ISN

test

Hello

Title

Kent Odde, Stian Onarheim, Tarald Vestbøstad

October 24, 2020

Contents

1 Abstract

This is our submission for the semester assignemnt 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 calucations, 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 desinfecting liquid.

The initial design looked like this.
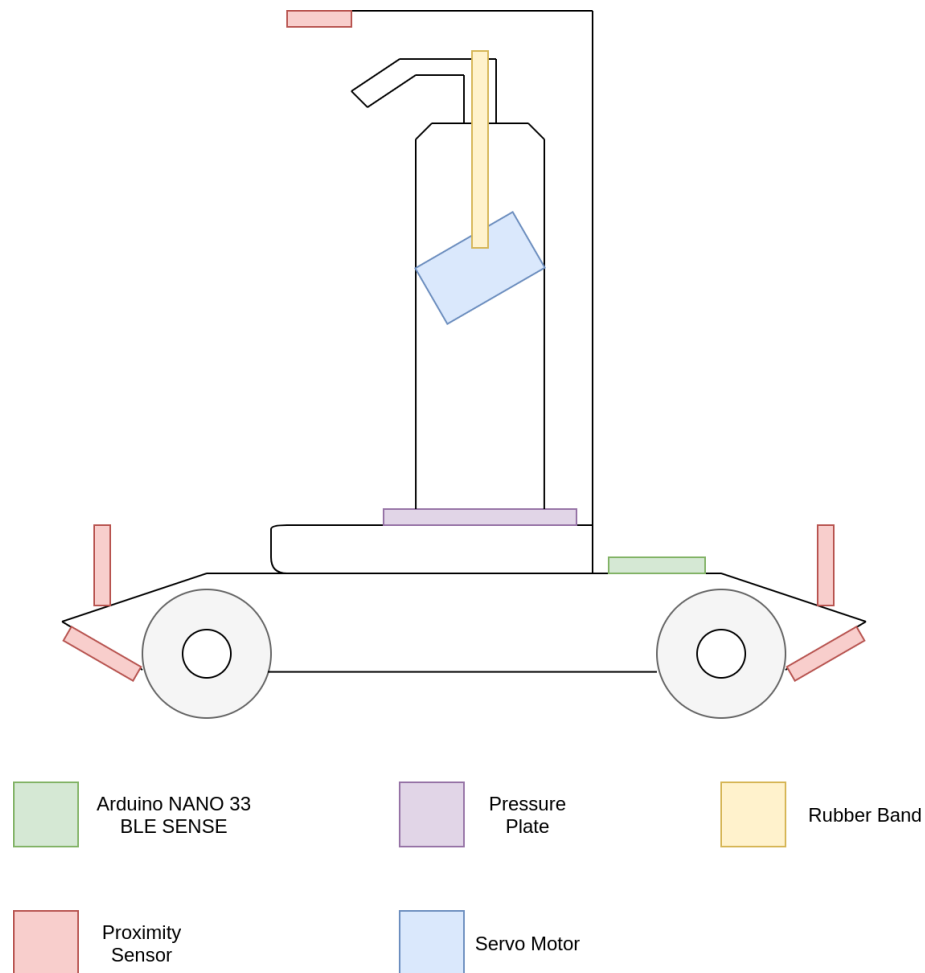
Figure 1: Prototype drawing of our vehicle

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 downpointing 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 to comprehensive, given the small timefram of the assignment.

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

Arduino NANO 33 BLE SENSE   Proximity Sensor

Brushless Motor ESC

Figure 2: Missing Caption.

