Automated Cruise Control

Team Members

- Anthony Savitt - 818054985 - theanthonysavitt@gmail.com
- Jacob Carver - 817520802 - jacobmcarver@gmail.com

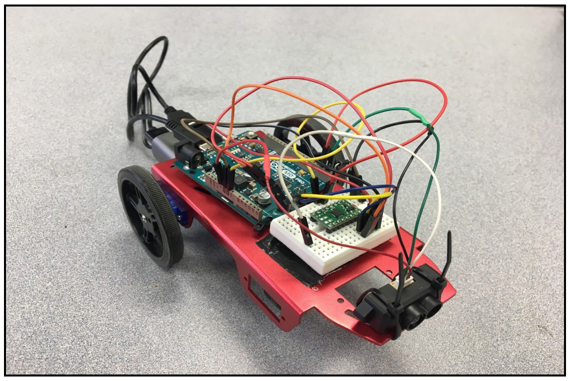
Accomplishments

Hardware

We were able to complete the hardware design, and we got everything working in the end, however there were some hiccups we experienced that are detailed below.

1. 2WD Mini Robot Platform Kit
    a. Main Chassis
    b. Assembled and working.
    c. Figuring Out how attach electronics did pose a challenge. We utilized velcro to bind our 5V power banks and zipties to hold down our microcontroller and optical sensor. Due to fears of stripping screws with a redesign, one of the vehicles has less optimal battery placement but works just as well.
    d. Screws and Wheels are low quality, we lost a screw and we were forced to order a new chassis
2. Arduino Zero Pro
    a. Main Controller
    b. SAMD21G18A ARM Cortex-M0 Based Microcontroller
    c. Working as intended, *zero* issues encountered
    d. Since we opted out of using the Arduino IDE, the pin assignments on this board were useless to us, however we got the extra experience of cross referencing the Schematic and Layout files with the datasheet in order to program each peripheral.
3. Benewake TFmini Lidar Sensor
    a. Working
    b. We encountered a ton of problems with these sensors, that we attribute to both user and design error.
        i. The first two sensors we got both died after a few uses.
        ii. Our best guess is that a negative voltage spike caused by our servo motors destroyed the internal IR led that Lidar depends on.
            1. We believe this is the cause as the actual device itself was functioning and able to send back serial data.
            2. The serial data itself was garbage, sporadic, and eventually stopped completely.
            3. Powering the motors on a separate power supply from everything else remedied the problem. We also separated the two grounds of the power supply with a diode to stop any reverse current
        iii. Sensors now work, and have worked ever since we fixed what we believe to be the problem.
4. FS90R Micro Continuous Rotation Servo
    a. Working as intended
    b. Killed two of our sensors, although this was not the fault of the motors.
5. DRV8833 Dual Motor Driver Carrier
    a. Worked instantly, and has worked without fail
    b. Fast Shutdown mode on the internal control transistors is what we believe caused the reverse current to flow into our sensor.
6. Bi-Directional Logic Level Converter
    a. Deprecated
    b. TFmini actually has a 3.3V TTL level for its data line, which we noticed late into the project. This made this part unnecessary.
7. Demo Track
    a. Wooden track we made for demonstration purposes.

   b. Works decently well, however our robots have a lot of trouble moving in a straight line even with this guide.

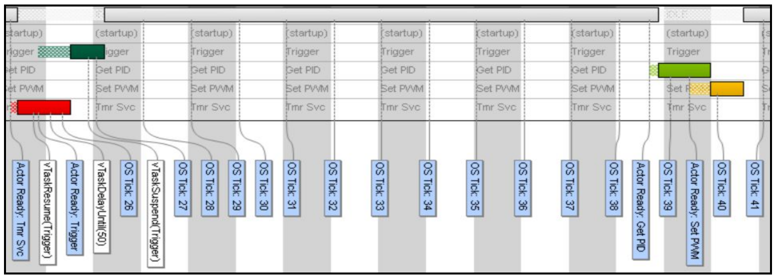

Finished Hardware Design

Software

1. Atmel Studio 7.0
   a. This is what we used to program and debug our microcontroller.
   b. Provides necessary files and libraries for us to program our ARM processor and its peripherals.
   c. Was a nice way to add complexity to our project, rather than using the much simpler Arduino IDE

