DSA3102 Intelligent Real-Time Systems Project

Kent Odde, Stian Onarheim, Tarald Vestbøstad

October 27, 2020

Contents

1 Abstract **3**

2 Introduction **3**

3 Specifications **3**

4 Technologies **5**
- 4.1 The Ada Programming Language . . . 5
- 4.2 Arduino Nano 33 BLE Sense . . . 5
- 4.3 Servo Motors . . . 6
- 4.4 Ultrasonic Sensors (HC-SR04) . . . 6
- 4.5 The Turnigy Trooper . . . 7

5 Toolchain & Testing **8**
- 5.1 Ada on the Arduino . . . 8
  - 5.1.1 Ada Drivers Library . . . 8
  - 5.1.2 Ravenscar runtime Library . . . 9
- 5.2 Flashing the Arduino . . . 10
  - 5.2.1 Our Ada Drivers Library Fork . . . 10
- 5.3 Minimum working examples . . . 11

6 Implementation **12**
- 6.1 Hardware . . . 12
- 6.2 Software . . . 12
  - 6.2.1 Architecture . . . 13
  - 6.2.2 Scheduling . . . 14

7 Various problems we've encountered along the way **17**

8 Final Product **17**

9 Conclusion **18**

References **19**

10 Appendices **20**
- 10.1 Arduino Nano 33 BLE Sense Pin Layout . . . 20

1 Abstract

This is our submission for the semester assignment in the course Real-Time Systems DSA3102, fall of 2020.

2 Introduction

The semester assignment as given, was to design and implement a real-time system, which would run on an Arduino Nano 33 BLE Sense, with the requirement that it would be programmed in Ada.

A broad definition of a real-time system, is a system that reacts to external stimuli, make calculations, and performs some output, within a given deadline. In other words, the challenge here is not only to write a logically correct program, but also that it finishes all tasks in time. Which naturally increases the difficulty.

This report will explain our project choice as well as the design. It will give a detailed description of the challenges, especially concerning running Ada on an Arduino Nano. Finally we will explain how the final solution works, and how we can prove that it is safe, and that the system will be able to meet all its deadlines.

3 Specifications

Very early on, we decided on building an autonomous car. We initially had a plan on using a Hololens to interact with the car in some way, but we quickly moved away from this, in order to reduce the overall complexity.

The car should not only drive, but it would also have a bottle of antibac on top of it, with a sensor that would stop the car, and provide a person with a few drops of disinfecting liquid.

The initial design is illustrated in Figure 1.

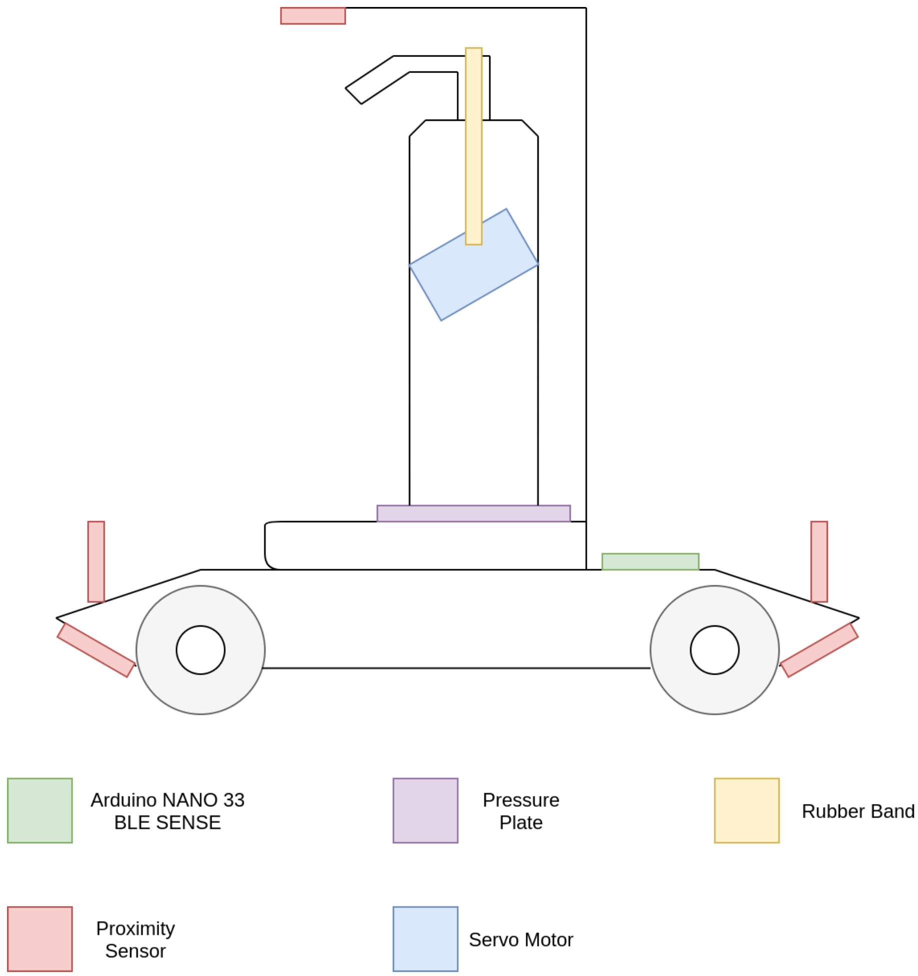


Figure 1: Prototype drawing of our vehicle

As the car would provide people with antibac, we naturally wanted the car to be able to drive on top of a table. That is why it had down-pointing sensors, to be able to know when it would drive of an edge. However, after receiving the car from our professor, we quickly found, that the navigation system we would need to implement would be way too comprehensive, given the small time frame of the assignment.