4   Technologies

4.1   The Ada Programming Language

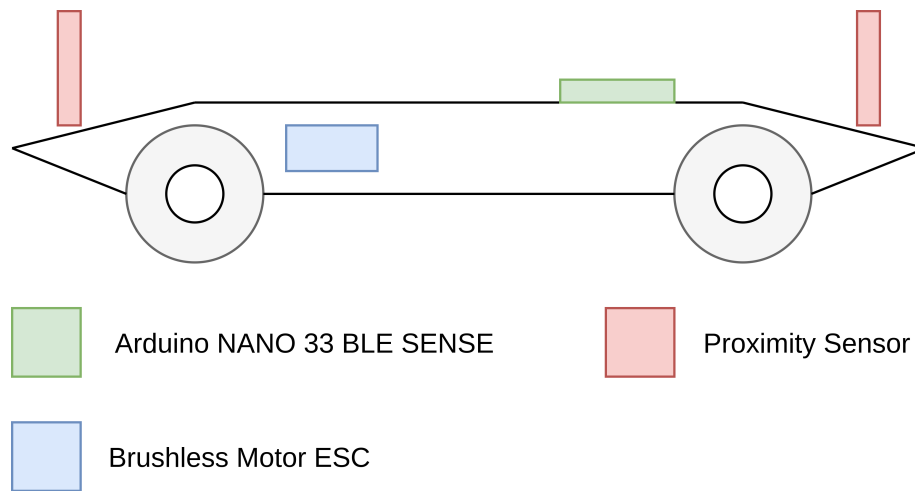Ada was development for the American department of defence in the 80's in order to try and get a standard programming language for all of their projects. As its main use-case was intended to be on critical military systems, the language would be designed from ground up, to accomodate the creation of 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

Her må noen andre finne på noe lurt***************************Kent

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 outputing 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.
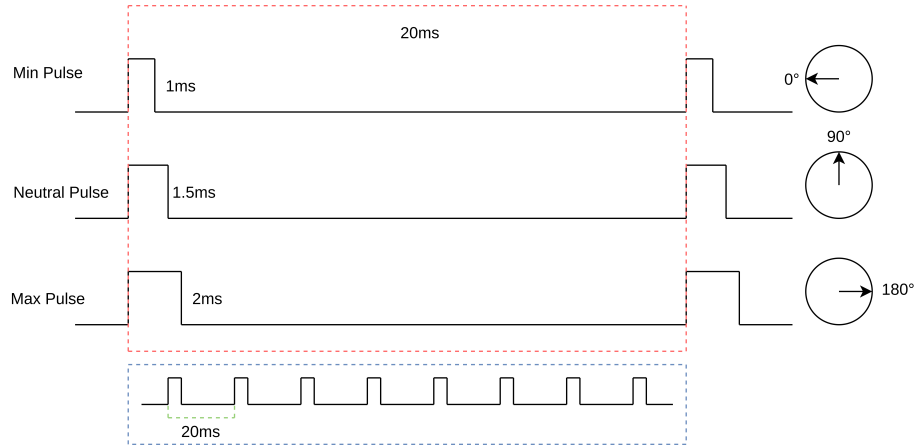
Figure 3: Missing caption.

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.
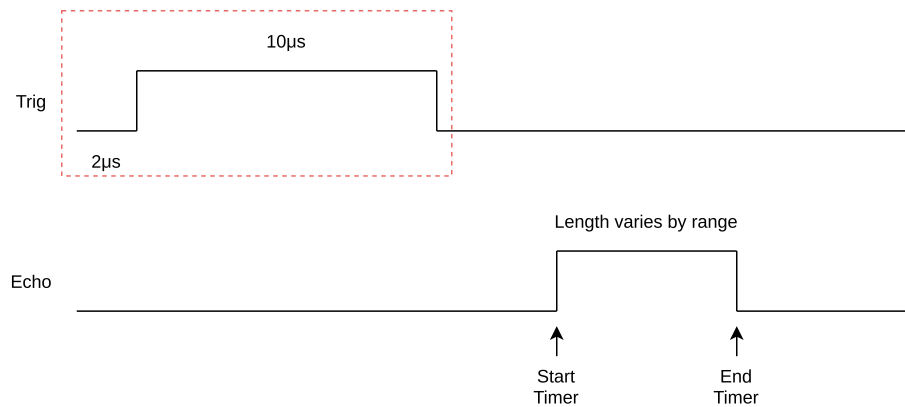
Figure 4: Missing caption.

The signal.... ****Her må du skrive litt stian********

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 components on the car can be seen in figure **??**
****Diagram av ECD, servo osv fra manual***********
The servo motor for the steering is traditional one, which takes the PWM that was desribed previously.

