase. The Bafang sensor was initially selected for torque measurement, while pedalling cadence was expected to be measured separately with a Hall sensor. However, the CAN frame analysis showed that the Bafang sensor already provided cadence information. A separate cadence sensor was therefore not necessary, and the Hall sensor could be kept only for rear-wheel speed measurement.

In the final architecture, the Hall sensor was therefore assigned to rear-wheel speed measurement. This information is not used to choose the shift direction. It is instead used as a protection input, allowing the firmware to inhibit gear changes when the bicycle is stopped or nearly stopped.

On the actuation side, the servomotor was selected because the derailleur cable requires controlled positioning under significant mechanical load. Each gear corresponds to a stable cable position that must be reached and held reliably. A small stepper motor or standard DC motor would not have been suitable for this constraint, since the actuator must continuously resist the cable tension. Reaching the required torque with another motor architecture would have required a larger motor, higher current consumption or an additional reduction mechanism. The servomotor was therefore the most appropriate choice, as it directly combines sufficient torque, internal position feedback and holding capability.

2.3 Risk Assessment and Mitigation

The risk assessment identified the technical points that could compromise the prototype if they were not validated early. The first major risk was mechanical: the actuator had to pull and hold the derailleur cable reliably. This was mitigated through preliminary torque estimation and direct testing on the real drivetrain.

The second major risk was related to the Bafang sensor. Since the available documentation did not clearly describe the CAN frames, the useful data fields had to be identified experimentally. The decoding of torque and cadence was therefore validated before being integrated into the final PIC firmware.

A third risk concerned the behaviour of the automatic control itself. A gear-shifting algorithm must not react too quickly to noisy measurements or short transient conditions, otherwise it may trigger repeated or unnecessary shifts. To mitigate this risk, several control mechanisms were implemented to limit erratic behaviour and make the shifting decisions more robust on the real setup.

The final architecture is thus the refined implementation of the original objective. The transition from a test bench to a real bicycle made the validation more demanding, but also more meaningful. Cadence was selected as the main gear-selection variable, power was kept as an effort-dependent adjustment, rear-wheel speed was assigned a protection role, and the servomotor was retained as the actuator best suited to the derailleur mechanism. Together, these choices form a coherent system for automatic gear

shifting on the real bicycle prototype.

3 Hardware Implementation

This section describes the main building blocks of the system. For each block, the early calculus are presented. Then, the working principle and how they were useful in building the project. Implementation details and practical issues encountered during development are covered in the sections that follow.

3.1 Servo Motor

A servomotor is used to actuate the derailleur by pulling or releasing its cable, allowing the bike to change gears. Since the derailleur features a spring that put under tension the cable, that means we don't need rigid linkage between the actuator and the gear selecting mechanism. This allowed to keep the flexible cable, and with the help of sheaths, allowing to put the servomotor wherever we wanted. The rear rack was quickly selected for the convenience of access and amount of space it features. From this decision, it was chosen that if possible, component would go in that same place, resting firmly on the custom made board.

One of the main questions was whether the actuator would be strong enough to pull the cable and hold it under load. Preliminary measurements showed that the required torque is within the capabilities of the selected servomotor : early measurements with a torque wrench have shown that $1[Nm]$ was enough to actuate the derailleur and the servomotor could provide $2.5[Nm]$ at full power.

Mechanically, the derailleur cable is kept under tension by a return spring. The servo pulls the cable to shift down and releases it to shift up. A custom wheel was designed and 3D printed to translate the servo rotation into cable displacement, mapping the full derailleur travel to a fixed angular range or around 90°. This approach also reduced jitter compared to a direct linear traction system, as the wheel acts as a pulley and smooths out irregularities in the cable path. The mechanical design is shown here 2

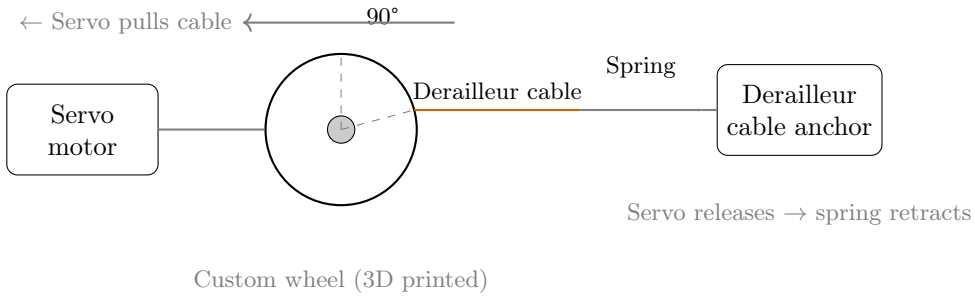


Figure 1: Schematic of the derailleur cable mechanism.

the tension provided also allow to design a wheel where the cable in the wheel has to be locked in only one direction, which allow for simpler design, no adhesive or easier replacement, which came really helpful during the gear shifting fine-tuning.

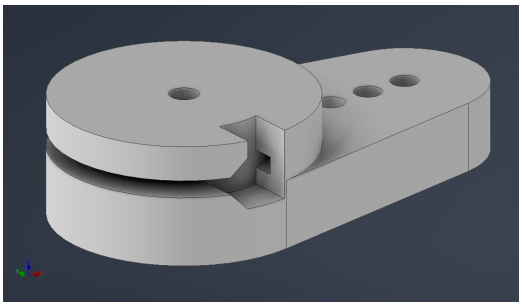


Figure 2: Custom servomotor wheel. The hole allow the cable to pass while blocking the larger head of the cable, providing a one-way locking mechanism

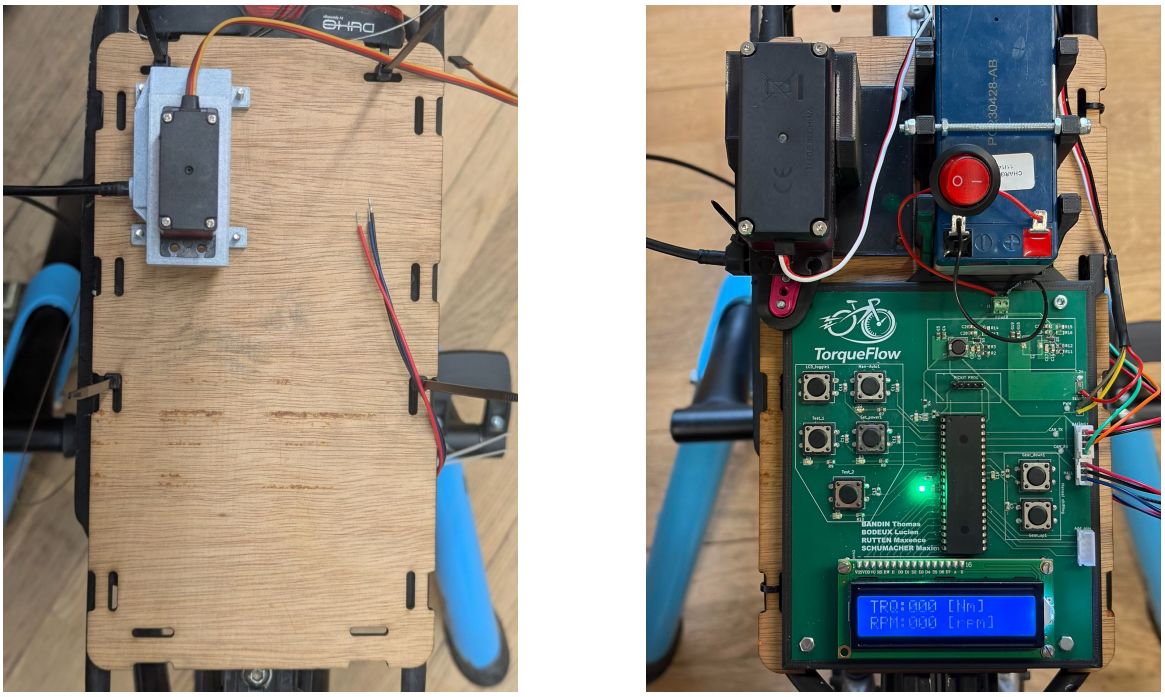
The servo requires only two inputs: a supply voltage and a PWM signal. Most servos respond to duty cycles mapped across a frequency range of roughly 180 Hz to 1.1 kHz, referred to throughout the code as "duty_cycle". The signal is generated using a Timer interrupt preloaded with a prescaler value, which modulates the output frequency directly. An alternative would have been to use the dedicated PWM peripheral with its Capture and Compare registers, which offers finer regulation. The chosen approach trades that precision for flexibility: since the signal is generated through a general-purpose timer, it can be routed to any pin the timer can reach, without being tied to a specific PWM-capable output which in our cases are used elsewhere (i.e : CAN communication and early I2C communication). This implementation is better described here 5.1.4

3.1.1 Servomotor support

The support fulfils 2 main roles: absorbs the cable tension forces and keep the servomotor-wheel assembly steady on the board. Since the direction of the cable change between the servomotor support and the derailleur, the sheath absorbs the tension forces of the cable, meaning the support has to feature and fixation for the sheath strong enough.