Throughout the project, the system went through several iterations, and as we met challenges along the way, the antibac idea was abandoned. The final design is illustrated in Figure 2.

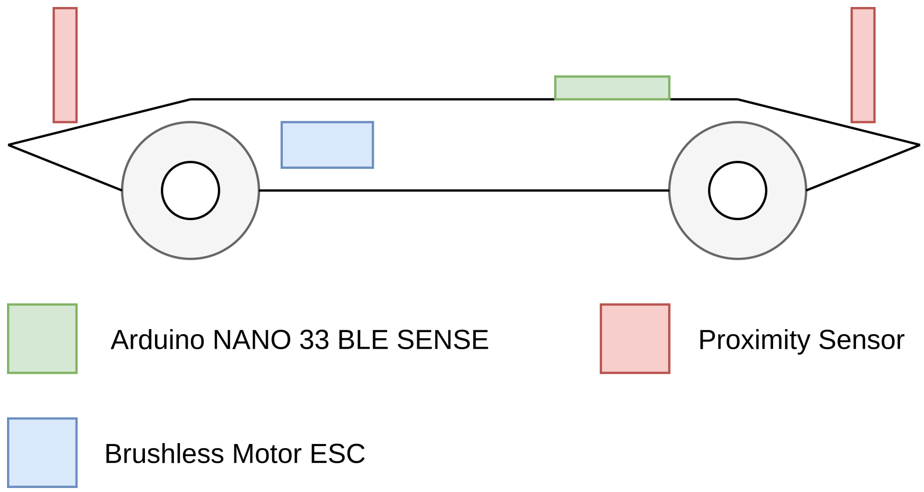


Figure 2: Drawing of final vehicle design

4 Technologies

4.1 The Ada Programming Language

Ada was development for the American department of defence in the 80's in order to try and get a standard programming language for all of their projects. As its main use-case was intended to be on critical military systems, the language would be designed from ground up, to accommodate the creation of robust and safe software. This resulted in a language with strong typing, a strict compiler, and a syntax that is meant to improve the readability, resulting in a easier maintainability in large projects. Today Ada is still mostly used in the defence and aerospace sector, and is very suitable for embedded systems.

4.2 Arduino Nano 33 BLE Sense

The Arduino Nano 33 BLE Sense is a small microcontroller based on the nRF52840 microcontroller. It comes with plenty of peripherals:

- Bluetooth
- IMU
- Microphone
- Gesture/Light/Proximity detector
- Barometric pressure gauge

- Temperature and humidity gauge

4.3 Servo Motors

Usually, when programming Arduinos in C, one makes use of a library to control servo motors. However, as we were doing this in Ada, we would have to write our own library. Servo motors are controlled using a pulse width modulated signal (PWM). A PWM has a fixed period, where the amount of time the signal is high defines the signal.

In the case of a servo motor, the fixed period is 20 milliseconds, and the signal must be high in a range between 1 and 2 milliseconds. Sending a 1.5 milliseconds high signal to a servo engine, is defined as the input for outputting neutral position. If we send a 1ms signal, the servo will move to the extreme left, and 2ms will output the extreme right. An illustration of this can be seen in Figure 3.

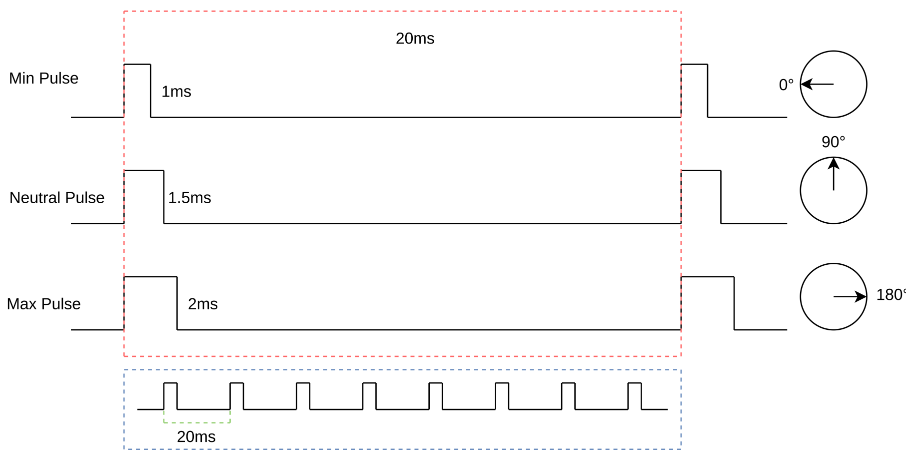


Figure 3: PWM for a Servo

4.4 Ultrasonic Sensors (HC-SR04)

As the ultrasonic sensors has both input and output, they are a bit more complicated than the servo motors. They have a TRIG pin and an ECHO pin.

The TRIG is its input, where we send a signal letting it know that we want to perform a measurement. As illustrated in Figure 4, it needs a sequence of a low signal for 2 microseconds, then a high signal for 10 microseconds. The ECHO pin responds with a signal, that when interpreted correctly, will tell us the distance to the closest object in the sensors facing direction. When the ECHO pin turns high, we initialize a timer which is active until the signal turns low. Dividing the time by 58 gives us the distance in centimetres.