The motor which will provide us with propulsion however, is actually a DC 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 deegree, but to the speed of rotation. As can be seen in figure 5, 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 5: Missing caption.

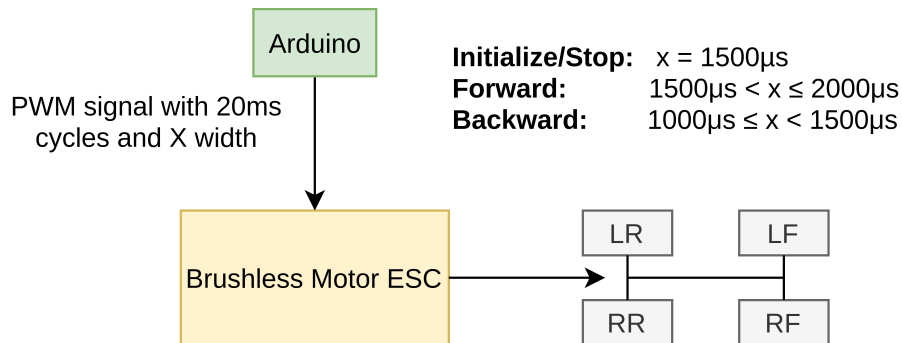The ESC in the Turnigy Trooper is also possible to program quite extensively, and after receiving the manual, we played around with the settings for a bit, before deciding that the default settings would suit us fine.

5 Toolchain & Testing

5.1 Ada on the Arduino

As we were told early on that almost noone 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 micro-controllers, however 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, *NRF52840*, 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 be quite a few hours of adjusting 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.

5.1.2 Ravenscar Run-time Library

When flashing a microcontroller with code, you also need to have a run-time library. As a micro-controller is limited in space, you usually won't include ev-

erything that the language normally offers.

For most of the micro-controllers 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 run-time 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 Ravenscar full 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 blinkin test. As we had'nt 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 impatiance 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, as can be seen in the picture below:
********Bilde av flashing med manuell holding av ledninger******************
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 Arduino library

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 pins on the Arduino range from ... . Mer om pin layout fix jfkdl-sjfdklajfkdlasjfkldsajfkdlasfjdklasfjdklasfjdklasfjdkla

TARALD**

We also wanted to include the ability for analog writes and interrupts, but this proved to be overly complicated and timeconsuming 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 analog 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. Because of our approach of test early, fail early, we were able to develop workarounds for this at an early stage.

The code for these tests can be seen in appendix **??**, and an image of the testing can be seen below.

Bilde av Servoer og sensorer?????***********

6 Implementation

6.1 Hardware

6.2 Software

6.2.1 Architecture

6.2.2 Scheduling

7 Final Product

To sum it up, the final product is a car, that will drive forwards and backwards, and avoid hitting walls.
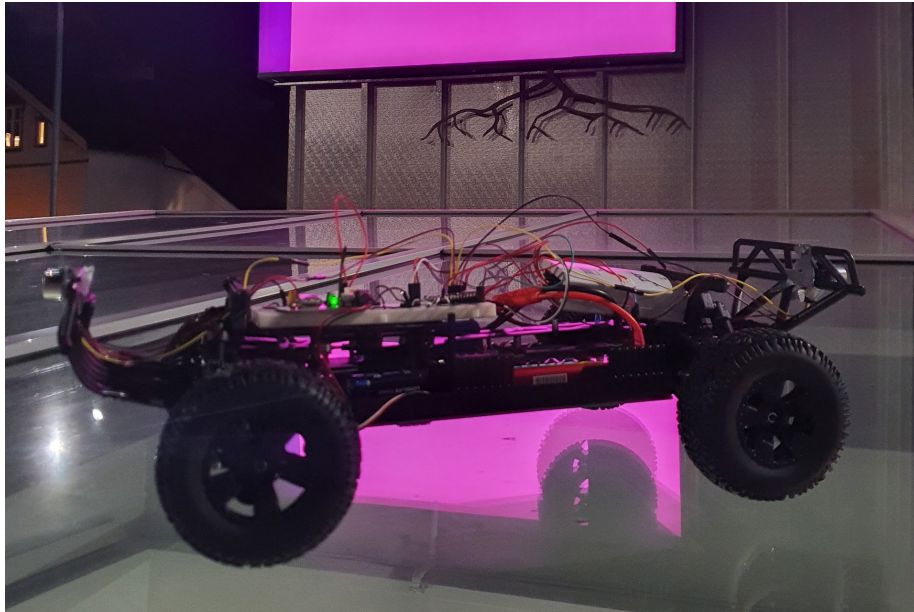
Figure 6: Missing caption.

