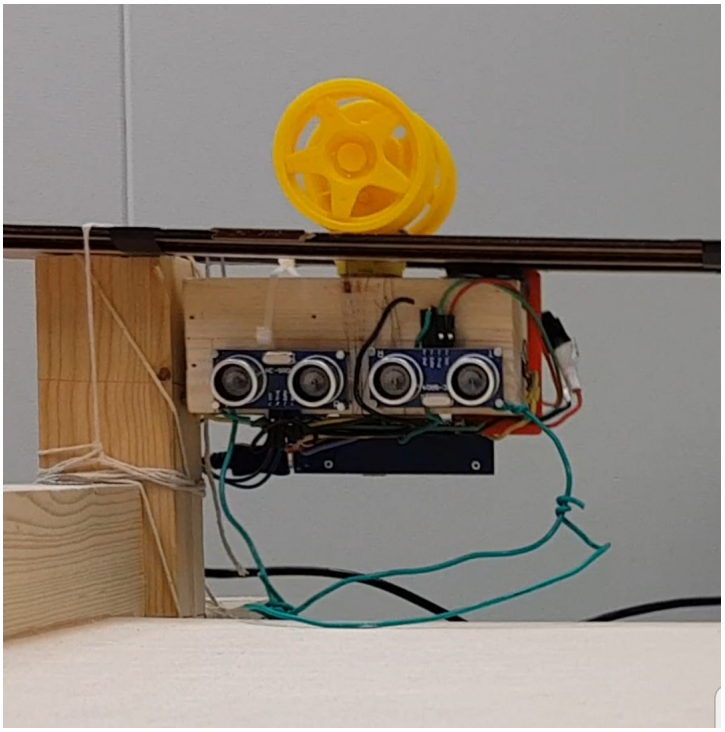
Project What A Save

Real-Time Goalkeeper

Runar Eckholdt — Markus Ø. Leander — Jørgen W. Søbstad
github

Semester project — Real-Time systems
DSA3102



University of South-Eastern Norway

Faculty of Technology, Science and Maritime
19/11/2021

Contents

1 Abstract

This is our prototype submission for the course assignment in Real-Time Systems DSA3102-1, Fall 2021. Where we create a real-time goalkeeper programmed in Ada using the Micro:Bit v2.

2 Introduction

A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within the intervals dictated by the environment. [1, p. 2]

Our tasks was to create a real-time system programmed in ADA using the Micro:bit-v2. It was stated in the task that a prototype of the project was required. We decided to go with project *"what a save!"*. *WTS* in short is an automated keeper, where we use ultrasonic sensors combined with a dc motor to detect an ball, then act by blocking it's path.

The difficult part is creating a system that meets all deadlines as well as having fault -detection and -handling. Seeing as the future holds more and more technological solutions, Real-Time becomes more important with it. Hence the understanding required of tasking, scheduling as well as fault management also increases.

In this paper we present our discoveries, trials and experience with making a goalkeeping robot that can both probe and track a ball using real-time criteria. There exists quite a few research papers out there addressing this, few made in Ada following real-time specifications, zero public papers using the Micro:bit v2.

This report will address our decisions when creating packages, component choices, decision's of priority scheduling, tasking and challenges of running Ada on the Micro:bit.

3 Ada and Real-Time

Today in 2021 Ada is still heavily used in defense, mission critical , safety critical and Real-Time embedded-systems. Recently the Scandinavian Real Heart group chose Ada as their language for their artificial hearts. Their reason for this as stated by their main software architect; Lars Asplund: *"Our heart pump has to work uninterrupted throughout the life of the patient"*. [2] ..." *"The quality and reliability of all parts of the system are crucial. We want to create software with the highest level of safety, and we know that SPARK together with Ada is the best option."* [2]

So there is no doubt about the relationship between Ada and the creation of Real-Time systems. Seeing as Ada was originally developed by the US department of defence with safety and security in focus there is no doubt about its integrity. When using Ada with the Micro:Bit we noticed how easy it made the creation of a reactive time aware system with both time triggered and event-triggered invocations. It made the measurements of computation time quite easy. Ada is also quite verbose making it easy to read, understand and maintain. The strong typing and related features ensure that most errors are detected at compile time and of the remainder many are detected by run-time.

4 Process

This part of the report summarizes our logs that we wrote most of the days we worked on our project. It will tell our thoughts and discoveries, and methods on our path to ball detection. We will divide the process into x phases.

4.0.1 Brainstorming

Phase 1 is the brainstorming part of our project. Initially we where choosing between 7 different ideas.

- Idea 1: Ping pong balancing device
- Idea 2: Remote controlled car
- Idea 3: Jumping rope robot
- Idea 4: Robot that is able to ascend,descend and walk straight
- Idea 5: Multiplayer game using the micro:bits
- Idea 6: Non Stop sorting device
- Idea 7: Automated goalkeeper

To decide what project we wanted to work with we used a Pugh Matrix where we evaluated different ideas after different weighted parameters.