2. Atmel Software Framework/Embedded C
   a. Provides an interface to all the microcontroller peripherals
   b. Prepackaged libraries and tutorials that we used to navigate the complex programming environment of the SAMD21G18A and ARM microcontroller in general.
   c. We avoided Arduino IDE, even though we have an Arduino device, we felt the extra challenge of navigating an actual Embedded C environment improved our project.
   d. Within this environment we were forced to make heavy use of the datasheet, and the Eagle CAD files to reference pin assignments.
3. FreeRTOS 7.4.2
   a. Real Time Operating System
   b. Open-source and FREE, other RTOS implementations cost big money and we do not have that.
   c. The Atmel Software Framework had a preexisting port of FreeRTOS for our hardware, which made it very easy for us to install.

4. Tracealyzer 4
   a. This is a software and tool that we used for our data acquisition.
   b. Took a while to get working, but we were eventually able to get the memory dumps from our processors and view them in this program.
   c. Required a lot of setup and configuration, lots of configuration files and modifications that allow the Trace to be read.
   d. The most frustrating documentation on the planet.
   e. We were never able to get the live streaming mode to work for this software due to a lack of debugger. This severely limited the data we would have been able to collect such as Communication Flow between tasks to better show our data being sent between them and I/O Intensity



Tracealyzer 4 Trace View Output

Experimental Design

We were able to seperate the progression of our project into a series of experiments, each detailing our increase knowledge of FreeRTOS, and the performance increases we were able to obtain with each iteration and improvement.

1. **Software Design**

   Brief Explanation of Non-FreeRTOS Elements

   a. LIDAR

      i. Distance data is sent over USART line clocked at 115200 baud. Our microcontroller has a USART peripheral that we used to collect each data packet we received.

      ii. The packet structure is shown below, we only used the 2 distance bytes in our project.

Byte1-2	Byte3	Byte4	Byte5	Byte6	Byte7	Byte8	Byte9
0x59 59	Dist_L	Dist_H	Strength_ L	Strength_ H	Reserved	Raw.Qual	CheckSum_ L

      iii. This information is handled in a variety of different ways depending on the configuration we are using. Some experiments use USART as an ISR and some use a polling configuration. The final implementation we went with uses callback mode for efficiency that is triggered from an rx_job that is triggered every 25 clock ticks that checks for double the buffer length worth of data, to ensure the 8 bytes we want are there.

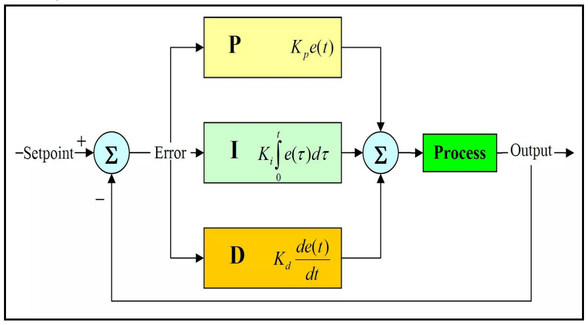
      iv. After this, the callback is triggered and locates the first two header bytes from the data received from the job to then gather the proper distance data.

   b. PID

      i. We needed a way to relate the information received from our Lidar sensor to the speed of our motors. Even the our control system requirements are very simple, we chose to use a PID controller to perform this function.

      ii. In summary a Proportional Integral Derivative Control Loop is a function that calculates the immediate error, summed error, and change in error over time, and then relates the sum of all three of those to some control signal based on 3 coefficients.

      iii. In this implementation, given the simplicity, we found using anything other than the P value to be unnecessary.