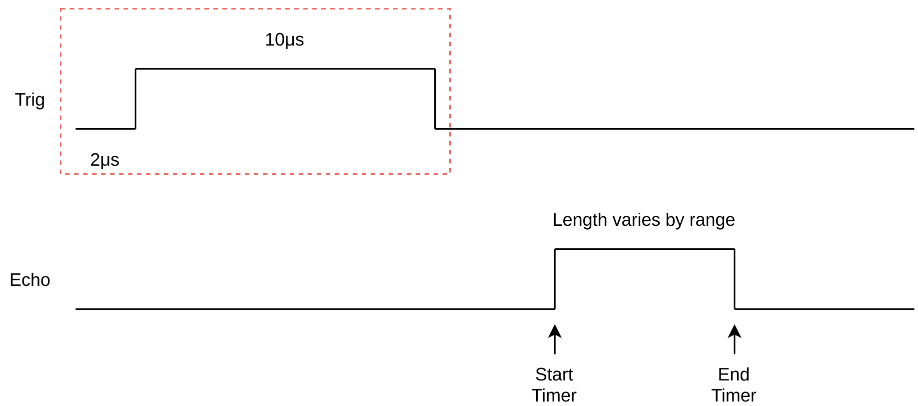


Figure 4: Trig and Echo pulse for the HC-SR04

4.5 The Turnigy Trooper

The Turnigy Trooper is the name of the car that our professor provided us with. It was originally a remote controlled car, but a lot is removed, so nothing but the chassis, motors and wheels remain.

A diagram of the internal components on the car can be seen in figure 5
The Receiver from this diagram has also been removed from the car.

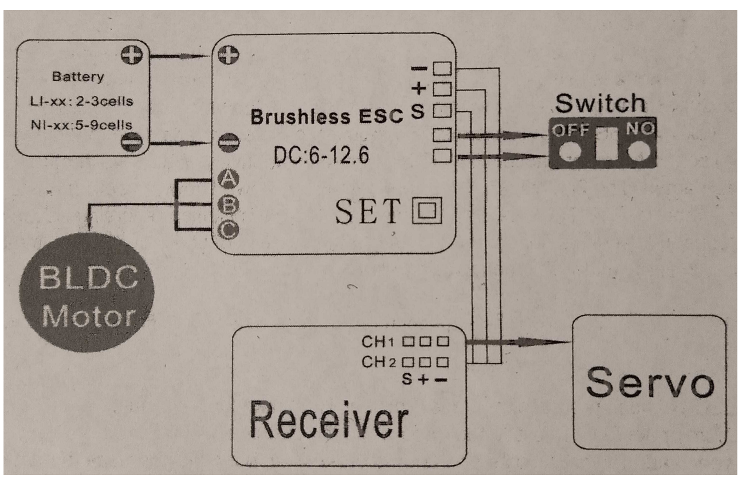


Figure 5: Diagram of car internals from manual

The servo motor for the steering is traditional one, which takes the PWM that was described previously.

The motor which will provide us with propulsion however, is actually a BLDC motor. Luckily for us, the ESC, provides a PWM interface for us, so that we can treat the motor as a fully rotating servo motor.

As in the case with regular servo motors, we send it a standard PWM signal, but in this case, the high time of the signal does not refer to a static degree, but to the speed of rotation. As can be seen in figure 6, a high signal of 1.5ms means that the motor will not rotate. If we send a 2ms signal it will rotate forward at the highest speed, and the opposite applies for 1ms.

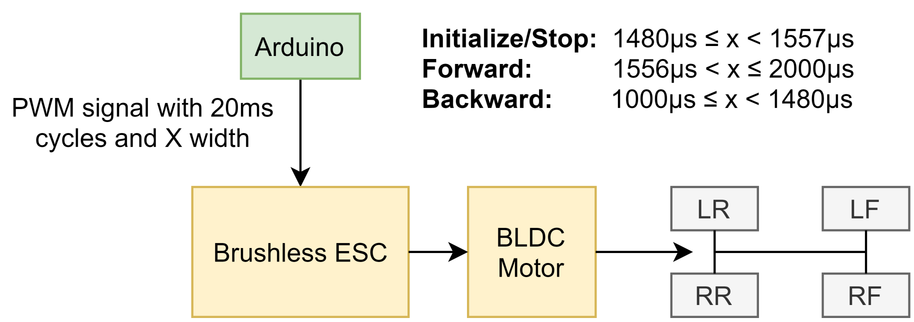


Figure 6: Block diagram of signal path from Arduino

The ESC in the Turnigy Trooper is also possible to program quite extensively, and after receiving the manual, we played around with the settings for a bit, before deciding that the default settings would suit us fine.

5 Toolchain & Testing

5.1 Ada on the Arduino

As we were told early on that almost no one had programmed Ada on the Nano, we knew early on that this would be a challenge. We didn't really know were to begin, until our assistant professor, Steven Bos, pointed us in the right direction.

5.1.1 Ada Drivers Library

The *Ada Drivers Library*, is a collection of drivers and examples for programming Ada on a wide range of microcontrollers, but the Arduino Nano was **not** one of them. However, as Steven Bos had discovered, it does provide drivers for the *NRF52_DK*. This board has the same CPU, *Arm Cortex-M4* and is a similar board as the Arduino Nano. Steven presented a working solution for a classical led blink example using this library, and after that we at least had a path to follow, as well as expanding on.

We knew that it would take quite a few hours to adjust the existing drivers, to suit the Nano. In the hope of being able to contribute to the open-source community, we forked the *Ada Drivers Library* , and used their script to build basic board files for the Nano, based on our inputted specifications.

