“Uncrashable”

Remote Controlled Car

Thomas Chau
Ben Sack
Peter Tsonev

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Overview

The original inspiration for this project came when Peter and Thomas, stepping out of their favorite burrito joint into a chilly October evening, witnessed a Nitro RC Car blazing down Amsterdam Avenue. The initial idea was to take one of these cars, override its original hardware, and to interface a *truly* remote control via the 3G network; they had imagined attaching a camera and driving an RC car from New York to Atlantic City via the cell network from the comfort of their dorm room.

Of course, the feasibility of doing this within a semester reduced the project to developing a car controlled by FPGA and sensors. The new goal is to have a car that drives quickly at an obstacle and halts just in time to prevent a collision.

An RC car was purchased from the Radio Shack. The ultrasonic sensor, which samples every 50 milliseconds and returns the distance to an object in inches, is used by the FPGA in its calculations that control the car hardware. We hacked the car hardware in order to use the Altera board rather than the original RF controller; the board is connected via Ethernet cable to a breadboard on the car.

We used an oscilloscope and discovered that the car used PWM to manage both steering and throttle at a frequency of 50 Hz. As the FPGA is capable of 50 MHz, we could supply a PWM signal to the car by converting a PWM width with a hardware counter.

With this module in place, having proven that the board can control the car hardware, the next step was to program the car to slow down as it approaches an object, preventing a collision. We integrated sensor output into a software algorithm that computes a new throttle level with each sensor reading. The resulting feedback loop accounts for distance and speed in order to regulate the engine output.

Design

Hardware

Needless to say, the hardware system consists of a NIOSII processor, memory, jtag uart (for debugging), and custom peripherals to interface with the sensor and the car hardware. We needed two custom SOPC components:

PWM

The system has two servos which control the steering and the direction in which the range sensor is pointing to and one throttle control. To enable control over these subsystems, we used PWM signals. The PWM units are VHDL components that are connected to the Avalon bus independently from the sensor unit.

The module uses a counter that increments on every “PWM Clock” cycle which is a signal that is derived from the main system’s 50mhz clock. When the clock reaches a count that corresponds to the end of the PWM period of one cycle, it resets. A register representing the width of the pulse is compared to the counter value and if the counter is less than the register the system generates a logic high signal. The duty cycle can be varied in discrete steps representing less than .1 % and is directly related to the counters resolution.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

```
-- Create a PWM clock. 500,000HZ.
 Counter : process (clk)
  begin
    if rising_edge(clk) then
      if reset_n = '0' then
              count <=  0;
              pwm_clk <= '0';
       elsif count = 49 then
              count <= 0;
              pwm_clk <= not pwm_clk;
       else
              count <= count + 1;
      end if;
    end if;
  end process Counter;

--Count 10000 steps for each clock cycle. for 0.01% increments of duty cycle.
  pwm_clock : process (clk)
      begin
          if rising_edge(clk) then
                                    if pwm_clk = '1' then
             if reset_n = '0' then
                pwm_count <= 0;
              elsif pwm_count = 9999 then -- should be 9999 to generate 50Hz PWM.
                pwm_count <= 0;
              else
                pwm_count <= pwm_count + 1;
              end if;
                                    end if;
                            end if ;
   end process ;
--Based in the Duty cycle register (RAM(0)) create a wave for the steering controls.
 steering_pwm_generate : process (clk)
      begin
                                    if pwm_clk = '1' then
                                            if pwm_count < RAM(0)  then -- duty cycle : 0 - 9999, so can control .01 of a %
                                                      steering_pwm <= '1';
                                            else
                                                      steering_pwm <= '0';
                                            end if;
                                    end if;
 end process ;
```

the module contains two additional processes to create PWM signals for throttle and servo controlling the direction the range finder points to.

Sensor

The peripheral for the sensor is implemented as a SOPC component in VHDL. The sensor provides distance measures every 50ms and it does so in three ways. 1) analog 2) UART serial 3) pulse width modulation. We used the third way. At the beginning of each 50ms sensor cycle, the sensor pulls the signal high. While high, and every 147us duration correspond to 1 inch measured. Once the until the sensor pulls the signal down, the distance can be determined by counting these 147us time periods.. The furthest detectable distance is about 250 inches which corresponds to about 37 ms of high pulse. During the rest of the cycle the pulse is guaranteed to be low and the sensor uses the remaining time to send the readings via UART and adjust the voltage levels for the analog output.

Our sensor peripheral counts how long the pulse described above is high and converts this count to inches. As soon as the pulse goes low, i.e. we have a new reading, the peripheral dumps the new distance reading into a register and raises an interrupt to be serviced by the NIOSII processor. At this