PID System Diagram

iv. Our setpoint for our project was the target distance that we wanted to maintain between vehicles. Our Error was our (Target Distance - Actual Distance). This proposed error distance of (Target-Max Distance)/Max PWM value of 1000 yielded our decided proportional constant value.

c. Motor Control(PWM)

i. Our Motor Driver IC we purchased allows for the easy control of a motor through a PWM signal.

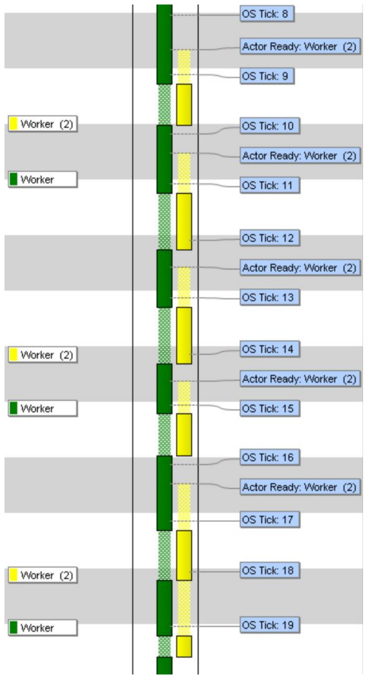
ii. We used the TCC peripheral on our microcontroller to implement this.

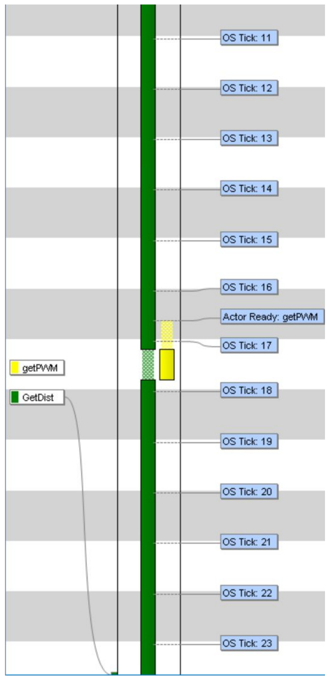
iii. The Duty Cycle of this signal, which is what controls the speed, can be altered by changing the match value relative to the counter overflow value.

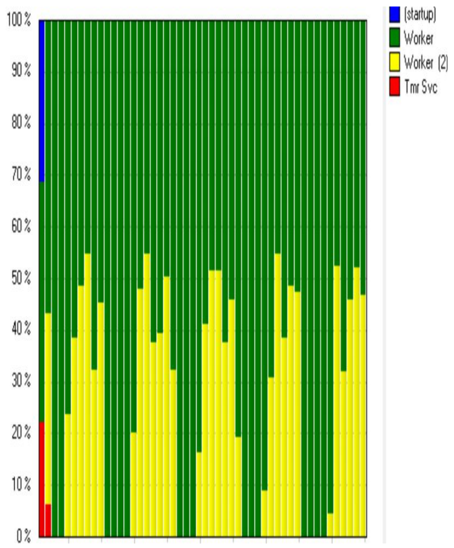
iv. This TCC signal runs continuously in parallel with the program, the match value is periodically updated to whatever value the PID is outputting.

2. **Success Metrics**:
   a. Distance and Speed maintenance between two vehicles.
   b. Implement FreeRTOS and use Trace information for data collection
   c. Utilize various FreeRTOS features to test differences in performance
      i. Find the best configuration for our application
      ii. Show the differences in performance across different configurations via Trace
   d. Evaluate whether or not an RTOS was appropriate for this project.
3. **Experiments**
   a. Initial non RTOS hardware/software test
      i. Initial Barebones test of hardware and software to see if they function
      ii. PID control implementation
         1. Calculate Setpoints and P coefficient
         2. Evaluate the utility of also including the I & D coefficients
   b. Initial FreeRTOS Implementation
      i. Basic use of Queues
   c. USART Callback Mode & Semaphores
      i. Semaphores
      ii. Callbacks and ISR Implementation
   d. Optimized use of Priorities
      i. Determine which tasks should take priority over others
      ii. Alter their priorities to see if performance increases.
   e. Disabling Preemptive Scheduling
      i. Disable Preemption to test effects on performance
   f. Timers
      i. Use of Timer objects to test if performance is improved.
4. **Results**
   a. Initial non RTOS hardware/software test
      i. PID
         1. Target Distance = 40cm
         2. Max Distance when full speed is reached - 90cm
         3. Proportional Constant = -20.00f
         4. Integral/ Derivative Constant = 0