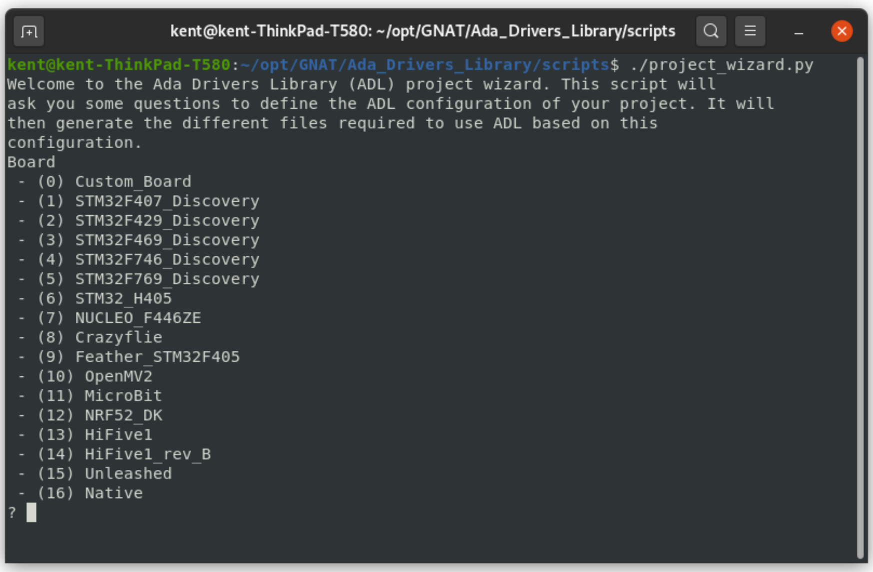


Figure 7: Screenshot of the project wizard from Ada Drivers Library

5.1.2 Ravenscar runtime Library

When flashing a microcontroller, you also need to have a runtime library. As a microcontroller is limited in space, you usually won't include everything that the language normally offers.

For most of the microcontrollers in the *Ada Drivers Library*, there exists three kinds:

- ZFP (Zero footprint library)

- Ravenscar SFP (Small footprint library)
- Ravenscar Full

For the NRF52840, the only runtime library that was provided to us out of the box, was the ZFP. The ZFP has (as the name implies) almost none of the features we would need. The most crucial one being the ability to have multithreading (called tasks in Ada). As much of this course's theory is based around having several tasks, and how to schedule them etc, a sequential approach would not do.

In order to have access to the full Ravenscar library, we cloned *bb-runtimes* from GitHub, and used a script they provide in order to build it. This proved a tedious task, and there were quite a few errors within the script that had actually already been fixed by the developers, but not released at that time.

Luckily for us, this eventually worked out okay. The Ravenscar library still has its limitations which would change the way in which we structured our software. For instance, it does not allow entry points into tasks. This means that we would not have access to dynamic tasking, and that all tasks would have to be created statically. It also meant that we would not (without extreme difficulty) be able to take advantage of the multiple core processor on the Nano.

5.2 Flashing the Arduino

After getting our software toolchain together, we started trying to flash our Arduino with a blinking test. As we had not received the JLink debugger yet, we wanted to see if this was possible to accomplish using only USB. After spending way too many hours on this, we found that this would not be feasible.

When we received the JLink debugger, we would unfortunately also have to wait for a 3D-printed clamp holder, as the soldering on the Nano had proved to be quite unstable for some of the other students. Our eagerness and impatience luckily drove us to try and flash the Nano whilst holding wires to the pins on the Nano by hand. This took a lot of trial and error, but eventually some of the members of the group developed great skills for this technique.

When we finally received the clamp holder a few weeks later, flashing turned into a dream, and we could with much greater ease debug our program, as it no longer required someone manually holding the wires in place during runtime.

5.2.1 Our Ada Drivers Library Fork

As mentioned, we built source files for our own drivers for the Arduino, using the scripts provided by the *Ada Drivers Library*. Most of the files required only small changes, but the pin layouts are quite different on the Arduino compared to the NRF52_DK.

The nRF52840 has two ports, P0 has 32 pins while P1 has 16 pins.
We found the P1 base address for the nRF52840 seen on page 23 in the datasheet [9]. Without it we were only able to use the pins with P0 seen on figure 10.1. After adding the base address to our Drivers Library and the option to chose between 0 and 47 pin number we were able to make all the pins function.

We also wanted to include the ability for analogue writes and interrupts, but this proved to be overly complicated and time-consuming to justify prioritization. More on this later.

As the code is quite extensive, we have chosen not to include it in the appendices, but our fork of the *Ada Drivers Library* can be seen on GitHub.

5.3 Minimum working examples

During the difficulties of flashing the Arduino, we spent our spare time writing both the full program, but also creating small testing suites. This would help us more efficiently test all the components, and how we interfaced with them, in a limited environment.

This helped us find quite a few bugs early on, as well as figuring out what would, and would not work.

For instance, the Ada Drivers Library has a servo motor example using analogue write and interrupts. We initially tried doing it this way, to ensure steady PWMs for the servo motors, but as mentioned this proved to be much harder than anticipated.

We developed testing projects to test the servos and the ultrasonic sensors. To test every pin on the board, we developed a blinking example, turning every pin on and off. The code for these tests can be found in our GitHub repository[5].

Because of our approach of test early, fail early, we were able to develop workarounds for this at an early stage.

6 Implementation

6.1 Hardware

The car has a 7.4V lithium battery, which initially powered the ESC and the motor. We chose to power the Arduino with the lithium battery as well, but the sensors require 5V. We received a converter from Henning and used it with a 9V alkaline battery to power the two sensors.

