Lowering Energy Consumption by Implementing a Custom Engine Control Unit in Place of a General Purpose Controller

Asger Kühl *s163933*, Berk Gezer *s163916*, Frederik Ettrup Larsen *s163920*, and Irene Danvy *s163905*

Introductory project – Electrotechnology 2018

***Abstract*—In this paper we will discuss how we replaced the generic motor control unit in the 'EcoCar' by special-purpose embedded electronics and lowered its energy consumption by one third, providing an extra 23.5 km/l in nominal conditions. The EcoCar is a fuel efficient car built by DTU Roadrunners to participate in the annual Shell EcoMarathon competition. For the first time this year, the competition takes electrical energy used into account, an aspect of the EcoCar which has not been optimized at all. The component in the car which consistently consumes the most energy is the engine control unit, which is currently a National Instruments reconfigurable I/O, shortened 'RIO'. Our approach to lowering the EcoCar's energy consumption was therefore to replace the RIO with our own embedded electronics, without losing any functionality. The embedded electronics was made using only one microcontroller, and removed no functionalities of the car.**

I. Introduction

DTU Roadrunners is a DTU student driven project aiming to create the most fuel efficient car possible and to participate in and win the annual Shell Eco marathon. Initially only mechanical engineering students where involved in this project. But over the years, it became more apparent that electrical engineers were needed to design the electronic components of the car. This year, DTU Roadrunner team was seeking to replace the electrical engine control unit in the ecocar. So far, the solution has been the so called RIO (National Instruments Reconfigurable I/O), which has been the primary engine control unit. But, due to a rule change in the Shell Eco Marathon, electrical energy consumption is now measured along with the fuel consumption to determine the fuel efficiency of the car. Combining this with a goal to reduce the overall weight of the car it became apparent that the RIO would have to be replaced with our own embedded electronics. This will hopefully also come with a multitude of advantages, such as the boot time for the car decreasing drastically, and the engine control in general being more customized to the specific needs of the mechanical team.

A. Project Specification

How can the Ecocar's engine control be redesigned and optimized seen from an energy perspective, when electric energy is included in the energy accounting, while fulfilling the "Shell's Eco-marathon Urban concept" requirements?

The engine control unit must:

- Perform time critical injection and ignition of fuel in the motor, depending on the motor's RPM.
- Make the smaller starter motor run until the main motor has enough inertia to sustain itself.
- Control the gear servo, in both manual and automatic mode.
- Calculate and detect the car's state, such as speed, temperatures, oxygen to fuel ratio and battery voltage.
- Send and receive relevant data over the CAN-bus.
- Communicate with a computer application through USB.
- Log data on an SD card.
- Have an emergency stop function, which halts the motor until restart.
- Control the motor such that it only drives as long as a continuous signal is received from the driver.

B. Scope of project

We do not need to design a UI that the board can communicate with, neither do we need to construct the rest of the car, or any of the other electronics in the car. We do, however, need to control the engine of the car and communicate with the other microcontrollers in the car.

II. Design process

A. Understanding the RIO

In order to replace the RIO, we had to understand what the RIO was, and what it could do. So we started by analyzing the code and documentation to learn the nature of its role. From this research, we became acquainted with the crucial functions of the RIO: injection and ignition of fuel, which both had to be done at precise moments to ensure optimal acceleration. We also learned which functions were not immediately crucial, but still played an important part in the operation of the engine, such as data logging and test functions used for calibration of the engine constants. In addition we learned that the RIO cannot communicate over CAN-bus, the protocol used by all other control units in the car. Therefore a 'motor board' was built, which acted as a translator for the RIO, receiving commands from the rest of the board and sending to the RIO over r232, as well as receiving measurements treated by the RIO via r232 and transmitting them out using CAN-bus. This board was also given a small number of other functions, such as calculating RPM from the data of a tachometer, and

playing melodies as an error detection mechanism when the RIO wasn't connected or emergency stop had been pulled.

If our project succeeded this board would become mostly superfluous, and we decided to replace it's functionality as well as the RIO's.