For this we designed a first version of the support that would encapsulate the wheel, in order to align it with the sheath support. The servomotor was placed upside down in order for the wheel axis to come trough the board below to it clamp of both end and relieve stress on the servomotor head. It was then 3D printed at the Fab52 along with the wheel.

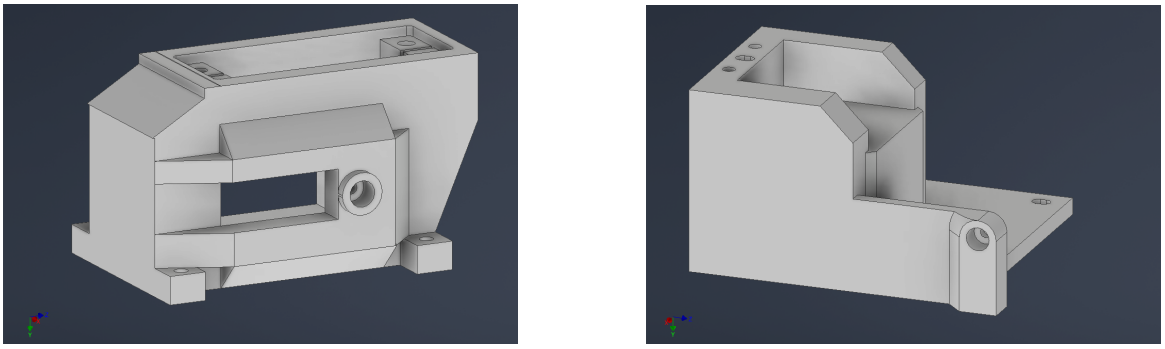
A few days before the deadline, our servomotor burned up, forcing us to find a replacement. The only one we managed to find in time was no longer a 20 kg.cm servo but a 150 kg.cm, which is substantially larger.



(a) First small servomotor (20kg.cm) (b) Second bigger servomotor (150kg.cm)

Figure 3: Size comparison of the 2 servomotor, board for scale

This meant that the support had to be redesigned, since the second servomotor could not fit into the first one even with design modifications.A new version of the support was therefore designed, together with an adapted wheel, and the rapid manufacturing capability of 3D printing allowed both parts to be ready just in time for Friday, May 15, so that we could test just in time the new servomotor and adapt the system, the firmware and the power supply accordingly, the latter being changed from 7 V to 12 V.



(a) First design (b) Second design

Figure 4: Servomotor designs for the first and second servomotor

3.1.2 Validation

A two-button interface was implemented to manually trigger up-shifts and down-shifts during validation. Tests confirmed that the servo changes gears reliably across the full range, with a response time under one second and no noticeable mechanical difficulty.

A test bench video was recorded with the initial 20 kg.cm servomotor in order to confirm that this first actuator choice was already able to shift the drivetrain under real load. At that stage, a discrete servo rotation of about 27° was used as a first practical step. This was sufficient to trigger gear changes, although it did not yet provide an exact one-to-one mapping for all gears and some transitions could

still skip intermediate positions. A more precise gear-to-position mapping was introduced later in the project. The replacement 150 kg.cm servomotor naturally handled these gear changes with even greater ease.

3.1.3 Servomotor feedback

A conventional RC servo contains three internal components: a DC motor,a reduction gearbox, and a potentiometer mechanically coupled to the output shaft. The potentiometer forms the feedback element: as the shaft rotates, its wiper moves and the resulting voltage directly encodes the current angular position. An internal control circuit continuously compares this measured position against the target position derived from the incoming PWM signal. When a difference exists, the motor is driven until the two match, at which point the motor stops. This closed-loop behaviour is entirely self-contained inside the servo housing; the microcontroller only needs to maintain the PWM signal at the desired duty cycle.

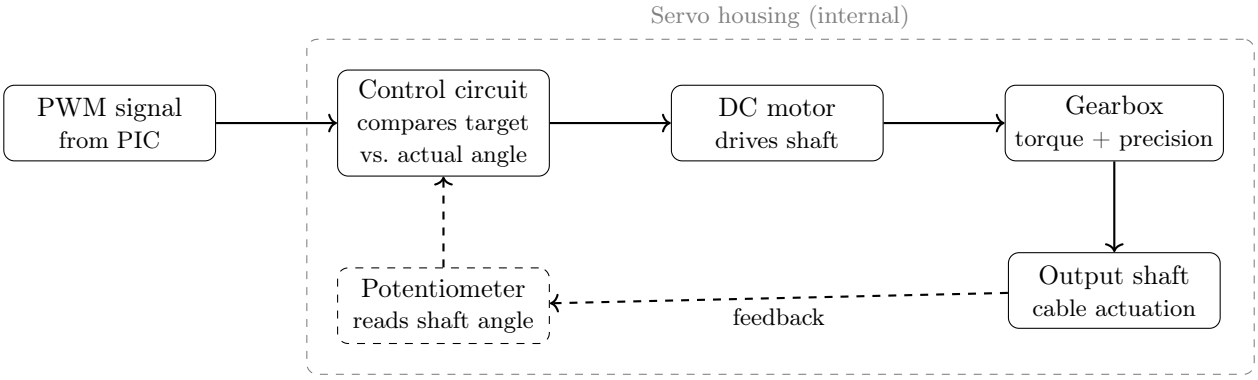


Figure 5: Internal feedback loop of a conventional RC servo.

3.2 Bafang CAN Sensor (Torque and Pedalling Cadence)

This sensor is one of the most important components of the project, as it provides measurements of the torque applied to the crankshaft and the pedalling cadence, which are the main variables used for gear shifting control.

It is part of a commercial kit designed to convert standard bicycles into electric bikes. However, the manufacturer provides only limited documentation, and the datasheet does not clearly describe the content of the CAN messages. This meant that going forward with this sensor would mean having to reverse-engineer the communication frames and adapting the bike.

3.2.1 Brief overview of the CAN protocol

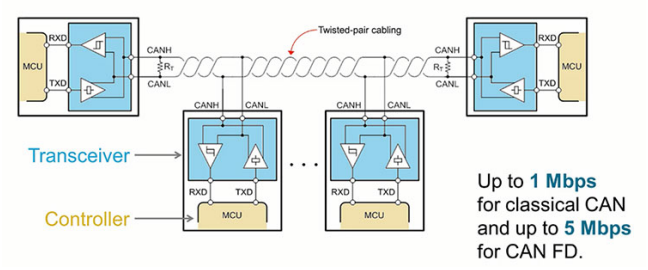


Figure 6: CAN bus protocol. Source: Digikey

CAN bus operates over two differential lines, CAN-H and CAN-L, and requires a termination impedance of 60 Ω or 120 Ω at each end of the bus for proper signal integrity. This differential pair cannot be read directly by the PIC, which expects single-ended logic levels on its CAN RX and CAN TX pins. This requirement was not anticipated during the initial design phase: the chinese datasheet consulted described the sensor interface in terms of CAN RX and CAN TX signals, whereas the physical connector on the sensor exposes only CAN-H and CAN-L. A dedicated transceiver (namely the DollaTek TJA1050) was therefore required to bridge the two interfaces, converting the differential bus signals into the single-ended logic levels expected by the PIC peripheral. It supports a multi-node topology with one master and multiple slaves. However, in this application, a single slave is used. Frames are 8 bytes long, each byte serving a distinct role: padding which help eliminate any corrupted frames, measured data, or a rolling counter for temporal tracking.