The sensors have two connections to the Arduino, in and out while the ESC has a single connection.

In Figure 8 we have drawn our wire setup, with common ground for both power sources.

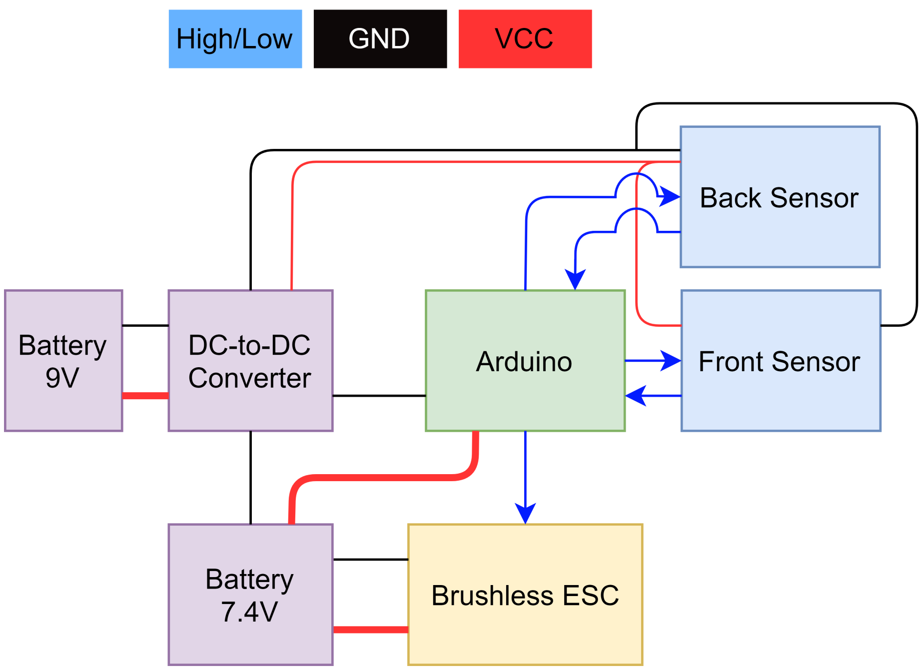


Figure 8: Block diagram of our hardware wiring

6.2 Software

Our software is written with the Ada language. To function it is heavily dependent on the open source Ada Drivers Library. To keep track of time and execute delays, we have used the Ada Real-Time library. As there were no libraries for the distance sensor and the servo, we have written them ourselves.

6.2.1 Architecture

The system architecture is illustrated in Figure 9. The Distance Sensor Controller keeps a hold of the ultrasonic sensors. It operates them all in one task, reading the values sequentially and stores them in the controller. The servo controller was initially made to control the steering servo, the ESC and the two servos used in operating the dispenser. As the project's deadline got closer, we had to remove the three servos and were left with the ESC. Even though the Brushless Motor ESC is technically not a servo, it uses the same operating signals as the other servos. At the time it made sense to fit it in with the others under the same controller.

The main component is the Vehicle controller. It reads the sensor values from the Distance Sensor Controller and uses them to calculate the next sate logic. Each state writes their own unique values to the Servo Controller. These values indicates the signals sent to the ESC, altering its speed.

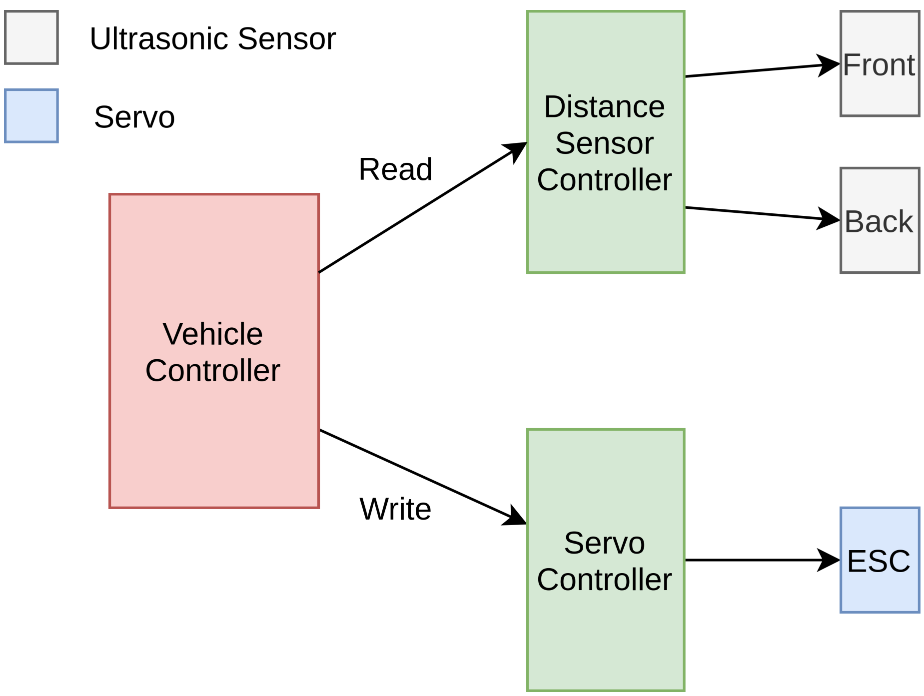


Figure 9: Block diagram of our system architecture

The system is built with a finite state machine in mind. Its states and conditions are illustrated in Figure 10. A built-in led on the Arduino is actively blinking the current state's colour, as described by the state's colour in Figure 10.