B. Understanding the EcoCar

When the EcoCar begins to drive, it starts the starter motor, which is an electrical dc motor. The starter motor accelerates the crankshaft up to where the RPM of the crankshaft is higher than threshold given by the MEK team, and if the RPM should dip below this threshold, the starter motor is started again. This limit is approximately where the injection and ignition of the motor is fast enough for it to drive itself. The car continues to drive as long as the driver keeps the 'burn' button on the wheel pressed. As long as the motor is burning, the RIO will do the injection and ignition calculations and send them to the motor board, which sends the data along to the mini board. The mini board amplifies a signal, and sends it on along to one of two outputs, either ignition or injection. This process will run until the driver stops pressing burn, or if the emergency stop is pressed.

The car can also be set to idle. The idle state forces the gear in the neutral state, and injects a lot more fuel than pressing burn. The idle state was created to heat up the motor before racing, since the fuel efficiency of a warm motor is higher than that of a cold one. Trying to change the gear or burning while in the idle state, will exit the state.

The primary strategy of our EcoCar is to burn very rarely, and then roll for as long as possible.

C. Work flow

Once we had understood the RIO and the car, we started working on implementing the code for our own ECU. We used the Teensy 3.6's quadrature decoder functionality to ensure that we would always know the exact position of the encoder. Using a system of interrupt service routines and quick calculations, we were able to precisely control ignition and injection. Once this was in place, we started making the rest of the various functions, we enabled the board to log data to an SD card, and allowed it to communicate with a user interface on a computer, in which we could set different variables used for control and calibration, and activate modes (such as idling, burning, autonomous gear change, etc.).

Following this, we started running various tests on the board, in order to measure the current going into the RIO, the ECU and out of the battery in both cases. We also drove around with our engine control to ensure that it worked as intended, which it did.

III. Framework

A. Hardware

We chose to use a Teensy 3.6 microcontroller, as these are already widely used by DTU roadrunners for controlling other parts of the Urban Concept car (lights, steering, etc.), and had proved reliable and easy to program. The technical specifications can be seen in table I.

TABLE I
TECHNICAL SPECIFICATIONS OF THE TEENSY 3.6 [6]

Teensy 3.6
MK66FX1M0VMD18, core with an 180MHz, 32 bit Cortex-M4F processor
1024 kbytes flash memory
256 kbytes RAM
4096 bytes EEPROM
58 pins
32 channels for direct memory access
19 timers
Options for USB, I2C, SPI, CAN-bus, ethernet and SD card.

A prototype printed circuit board, or PCB, for the Teensy to control, had already been designed and printed by PCBWay when we started the project. This PCB was soldered and it worked as our main platform for testing and debugging. After many weeks of working on this board, we did however run into a serious problem when we tried to communicate with other boards through the CAN-bus protocol. Initially we wanted to use both quadrature decoders in the Teensy - one for the motor encoder and one for the wheel tachometer. Unfortunately the quadrature decoders shared pins with the dedicated CAN-bus pins. Fortunately the Teensy 3.6 should have the ability to talk over two CAN-bus channels: CAN0 and CAN1. We therefore simply designed for communication through CAN1 and hoped we were able to change the software CAN-bus library to effectively use the new CAN channel, despite all the other boards are using CAN0. We did not succeed. In order to counter this problem, we designed a new board with almost the same layout. The main difference was the switching of the communication port pins and the pins used for the wheel tachometer input. In addition a few minor mistakes where fixed, such as connecting extra input ports to unused pins.

The wheel tachometer signalers are now handled though interrupts, since the signal frequency is low enough that the resulting interrupts do not to significantly impact processor performance time. The new design proved to work as well as the old, and fixed the communication problems, and it has thus become the board which we expect to on the car in London.

We used an encoder mounted on the crank shaft of the motor to keep track of it's position. This position controlled the injection of fuel into the motor, the discharge of the inductor in the spark plug and the RPM of the shaft was used to control the electric starter motor.

The board has an onboard LED display for printing status messages and for debugging purposes, and room for four of thermocouplers for measuring temperatures of oil, water and exhaust.

The board also has various inputs for different sensors, which will be introduced later in this paper.

B. Software

Where it was possible, we wrote the code of this project in the Arduino language, with the Teensyduino extension necessary for compiling code the Teensy can run. The Arduino language is based on C++, and can import and use C++ and C code. This language is very basic, and we found that the available functions proved insufficient, so when it fell short various open source libraries written in C++, such as the TeensyDelay library[1] [5], where found, adapted, and extended as necessary. If this method also proved insufficient our own libraries or classes where implemented. This happened when using the hardware quadrature decoder, and when implementing a queue.