Table 1: CAN bus frame structure and example (rest state: 0.0 Nm, 0 RPM)

Byte(s)	Field	Example	Description
1–3	Padding	`0x00 0x00 0x00`	Reserved / unused
4–5	Torque	`0xEE 0x02`	16-bit little-endian (LSB first); τ [Nm] = $(x_{\text{raw}} - 750)/30.625$, where x_{raw} is also the sensor voltage in mV. Range: 0–80 Nm (750–3200 mV).
6	Cadence	`0x00`	Pedalling cadence in RPM
7	Counter	`0x6A`	Rolling frame counter
8	Padding	`0xFF`	Reserved / unused

Sensor Data Frame (Bafang PA261) Example raw frame at rest:
`0x00 0x00 0x00 0xEE 0x02 0x00 0x6A 0xFF`

3.2.2 Implementation

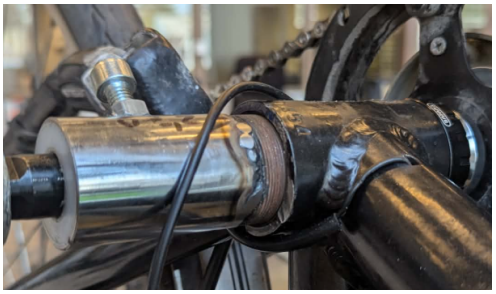
Due to the lack of documentation (in datasheets and on the website), it was necessary to reverse engineer the CAN communication used in the sensor in order to identify which bits correspond to torque and cadence. In brief, we needed to create a "sniffer" that was able to read the bus clearly.

To do this, we first ensured that the CAN communication was correctly initialized, so that the received messages could be reliably linked to the sensor. We then used publicly available resources, in particular the Open Bafang project, and compared their decoding with our own observations. This process, combined with trial and error, allowed us to identify the relevant data fields.

The bike had to be substantially modified to accommodate this sensor, since the entire shaft and bearings had to be replaced. When selecting the Bafang sensor, we could confirm the diameter on the assembly was compatible on the bike, but was wider and therefore modification were to be made on the shaft hub. First a hole had to be drilled to pass the cable through. Then a metal tube had to be installed to lengthen to shaft hub. At first we believed this would make the bike less pleasant, as the pedals were not properly centred anymore, but after testing we were proved otherwise.



(a) Chain Ring teeth connection



(b) Contraption to lock the sensor into the bike

Figure 7: Mechanical aspect of the torque sensor

3.2.3 Validation

During testing, cadence values were immediately coherent, but torque readings were initially erratic. The cause was traced to the chainring originally mounted on the bicycle, which lacked the mechanical tooth interface required by the Bafang sensor to work properly. Replacing the chainring with the OEM Bafang part resolved part of the issue. However, the torque value still seemed to be overestimated. The variations in effort nevertheless remained clearly visible, but shifted upward. At this stage, it is not clear whether this comes from the mounting conditions or from the sensor accuracy itself. The signal also remained highly variable, which is normal since torque depends on the pedal position during the rotation. In practice, averaging was therefore needed to obtain values stable enough for the controller.

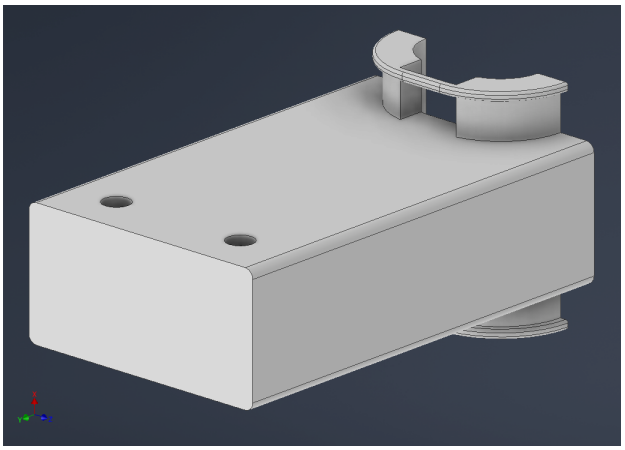
The decoding routine was first prototyped in C++ on an Arduino to validate the communication protocol and sensor behaviour before porting to assembly. The main difficulty was understanding the timing requirements of the CAN peripheral and implementing the conversion from raw voltage to Newton-metres in integer arithmetic. The resulting implementation is presented as pseudo-code in Section 5.1.1.

3.3 Hall Sensor (Rear Wheel Speed)

The Hall effect sensor detects the passage of magnets fixed to the rear wheel, generating one digital pulse per magnet pass. Counting these pulses over a fixed time window gives a direct estimate of wheel rotational speed, with pulse frequency proportional to velocity. The total count also depends on the number of magnets mounted on the wheel, which is nine in this implementation.

The sensor was originally intended to estimate the ratio for the current gear by comparing wheel speed against pedalling cadence. This turned out to be unnecessary for the implemented control strategy, which does not require precise gear knowledge. The sensor instead serves two safety roles: it blocks gear shifts when the wheel speed is too low or near zero, and it can detect free-wheeling, the condition where the bicycle is moving but no torque is applied to the crank in which case gear changes are also inhibited.

In order to get reliable readings, the sensors had to be as close as possible to the magnets placed on the wheel. Measurements showed us that a $2,5$cm distance was ideal in order to read all spikes consistently. This measurement also matched the data sheet of said sensor. Therefore, instead of mounting the sensor on the rack mountings, and adaptor that would get the hall at around $2,5$cm was designed and 3D printed. With the latter, supports for the magnets were also designed, in order for them to not move out of the hall detection range due to the centrifugal force. In total, 9 magnet were attached to the spokes. This is a compromise between resolution at low speed and precision at high speed were inputs from 1 magnet can interfere with the following and previous.



(a) Hall sensor support



(b) Implementation on the bike.

Figure 8: Hall sensor implementation

3.3.1 Implementation

The sensor is connected to pin RB2 (INT2) of the PIC18F47Q84. The firmware (`hall.inc`, cfr to 5.1.2) configures INT2 to detect rising edges, so that each pulse (i.e. each magnet passing) triggers an interrupt. Timer0 is used to define a fixed measurement window, during which the pulses are counted.

The counting logic is as follows:

- `hall_counter` is incremented at each pulse.
- On overflow ($255 \rightarrow 0$), `hall_overflow` is incremented.
- After a fixed number of Timer0 overflows, the total number of pulses is stored and the counters are reset.

Since the measurement duration is known, the number of pulses counted during this interval is directly proportional to the wheel speed and can be deduced by dividing the counter by the time frame giving us an estimated speed but also confirming or not the fact that the wheel is moving or not.

3.3.2 Validation

Correct operation of the sensor was verified using visual indicators, first LEDs and later the LCD display, confirming that pulses are detected reliably and that speed measurements are updated at a consistent rate. The sensor proved stable across a wide range of speeds, and its digital output interfaces directly with the PIC without additional signal conditioning. One design choice was the number of magnets to fix on the rear wheel. Too many magnets cause the logic input to remain permanently high at higher speeds, making individual pulses impossible. Too few magnets result in insufficient pulse resolution at low speeds, making slow motion undetectable.

The choice of nine magnets comes from the dual role of the sensor. At low speed, the pulse count must be high enough to detect motion reliably and trigger the speed-based protections, which requires a high magnet count. At high speed, the sensor is only used to confirm that the wheel is turning, so resolution matters less. Nine magnets was the value that worked well at low speed without causing the logic input to saturate at higher speeds. This was validated by the fact the the speed of the bike is printed on the LCD screen in the final implementation and that this speed did not seem to saturate at the speed at which a bicycle can go.

3.4 LCD screen

The LCD screen enables one to see in real time the different variables that are stored inside the PIC. In fact, it is easier and clearer to have those values shown rather than to simulate using watch tables and the built-in protocols to recover variable changes (i.e using UART to send to another terminal the different values). Therefore, the LCD is implemented to let the user easily see the changes and also the feedback on the different sensors.

3.4.1 Implementation

The LCD that was sourced used an I2C backpack in the form of the PCF8574. This enable to reduce the number of required pins to 2 at the cost of software programming.
On the final PCB, the LCD screen pins were directly routed to the PIC, eliminating the need for an I2C backpack.
We implemented several functions in `lcd*.inc` in order to interact with the LCD:

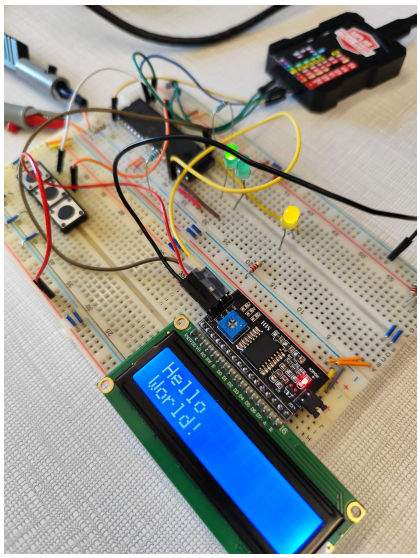
- `LCD_Init`: implement the initialisation sequence and set the entry mode.
- `LCD_GotoLine*`: Move the cursor to the first character of line 1/2.
- `LCD_Clear`: clear the screen and reset the cursor position
- `LCD_SendChar`: write the Ascii character currently in the working register on the screen.
- `LCD_SendDec`: Write the base 10 numeral in the working register (up to 999) as 3 Ascii characters for hundreds, tens and units.