Weight	Parameters	Idea 1	Idea 2	Idea 3	Idea 4	Idea 5	Idea 6	Idea 7
4	Component:Price	-1	+1	0	0	+1	-1	0
3	Component:Complexity	-1	0	0	-1	+1	-1	-1
4	Coding complexity	-1	+1	+1	0	0	0	-1
5	Professional Relevance	+1	+1	+1	+1	+1	+1	+1
3	Upgrade potential	+1	0	-1	+1	+1	-1	0
	Total Score	-3	+13	+6	+5	+15	-5	-2

The idea with the matrix is not to decide it with the resulting score. It visualises the pros and cons with the different ideas, giving us a better idea of what we can expect when choosing the project. Some ideas got a pretty high score but was not chosen anyway. The reason why we went with the goalkeeper project was because of it's professional relevance and it's variance in complexity. In addition it's a project that where certain we could finish within the deadline.

We then started by creating an initial sketch of the system (1). Our idea was to use lasers to measure the position of the ball at two locations. Using these two positions and the time difference we could calculate a speed vector of the ball. We wanted to send pulses and calculate time of interference of our signal. If there was an interference we would know the ball has arrived at location y_1 or y_2. We then thought that we could use this interference to calculate the balls x position if we scaled the laser signal correctly to the track.

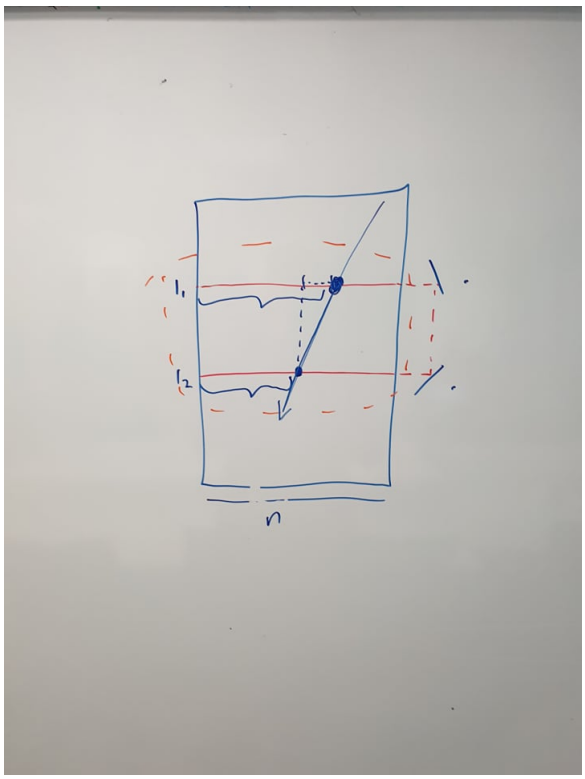


Figure 1: First sketch of the system

4.0.2 Scoping in

We eventually realized that we needed a simpler way to track the ball location. Just designing the laser signal was difficult. We also realized that to calculate the time of interference in a light signal we need quite high resolution on the clock. This is where the limitations on the micro:bit comes in.

We thought about swapping the lasers out with ultrasonic sensors as they can measure distances pretty accurate. However this does not give us it's y position on the track, because it does not only detect strait forward as sound moves in a cone shape. If the ball had moved parallel to the y axis we could simply just use the distance, but it does not work that way. Using an IR object detection sensor we could detect when the ball reached a certain y location. However this was just a true false response, no distance. Therefore we used it to activate the ultrasonic sensor to measure the distance for us. But the IR was too unstable and too quick at the same time. In some rare situations they synced perfectly, but most of the times the IR would activate the ultrasonic too early. The next solution was a cheap camera module.

The keeper needed a mechanism to move. The mechanism has to be fast as we need quick response and positioning. As we wanted our prototype to be as cheap as possible, we decided to use some spare fishing line to pull the robot as fishing line is non-elastic and strong. *Figure*(2) shows a sketch of the system using these specifications.

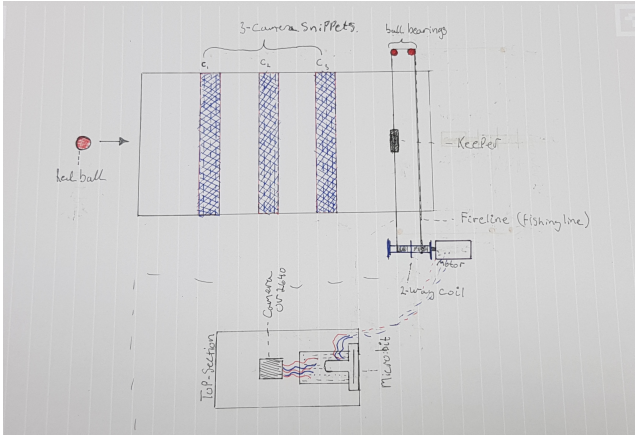


Figure 2: WTS using camera, dc motor and fishing line

4.0.3 Camera phase

We knew that the solution we had in this phase was highly dependant upon the camera module. The module we went with was the **OV2640**. We started right away to try make it work. The module uses SCCB [3] protocol to communicate with the micro controller; which seems to be a specialized version of I2C. Even after initializing the I2C on the micro:bit, and finding the correct address to the camera's SCCB unit, we could not get any signals out of the camera. We used the analog discovery at all times monitoring the different cables between the camera and the micro:bit.