In classic microcontroller fashion the code contains three main elements: an initialization run once at startup, a main loop running continuously forever, and finally a number of interrupts for time sensitive functions. Please see fig. 1 for an overview of the code structure.

Communication describes two processes. The first is getting data from attached sensors and sharing this data over CAN-bus with other elements of the car, over bluetooth with an app designed by a different team or over Json with an UI if a PC is attached through USB. The second is taking commands from other control modules or the PC. These could be manual gear changes from the steering wheel sent over CAN-bus or requests for engine control values from the UI.

There are a number of engine constants, such as gear down speed and up speed to the autogear and the RPM at which the starter motor should stop running, which can be set from the PC and saved in the Teensy's EEPROM in order to store them between reboots. These can be set, through communication with a PC connected via USB serial.

IV. Implementation

A. Tracking engine position using a Quadrature encoder and decoder

A SCANCON type SCA50 quadrature encoder [1] is mounted on the motor's crankshaft, giving 720 ticks per full rotation of the crank shaft in the motor, with 720 A- and B-pulses a turn, shifted compared to each other. Each pulse corresponding to a change in angle by half a degree. A full rotation is marked by a special tick called the Z-pulse.

The first version of the position tracker was implemented by running four interrupts, three of which were used for each of the pulses given by the encoder. On a Z-pulse, the necessary calculations for finding the fuel management timings were done, and the fourth interrupt was called every $10\mu s$, checking if the desired position (defined by the amount of A and B pulses that had passed) was reached. This idea was implemented and did work, but was scrapped, as it used up way too much processing power. A much more processor efficient way to track the positioning was implemented afterwards.

[1]The Teensy Delay library allows for delay triggered interrupts that are not periodic, as they would be using the standard timer interrups. We used this to implement the ends injection and ignition, and to time the pause in the motor associated with gear shift, respectively explained in section IV-B and IV-C.

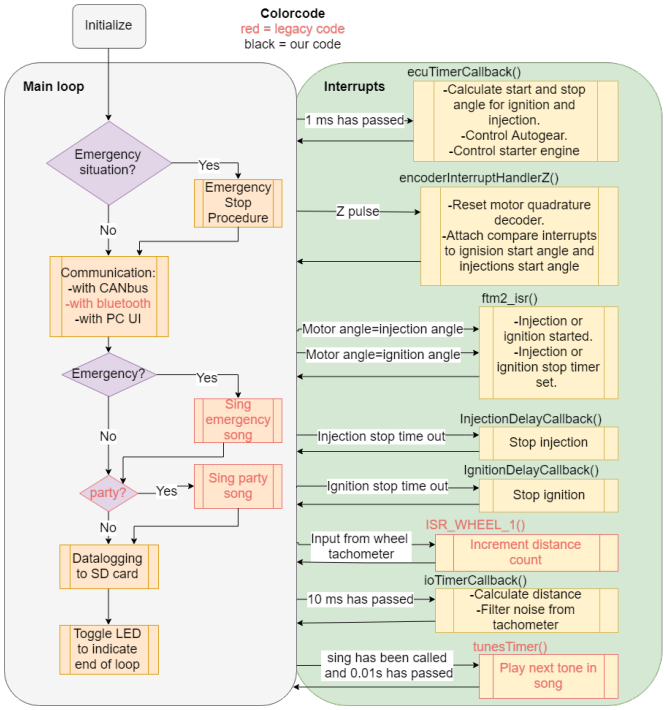

Fig. 1. All red text describes legacy code, taken over from previous students working on ecocar electronics. All black text describes code designed and implemented by us as part of the project. Please note that all functions that result form communication, such as emergency, gearshifting and turning on and off the autogear, happen under "Communication".

A hardware quadrature decoder was used, which unlike interrupts, makes no use of the processor. The hardware quadrature decoder is built into the Teensy's ARM chip[2], using a re-purposed timer, which also enables us to use interrupts when the encoder tick reaches certain counts; this is used for the fuel management and ignition.