In the final firmware, the LCD is used to display a few simple but useful screens. The main pages show either torque and cadence, or estimated power and rear-wheel speed. When the user adjusts the cadence target, the LCD also shows the target value directly, followed by a short confirmation message once the new value is saved. The LCD is therefore used both to monitor the system and to give the user direct feedback during interaction.

3.4.2 Validation

The LCD first proved useful during development through the I2C-backpack version, which allowed us to display internal variables while the direct parallel interface was still being implemented. In this way, it was used as a practical debugging tool for the rest of the firmware, since torque, cadence and other useful values could be visualized directly during testing.

Once the direct LCD interface was implemented and integrated on the PCB, the same role was kept in the final system. The screen was then able to provide both runtime monitoring and direct user feedback through the different display pages.



(a) Preliminary test using the I2C backpack



(b) LCD on the PCB showing Torque and cadence values

Figure 9: LCD implementations throughout the project

3.5 Buttons

Buttons let the rider control the system: One is used to switch between automatic and manual mode; in case of manual mode, 2 buttons change gears manually; 3 buttons are used to set the desired target velocity (displayed on the LCD screen); one is used to change the values displayed on the LCD display. This makes the system easier to test and more flexible to use.

3.5.1 Implementation

Each button is connected directly to the microcontroller. Because mechanical buttons bounce, a software debounce was added. When a press is detected, the firmware waits a short time and checks again. If the press is confirmed again, only then is the press accepted. This prevents false triggers. In manual mode, two buttons control upshifts and downshifts. Another button toggles between automatic and manual mode.

3.6 PCB and power supply

The following list presents the different features of our Printed Circuit Board:

- The PCB features 2 voltage Rails, one to supply the MCU, the Hall sensor and LCD display and another to supply the servomotor independently. This helps the servomotor to be provided with full power. This is done with two buck controller that step-down 12V. This input voltage has been chosen as the bike is provided with a 12V battery.
- All the pins have been exported in a way that the mapping can be switched easily. Since a PIC18FXXQ series is used, all pins can be remapped independently to any pins on the same register. This could come useful as we also made sure to add holes to reroute if needed.
- Pin headers connect the different sensors neatly on the outside of the PCB such that it is easy to connect.
- A connector is provided to connect the PICKIT basic easily.

- A place for the LCD combo is also put so that the LCD screen is mounted properly on the board that will act like a “dashboard” in our vision.

The following picture depict the overall placement of the PCB. For further reference, please refer to the schematics that are available on the Cadlab website mentionned in the annexes 7

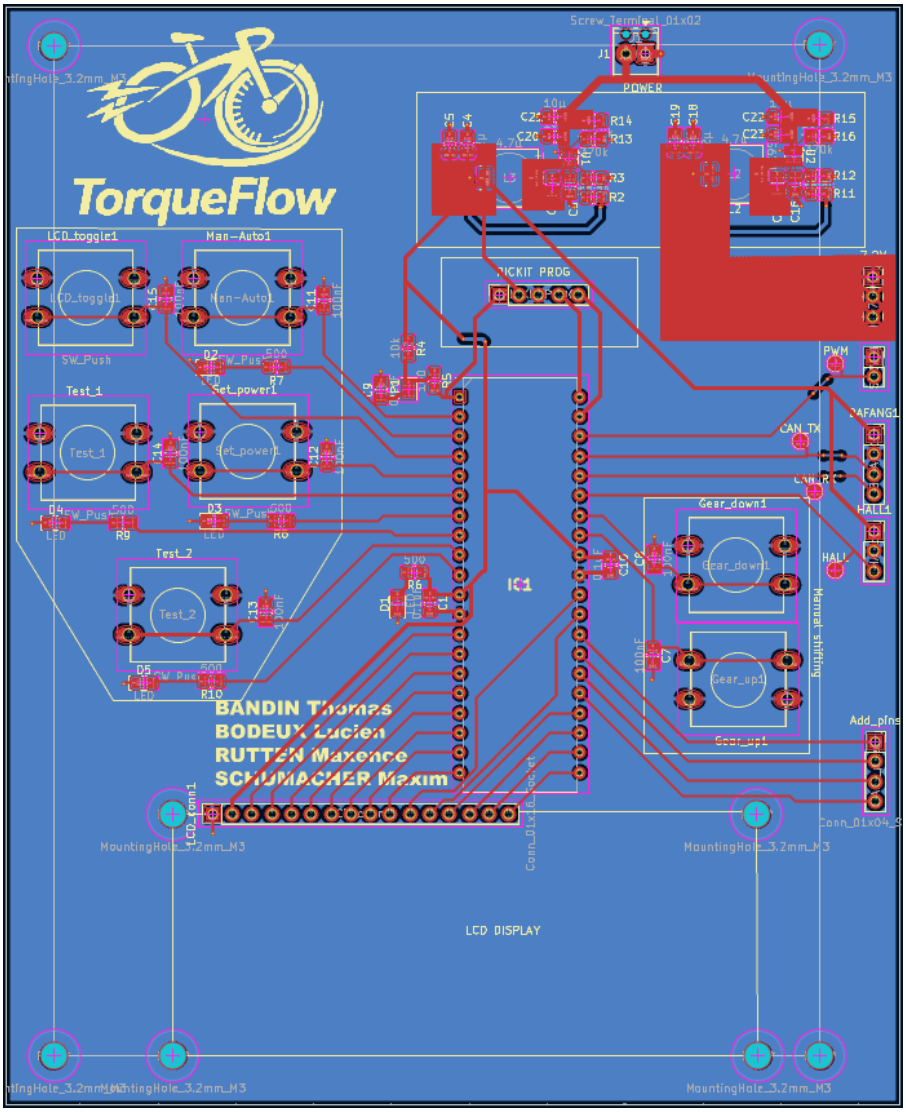


Figure 10: PCB schematic in Kicad

To contextualise, below is an annotated schematic of the whole PCB as an overview of the overall design.

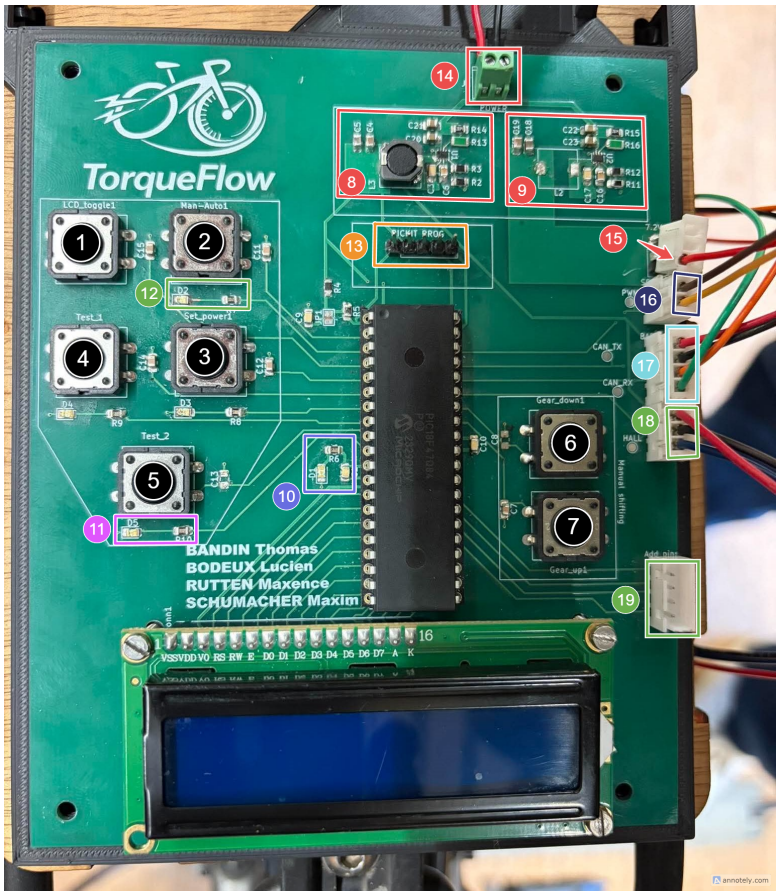


Figure 11: Assembled PCB and connections

Buttons:

1. Cycle through screen information options
2. Toggle between automatic and manual mode
3. Set cadence target
4. Adjust cadence target up
5. Adjust cadence target down
6. Manually shift down
7. Manually shift up

Converters:

8. 12V to 5V buck converter
9. 12V to 7.2V buck converter (inoperational at the time of picture)

Informative Leds:

10. ON led
11. "Sport" mode ON (a.k.a Remco Evenepoel mode)
12. Automatic mode ON

Connectors:

13. PICKIT connector pins
14. 12V power screw terminal
15. Servomotor power JST connector (7.2V planned, 12V for the last minute changes)
16. ground and servomotor PWM JST connector
17. CAN bus (RX, TX, ground, 5V) JST connector
18. Hall sensor (ground, 5V, signal) JST connector
19. additional JST connector, not mapped to any function

3.6.1 Power converters

2 power converter were needed to lower the 12V voltage to 5V and 7.2V. During testing the 7.2V rail had a failure that was attributed to a short near the inductance, which was removed. With the subsequent failure of our first servomotor and its replacement by a 12V model, it was decided to keep the left converter inoperative in the final version. A short wire was soldered inside the PCB support from the power screw terminal to the no longer 7.2V JST connector, now 12V.