After two weeks of debugging and frustration we decided to move away from the camera, because of time limitations.

4.0.4 Triangulation

Moving on from the camera idea we decided to attempt on finding a solution using the ultrasonic sensor HCSR04.

We placed two ultra-sonic sensors(HC-SR04), one in each lower corner. They are separated by the short side of the table which is 40cm. We would use tasking to synchronize the two so they wouldn't interfere with each other. As the distance between the sensors are static, we could calculate the coordinates from this and the measured distances. We then used some time sketching and derived a formula that solved our problem.

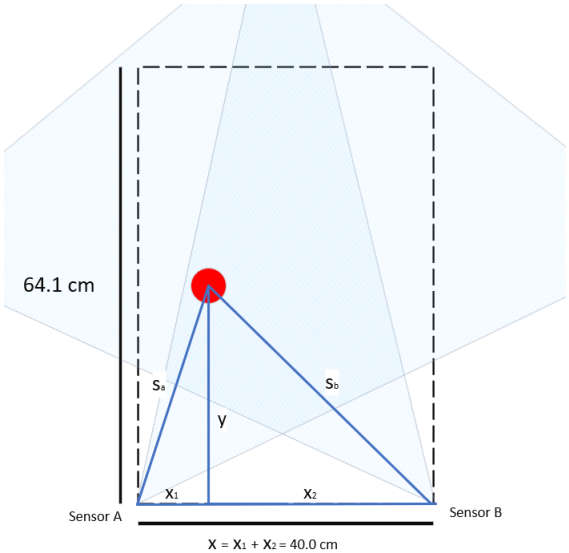


Figure 3: Using triangulation to calculate ball position

As you can see in *figure* (3), we only have three known values: Both Ball sensor lengths(s_a), (s_b) and the total length between the sensors($x = 40cm$). With the intention of using Pythagoras we derived this formula:

$$s_a^2 = x_a^2 + y^2 \Rightarrow y^2 = s_a^2 - x_a^2$$

$$s_b^2 = x_b^2 + y^2 \Rightarrow y^2 = s_b^2 - x_b^2$$

$$s_a^2 - x_a^2 = s_b^2 - x_b^2 \Rightarrow x_a^2 = s_a^2 + x_b^2 - s_b^2$$

We then find an expression for x_b^2:

$$x = x_a + x_b \Rightarrow x_b = x - x_a \Rightarrow x_b^2 = (x - x_a)^2 \Rightarrow x_b^2 = x^2 - 2x \cdot x_a + x_a^2$$

Then we substitute x_b^2 in the previous expression:

$$s_a^2 - x_a^2 = s_b^2 - (x^2 - 2x \cdot x_a + x_a^2) \Rightarrow x_a^2 = s_a^2 - s_b^2 + x^2 - 2x \cdot x_a + x_a^2 \Rightarrow 2x \cdot x_a = s_a^2 - s_b^2 + x^2 \Rightarrow x_a = \frac{s_a^2 - s_b^2 + x^2}{2x}$$

We can then find the height by:

$$y = \sqrt{s_a^2 - x_a^2}$$

Which we combine to get the coordinate of the ball ($B_n(x_n, y_n)$) for each measured time(t_n). This means we can simply calculate the movement and coordinate of impact using the vector:

$$\bar{r} = [B_n - B_m]$$

.

There is one big problem with this solution. To know the ball's position we need detection by both sensors. As you can see in *figure* (3) there is blind zones all around on the field. The only place both sensors cover is in the middle of the field, and after some testing we could confirm this. We therefore needed yet another solution, which will be presented in coming section.

5 Final build

We will in this section describe the final build of our project.

5.0.1 Budget Wall-E

Our final bot design works by hanging it upside down from rails above the soccer field, *figure*(4).

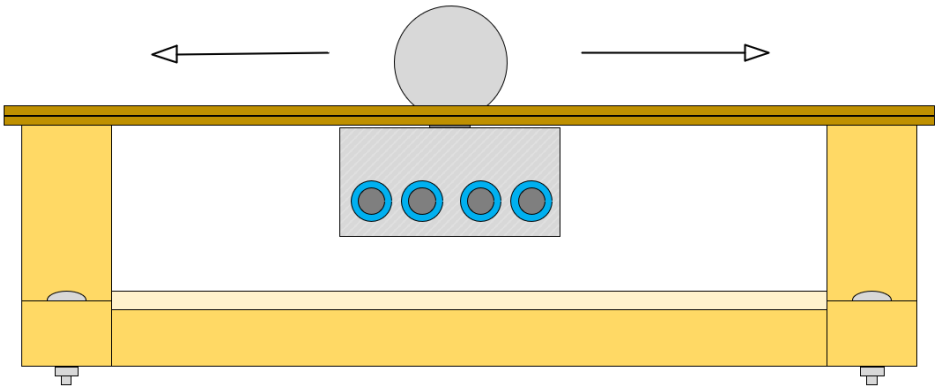


Figure 4: Front view of setup

This setup works very well, the only cons about this is that the bot currently hangs a bit to high above the field; which gives a blind zone close to it. If the ball is rolling with an angle too steep, it might go past the keeper undetected. Other than that the setup yields us the best results from what we have tried so far. The result is increased stability and with 4 nails on top we lock it from rotating around the Z-axis.