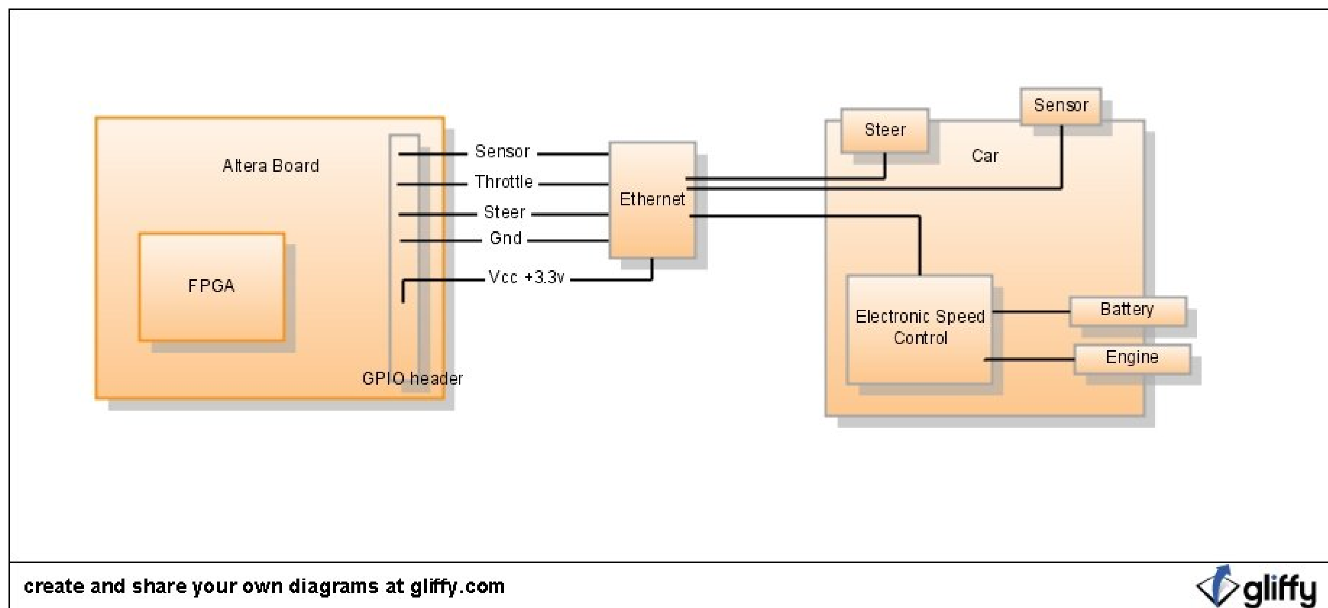
created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

point, the processor can go and fetch the most recent distance reading from the register and once done it clears the interrupt signal. In terms of timing, the worst case occurs when the sensor holds its pulse high for the maximum allowed duration in one 50ms cycle – 37ms. The remaining 13ms are more than enough to carry out any reasonable processing in the interrupt handler without missing further readings from the sensor.

Physical Hardware of Car and Board Interface

Besides the FPGA setup, we needed a lot of low-level hardware interfacing. Since the car is intended to move around, we thought of sending the control signals and receiving the sensor readings wirelessly, but doing so would require another processor on the cars end. Thus, we used an Ethernet cable with 4 twisted pairs to transmit the two pwm signals for steering and throttle, the power for the sensor (3.3 v from the FPGA), the pulse generated by the sensor, and the common ground. The GPIO ports from the FPGA were taken to a bread board via a ribbon cable to preserver the pins. The wiring on the car itself used a small breadboard too (We cut a piece from a regular bread board with a saw. Yes we did!).

The schematic below presents an illustration.



Software

We can define two layers in the software: data filtering and feedback control.

Data Filtering

Sensor data filtering for the projects took a few generations of code to develop. Initially, we did not suspect the sensor to be too faulty nor did we suspect that many reading would be distorted based on the conditions of the environment around the sensor. We spent time looking at printouts of sensor readings on the screen and tried to build a filtering mechanism based on our assumptions of the types of

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

errors that were occurring. One of the problems in this technique is that patterns are very difficult to notice just by inspecting the raw data. In addition, as the project developed, we used three different wiring schemes and two different power supplies. In all but the last wiring/power configurations, the sensor readings were affected by interference. This made the process very difficult. Another issue rises from the testing conditions of the sensor: Initially, the sensor was tested away from the car, and from the environment it would be used in, which in hind sight was a mistake. The sensors behavior changed once it was attached to the car and signaled to the FPGA which was 15ft of Ethernet cable away.

We started by writing sensor and control module in VHDL which ran independently from the NIOS 2 processor. The system was controlled form the DE2 keys much like the first lab in the class and displayed the distance reading and a speed derived from the last two readings on the 7 segment displays. We then wrote a module to communicate with the nios II processor and

finally decided the best way to proceed was to record large amounts of readings while driving the car continuously back and forth. Since we had access to an embedded system running UcLinux, we wrote a module and software to communicate with it and dumped the reading into a file. We used the files generated to create a filter.

Once we eliminated the condition which generated bad readings such as sensor location on the car, wiring and powering schemes, the reading errors could be dealt with easily. We either had really big readings, or one single bad reading between two good readings and finally, many consecutive similar readings in a row, which can either mean the car is not moving or moving too slow.

Elimination of similar readings:

```
while(!flag)
// there is a problem that two consecutive readings are the same if car is not moving fast. so, lets eliminate some of those.
// from inspecting the graph and experimenting with the car, the difference between movement and slow speed is the //number of similar
readings. above 6 readings, the car is stationary for certain.
{
        readSensor();
        if (prevReading == distReading){
                similar_count++;
                no_movement_flag = 1;
        }
        else
        {
                flag = 1;
                similar_count = 0;
                similar_counter_threshhold = 5;
                no_movement_flag = 0;
        }
        if (similar_count > 0) // maintain notion of time for speed calculations
                timeDeltaReading += timeReading;
        else
                timeDeltaReading = timeReading;

        if (similar_count > similar_counter_threshhold)
        {

                flag = 1;
                //report speed every 6 readings even if car not moving,
                //otherwise the algorithm will not be able to detect the car has stopped.
                similar_counter_threshhold+=5;
        }
}
```