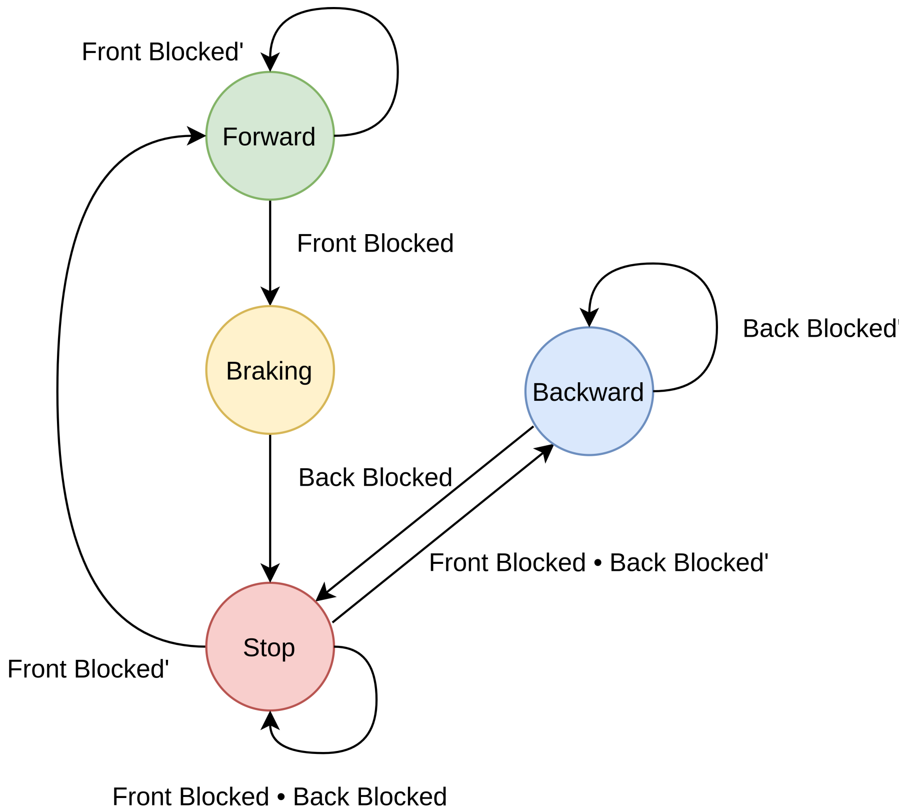


Figure 10: Finite State Machine

6.2.2 Scheduling

As the software for the system consists of several concurrent tasks, it is crucial to make sure that the system is schedulable, and is able to reach all of its deadlines.

As mentioned previously, we were not able to get interrupt signals to work on the Arduino. This increases the importance of our scheduling even more, as we have hardware components that rely on steady and exact signal interaction. Where as the ESC is the the most crucial component.

The simplest and most suitable scheduling scheme for our system, we found to be fixed-priority scheduling (FPS). This is a scheme where all the tasks are given a priority at compile-time, and is very simple to implement.

The prioritization of the tasks in our system, can be seen in the table below.

Task	Priority
Engine	4
Compute	3
Sensors	2
Lights	1
Main	0

In order to not ever break the PWM sent to the ESC, the task that controls the engine has been given the highest priority. Next is the task that makes the computations. If we had access to interrupts, this would naturally have had the highest priority.

The sensors and lights comes next, and finally the main loop. The good thing about FPS, is that one can add arbitrary tasks and assign them a priority of 0. In our case the system will run forever, so the main loop is just a loop with a null statement, which will never still any CPU clock cycles.

We quickly found out that if a high priority task like Compute didn't have any delays, it would lead to starvation on the lower priority tasks. First we played around with these numbers a bit, before settling on values that seemed to work.

However, a bit of trial and error is not good enough for a real time system. We needed to be able to prove that our system will never miss any deadlines.

In order to find out if our system was schedulable, we decided to make use of the utilization test. As we had five tasks, the formula states that the combined utilization factor must be:

$$U_{Sum} < 74.3\%$$

The utilization factor for a given task is given by the formula:

$$U_i = \frac{C_i}{P_i}$$

We found the computation time for each task in Gnatstudio, by opening the assembly code window during debugging. Here we went through each task step by step, and counted the lines. In cases where the code had the possibility of moving to different branches, we always chose the *worst-case* option. Merely counting lines in assembly is probably not 100% accurate, but for our use, this was good enough.

The periods were set in advance based on guesses. Luckily for us we were able to see just how effective this tool is when developing real-time systems. When we started the scheduling analysis, we were quite satisfied with our system, even though it had a few bugs, which would appear in irreproducible conditions. The reason became clear when we saw that we had a utilization factor of 0.92.

We played around with the periods of the Compute and Sensor task, as well as adjusting the computation time for the sensors by letting the task timeout on distances bigger than 1 meter. This led to the table that can be seen below.

Task	Period(T)	Comp. time(C)	Priority(P)	Util(U)
Engine	20ms	4094ns	4	0.0002
Compute	16ms	844ns	3	0.000052
Sensors	16ms	11600190ns	2	0.72
Lights	200ms	13438ns	1	0.000067
Main	N/A	16ns	0	0

The utilization factor for this system is

$$U_{Sum} = 72.0319\%$$

which is lower than the limit of 74.1% for five tasks.

After implementing these changes in the code, we noticed that the bugs we have previously, had disappeared. We ran the system for five minutes continuously, exposing the sensors to all kinds of behaviour without any errors.