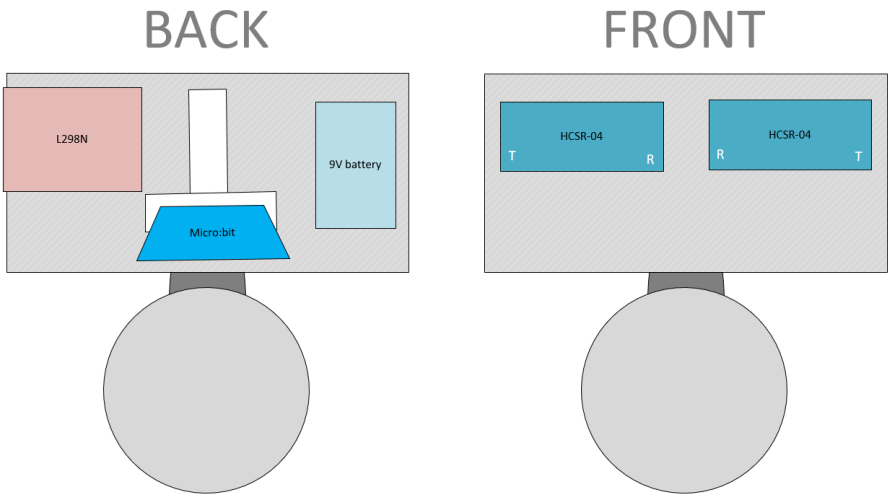


Figure 5: front and back view

part	quantity	function
wooden block	1	Main body of keeper
metal nails	4	Rotation blocking
Micro:Bit v2	1	Main micro controller
Micro:Bit GPIO board	1	Port management
HCSR-04	2	Object detection.
L298N	1	Motor controller
9V Battery	1	Power for L298N
DC motor	1	Motor
plastic wheel	2	keeper movement.

5.0.2 Circuit

We first tested out each part using breadboards and checked each component thoroughly using the Analog discovery kit. We designed a circuit diagram when we could confirm that our wiring was correct, and ready for soldering to a PCB board. After soldering we re-checked all components with the analog discovery kit before we put the bot back together, to be sure we wouldn't burn anything.