   b. Initial FreeRTOS experimentation
      i. The first implementation of FreeRTOS integrated into our baseline project was based solely around basic queues to allow tasks to communicate and proceed after one another.
      ii. We separated our baseline implementation into 2 tasks with the same priority and had 2 variations on its implementation. The first, getting our distance value and putting it into global buffer based on a boolean. Results detailing the task execution order and CPU Utilization shown below both bullets.
      iii. The other directly sent the distance data between tasks instead of relying on global values.







Trace View of bool+global var *Trace View distance sent directly* *CPU Utilization*

iv. As it can be seen, the removal of a messy global variable to a direct sending of distance data drastically improved the task execution order. At this time, our CPU Utilization for both implementations was abysmal, utilizing no Idle time and all around not very efficient. Execution times below are based on the task name that performed the set of the PWM to make the vehicle move. For our different version, we utilized different task names and believe 'Worker2' could have been more appropriately named.

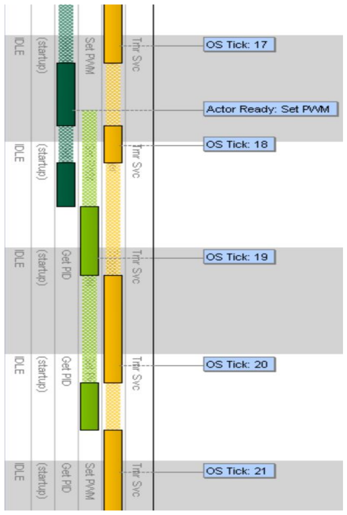
v. Boolean/Global Variable based Results
   1. Worker2 Execution Time: 1.195ms
   2. Worker2 Response Time: 2.323ms

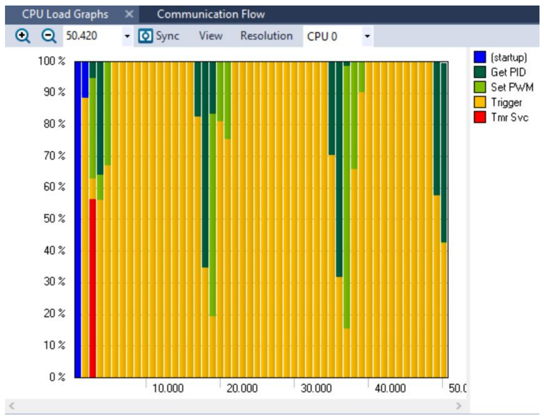
vi. Direct pass of distance data between tasks
   1. getPWM Execution Time: 571µs
   2. getPWM Response Time: 1.475ms

vii. As it can be seen, the difference between using messy global variables based on sending bools vs directly sending our distance between tasks proved very significant in improving execution, response time, and task execution order.

c. USART Callback Mode & Semaphores

i. Our implementation utilizing Callback and Semaphores sent us down a good path of slight improvements as we learned to better utilize what FreeRTOS had to offer. Semaphores were used to check for the first header bit in a job before being sent to the Callback.





Lots of Fragmentation *Still terrible CPU Utilization*

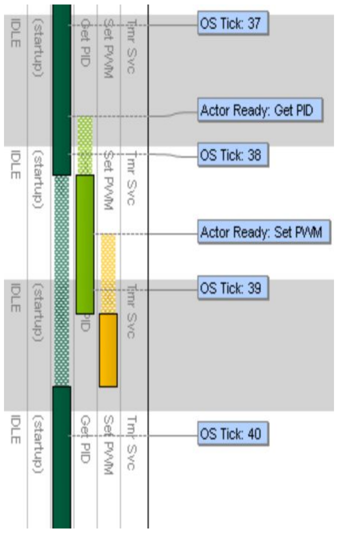
ii. While yielding some Fragmentation and non ideal CPU utilization, these results proved promising as they had improved synchronization. The tasks switching back and forth were in line with the implementation of our semaphores (detailed above) but ultimately while logical, was way too chaotic of an execution order.