Even though our system was sure to meet its deadlines, we wanted to also try and prove this using response-time analysis.

The equation for this is given by:

$$R_i = C_i + \sum_{j \epsilon hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

When calculated with a C++ implementation, we found these results:

Task	Period(T)	Comp. time(C)	Priority(P)	Response Time(R)
Engine	20ms	4094ns	4	4094ns
Compute	16ms	844ns	3	4938ns
Sensors	16ms	11600190ns	2	11605128ns
Lights	200ms	13438ns	1	11618566ns
Main	N/A	16ns	0	N/A

The fact that all the response times are much lower than the deadlines (periods in this case), means that our system has passed the response time analysis.

Because our system eventually passed the utilization test and the response time analysis, we can say we pretty high confidence that all our tasks will meet their deadlines.

7 Various problems we've encountered along the way

In the list below are some problems we have encountered during our development phase from 17.09.2020 - 23.10.2020. Most of these issues have occurred using Linux.

- Flashing to the Arduino board without a debugging probe might be impossible.
- After installing software for the Segger J-Link debugger from their official webpage, a new .rules file for the debugger appears in */etc/udev/rules.d* containing the necessary information for the debugger to be found by *pyocd list*. This has only been successful with one of our computers. A workaround is to run pyocd with sudo privileges.
- When trying to flash to the board without having the cables soldered, errors in the list below have appeared as a consequence of unstable hands.
  - Unable to start CPU core.
  - Target System has no power.
  - Unspecified Error.
  - J-Link is already open.
- Including Arduino_Nano_33_Ble_Sense.Time makes the microcontroller crash during runtime.
- Only Digital I/O works. Possibly because Analogue relies on interrupt/-timers which seem broken in our drivers or runtime.
- Generating a pulse accepted by the servos and ultrasonic sensor is more challenging than setting pins to low/high and using a delay.
- Using a copied bb-runtime library will not work correctly on other computers under debugging. They must be generated on the debugging environment.

8 Final Product

To sum it up, the final product is a car, that will drive forwards and backwards, and avoid hitting walls.

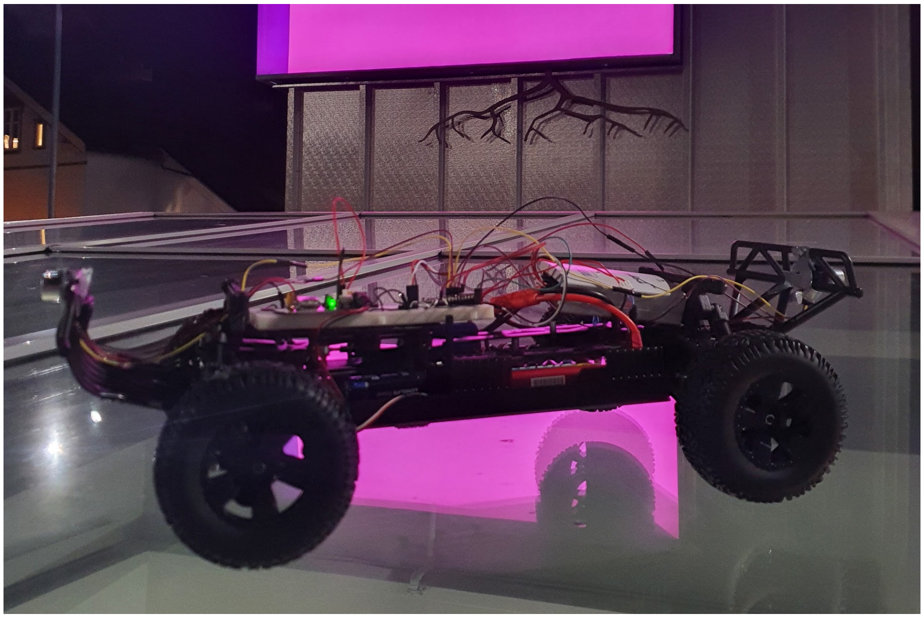


Figure 11: Final product

9 Conclusion

The purpose of this assignment was to try and implement the theory relating to real-time systems in practice, in order to better understand it. In spite of the challenges along the way, and how we had to downscale the project several times, we would still say that this goal was accomplished.

One can read about the difficulty of making a system meet all its deadlines, how scheduling works and what happens when a thread is starving. However, nothing can beat the feeling of actually seeing it in real life. As mentioned, we had several tasks that didn't deliver the output we expected, and many times this was due to the task not getting any time on the CPU. This was a powerful experience.

Learning to program in Ada along the way, as well as building the the Ravenscar run-time library and the library for the nRF52840, was a great experience, and is guaranteed to prove very valuable in the future.

References

[1] Ada_Drivers_Library, github.com. https://github.com/AdaCore/Ada_Drivers_Library. Accessed: October-20.

[2] Arduino Nano 33 BLE Sense, arduino.cc. https://store.arduino.cc/arduino-nano-33-ble-sense-with-headers. Accessed: October-20.

[3] bb-runtimes, github.com. https://github.com/AdaCore/bb-runtimes. Accessed: October-20.

[4] JLink Arduino Nano setup, segger.com. https://wiki.segger.com/Arduino_Nano_33_IOT. Accessed: October-20.

[5] Stykk-Gruppen, github.com. https://github.com/stykk-Gruppen/rts-project. Accessed: October-20.

[6] turnigy trooper, hobbyking.com. https://hobbyking.com/en_us/turnigy-trooper-sct-4x4-1-10-brushless-short-course-truck-arr.html?___store=en_u. Accessed: October-20.