******Images**********

8   Conclusion

The purpose of this assigment 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, nothing can beat the feeling of actually experiencing it. As mentioned, we had several tasks that did'nt deliver the output we expected, and many times this was due to the task not getting any time on the CPU. That is 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 great, is guaranteed to prove very useful knowledge 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] 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.

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

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

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

9 Appendices

9.1 Stacktrace

9.2   Arduino Nano 33 BLE sense PIN layout

ARDUINO SYMBOL

| BUILT_IN LED | P0.13 | LED_BUILTIN |
| Power | P1.09 | LED_PWR |

SCK	P0.13	D13~		~D12	P1.08	MISO
		+3V3		~D11	P1.01	MOSI
		AREF		~D10	P1.02	
P0.04	A0		~D9	P0.27	
P0.05	A1		~D8	P0.21	
P0.30	A2		~D7	P0.23	
P0.29	A3		~D6	P1.14	
SDA	P0.31	A4		~D5	P1.13	
SCL	P0.02	A5		~D4	P1.15	
P0.28	A6		~D3	P1.12	
P0.03	A7		~D2	P1.11	
		+5V		GND		
		RESET		RESET		
		GND		RX	P1.10	
		VIN		TX	P1.03	

ARDUINO.CC

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

NINA-B3X

| +5V |
USB_DM	USB N
USB_DP	USB P
	USB ID
GND

BOTTOM

LSM9DS1
Inertial Motion Unit sensor

P0.22	VDD
P0.22	VDDIO
SDA1	P0.14	SDA
SCL1	P0.15	SCL
P0.22	CS_A
P0.22	SDO_A
P0.22	CS_M
P0.22	SDO_M
GND

| ~D8 | P0.10 | NFC2 |
| ~D7 | P0.09 | NFC1 |

Making a short circuit using the solder jumper allows only the function in the SJ Pin cells and also it changes the Pad connection of the pins D7 and D8 in P0.09 and P0.10.

| + | +3V3 |
| - | GND |

Cutting the solder jumper allows to power via battery connecting the battery's ground to the GND pin and the battery's positive to the 3.3V pin.