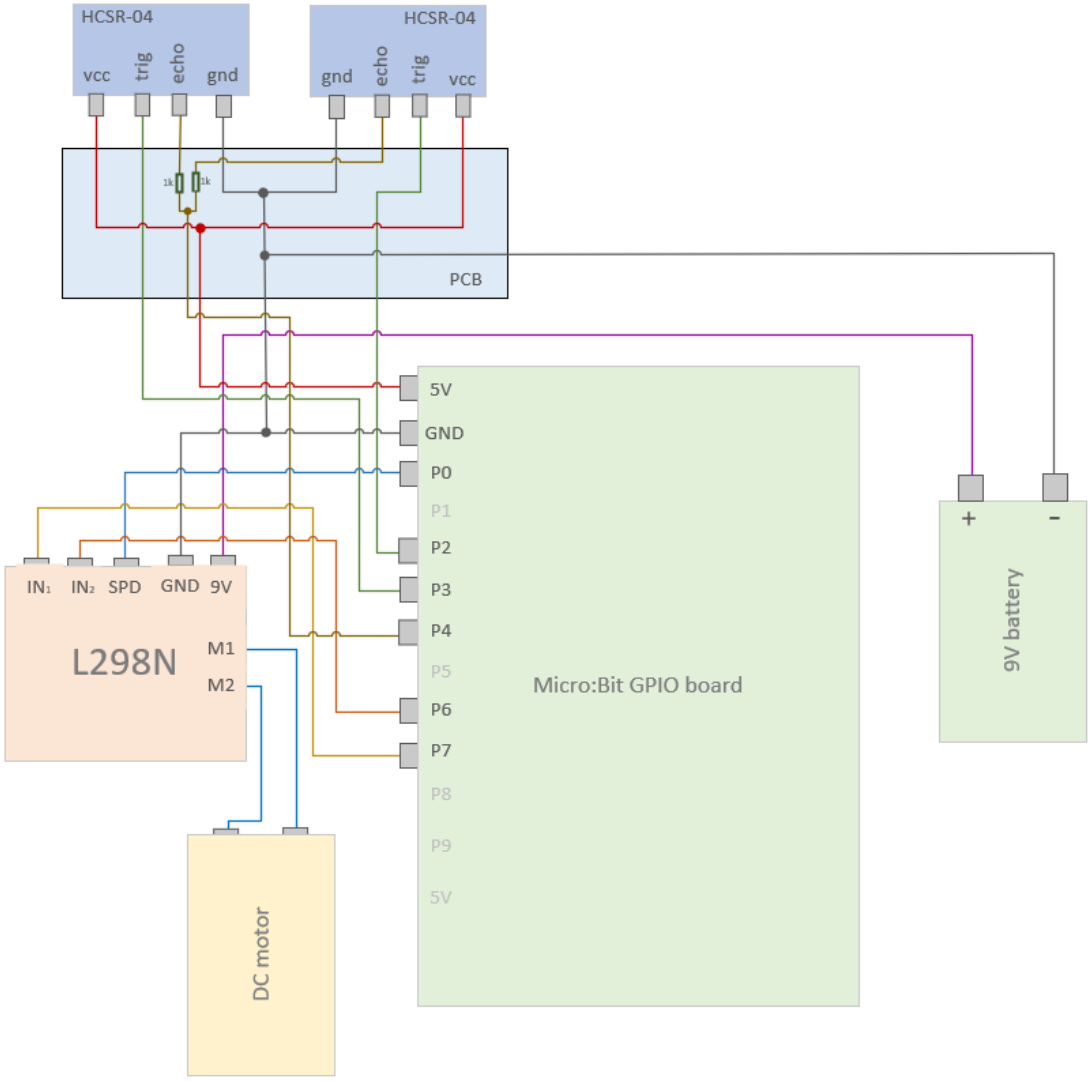


Figure 6: circuit diagram of Wall-E

As you might notice the ultra-sonic sensors are mirrored, this is a design choice we made as a way to get the signals as centered as possible.

5.0.3 Tasks

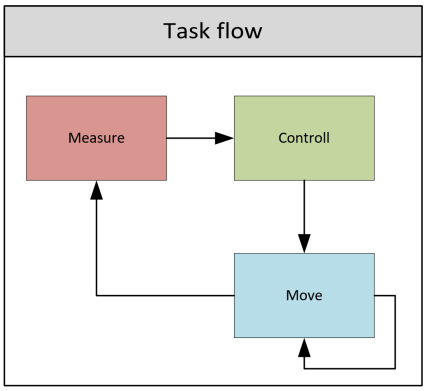


Figure 7: Task Flow Chart

Measure — Priority 3

The functionality of measure is to trigger both ultra-sonic sensors, and write the retrieved distance data to a resource shared between the tasks. This was at first our main bottle-neck, as we had to wait for the sound to reach back to the sensors twice each task set; further explained below *fig* 10. We solved this by letting the task sleep while the sound is traveling, and wake it up by an interrupt triggered on their shared echo pin (see 6). This small feature actually gave us a reduction in our average worst case response time by 87.7%! Illustrated in *figure*(12); this allows our *move* task to work during the sound travel time. We decided to not include the blocking time in our final calculations, as the potential blocking time is negligible.

Controller — Priority 2

The controller fetches the shared data, then does various calculations and mode assignment. We have two modes: TRACK and PROBE, where *TRACK* is controlled by this task; it simply set the direction equal to the side relative to the keeper, where the bot stops within a threshold in the middle. As the controller only process new data, it shares the same period as Measure, but it is also controlled by an entry at the end of measure. This is to prevent controller from accessing old data during the sound travel period.

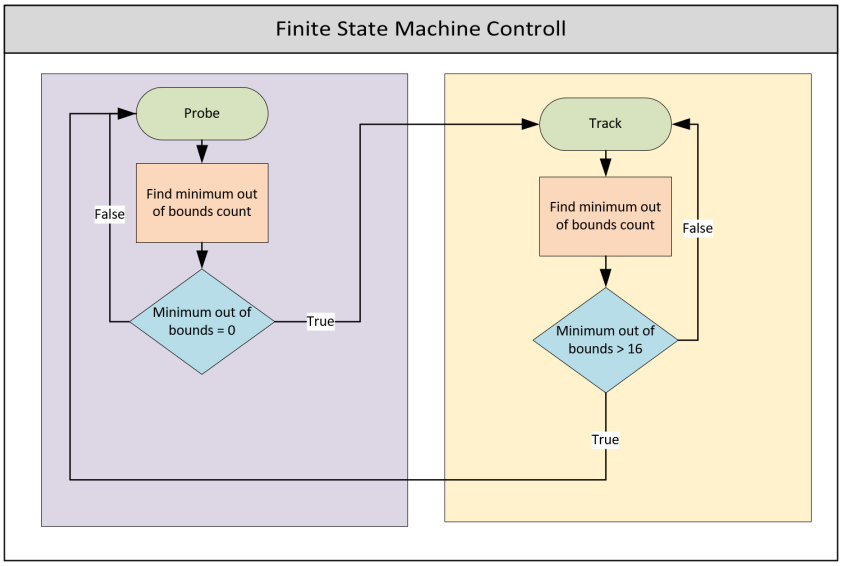


Figure 8: Finite State Machine - Controller

See `Calculate` at line *123* to view the track logic source code.

Move — Priority 1

Move is our lowest priority task, it has two modes as mentioned above.