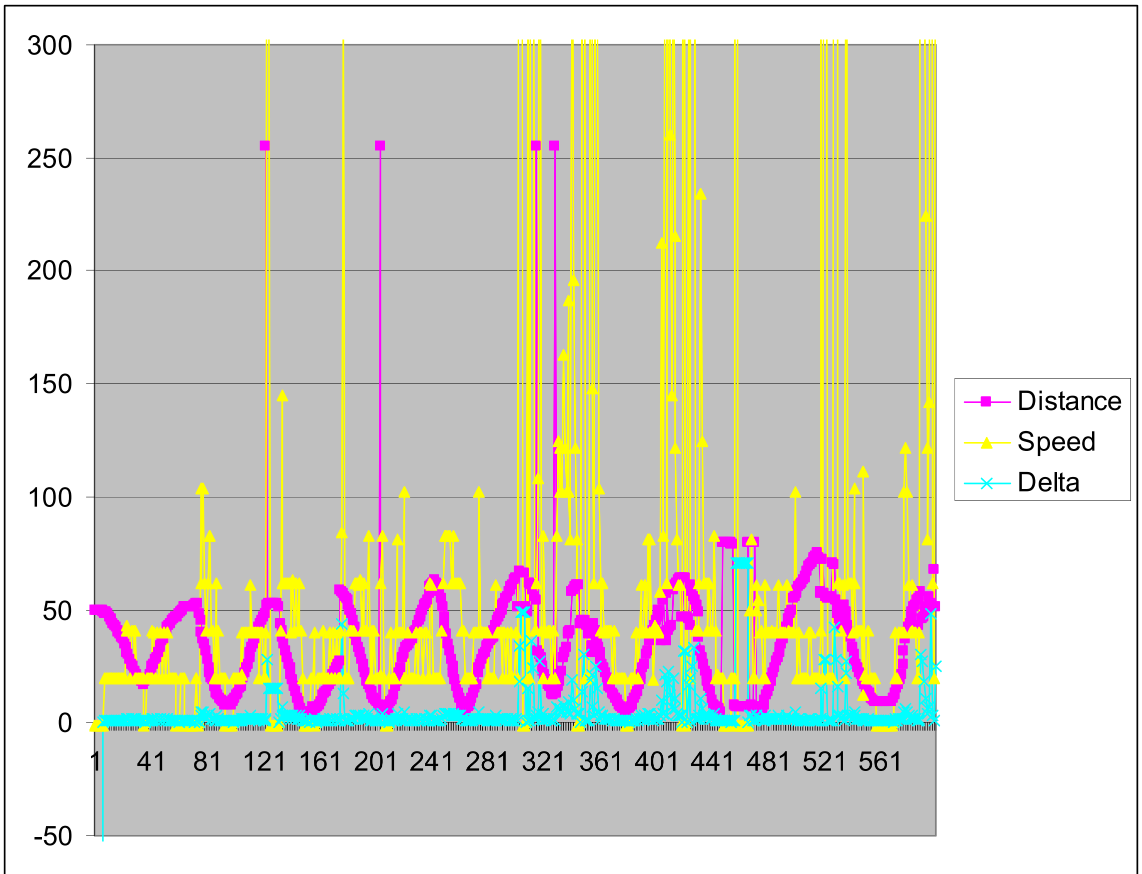
the second filtering eliminates small spikes in the graph, :

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

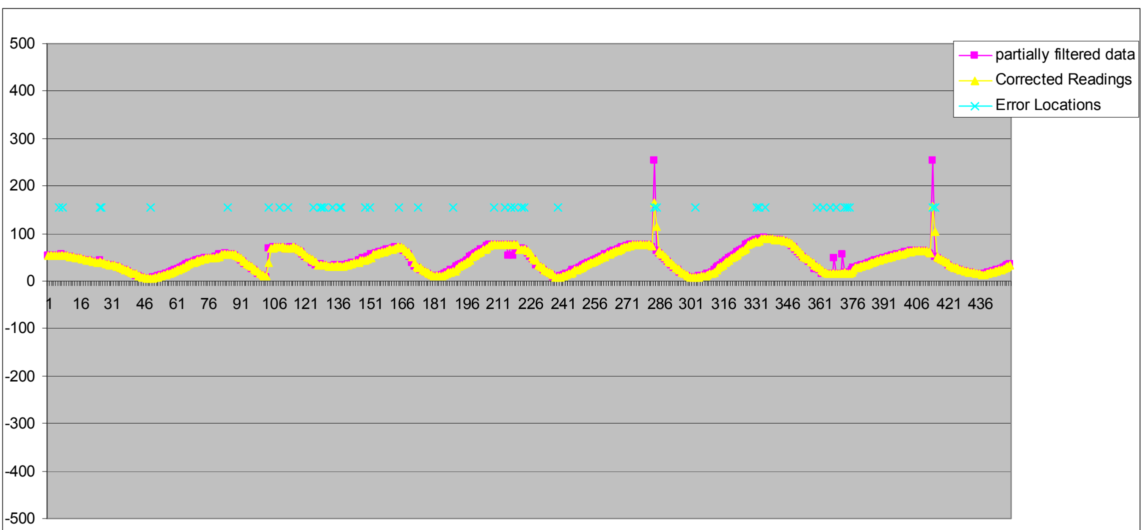
```
//why use the present? lets use the future.
		latest_results[0][3] = latest_results[0][2];
		latest_results[0][2] = latest_results[0][1];
		latest_results[0][1] = latest_results[0][0];
		latest_results[0][0] = distReading;

		latest_results[1][3] = latest_results[1][2];
		latest_results[1][2] = latest_results[1][1];
		latest_results[1][1] = latest_results[1][0];
		latest_results[1][0] = elapsedtime;

// latest_results[0][3] assumed to be clean.
/*
		if (latest_results[0][3] - latest_results[0][2] > 0 ) //moving forward.
			{
				if (latest_results[0][2] - latest_results[0][2] > 6 ) // too big of a delta!
				latest_results[0][2] = latest_results[0][3];
			}
		else if  (latest_results[0][3] - latest_results[0][2] < 0 )
			{
				if (latest_results[0][2] -latest_results[0][2] -  > 6 ) // too big of a delta!
				latest_results[0][2] = latest_results[0][3];

			}
// Removed for final demo, most bad reading NOW are single, and this method  interferes more in that case*/
		// simplelest way is to get rid of parabula peaks i.e. the point is bigger than previous and next by a big factor.



			//this	reading		//next	reading		//this	reading		//previous reading
		if (	((latest_results[0][2] > latest_results[0][1]) && (latest_results[0][2] > latest_results[0][3]))  ||
			((latest_results[0][2] < latest_results[0][1]) && (latest_results[0][2] < latest_results[0][]))		)
		//then
			{
				flag_raised =1;
				latest_results[0][2] = (latest_results[0][3] + latest_results[0][1]) / 2;
			}
		else if (	((latest_results[0][2] > latest_results[0][0]) && (latest_results[0][2] > latest_results[0][3]))  ||
			((latest_results[0][2] < latest_results[0][0]) && (latest_results[0][2] < latest_results[0][3]))		)
		{
				flag_raised =1;
				latest_results[0][2] = (latest_results[0][3] + latest_results[0][0]) / 2;
		}
```

Graph 1: unfiltered results.
Notice the distance (pink) curve which represents forward and backward motion. It contains spikes and non-continues data which affects the “distance delta”(blue) and the speed(yellow).