3.6.2 Power Supply

A 12V lead-acid battery was generously lent to us by MicroSys. This was crucial for our project since the bike obviously need to be free of cable to properly function. This powers the 12V rail mentioned previously. To safely use the bike without having to physically reconnect the battery each time, a switch was soldered to the power cables.
On the final project, it is positioned next to the servomotor, close to the buck converters. It is also worth mentioning that the battery datasheet indicates a limited current capability at startup. To take this into account, a delay was added in `main.asm` (see 5.2.1) before initializing the most power-demanding peripherals.

3.7 Overall package

On the final project, the PCB, battery and servomotor support were mounted on the bike rack via a laser cut board to be firmly attached and for ease of access. As the servomotor was one of the first to be tested on the bike, a separated 3D printed piece was designed, with latter a second one for both the battery and PCB.

The use of a wooden board was motivated by its adaptability to future changes: At the moment it was cut out, the size, layout and position of the PCB and battery was not totally sure. With a wooden board, we were able to cut out more mounting holes afterwards. This was proved very useful when we had to mount the second version of the servomotor support. We could easily drill new holes to accommodate its different dimensions.

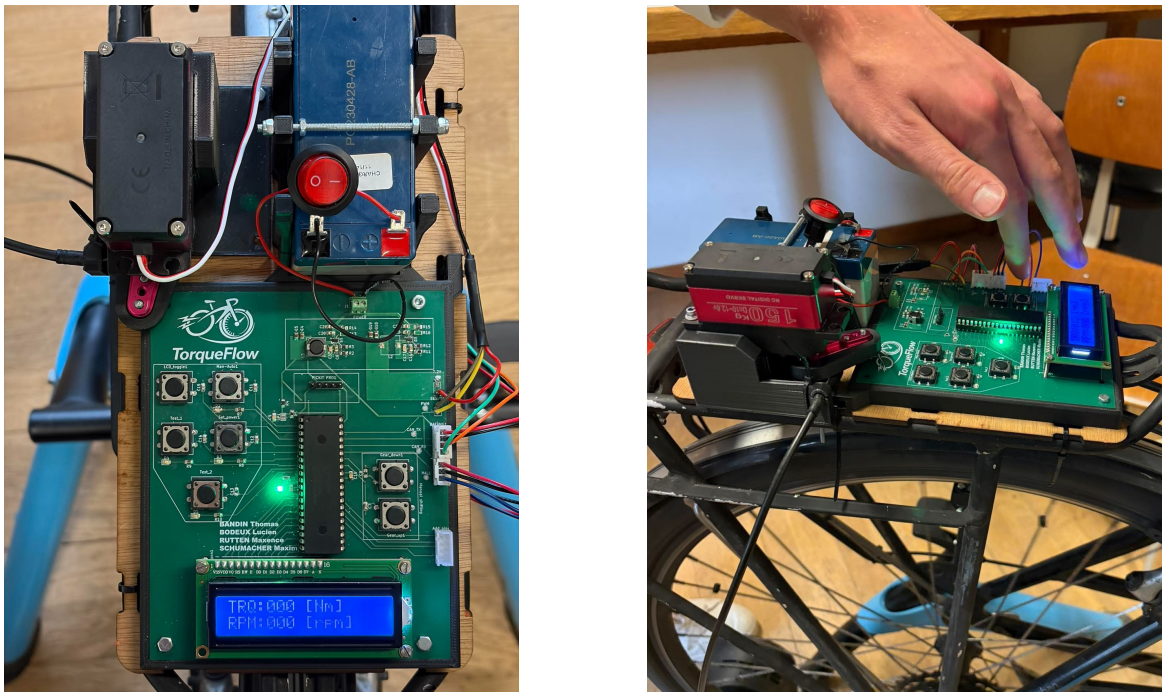


Figure 12: Final board layout

On the final version, the board is attached to the bike with 4 zip-ties, and the 2 3D printed piece are attached to the board with bolts, with the PCB secured on its support by those same bolts. The battery is kept in place by 2 bolts attached horizontally.

4 Control Strategy

The aim of the controller is to keep the rider in a suitable cadence range while adapting this range to the current effort level. In practice, the behaviour must remain natural for the rider and consistent with the riding situation.

The implemented strategy is based on cadence targets modulated by power. The rider first selects a preferred cadence target, and the controller then adjusts the active target according to the estimated power. To keep the implementation simple, only two power zones are used. In the low-power zone, the selected cadence target is kept unchanged. In the high-power zone, the target is shifted upward so that the controller favours a slightly higher cadence when the rider effort increases.

This idea is inspired conceptually by the literature on cadence and effort, in particular the article *Cadence, power, and muscle activation in cycle ergometry* [1], which shows that the cadence associated with reduced muscle activation tends to increase with power. In this project, the goal is not to reproduce a detailed physiological model, but rather to show in a simple and understandable way that power can be used inside the control loop to adapt the cadence target.

With this strategy, the controller gives an intuitive response in the main riding situations:

- **Steady riding:** when the rider effort remains moderate, the controller keeps the cadence around the user-selected base target.
- **Acceleration or strong effort:** when the rider pushes harder, the estimated power increases and the controller switches to a higher cadence target, which favours a more dynamic pedalling rhythm.
- **Difficult climb:** when resistance increases, cadence tends to fall if the gear becomes too hard. The controller then interprets this as a need for an easier gear.
- **Downhill or very easy section:** when the resistance becomes low, cadence can rise more easily. The controller therefore interprets this as a need for a harder gear.

This makes the control both robust and adaptable. The rider chooses the preferred cadence reference, and the controller adapts its behaviour around this reference according to the current effort level. In this way, the same system can suit a wide range of uses, from relaxed everyday riding to a more dynamic riding style.

4.1 Robustness mechanisms and protections

Since the control relies on human inputs, which are naturally variable and unstable, the strategy cannot be applied directly without additional precautions. Several mechanisms are therefore needed to make it usable in practice and to avoid erratic behaviour on the real bicycle.

The main protections included in the implemented controller are the following:

- **Pedalling detection:** the control is only active when cadence and torque indicate that the rider is effectively pedalling. This prevents the controller from reacting to near-zero values when the bicycle is coasting or stopped.

[1] Available at http://ovidsp.ovid.com/ovidweb.cgi?T=JSCSC=YNEWS=NPAGE=fulltextAN=00005768-200007000-00015D=ovft

- **Hysteresis:** the controller does not react to one exact boundary. A cadence window is defined around the active target, and the transition between the low-power and high-power regimes also uses separate entry and exit thresholds. This avoids repeated switching when the measured quantities fluctuate around a limit.
- **Consecutive validation of the conditions:** a shift is not triggered as soon as one condition becomes true. The corresponding condition must remain valid for several consecutive control evaluations before the request is accepted. This filters out short disturbances and isolated fluctuations.
- **Post-shift cooldown:** after each gear change, a delay is imposed before the controller can request another one. This gives the drivetrain time to settle and prevents several shifts from being triggered too close to one another.
- **Torque override:** in addition to cadence, a high-torque condition can directly favour an easier gear. This allows the controller to react when rider effort becomes too high even if cadence alone would not yet be enough to trigger a shift.

Together, these mechanisms keep the controller from reacting too aggressively to short variations and make the behaviour compatible with the real mechanical response of the bicycle.

4.2 General control algorithm

At a general level, the control follows a simple sequence. It first checks whether the current measurements correspond to an actual pedalling situation. If not, no automatic action is taken. If pedalling is active, the controller determines the current effort regime and selects the corresponding cadence target.

The measured cadence is then compared with the acceptable range around this target. If cadence stays too high, the controller tends toward a harder gear. If cadence stays too low, it tends toward an easier gear. A high torque can also reinforce the request for an easier gear. If none of these conditions is sufficiently confirmed, the current gear is kept.