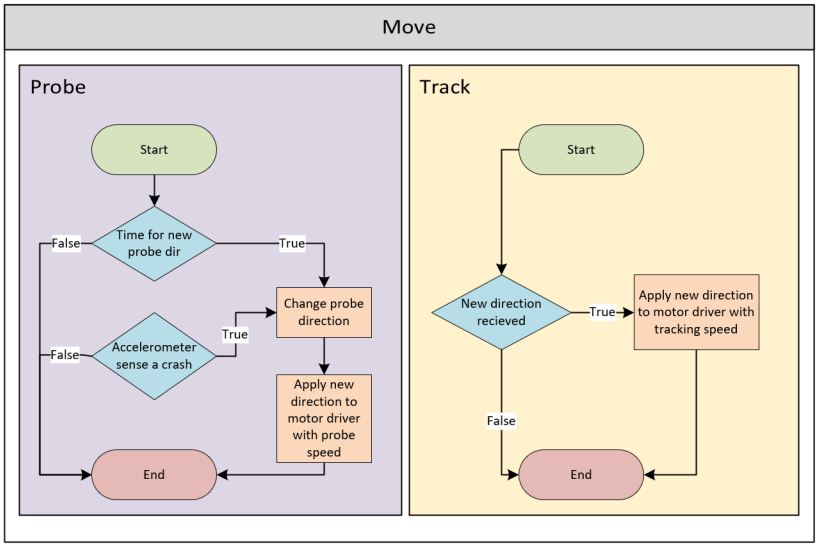


Figure 9: Finite State Machine - Move

TRACK — the tracking mode has the lowest computation time of all parts of our tasking. The reason for this is because it just enables the motor in the assigned directions, and only when the direction is changed.

PROBE — This mode is enabled when the bot lose track of the ball. It is controlled by a tick-counter so it requires no target-detection for at least n ticks to enable; which is 16 in our final build. The counter is monitored and calculated in the *Controller* task. Probing works by monitoring the current acceleration; if the bot hits a wall, the acceleration will spike, if that spike exceeds our set threshold the bot will simply beep and switch direction. We added a de-bounce to keep it from constantly switching directions as the acceleration will spike on direction changes as well. We also added a timer as a fail-safe mechanism that will switch the direction if it runs out, the timer is reset at each acceleration caused switch. If an object is detected at any point during probing it will switch to *TRACKING* at the next *controller* task cycle.

5.0.4 HCSR04

Figure 10: hcsr04 ultra sonic sensor

The ultrasonic module HCSR04 calculates the distance using sound travel time. The trig pin is used to trigger the module making it send the sound signal. It needs a pulse of 5V for at least 10 micro seconds to trigger. When it triggers it will then start sending 8 pulses of sound. When the 8 pulses is sent, the echo pin goes high. The moment the last of the 8 pulses is received, the echo pin goes low.

The pulse of the echo pin then determines the time it took for the sound signal to bounce back. We can use this time to determine the distance between the module and the closest object. The time we receive is for the sound to travel both ways. Therefor we start by dividing the time by 2. We then multiply it by the speed of sound and this should give us the distance in meters. We want to work with centimeters so we multiply this again by 100.

$$\frac{time\ s * 343m/s}{2}$$

We created our own package for this module; see `HCSR04.ads` and `HCSR04.adb`

5.0.5 L298N

We use the motor driver L298N for our dc motor. There is two factors the dc motor is controlled by, current and direction of current. The higher the current the faster the motor goes. The direction of the current determines the direction of the motor rotation.

The L298N controls all of this for us. We use only half of the driver (one motor). It has two **IN** pins we can use to determine the direction. In *figure*(11) you can see the circuit and how it will change direction based on the IN signals. If both **IN** are disabled no current will be allowed into the circuit. If both is enabled the circuit will not be connected to ground. We can therefore use this to stop the motor.