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image



Graph 2:
The resulting graph obtained from over 700 readings. about 20% of all reading do not represent change from their previous readings and are filtered out of the graph.
The blue dots signal locations of reading that were dropped from the original graph.



created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

FeedBack Control:

PID theory

One approach to controlling a physical system is via dead reckoning which means that you control the system without verifying that the controls result in the desired results. If you know all aspects of the system, in theory, you can predict the behavior for any input. However, most systems are way too complex to fully describe. Thus, if we try to control such systems, the results will differ from our intentions and unless we monitor those deviations and act to correct them, dead reckoning is not a good option.

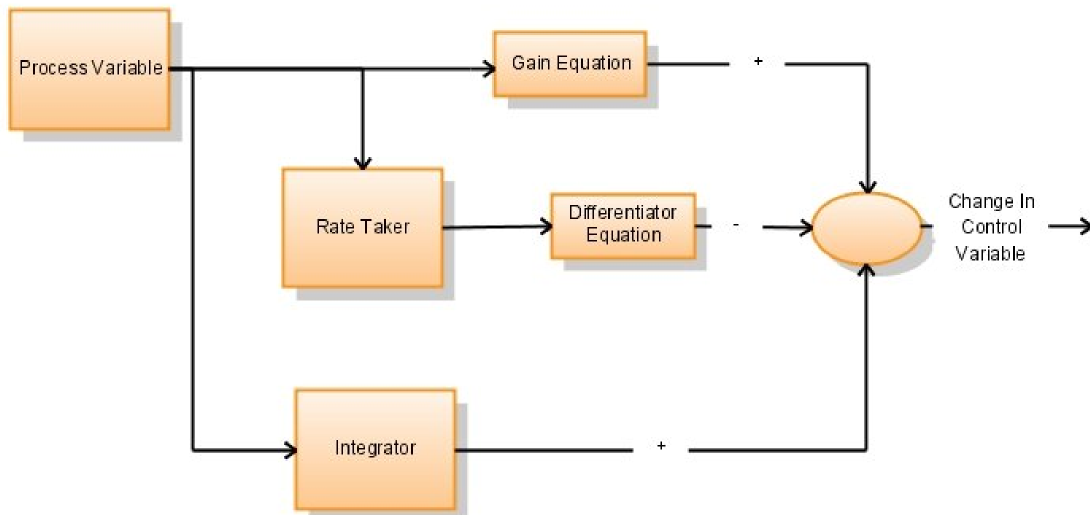
Another approach is to monitor the aspects we are trying to control and take corrective action if the results differ from our expectations. Thus, the way we control the system in the future depends on observed errors in the past which creates a feedback loop. In general, you have a ***process variable*** and a ***control variable***. The corrective action is taken on the control variable and the monitoring of the system is done via the process variable. The purpose is to update the correct the control variable in a way that gives the system the desired behavior.

The heart of our algorithm employs feedback approach since the physical system of the car is very complicated to predict how it will behave for certain inputs (e.g. throttle and steering). More specifically, we are using the PID model of feedback control. It stands for proportional-integral-derivative because the feedback is determined by three independent equations, e.g. three degrees of freedom. Each of the equations is sensitive to the error in a different way and suggests corrective action independent of the others. The sum of those is the total corrective action.

The three equations are generally referred to as the GAIN, the DERIVATIVE, and the INTEGRATOR. The GAIN equation always tries to correct the control variable in proportion to the error in the process variable. The DIFFERENTIATOR corrects the control variable proportionally to the rate of change of the process variable and usually opposes the GAIN (corrects in the opposite direction). The INTEGRATOR provides a bias to the control variable based on the persistence of deviations in the process variable throughout time. So the three equations fight and help each other in correcting the control variable.

Another important aspect is frequency of the feedback loop – how often you measure the process variable and correct the control variable. The control system slow can be described best by the following diagram:

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image



PID Specific to our case

In our system, we control the throttle and observe the distance to the wall. All other variables are ultimately derived from these two. So the process variable is the sensor distance and the control variable is the throttle. The error = distance – desired_distance_from _wall.

The schematic below describes the functionality of our feedback loop.
The GAIN uses filtered distance and the rate taker for the DIFFERENTIATOR uses a moving average on the differences in the filtered distance readings to produce the rate of approach, which is then used by the DIFFERENTIATOR itself. The INTEGRATOR is not applicable to this system.

As noted above, the PID equations are best suited for a linear system. However, ours is highly non-linear mainly because of the throttle. The duty cycle of this signal doesn't map to the torque produced by the engine in a linear way. Thus, small changes in throttle have almost no effect on the engine, and at certain values they produce dramatic effects. In essence, we have a 5 level speed control: neutral, steady forward, steady reverse, fast forward, fast backward. The PID equations cannot adjust the system smoothly in a timely manner. Therefore we introduced non-linearity in the DIFFERENTIATOR by making it inversely proportional to the error.