LSP22
Pressure sensor

P0.22	VCC
P0.22	VCCIO
SDA1	P0.14	SDA
SCL1	P0.15	SCL
P0.22	CS
GND	SDO
GND
GND	RESET

APDS9960
Digital proximity, Ambient light, RGBa and Gesture sensor

SDA1	P0.14	SDA
P0.19	INT
+3V3	LDR
+3V3	A
+3V3	K
SCL1	P0.15	SCL
GND
+3V3	VDD

| +5V | VBUS |

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

1	+3V3	NINA-B3X
2	SWDIO	15-SWDIO
3	SWCLK	11-SWCLK
5	GND
6	RESET	P0.08

MP34DT06JTR
Microphone

P0.17	VDD
P0.26	CLK
P0.25	DOUT
P0.17	LR
GND

HTS221
Humidity and Temperature sensor

P0.22	VCC
SDA1	P0.14	SDA
SDA1	P0.15	SCL
P0.22	CS
GND

ARDUINO SYMBOL

red	P0.24	LEDR
RGB LED	green	P0.16	LEDG
blue	P0.06	LEDB

nRF52840

ARDUINO.CC
Last update: 16/09/2020

Legend:
- 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
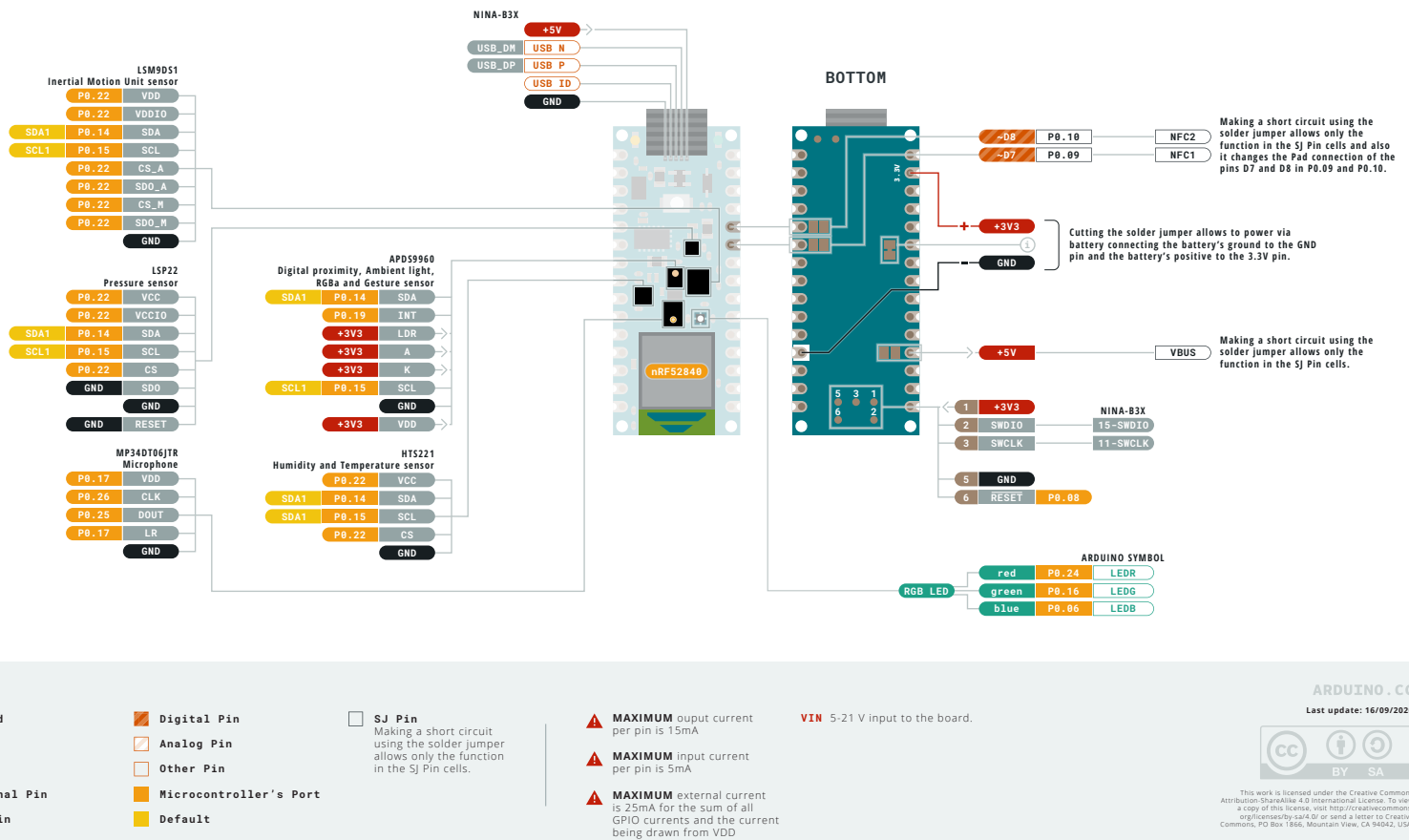
VIN 5-21 V input to the board.

10   Maybe useful text

Thursday 20. August 2020, the group gathered and had a meeting with Steven Bos. We discussed our project idea and the potential use of the Microsoft HoloLens 2. It was in the group's best interest to put our resources into the microcontroller rather than the goggles, dismissing any development with the HoloLens.

On September 10. the group met to plan the project. After deciding on the initial design, we drew a sketch and wrote a short description. This was sent to our professor, along with a list of required parts as well as the associated budget. The initial design can be seen in section XX.