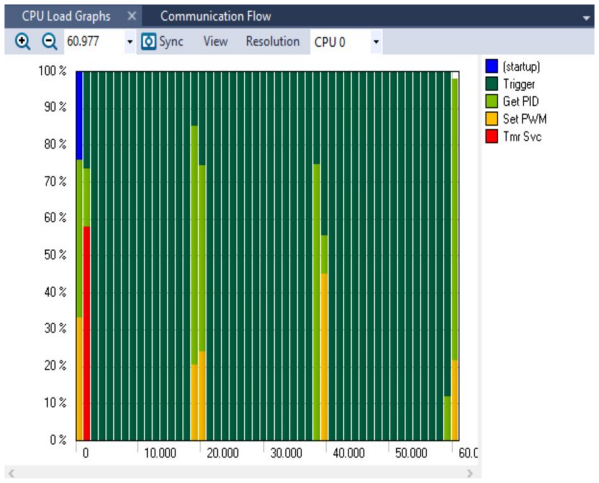
iii. We decided it was best to measure execution and response time in regards to our SetPWM task as that in when our PWM value is set that makes our vehicle go. The values below are provided averages from Tracelyzer

1. SetPWM Execution Time: 773µs
2. SetPWM Response Time: 2.169ms

d. Optimized use of Priorities

i. One major flaw that gave us the results in the previous experiment was that we did not utilize task priorities. Setting the task priorities higher for the GetPWM and SetPWM tasks performed after our Rx_Job Task yielded a significant jump in results all across the board.





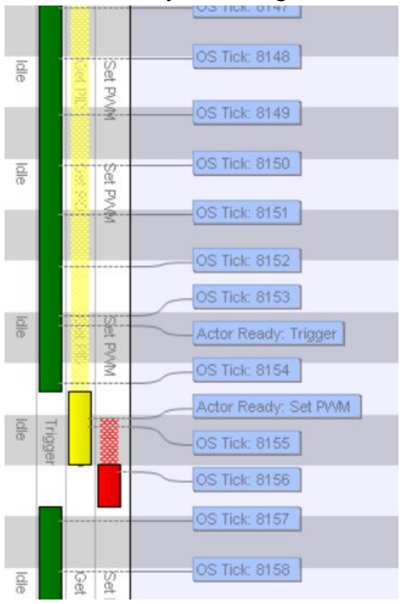
Improved Execution Order *Still Bad CPU Utilization (don't worry this gets fixed)*

ii. Changing the priorities so that the GetPWM and SetPWM performed as soon as they were able to, preempting the Rx_Job task, removed all of the scattered fragmentation seen before and drastically improved our execution and response times. CPU utilization is still terrible due to tasks always waiting to be executed but is eventually accounted for in our last experiment.

1. SetPWM Execution Time: 444µs
2. SetPWM Response Time: 787µs

e. Disabling Preemptive Scheduling

i. Not really crucial to our results but for self confirmation purposes we disabled preemption to see what would occur. As we expected, this did not yield any improvements showed tasks always awaiting for their occurence, yielding poor results.