In general, a system controlled by PID will have an oscillatory behavior as noted above. We want critical damping, e.g. stop at the desired distance from the wall without overshooting and without being conservative. An approximation to this behavior was achieved by experimentally changing the parameters under our control. These were:

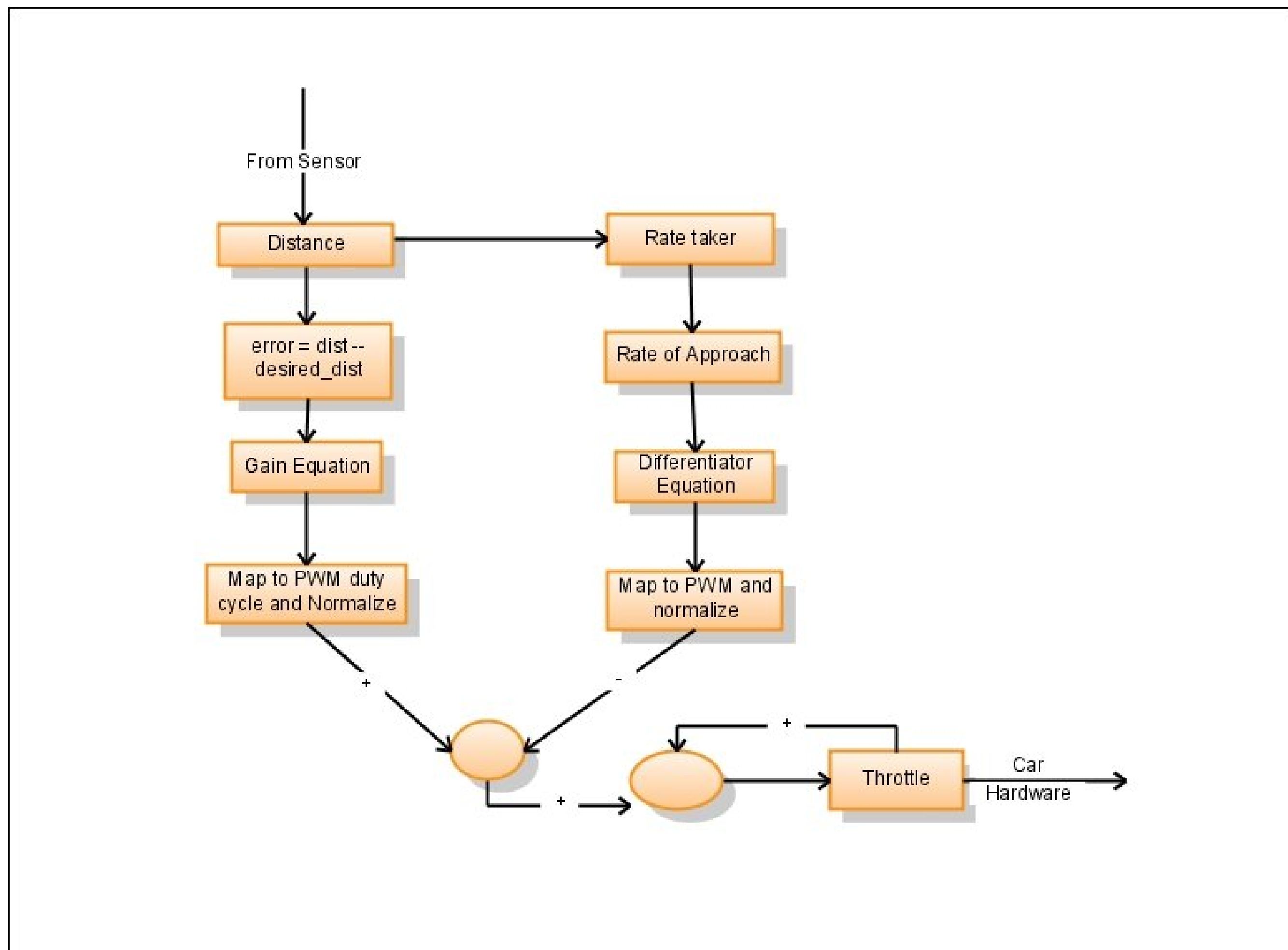
created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

a) frequency of feedback loop – every 250ms
b) maximum increments and decrements in throttle
c) maximum and minimum values of throttle
d) scale factor of the GAIN equation
e) scale factor of the DIFFERENTIATOR
f) normalization and mapping from to pwm units (duty cycle)

All of the above are degrees of freedom and had to be determined experimentally to enable the equations to react quickly enough without being over conservative.