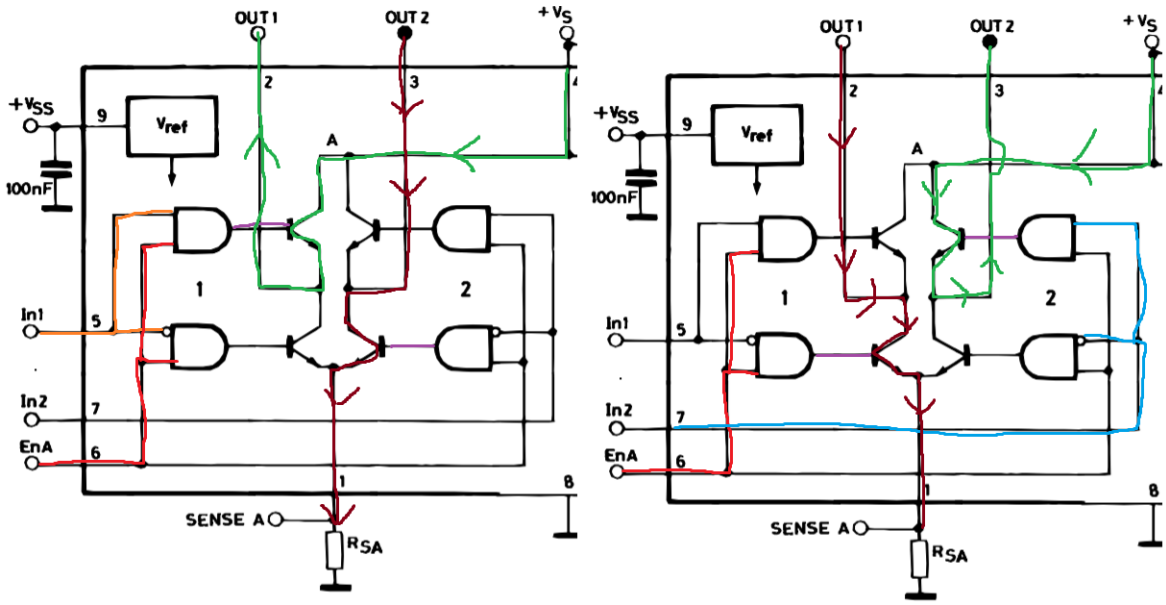


Figure 11: Left picture shows behavior when IN1 is enabled. Right picture is when IN2 is enabled. Circuit is an edited picture taken from [4]

The module also supports speed control. The pin need a pwm signal to simulate an analog signal. Lower analog signal gives a low speed. Higher analog signal gives high speed. The frequency it expects is at 20 kHz.

We made our own ada package to use this module, you can view the source code here `adb` and `ads`.

5.0.6 Micro:bit

The use of the Micro:bit gave us access to a lot of sensors. One we had the fortune of using was the accelerometer. This allowed for the probing mechanism to circumvent having to know it's own position. Once it hit a wall there is a spike in instantaneous acceleration. This allowed for the direction to be changed easily. When using the Micro:bit the documentation was easily found at the nRF51 [5]

We have used Steven Bos's fork of the `Ada driver's library`, and `bb-runtime` to be able to program the Micro:bit-v2 and use the Ravenscar profile.

5.1 Scheduling and tasking

5.1.1 Fixed Priority Scheduling — FPS

The group decided to follow in the steps of many others using the most common approach. A Preemptive Fixed-Priority Scheduling. The choice was made for us, by the Ravenscar profile. The Ravenscar profile restricts the use of dynamic priority scheduling algorithms such as Earliest Deadline first and Least Laxity First. When using FPS we gave each task a set fixed, static priority that where computed before run-time. The advantage of a Preemptive approach is that the higher priority tasks become more reactive. Section 5.1.2 and 5.1.3 talks about how FPS with interrupts do when tested against the Utilization and Response-Time test. With the implementation of Interrupts we don't have to follow the Rate Monotonic priority assignment. Usually the lower periods get higher Priority but with the interrupt implementation we circumvent this (as proven in tasks).

5.1.2 Utilization-based Schedulability test

After our tasks where finished programming-wise; we tested each tasks worst case computation time in an isolated environment. We then used a simple `python script` to test against the utilization bound, given as:

$$\sum_{i=1}^{N}(\frac{C_i}{T_i}) \leq N(2^{\frac{1}{N}} - 1) => 0.493 \leq 0.780$$

Hence the Utilization-based test is passed; details are found in the table below.

Task	Priority(P)	Comp. time(C)	Period(T)	Utilization Factor(U)
Measure	3	12.323700ms	32ms	0.385115625
Controller	2	0.030518ms	32ms	0.0009536875
Move	1	0.427246ms	4ms	0.1068115

After modeling our timeline we noticed a huge part of measure and controller is idle after its has run. We realized we can easily half the period and still have plenty time to meet all deadlines. In this case the Utilization test actually fails, but the response-time analysis is successful.

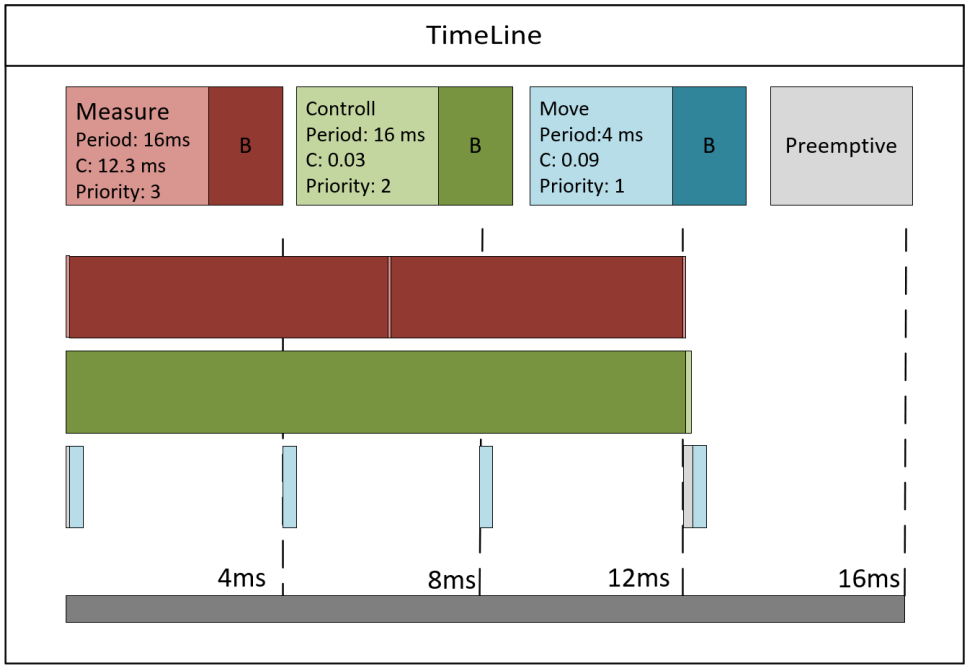


Figure 12: Task timeline

Task	Priority(P)	Comp. time(C)	Period(T)	Utilization Factor(U)
Measure	3	12.323700ms	16ms	0.77023125
Controller	2	0.030518ms	16ms	0.001907375
Move	1	0.427246ms	4ms	0.1068115