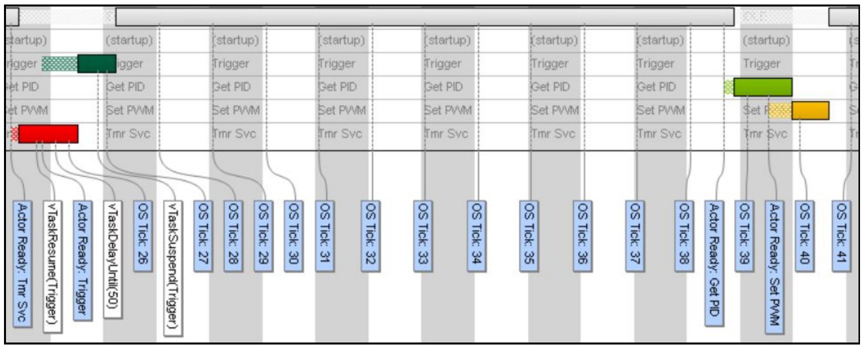


As Expected

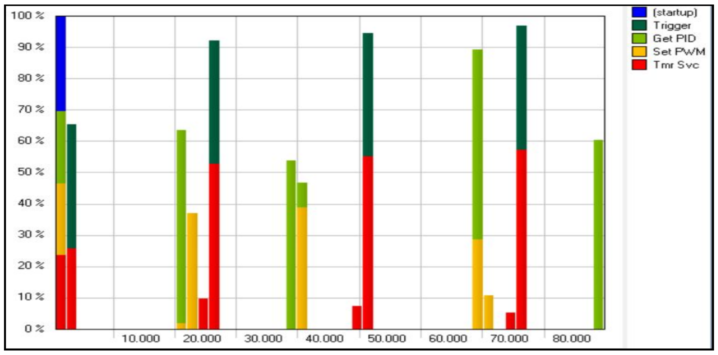
f. Timers

i. This experiment had the results we were the most satisfied with and this is the configuration we decided to go with for our demonstration.

ii. We finally saw a significant decrease in total CPU utilization, and the actual appearance of the IDLE task which was absent from our previous experiments.



Trace View of Control Events



CPU Utilization for Timer Configuration

iii. Since the distance data is now only sent periodically, using a timer makes sense here as we can tune the rate at which new data is sent to the control tasks. While we had this increase of CPU Utilization time and appearance of IDLE, our Execution Time and and Response Time increased slightly. We hypothesize this slight increase to use of the timer triggering every 25 Clock Ticks. This helps our CPU Utilization time but since the the tasks are based around this instead of triggering exactly when they can, it yields a slightly larger time that we think for this project, is marginal enough to deem this implementation an overall improvement. With more tweaking of the timer, getting an improved Execution and Response time over the previous implementation is easily doable.
   1. IDLE now accounts for 83.3% CPU Usage!
   2. SetPWM Execution Time: 502µs
   3. SetPWM Response Time: 877µs

5. Extensions and Additions
   a. We are very happy with our final product and learned a lot throughout each of our implementations and it was very cool to see gradual improvements to our system with each new one, but like all final products, more improvements definitely would have been done, time permitting.
   b. Future additions that would serve this project well would be an implementation of another PWM to allow reversing movement to better maintain distance if our device got too close.
   c. The addition of an Inertial Measurement Unit (IMU) that would ideally contain an accelerometer, compass, and gyroscope would have allowed us to obtain a whole slew of extra data that could be used to further enhance our understanding and micro-improvements. An example of this would be using the compass to autocorrect the wheels to prevent gradual turning.

Missing Milestones

Overall we feel we met all of our milestones, however we did have to make some slight adjustments to our overall goals as the project developed.

Failures

We believe that we have met all the previous milestones we set for ourselves in our initial proposal. Initially we wanted to compare the performance of a FreeRTOS implementation and a vanilla approach, but this proved difficult to quantify. We were however able to use our project as a foundation for analyzing and comparing the different performances we got out of FreeRTOS. This approach provided more than enough substance and data, and we feel that we were able to provide good information on how to implement FreeRTOS in a similar application.

Negative Results

We had a few results we were a little unsatisfied with in the end, however we believe them to be largely out of our control. Due to the cheap nature of the robot chassis we purchased, our actual distance maintenance was very finicky, and the robots had a hard time maintaining a straight course. To remedy this we even built a track, and they still tended to bump into the walls. However, the actual physical utility of our project was not our main success metric, and we are able make a ton of conclusions about the different ways an RTOS can be used in a project similar to ours.