B. Time critical fuel management

The injection and ignition of fuel is the most time critical aspect of the car, since they have to be timed very precisely with a high resolution, regardless of the motor's RPM. When the motor is running at it's highest RPM, a single revolution takes less than 10ms. To achieve the most precise timing, five interrupts were set up; one for the Z-pulse, and one for each of the four scenarios described in fig. 2. The interrupt which is triggered by the Z-pulse is a very small function, which calculates all of the timings and constants for the other interrupts; at which position to inject and ignite fuel, and for how long. Starting ignition and injection takes one interrupt each, while stopping the ignition and injection also uses one interrupt each. Starting the injection and ignition is done by using the interrupts on the quadrature decoder, to ensure they are activated at correct positions, while stopping the injection and ignition is done by a timer, since these are calculated to be a certain length in time, not in degrees.

It should be noted, that the timing of the injection cannot be exactly calculated, as the injection duration also changes with various values we cannot control or accurately measure, such as air temperature or humidity, to compensate for these

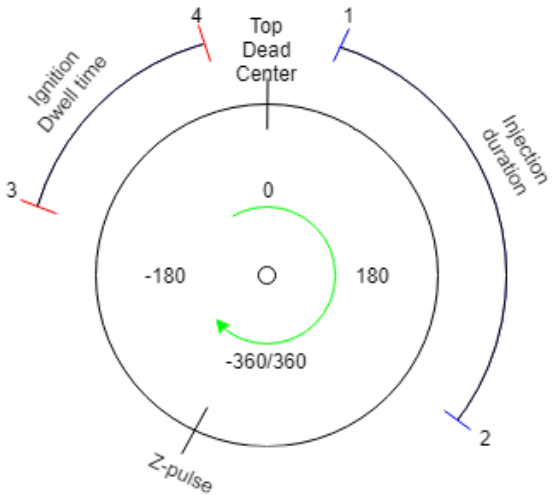

Fig. 2. A representation of a motor cycle, as read on the encoder. 1: Start of fuel injection. 2: Stop fuel injection. 3: Begin charging ignition coil. 4: Discharge ignition coil, giving a spark and igniting the fuel. Note that the Z-pulse is placed at an arbitrary angle, dependant on the physical installation of the encoder.

changes, a easily configurable variable is have been introduced. This is a simple factor by which the calculated injection time is multiplied by.

C. Controlling the gear

The gear of the car is mechanically controlled by a servo motor, with 3 available positions, two gears and one neutral. The servo is powered externally, and controlled by a pulse width modulated (PWM) signal from the Teensy, where the pulse width determines the target position for the servo; this method is generally considered to be the standard [4]. Generating the PWM signal is done by using a hardware timer, from a standard library. When receiving a command to change the gear, from either the CAN-bus or the connected PC, the Teensy will change the output to correspond with the request.

The automatic gear can be enabled or disabled by signals from either the CAN-bus or the PC. If the gear is changed while the motor has a high RPM, the highly different rotational speeds of the crankshaft and the gear wheel can break the gearing. To avoid this happening the gear change is regulated according to the RPM of the motor, if the speed is above a threshold, the motor is paused until the RPM have fallen below a threshold, the gear is then changed and the motor is resumed again.

D. Burn and Idle

We wanted the ECU to be able to control burn and idle in the same manner as the RIO, and so we have recreated this functionality to our best ability. The engine has two main driving techniques: burn and idle. These techniques combines the ignition, injection and gear control to start and stop the engine. Burn is used to accelerate the car up to driving speeds while gear control is handled either manually or automatically. The ECU will only continue to burn as long as the button on the steering wheel is pressed.

Idle is used to warm up the engine, and is controlled by a toggle switch, it keeps the gear in neutral while injecting more fuel than when burning. This is used before a race, since the motor can run on less fuel when warm. The ECU exits the idle state and stops the engine if the driver attempts to either change the gear or press the burn button. This doesn't trigger the emergency mode, but the driver needs to release all buttons before the ECU can perform a new burn or enter the idle mode again.

E. Emergency Stop

The car is equipped with a button on the outside, which can be pressed in case of an emergency where the driver is incapacitated and the car needs to be stopped. The outside button cuts the connection between the battery and the amplifier for the injection and ignition signals. To figure out if there is an emergency, the Teensy measures the voltage over the amplifier, using a built in ADC. If the voltage drops below a configurable threshold, the software enters an emergency lockdown mode, where the engine is stopped and this state is communicated to all the other boards. The only way to exit the emergency state is to make a full system restart.