$0.879 \leq 0.780$ — test failed

5.1.3 Response-Time Analysis

We calculated The Worst Case Response Time; verifed by $R_i \leq P_i$. We managed to implement a working interrupt within the HC-SR04. To verify it's efficiency boost we calculated the response times with and without interrupts using the following formula:

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

Task	Priority(P)	Comp. time(C)	Period(T)	Worst Case Response Time(R)
Measure	3	12.323700ms	16ms	12.323700ms
Controller	2	0.030518ms	16ms	12.354218ms
Move	1	0.427246ms	4ms	12.781464ms

Worst Case Response Time ***without*** *interrupts in HCSR04.*

Task	Priority(P)	Comp. time(C)	Period(T)	Worst Case Response Time(R)
Measure	3	1.373291ms	16ms	1.373291ms
Controller	2	0.030518ms	16ms	1.403809ms
Move	1	0.427246ms	4ms	1.831055ms

Worst Case Response Time ***with*** *interrupts in HCSR04.*

The reasoning behind the huge reduction in computation time is because the *measure* task sleeps during the sound travel time; awoken by interrupt. As you might notice, we have decided to give the *Controller* task the same period as Measure. Because there is no reason for controller to calculate old data it share the period of measure. One could argue that measure and controller could be in the same task, but we found this to be the more organized approach.

6 Discussion and summary

6.1 Unresolved issues

At this state of the project we still have issues that we wish we had more time to explore. The robot sometimes sees objects where there are none to be seen. A hypothesis we have is that the sound waves are separated. One part of it might bounce back at the correct time telling the robot it's out of bounds. The other one might bounce around some before returning. If this signal arrives at correct time it might be interpreted as an object nearby.

The tracking of objects is definitely not perfect either. Sometimes it detects the ball but in the wrong location. We know that it is the ball it detects because, if we move it slightly to the sides the the robot also changes it's position. Our hypothesis on this issue is how waves bounces of convex surfaces. As we use the difference in distance measured by the two sensors to track the ball, a convex bounce might enable multiple locations to be the correct place at the same time.

Another issue is that in some rare situations the system freeze. We initially thought that this was caused by the entry waiting in the HC-SR04 package. However after a lot of testing and debugging we have debunked this theory. The system freeze happens even if we don't use this functionality. When we measured the timing for the measure tasks, we sometimes observed a very rare anomaly where the task used 137 ms to finish. Whatever caused this is probably responsible.

The robot itself is not balanced at all. Some sides of the robot is much heavier than others. This makes it much more "*wobbly*".

6.2 Shortcomings

6.2.1 Timeouts

There where attempts towards the end at creating a timeout for the HCSR04 sensor. This would have been able to handle some of the anomalous spikes in measure times. With a timeout we would be able to set a max wait time for the signal to rendezvous and possibly avoid certain anomalies. Failing the attempted entry call would result in a withdrawal and an alternative sequence would executed.

Implementation of the timeout would enable a maximum wait time for the sensor. It if exceeds this time it would be timed out and count as an out of bounds measurement.

In regular Ada you would use Select statements for the timeout but considering that we are using Ravenscar profile it brings with it the restriction of No Select Statements.

6.2.2 Track Scent

We started tinkering towards the end making an sort of "scent". The intention behind this was for the robot to remember the direction it lost sight of the ball. Making it so that the robot would continue in that direction for a little while.

6.2.3 Potential Upgrades

One wish for a potential upgrade is for the keeper to drive with no supporting rails. Returning to the original design of a keeper on the ground with the ability to self balance. For this, weight distribution should be reconsidered. A self balancing keeper that could be put in any goal with previous-defined parameters would be terrific. This project has tremendous amounts of potential upgrades we would love to explore further.

7 Summary

Project *What A Save!* has been a fun and challenging experience. We all fell in love with the ADA programming language; such a simple yet effective language to work with. We wish we had more time to further build and develop our keeper. The project changed its shape countless times. The group had to restructure code, models and even scratch it all a few times. But in the end we got a working prototype we are proud of. The entire group agree we have learned a lot about real time systems along the way and see the importance of good practice and how this field is applied in the real world.

References

[1] Alan Burns. Analysable real-time systems : Programmed in ada, 2016.

[2] AdaCore. Blog.AdaCore. 2018.

[3] OmniVision. OmniVision Serial Camera Control Bus. 2003.

[4] SGS-THOMSON Microelectronics. APPLICATIONS OF MONOLITHIC BRIDGE DRIVERS. 1995.

[5] Nordic Semiconductor. nRF51 Series Reference Manual. *Nordic Semiconductor, October*, 2014.