[7] Alan Burns and Andy Wellins. *Analysable Real-Time Systems, Programmed in Ada*. Createspace Independent Publishing Platform, 2009.

[8] Henning Gundersen. DSA3102 slides, canvas. https://usn.instructure.com/courses/22239/files. Accessed: October-20.

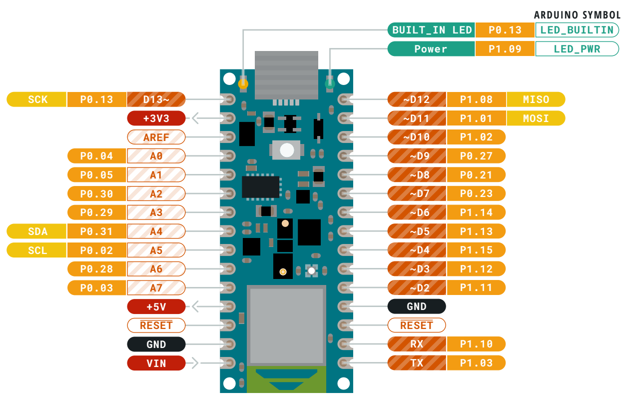
[9] Nordic Semiconductors. *nRF52840*, 2 2009. v1.1.

10 Appendices

10.1 Arduino Nano 33 BLE Sense Pin Layout

ARDUINO

ARDUINO
NANO 33 BLE SENSE
STORE.ARDUINO.CC/NANO-33-BLE-SENSE



Ground

Power

LED

Internal Pin

SWD Pin

Digital Pin

Analog Pin

Other Pin

Microcontroller's Port

Default

MAXIMUM ouput current per pin is 15mA

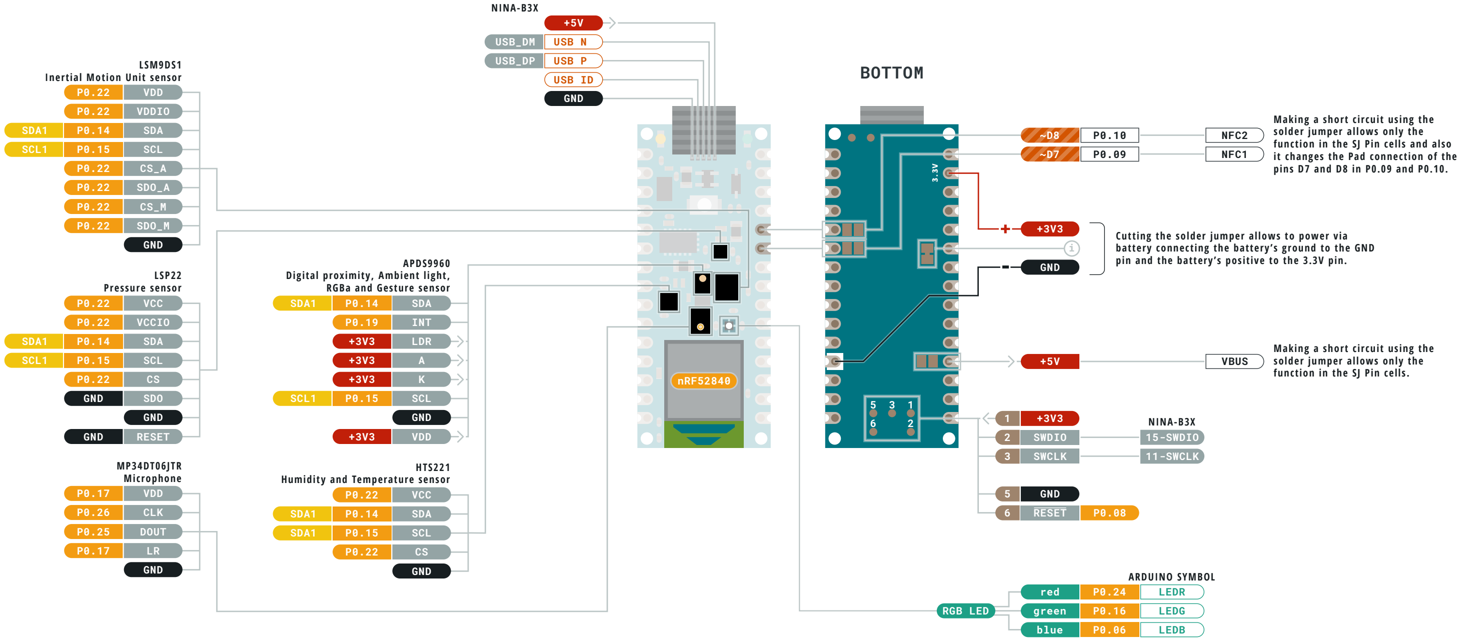
MAXIMUM input current per pin is 5mA

MAXIMUM external current is 25mA for the sum of all GPIO currents and the current being drawn from VDD

VIN 5-21 V input to the board.

ARDUINO.CC

Last update: 16/09/2020



Ground
Power
LED
Internal Pin
SWD Pin

Digital Pin
Analog Pin
Other Pin
Microcontroller's Port
Default

SJ Pin
Making a short circuit using the solder jumper allows only the function in the SJ Pin cells.

MAXIMUM ouput current per pin is 15mA

MAXIMUM input current per pin is 5mA

MAXIMUM external current is 25mA for the sum of all GPIO currents and the current being drawn from VDD

VIN 5-21 V input to the board.

ARDUINO.CC

Last update: 16/09/2020