Apart from checking if the outside button have been pressed, this system also stops the engine in case the battery voltage drops below an accepted threshold, for example if it isn't charged enough to drive or a fuse has been burned.

F. Sensors and sensor data logging

A mini SD card is mounted in the mini SD slot on the Teensy. The Teensyduino version of the Arduino SD library is used to manage it. It logs data into a buffer at a rate of 10 Hz. The buffer is then emptied into the SD card at a slower, changable, rate, and can be sent through serial when it receives a prompt from the user interface, to show live drive data.

The PCB has various sensor inputs besides the encoders, which the microcontroller reads, logs and reacts to. These can be seen in table II.

TABLE II
OVERVIEW OF SENSORS

Sensor	Description	Response
Brake	Detects if the driver is trying to brake.	Passes signal onto the brakes over the CAN-bus.
Thermos	Reads air, water and oil temperature.	None
Battery	Read the battery voltage.	Puts car in emergency state, if voltage it too low.
Lambda	Reads air ratio in the exhaust.	None
Wheel sensor	Reads the wheel sensor's signals	Car speed

V. Measurements and discussion

A. Feature test

The gear control was tested by setting the gear change threshold to $8\frac{km}{h}$ and accelerating the car to $10\frac{km}{h}$, so the

threshold were easier to reach by spinning the wheels. This way we could test if the automatic gear were able to pause the engine, change the gear and restart the engine. The test was repeated for the manual gear, but this gear wasn't changed exactly at $8\frac{km}{h}$ since the speed has no impact on the manual gear. The gear test showed that we were able to control the gear.

The car's sensors have been tested continuously as each sensor was implemented. The testing for each sensor consisted of reading the output, while changing the state, to see if the physical change was followed by a changed sensor reading. This included spinning the wheel to test the speed, heating up temperature sensors and connecting different batteries to test their voltage.

The process of sending data was tested by connecting the PCB to the car, and reading the sensor output using the screen found on the steering wheel; receiving data was tested by pressing the burn button found on the steering wheel, and seeing that the motor performed a burn.

Communicating with a PC was tested by connecting a PC and verifying the correct data was sent and received, by toggling an LED using signals from the PC, while simultaneously seeing the correct sensor data was sent to and displayed on the PC.

Logging was tested by running a burn 15 seconds after turning on the car, and then looking at the logged data to verify the logged timing matched the observed timing. The consistency of the log was tested by leaving the logging function turned on over night, arriving the next morning and seeing it had continuously and correctly logged various data from sensors, which resulted in a consistent stream of data, as expected.

The emergency stop was tested by idling the car, and then pressing the emergency stop button to stop the engine.

B. Measurements

We have measured the current going into the RIO and the motor board, and the current going into the ECU. This was done using a device of our own design which can log current with timestamps. The device is a Teensy with a low resistance Hall sensor and a lowpass filter to cleanse the signal, which according to our calculations give an accuracy of about $\pm 5mA$.

Most of our time critical code will only be used for a few seconds every round on the racetrack. Keeping in mind that we are not changing the current going into the starter motor and gear servo, we are only reducing the current going into the ECU. For this reason we are only showing the measurements of the motor board and RIO against the ECU, while both are inactive, and booting up. Our measurements can be seen in figure 3.

C. Discussion

From the data, it is clear, that the ECU as a whole uses less current than the RIO and motor board. the RIO and motor board use about $0.6A$ while the ECU uses $0.2A$. When we use the various more energy consuming features such as the starter motor or the gear servo, the ECU will at most use the same amount of energy as the RIO. This means, that the ECU is just as good as or better than the RIO, when comparing the power energy.