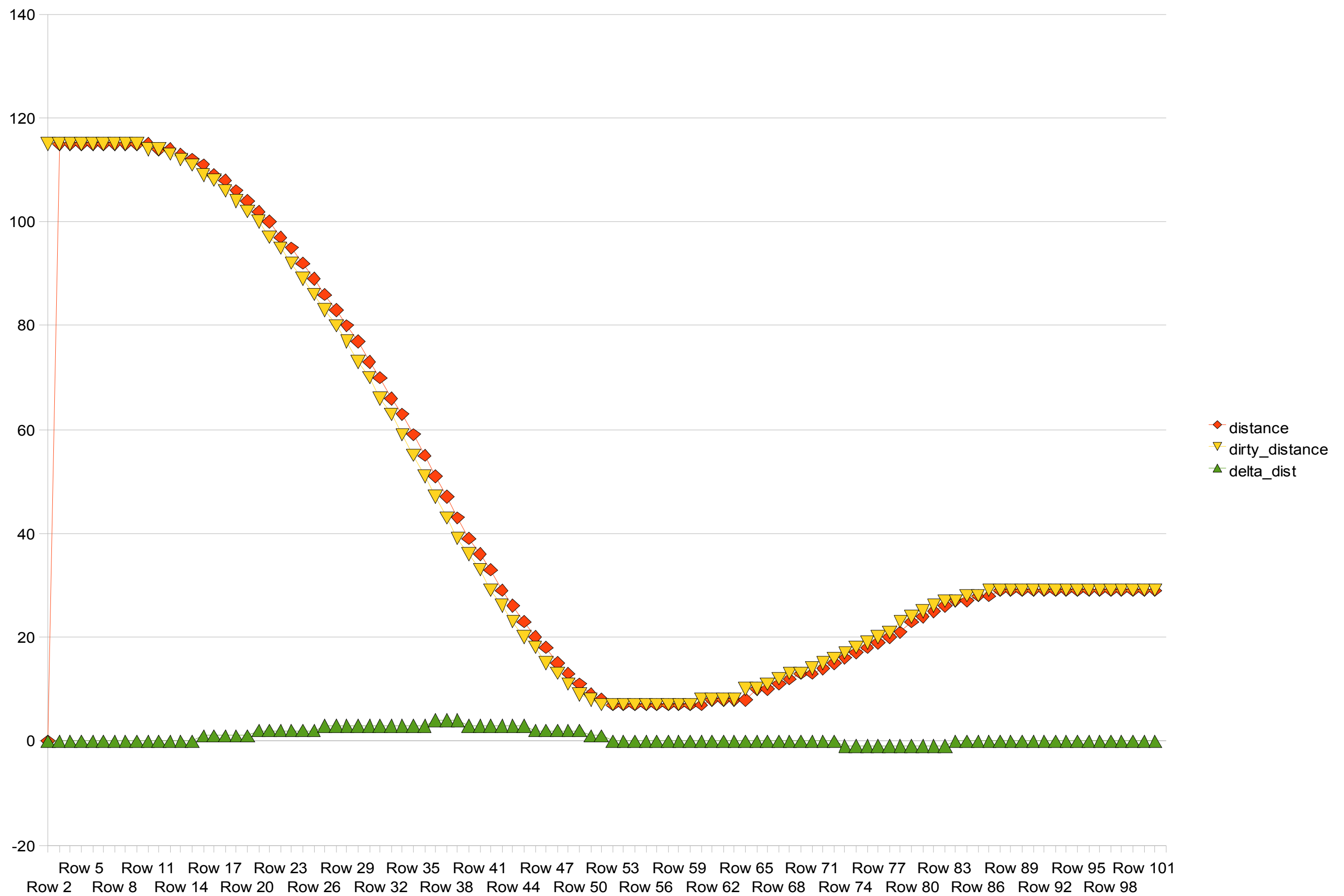
In its final version the algorithm uses the two PID equations as described above with the introduced non-linearity in the second one. The above parameters, we determined by trial and error. Since we logged data to file over jtag, we were able to analyze the data after each run of the experiment and examine which parameters need to be adjusted to get the behavior closest to the desired one.

The graph below shows the flow of control with the above parameters absorbed in the blocks.



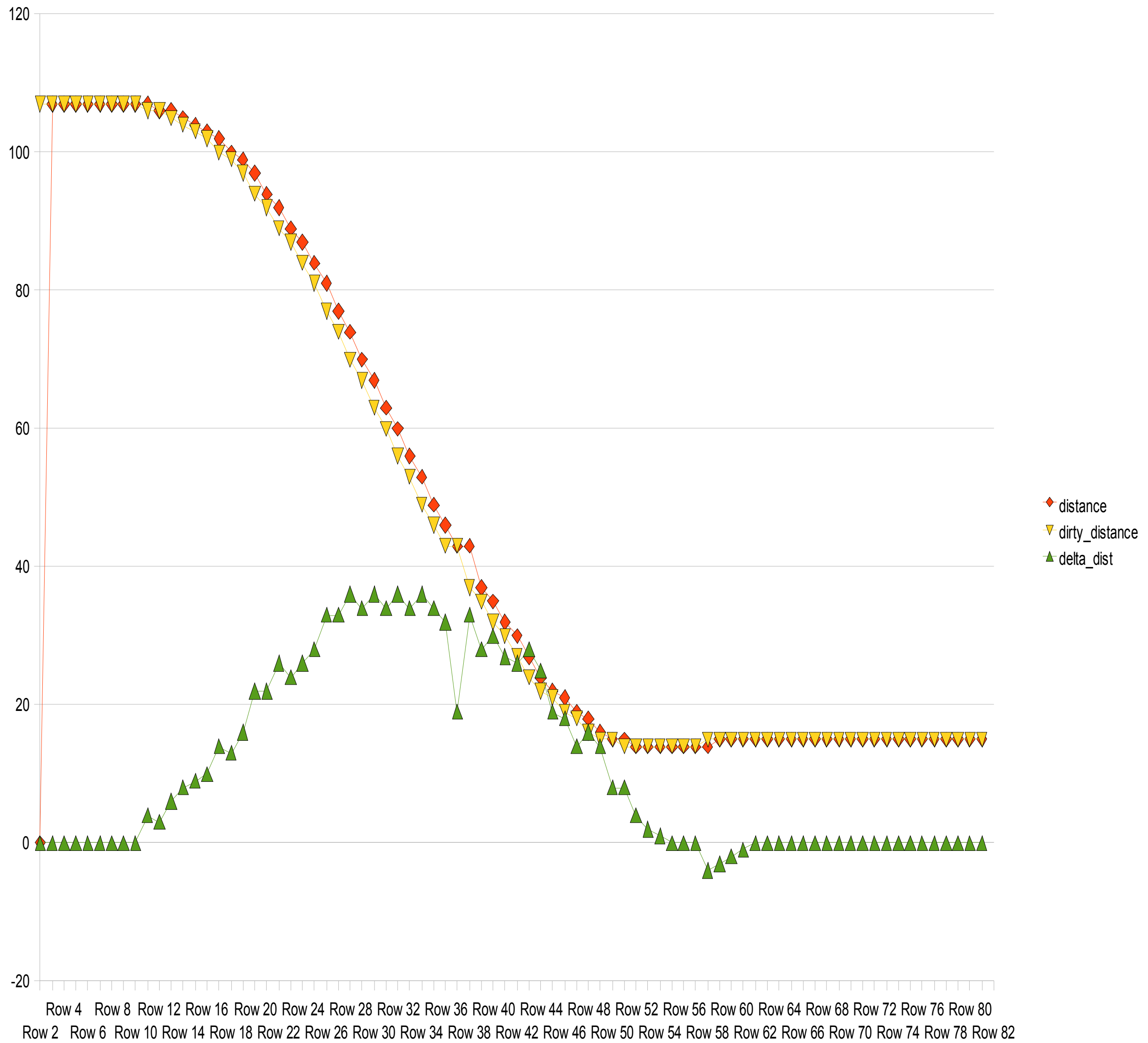
The following are the graphs produced for a few car runs:

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image



The yellow series is the measured distance from the wall and the orange series is the filtered distance. The green series represents the speed of approach. The desired distance was set to 20 inches (second horizontal line on the graph from the bottom. The graph above shows the results for a certain choice of the parameters. What can be seen is that the equations did not respond to stopping fast enough and the car overshot the target and then over corrected itself before halting.

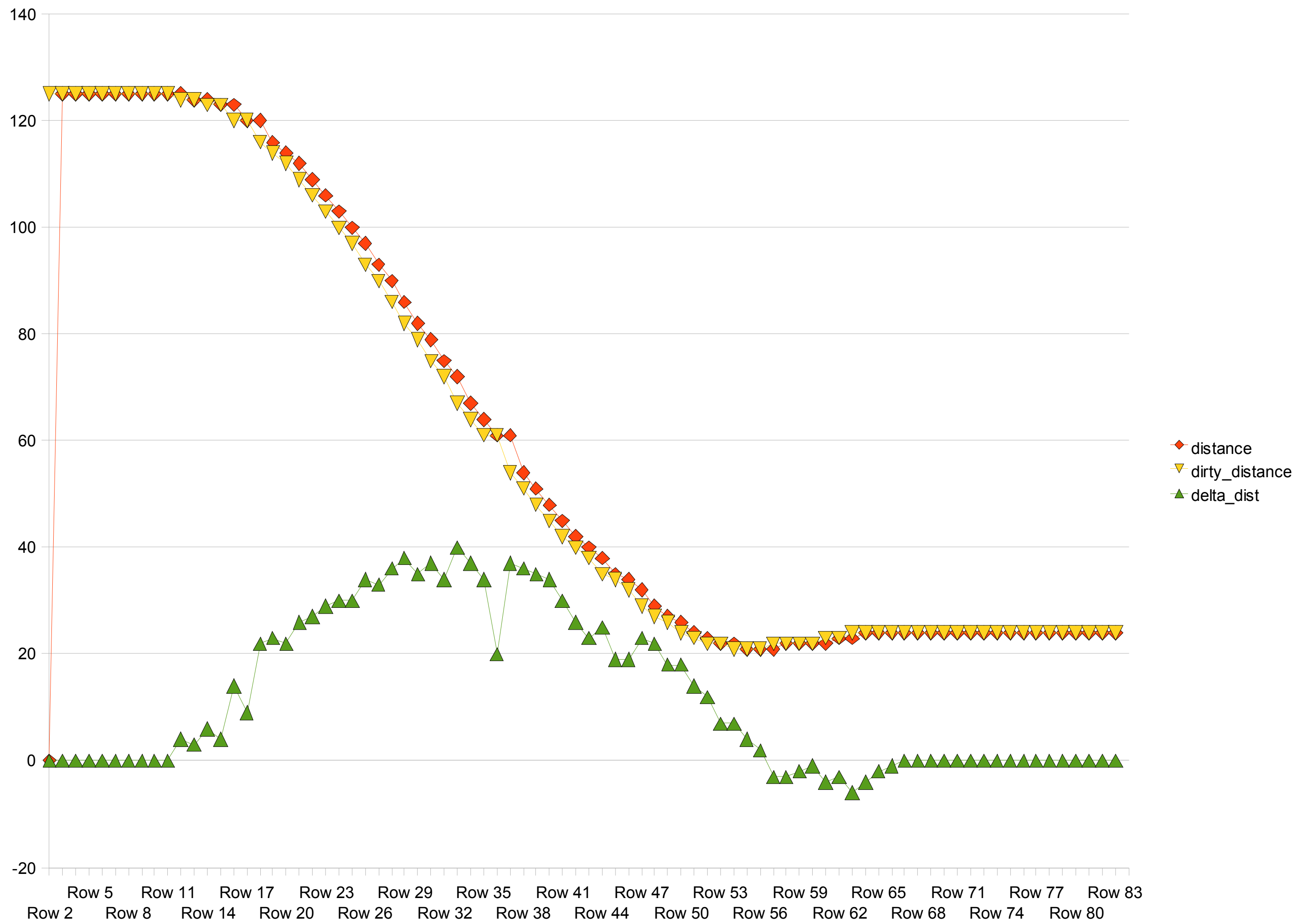
created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image



The speed is scaled in the graph above for better display. Here, the ringing effect from the previous graph is reduced. The car still overshot but this is within a reasonable margin of error having in mind the highly non-linear nature of the system.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

The following graph shows the best set of parameters we could find.