The overall decision flow of the controller is summarized in Figure 13. The diagram is simplified in the sense that it does not show all the robustness mechanisms described earlier, but it is sufficient to understand the general logic of the algorithm.

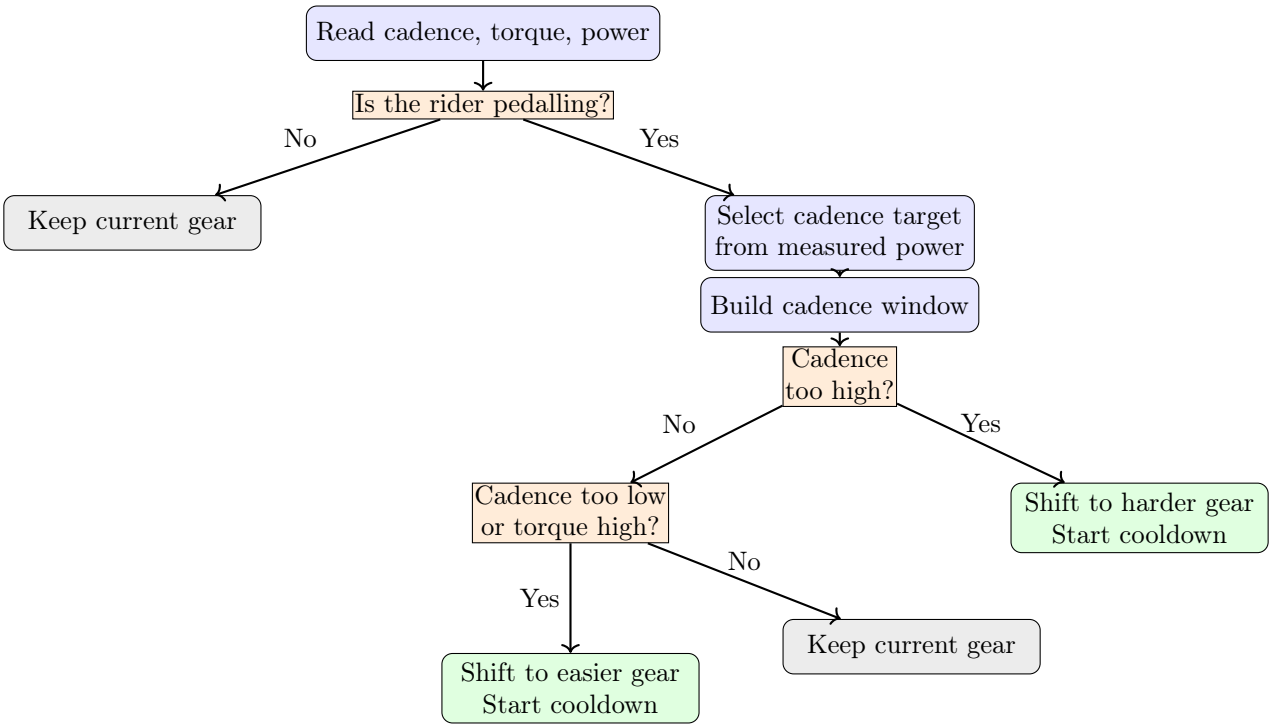


Figure 13: Simplified decision flow of the implemented controller.

4.3 Practical implementation parameters

To move from the general control principles to a real implementation, several numerical parameters had to be fixed. These values define how the controller behaves in practice and give concrete meaning to the targets, thresholds and delays introduced above.

In the current implementation, the default cadence target is set to 70 rpm. This value was chosen as a reasonable compromise for a standard rider, in order to provide a natural and comfortable riding feeling. The user can then increase or decrease this target according to personal preference.

The main parameters used by the controller are summarized in Table 2.

Table 2: Main parameters used in the practical implementation of the controller

Parameter	**Value**	**Role**
Default cadence target	70 rpm	Base reference selected at startup.
Minimum cadence target	60 rpm	Lower limit allowed for user adjustment.
Maximum cadence target	120 rpm	Upper limit allowed for user adjustment.
Cadence target step	10 rpm	Increment or decrement applied through the buttons.
High-power target offset	+20 rpm	Increase applied to the active target in the high-power regime.
High-power entry threshold	800 W	Power level above which the controller tends toward the higher cadence target.
High-power exit threshold	600 W	Lower threshold used to return to the base cadence target without oscillation.
Cadence window half-width	15 rpm	Defines the acceptable range around the active target.
Shift stability threshold	5 evaluations	Number of consecutive control evaluations required before accepting a shift condition or a power zone change
Control evaluation period	100 ms	Time between two normal control evaluations.
Post-shift cooldown	1.5 s	Minimum delay imposed after a shift before another one can occur.
Pedalling cadence threshold	5 rpm	Minimum cadence required to consider that pedalling is active.
Torque activity threshold	1 Nm	Minimum torque required to consider that pedalling is active.
Torque override threshold	65 Nm	High torque condition that directly favours an easier gear.

Taken together, these parameters define the practical behaviour of the controller. For example, a shift stability threshold of 5 evaluations combined with a control period of 100 ms means that a shift condition usually has to remain valid for about 0.5 s before a gear change is accepted. These values were chosen experimentally on the real setup and were found to give a robust and pleasant behaviour in practice. The power thresholds may seem quite high, but this is consistent with the current implementation, since the torque evaluation tends to overestimate the real torque and therefore the estimated power as well. The controller nevertheless remains intentionally simple, and these parameters could still be refined further depending on the rider profile and the intended use.

5 Software Implementation

The software is divided into several small modules, each handling one specific part of the system. Separate files are used for pin configuration, sensor acquisition, button handling, servomotor control, LCD driving and gear-shifting logic. This makes the firmware easier to build and test, since each part can be validated separately before being integrated into the final program. All the codebase can be found on the dedicated github of this project referenced in the annexes 7

In its final form, the firmware must satisfy the following requirements:

- receive torque and cadence data from the Bafang sensor through CAN communication;
- estimate rear-wheel speed from Hall sensor pulses;
- debounce the push-buttons and detect valid user actions;
- manage the automatic and manual operating modes;
- allow the user to adjust the cadence target used by the automatic controller;
- compute the estimated power used to adapt the cadence target;
- evaluate the control conditions and protections before accepting a shift;
- generate the servomotor command corresponding to the requested gear;
- refresh the LCD at a controlled rate without disturbing the rest of the firmware.

The following subsections first present the main software modules. The way they are combined, timed and executed in the final firmware is then described afterwards.

5.1 Main software modules

5.1.1 `can_torque.inc`

This file handles the communication with the Bafang torque sensor through the CAN peripheral of the PIC18F47Q84. It is responsible for receiving the sensor frames, identifying the useful message, extracting the torque and cadence information, and converting the raw torque value into a quantity that can be used by the rest of the firmware. It is therefore directly related to the CAN transceiver connected to the PIC, and more generally to the crank sensing part of the system.

In the implemented architecture, the CAN module is configured to receive frames from the sensor, and the firmware then checks whether the received identifier corresponds to the expected message. Once a valid frame is found, the cadence byte is extracted directly, while the torque bytes are converted into an integer torque estimate. In order to keep the torque measurement stable enough for both control and display, the code applies a filtering stage before publishing the final torque value. Each new torque sample is first checked against a minimum and a maximum acceptable range: values below the lower threshold are forced to zero, while values above the upper threshold are discarded. The accepted samples are then inserted into a circular moving average, so that the published torque evolves more smoothly and is less sensitive to short fluctuations from one CAN frame to the next.

The main routines of this file are `CAN_Torque_Init`, which initializes the CAN module and the torque filtering variables, and `CAN_ServiceTorqueSensor`, which is called repeatedly by the main loop. This second routine relies on `CAN_Receive` to read the incoming frame and on `_CAN_ComputeTorqueNm` to convert the raw sensor bytes into a usable torque value before the filtering stage is applied.

Algorithm 1 High-level behaviour of `can_torque.inc`

`CAN_Torque_Init`: initialize the CAN peripheral and reset the torque averaging variables
`CAN_ServiceTorqueSensor`:
if a new CAN frame is available **then**
    `CAN_Receive`: read the identifier and payload of the received frame
    if the identifier matches the Bafang sensor message **then**
        extract cadence from the payload
        `_CAN_ComputeTorqueNm`: convert the raw torque bytes into a torque sample
        if the sample is below the minimum accepted value **then**
            force the sample to zero
        else if the sample is above the maximum accepted value **then**
            discard the sample
        else
            update the circular moving average with the new sample
        end if
        publish the filtered torque value for the LCD and the controller
    end if
end if

Cadence and torque are not handled in exactly the same way in this file. Cadence is taken directly from the received frame, since it is stable enough to be used as it is, while torque is first converted and then filtered before being published. The filtering is implemented as a circular moving average, which keeps the torque signal stable without requiring a full recomputation each time a new sample is received.