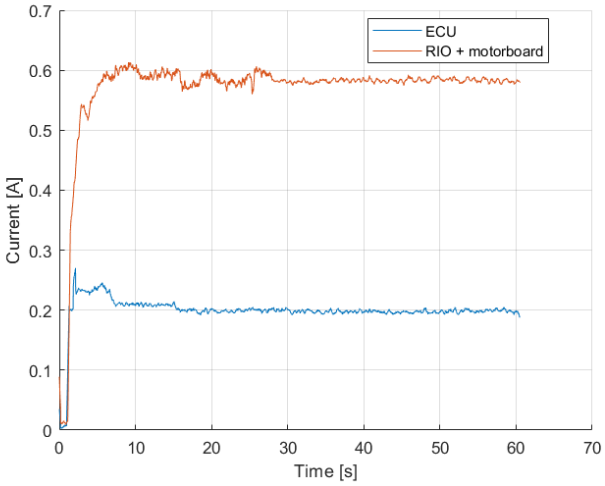

Fig. 3. The current going into the RIO + motor board and ECU, while not booting, the RIO is done turning on after 28s, which causes the noise at the beginning

On average, we can calculate how much fuel we expect to save, by using the formula provided by Shell[3]:

$$Gas[L] = \frac{Electrical\_energy[kJ]}{0.75 \cdot 0.25 \cdot 42900\frac{kJ}{kg} \cdot 0.7646\frac{kg}{L}} \quad (1)$$

The constants are provided by Shell, and have to do with efficiency equivalents, net caloric value and density values. Assuming that the EcoCar drives exactly like last year (16 kilometers in 39 minutes, resulting in a result of 460Km/L), replacing the RIO with our own system saves 23.5 Km/L in fuel equivalent, as the data and graphs show that the current consumption decreases by 390mA, when implementing the ECU instead of the RIO and motor board, which is a 67% decrease.

Looking at the other project specifications, the time critical injection and ignition were tested and proven to be fulfilled in these tests, since the car had to drive for the measurements to take place. We see that the embedded system can perform the time critical injection and ignition of fuel, as the motor was running, and it would stop if these operations were performed wrongly, and the system could control the starter motor, as it was able to start the car, and the starter motor is a vital component in this process.

D. Further work

Due to the fact that the our code logs data relatively slowly, it is very likely, that the car would, especially in an emergency situation, not log the data in the moments before it is turned off. This is not optimal, as some of the most crucial data, is the data before an unexpected shut down, for this reason, we will work on implementing a watchdog routine, which in case

of an unexpected shutdown would ensure, that as much data as possible was saved.

Further work would include other approaches to lowering the electrical energy used in other ways. When the current measurements were done on the different boards, an unidentified 200mA current was detected in the EcoCar. While this isn't related to ECU, it would be another way to reduce the energy consumption of the EcoCar which was the ultimate goal of this project.

VI. Conclusion

In this project we have implemented and coded an embedded engine control unit (ECU) for DTU Roadrunners' EcoCar. The ECU was designed to meet the requirements in the project specification.

Using interrupt service routines, injection and ignition was handled. This system is used when burning, as long as the system receives a signal from the driver. When burning, the motor is accelerated up to a speed high enough for the motor's own fuel burning to take over, this acceleration is handled by the starter motor. Once the car is driving, the gear can be handled either manually or automatically, in either case being controlled by a pwm signal. While the car is driving, it will communicate over CAN-bus with the other systems in the car such as the steering wheel, or over USB serial with PC through a UI. When communicating it sends out data read from the sensors, while also receiving commands such as burn or idle. Much of this data will also be logged on the on-board SD card. The entire process is stopped if the car is set into emergency, until the problem is handled.

The ECU also has a 67% smaller energy consumption than the previously used RIO and motor board - which we find to be a rather satisfying result. This decrease in current consumption corresponds to an extra 23.5km/L at the competition.

From this we can conclude the ECU is a strong improvement for the EcoCar, and ready for the competition in London.

References

[1] SCANCON Industrial Encoders. *Type SCA50 datasheet*. 2014. [Rev 1.5].

[2] Inc. Freescale Semiconductor. *K66 Sub-Family Reference Manual*. 2015, pp. 1252–1257. [Rev 2].

[3] Shell Eco-Marathon Shanna Simmons. *Official Rules Chapter I*. 2018.

[4] Jan Malášek. *Servo control interface in detail*. URL: https://www.pololu.com/blog/17/servo-control-interface-in-detail. (accessed: 14.06.2018).

[5] Lutz Niggl. *Teensy Delay*. URL: https://github.com/luni64/TeensyDelay. (accessed: 18.06.2018).

[6] Paul J. Stoffregen. *Teensy Technical Specifications*. URL: https://www.pjrc.com/teensy/techspecs.html. (accessed: 18.06.2018).