On the 16th of September, we ordered three Arduino Nano 33 BLE Sense w/headers from Arduino.cc. This came to 1080NOK. In addition to this the professor provided us with a car for the project. It is called Turnigy Trooper [5], and was originally a radio controlled car. However everything but the battery and servo motors for thrust and steering, have been stripped away.

The servo motors are controlled by a pulse-width modulated signal. This means in essence that we send discrete signals, where the signal will rise at a fixed frequency. The length of time the signal is high before going low will determine the behaviour of the servo motors. Luckily for us there is a standard regarding the behaviour generated by a specific pulse width.

We will need to write a library which will abstract this away in the code. Where the interface for steering will take an angle, and the motor interface will hopefully be able to take a velocity in m/s, where negative numbers will mean going backwards.

After Steven showed us a proof of concept that it will be quite doable to base our library for the Arduino Nano on the existing toolchain for NRF52840, we started developing on the 17th of September. However there quickly emerged a problem, as Gnat arm-elf initally only supports ZFP-runtime libraries for this card. This meant that it would only support barebone functionality, and we would never be able to make use of the Ada realtime library, or even multi-threading the software. Luckily, there exists support for full ravenscar run-time libraries for the card. We followed the official guide at the bb-runtime repo [3], to generate and build them ourselves.

```
./build_rts.py --rts-src-descriptor
    ~/opt/GNAT/2020-arm-elf/arm-eabi/lib/gnat/rts-sources.json
    --output=temp nrf52840
```

```
gprbuild -P
    ~/opt/GNAT/2020-arm-elf/lib/gnat/arm-eabi/lib/gnat/ravenscar-full-nrf52840/ravenscar-build.gpr
```

On the 18th of September we improved our library build, by letting the project_wizard script in Ada drivers library generate the gpr-files for our board automatically. All we had to do was to provide some basic information about the microcontroller, and the script did the rest. We also started adapting the pin mappings from the nRF52832 to the nRF52840.

Provided that our efforts lead to a useful library, we will create a pull request into the Ada drivers library, so that others may benefit from our work.

```
./Ada_Drivers_Library/scripts/project_wizard.py
```

October 5.

We received the Segger J-Link debugger from our professor. In order to get this to work, we installed the firmware from segger.com. However we also had to install a pack for pyocd in order to find the debugger:

```
pyocd pack --install stm32l476VG
```

As all development within the group is done on Linux, we also had to add udev rules, so that we were able to access the usb ports without being root. These rule-sets were found on pyocds git repository.

We also had to give the respective user access to the serial port by running the command:

```
usermod -a -G dialout MY\_USER\_NAME
```

Legge til utilization test og grafikk.

October 12.

We are able to flash and debug by manually holding cables to the debug connections on the Arduino. We used the diagram on the segger website [4] to find out which JLink pins to "connect" to the Arduino points.

AnalogWrite seems like the perfect choice to generate a pulse for both servo and sensor. In the Ada Drivers Library [1] there is a very good example for controlling a servo. It changes the analog signal period with **Set_Analog_Period_Us** to generate a steady pulse.

However we were never able to make it work on the arduino. Seems to be a problem with incompatibility between the drivers and the Arduino or the nRF52840 chip.
We suspect it is tied to interrupts since the stacktrace from the debugger kept stopping on a new line each time.

Progress is slow because of the manual labour required for each flash and debugging session.

October 19.

We recieved the clamps from Richard and Steven was able to put it together

with the Arduino and a breadboard. We are finally able to easily flash and debug.

We were able to create a steady pulse using digital I/O and tasks with a fixed priority. With a proper priority we were able to use three tasks for three separate servos and one task for three sensors.

We encountered some problems using a 9V battery power source, as the Ardino requires at least 5V and the peripherals need 5V.

October 21.
We wrote the code for the entire system.