5.1.2 `hall.inc`

This file is responsible for counting rear-wheel pulses and converting them into a speed value that can be used by the rest of the firmware. It therefore provides the software interface between the raw pulse events and the published rear-wheel speed information.

The logic is based on interrupts. One interrupt updates the pulse counter whenever a new event is detected, and another periodic interrupt defines the measurement window. At the end of each window, the current pulse count is captured, reset and converted into an estimated speed value.

Algorithm 2 High-level behaviour of `hall.inc`

`Hall_Init`: initialize the pulse counter and configure the interrupt logic
while the firmware is running **do**
    if a pulse event occurs **then**
        `Hall_INT2_ISR`: increment the current pulse count
    end if
    if the measurement period expires **then**
        `Hall_TMR0_ISR`: store the pulse count and reset it
        `_Hall_UpdateSpeedKmh`: convert the captured count into a speed estimate
        publish the updated speed value
    end if
end while

The main software choice in this file is to estimate the speed over a fixed measurement window rather than from isolated pulse timing. A sufficiently large window, approximately 0.524 s, was chosen in order to obtain a reliable measurement and to limit the influence of short-term fluctuations in the pulse count. It is also worth noting that the pulse count is stored as an 8-bit integer in the code. This

is fully sufficient for our application, since an overflow would only occur at an estimated bicycle speed of about 422 km/h, which is far beyond any realistic operating condition.

5.1.3 `debounce.inc`

This file handles the push-buttons used by the user to interact with the system. Its role is to acquire the raw button states, remove the effect of mechanical bouncing in software, detect valid button presses and update the corresponding flags or actions. In other words, it avoids relying on a dedicated hardware debounce solution and ensures that each user input is interpreted in a clean and reliable way by the rest of the firmware.

The main routines of this file are `Debounce_Init`, which initializes the debounce variables and the periodic sampling mechanism, `Debounce_ISR`, which regularly samples the raw button states and validates a stable state, and `Buttons_HandleMainLoop`, which is called in the main loop to detect newly pressed buttons and apply the corresponding actions. The file also contains the small action routines associated with each button, such as toggling the operating mode, requesting a manual shift or modifying the cadence target.

Algorithm 3 High-level behaviour of `debounce.inc`

```
Debounce_Init: initialize the debounce variables and sampling timer
Buttons_Init: initialize the button history and operating flags
while the firmware is running do
    if a debounce timer event occurs then
        Debounce_ISR: sample the raw button states
        if the sampled state remains unchanged for enough consecutive samples then
            update the stable debounced button state
        end if
    end if
    Buttons_HandleMainLoop
    compare the current stable state with the previous one
    detect which buttons have just been pressed
    if the mode button is pressed then
        toggle between manual mode and automatic mode
    end if
    if the page-selection button is pressed then
        update the display-selection flag
    end if
    if manual mode is active then
        process manual shift requests if the safety condition is satisfied
    end if
    if the target-adjustment buttons are pressed then
        increase or decrease the cadence target within its allowed range
    end if
end while
```

In this file, the interrupt routine is only used to sample the buttons and validate a stable state. This keeps it short and predictable. The accepted state is then interpreted later in the main loop, where the corresponding action is applied.

The debounce parameters were chosen so that a button state must remain identical over several consecutive samples before being accepted. With the current sampling period and threshold used in the code, this corresponds to a validation time of about 30 ms, which is long enough to reject the mechanical bouncing of the push-buttons while remaining short enough for the interface to feel responsive.

5.1.4 `servo_hw.inc`

This file is responsible for generating the command sent to the servomotor and for linking this command to the current gear selected by the firmware. In practice, it manages the internal gear state, converts a requested gear into the corresponding timer preload, and continuously generates the output waveform used to actuate the derailleur through the servo.

The main routines of this file are `Servo_Init`, which initializes the timer and the default gear, `Servo_SetGear`, which updates the selected gear while keeping it inside the allowed range, and `Servo_IncPos` / `Servo_DecPos`, which request a shift toward the neighbouring gear. The periodic waveform itself is produced by `Servo_ISR`, while the helper routine `_servo_load_ticks_from_gear` loads the timer preload corresponding to the current gear.

Algorithm 4 High-level behaviour of `servo_hw.inc`

```
Servo_Init: initialize Timer1 and select the default gear
load the timer preload corresponding to the selected gear
while the firmware is running do
    if a gear change is requested then
        Servo_SetGear: clamp the requested gear within the valid range
        update the current gear
        _servo_load_ticks_from_gear: load the preload associated with this gear
    end if
    if Timer1 overflows then
        Servo_ISR: toggle the output pin
        reload Timer1 with the current preload
    end if
end while
```

In this file, the gear is handled as a discrete software state rather than as a continuous actuator position. A lookup table is used so that each gear can be associated with its own precise timer preload value. This makes it possible to tune the command of each gear independently, which is useful when the different mechanical positions are not perfectly uniform. The rest of the implementation then remains simple, since the current gear only has to select the corresponding preload. The firmware initially included a short "boost" mode for downshifting(pulling the cable). In practice, this meant pulling the cable slightly further than the normal gear holding position for a short amount of time, in order to make sure the shift was effectively triggered. This was necessary with the first servomotor (20 kg). After this servomotor was damaged by an excessive current and had to be replaced last minute by a 150 kg model, the extra boost was no longer needed and was removed from the final implementation.

The timer interrupt is finally used only to maintain the output waveform, so the signal generation remains periodic and independent from the rest of the main loop.

5.1.5 `Control.inc`

This file implements the control strategy described earlier in the report. It uses the measured quantities provided by the other modules, mainly cadence, torque and estimated power, to decide whether the bicycle should shift toward an easier or a harder gear. It also contains the protection mechanisms described earlier.

The main routines of this file are `Control_Init`, which initializes the control variables, `ComputePower`, which estimates the rider power from torque and cadence, and `GearShiftControl`, which evaluates the current riding situation and decides whether a shift must be requested. The file also contains helper routines used to update the active cadence target, to prepare the current cadence window and to reset the internal counters when the control conditions are no longer satisfied.

Algorithm 5 High-level behaviour of `Control.inc`

```
Control_Init: initialize the target cadence and the control counters
while automatic mode is active do
    ComputePower: estimate the current rider power from torque and cadence
    GearShiftControl
    if the control delay has not yet expired then
        return without inspecting further
    end if
    if pedalling is not active then
        reset the stability counters
        reload the normal control delay
        return
    end if
    determine the current power zone
    select the corresponding active cadence target
    derive the cadence thresholds from this target
    if cadence remains too high for enough consecutive control evaluations then
        request a shift toward a harder gear
        reload a long delay before the next shift
    else if cadence remains too low, or torque is too high, for enough consecutive control evaluations
then
        request a shift toward an easier gear
        reload a long delay before the next shift
    else
        reload the normal control delay
    end if
end while
```

In practice, this file forms the decision layer of the firmware. It updates the internal control variables and decides whether a shift request should be sent to the servomotor module. The cadence thresholds are updated at each control evaluation so that any change in the active target is directly taken into account by the algorithm.

5.1.6 `wait.inc`

This file provides simple blocking delay routines in microseconds, milliseconds and seconds. They were used extensively during the early stages of development, when task concurrency was not yet a main concern and simple timing loops were sufficient to validate the different building blocks.

In the final firmware, these delays are used as rarely as possible in order to avoid blocking the microcontroller unnecessarily. They are mainly kept for a few low-level hardware timing requirements, such as the LCD driver and some startup sequences.

5.1.7 `lcd_direct.inc`

This file contains the low-level LCD driver used by the firmware. It initializes the display and provides the basic routines needed to write characters, numbers and complete lines on the screen. The higher-level display logic decides what must be shown, while this file only handles how the information is sent to the LCD.

The main routines provided by the file are summarized in Table 3.

Table 3: Main routines provided by `lcd_direct.inc`

Function	Role
`LCD_Init`	Initializes the LCD and places it in the required operating mode at startup.
`LCD_Clear`	Clears the display and returns the cursor to its initial position.
`LCD_GotoLine1`	Moves the cursor to the beginning of the first line.
`LCD_GotoLine2`	Moves the cursor to the beginning of the second line.
`LCD_SendChar`	Sends one character to the LCD.
`LCD_SendDec`	Displays an 8-bit unsigned decimal value.
`LCD_SendDec16`	Displays a 16-bit unsigned decimal value.

Some waits are required in this file because the LCD controller cannot accept successive commands or data writes without a minimum delay between them. For this reason, the driver inserts short waits after each transfer, and longer ones during initialization or after a clear command.

These waits are implemented with the blocking delay routines provided by `wait.inc`. In practice, this remains acceptable because refreshing a complete 16×2 screen only takes a few milliseconds, which is short compared with the time scales of the control and user interaction tasks.

5.1.8 `pinconfig.inc`

This file configures the microcontroller pins used by the firmware. It defines their direction, enables the required digital mode and pull-ups, and links each pin to its role in the final system. Since it mainly provides the interface between the firmware and the hardware connections, it is best read together with the functional pin mapping given in Table 4.

Pin	Function in the project	Direction	Type	Configuration / remark
RA0	Automatic/manual mode button	Input	Digital	Pull-up enabled
RA1	Mode indicator LED	Output	Digital	
RA2	LCD screen selection button	Input	Digital	Pull-up enabled
RA3	Cadence-target setting mode button	Input	Digital	
RA4	Cadence target increase button	Input	Digital	
RA5	Auxiliary LED output	Output	Digital	
RA6	LCD control line	Output	Digital	
RB0	Manual gear button	Input	Digital	
RB1	Manual gear button	Input	Digital	
RB2	Hall sensor input	Input	Digital	
RB3	CAN receive line	Input	Digital	PPS-mapped
RB4	CAN transmit line	Output	Digital	PPS-mapped
RB5	Servo command output	Output	Digital	
RC0-RC5	LCD data / control lines	Output	Digital	
RD0-RD3, RD6, RD7	LCD data and backlight lines	Output	Digital	
RE0	Status LED	Output	Digital	
RE1	Cadence target decrease button	Input	Digital	Pull-up enabled
RE2	Indicator LED	Output	Digital	

Table 4: Functional pin mapping used in the final implementation

5.2 Main loop organisation and timing strategy

Once the individual modules are defined, the next question is how they are executed together. In the final firmware, the recurring tasks are handled through a simple round-robin main loop, while the low-level timing constraints are handled in the background by interrupts.

At each pass through the main loop, the firmware first checks whether a new CAN frame is available and updates torque and cadence if needed. It then updates the power estimate, refreshes the LCD when its refresh period has expired, interprets the debounced button states, and finally evaluates the automatic gear-shifting logic when the control period has elapsed.

These main-loop tasks are summarized in Table 5. The execution times given in the table are approximate and were estimated from the timing constants used in the code, from the delays present in the LCD driver, and from the small number of assembly instructions executed by the shortest routines.

Table 5: Main loop tasks executed in the final firmware

Task	Effective period	Execution time	Role
`CAN_ServiceTorqueSensor`	every loop pass	short	Checks whether a new CAN frame is available and updates torque and cadence if a valid frame is received.
`ComputePower`	every loop pass	short	Updates the power estimate from the latest torque and cadence values.
`LCD_RefreshDisplay`	200 ms	about 1.5 to 2 ms when active	Refreshes the display at a human-readable rate.
`Buttons_HandleMainLoop`	every loop pass	short	Interprets the debounced button state and updates the corresponding actions or flags.
`GearShiftControl`	100 ms in automatic mode	short	Evaluates the automatic control conditions and generates a shift request when needed.

The CAN service is intentionally executed at every pass through the loop. This makes sure that the firmware always works with the freshest possible sensor data, which is important because torque and cadence are used both by the controller and by the display. It also avoids leaving received CAN data pending longer than necessary.

Not all tasks need to run at the same rate. The LCD refresh and the control evaluation are paced by software counters updated by the `Timer4` tick interrupt. In parallel, the other interrupts handle the low-level functions that either require precise timing, such as the servo signal generation and pulse counting, or benefit from a regular sampling period, such as button debouncing.

Given the small number of main-loop tasks, their short execution times, and their moderate timing requirements, these tasks can be handled reliably with the chosen round-robin structure. Their schedulability is not strict in the sense of hard real-time scheduling, and the few timing constraints that do exist are easy to enforce through simple interrupt-driven counters and dedicated low-level interrupt routines. In this context, an RTOS was not necessary, since it would have increased the software complexity without providing a decisive advantage for the present system.

5.2.1 `main.asm`

These principles are implemented concretely in `main.asm`. This file defines the startup sequence of the firmware and contains the round-robin loop around which the rest of the software is organized.

At startup, the low-level modules are initialized in a coherent order: first the pin configuration and the basic peripheral drivers, then the interface and acquisition modules, and finally the servo and control-related variables. A deliberate delay is inserted before enabling the servomotor, in order to avoid adding its current demand to the rest of the startup sequence. This helps reduce the risk of excessive inrush current when the system is turned on. Once this phase is complete, interrupts are enabled and the firmware enters its normal execution loop.

Algorithm 6 High-level behaviour of `main.asm`

```
initialize the low-level modules and shared variables
initialize the acquisition, interface and timing modules
wait briefly before enabling the servo
initialize the servo and the control layer
enable interrupts
while the firmware is running do
    service the CAN sensor
    update the power estimate
    refresh the LCD when required
    process the user inputs
    if automatic mode is active then
        evaluate the control logic
    end if
end while
```

This file is therefore where the overall software structure becomes explicit. The interrupt routines provide the low-level timing behaviour in the background, while the main loop coordinates the higher-level tasks in the order described above.

6 Lessons Learned and Possible Improvements

Several ideas for improvement appeared during the project but could not be explored further, mainly because of time constraints:

- The screen and buttons should ideally be placed at the front of the bicycle so that the rider can read the displayed information and interact with the system more comfortably while riding. In the current configuration, this also meant that debugging often required two students: one to pedal and the other to observe the screen and the system behaviour. This could also allow to display other types of information such as warnings for the gear changes. This design approach was valued because of simplicity (no wires running along the bike to be scared of) but it's therefore not ergonomic.
- More care should be taken in the choice of power electronics packages. The buck converters used in this project were very small, mainly because they were the only ones that satisfied both the cost and power requirements. However, this made measurements and handling much more delicate. While trying to measure the output of one of these converters, the probe slipped and caused a short circuit, which damaged the 7 V power stage and the initial 20 kg servomotor only two days before the deadline. In the future, more precautions should therefore be taken either in the choice of the packages or in the manipulation of such small components, in order to avoid this kind of last-minute stress and the resulting design changes. More thoughts could also be used when designing protection circuitry for future project to avoid those problems in the longer run.
- More care should be taken when designing the PCB. In our case, we didn't use any protection to limit the component amount but the probe failure explained just above proved that with more care and components that we could have designated as crucial, some of our practical mistakes could have been avoided.
- Although the servomotor already includes an internal feedback loop, it would have been enriching to design and implement such a feedback loop ourselves. This would have made the control aspect of the project even more complete from a learning point of view.

- The component that gave us the most trouble is clearly the Bafang torque sensor. We choose an already integrated model to save us conception time and to be sure to have something working reliably, but in the end to readings we could get from it were not satisfactory. In the final project it is used more as a qualitative measurement than a reliable quantitative one in the case of torque. For this reason designing a custom sensor, most likely with a strain gauges, would certainly improve this project although this would certainly be quite a challenge.

Acknowledgements

We would like to sincerely thank all the people who contributed to this project through their time, advice and support.

In particular, we thank the professors for their guidance and advice throughout the project, which made this work possible. We also thank the teaching assistants for their availability, their time and their helpful advice during the different stages of the project.

We would also like to thank Angel for his time, his hardware advice, and his sometimes unconventional but effective solutions to the bugs we encountered. We also thank Kleber Axel for lending us the 150 kg servomotor after the last-minute failure of our initial setup, without which it would probably not have been possible to present such a complete and functional system. Finally, we thank the Fab52 team for their help with the integration of the Bafang sensor on the bicycle and with the 3D prints, as well as Microsys team for their last-minute help with the 3D prints when Fab52 was closed.

7 Annexes

- Github page containing all the codebase for this project : Github
- Cadlab repository for all the PCB files ready to be viewed directly from the website : CadLab

The demonstration videos are also available on the Github and each of the videos represents the following features of the project:

1. Startup_and_LCD.mp4 shows the startup sequence, the toggle between the different screens available to the user as well as the cadence target setting interface.
2. Manual_1.mp4 and Manual_2.mp4 show the manual gear shifting interface and the effect on the actual gears of the bike, the inhibition of button press when the rear wheel is not spinning as well as active measures of torque and cadence on the LCD screen.
3. Automatic.mp4 Showcases once again the manual gear shifting, the transition to automatic shifting, the inhibition of manual button press in that mode and the global functioning in automatic mode.