October 22.
This was the first day we tested our servo code on the car. After a lot of trial and error on the steering we quickly moved on to the drive motor. We got the wheels to move fairly quickly after discovering that the motor needs a neutral (1500ms PWM) signal to intialize, before giving it signals which indicate torque. However, there quickly emerged problems. First of all the minimum speed was much to high, and secondly, it was impossible to get the car to switch between going forwards and backwards. After conversations with our professor, we learned that Joakim Bjork, had worked a lot with this car, and sat on a lot of documentation for the motors. This documentation would be made available for us on the next day.

Obtober 23.
D-Day (Documentation day)

11   Ravenscar

The full ravenscar run-time library offers what is called the Ravenscar profile. This is a subset of the Ada language, where the features and available libraries are limited. The reason for this is to ensure robust software on realtime and safety critical systems.

The specifications of the ravenscar profile are the following: https://www.ada-switzerland.ch/rm/RM-D-13.html

There are a couple of limitations here that will affect the way we initially planned our software.

The maximum number of entries in tasks are set to 0. This means that we won't be able do have dynamic tasking, and all tasks will rather have to be defined at compile-time.

Another thing we planned was to specify which tasks would run on the respective cores of the CPU. This however, is not supported within the ravenscar profile, and we will need to do all the scheduling as if we only had a single-core processor.

12 Various problems we've encountered along the way

By the time you're reading this, some of these issues might have been fixed. Down below are some problems we have encountered during our development phase from 17.02.2020 - ??. Most of these issues have occured using Linux, it is stated otherwise.

- 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.

- Including unused libraries in the Ada source-code makes the arduino crash under run-time.

- 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 in real time. –ref til Debug lstlisting i appendix–

- Only Digital I/O works. Possibly because Analog 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 every new computer.

12.1 Pin layout

11.10.2020
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 [8]. Without it we were only able to use the pins with P0 seen on figure 9.3. After adding the base address to the Ada code and the option to chose between 0 and 47 pin number we were able to make all the pins function.
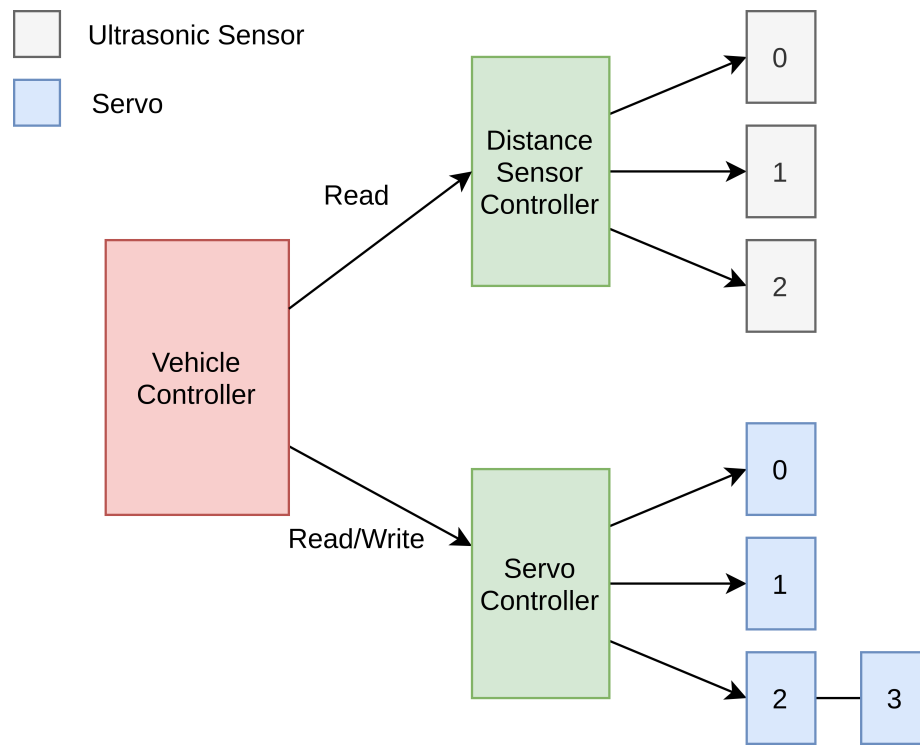
13   Maybe useful pictures

Figure 7: Block diagram of our system architecture