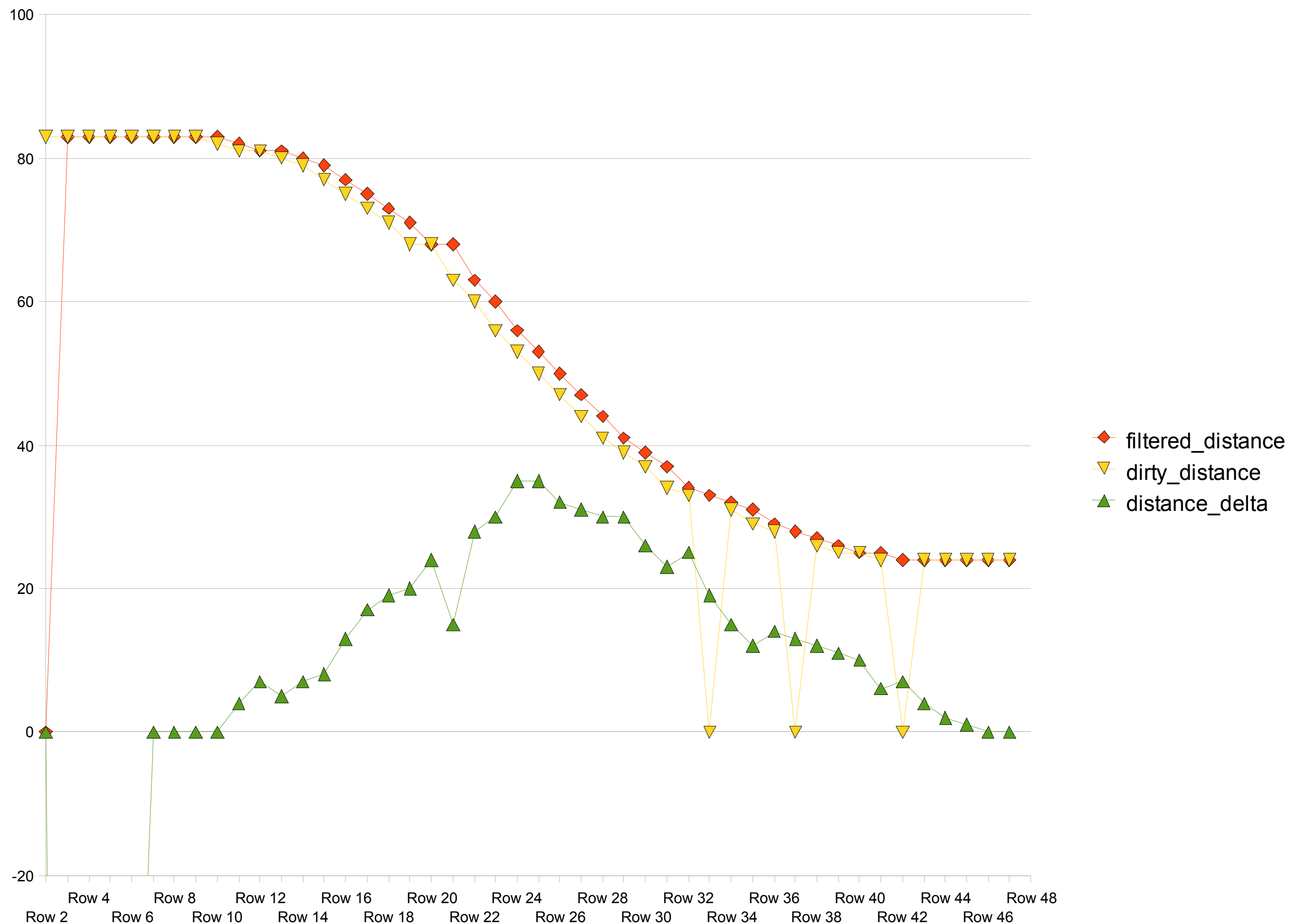


created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

The following graph demonstrates the performance of the algorithm with the same set of parameters as the previous graph but in the presence of very dirty sensor data.

As illustrated the PID equations, the non-linear modifications, and a set of parameters determined by experiment were combined to produce accurate feedback control without overshooting the target while maintaining conservative behavior in a highly non-linear environment and unreliable measurements.

The data filtering and smoothing provided a reliable underlying for the feedback control.



created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Difficulties

The biggest difficulty in this experiment and also a fundamental limitation to the accuracy is the non-linearity of the throttle. This makes it very hard to get the car to the desired distance without overshooting and being over conservative.

Another difficulty linked, to the first one, is mapping the corrective action suggested by PID to the right duty cycle for the throttle. The equations use distance and speed as inputs and it is almost impossible to translate duty cycle into real speed because of the severe non-linearity in the throttle. This non-linearity changes according to the battery charge level. since the engine is current intensive ,even if the battery discharges slightly the same duty cycle produces different torques.

Our project had an extra dimension – degree of freedom for mistakes – in the wiring and low level interfacing. At some point we tried to run power over the Ethernet cable or beside it to avoid the dependence on the battery condition. However, when the car is moving slower, the engine switches between on and off too quickly and draws huge amounts of current for short periods of time which introduces significant emf and contaminates the other signals in the Ethernet cable.

The hardest bug we had was a short circuit in the Ethernet cable which probably occurred from twisting and pressing by nearby chairs. The problem didn't prevent things from working and only exhibited itself under certain conditions.

At the beginning we did not have a safe way of changing the throttle and as a result we burned the transistors of the H-bridge. We replaced it, but unfortunately it defaulted again the night before the demo although the software framework enforced smooth and gradual throttle change. We know this because we did not see smoke, excessive heat or unusual behavior prior to the failure.

Finally, debugging our project was rather tedious due to the many layers. The software has to assume a trustworthy hardware, but we had to often go all along the debug chain down to the wires. As time passed and hardware stopped failing us, shortly, it had a come back with some very nasty bugs since they do not exhibit themselves at all the time, are not consistent and do not prevent the car from working. For example a series of bad reading at the initial acceleration phase of the test will not cause the car to hit the wall but would either make it miss the target or create an extreme breaking situation, which is dangerous for the throttle control.

MISCELLANEOUS

Manual Override
There is also a manual override mode. Another piece of functionality built into the software is the ability to drive the car via keyboard input; we decided to use the conventional computer gaming controls (keys WASD) for forward, left, back, right). Pressing the keys will increment the PWM levels in the appropriate direction. The function that writes to the PWM duty cycle register is designed to

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

prevent the user from changing the throttle too drastically, possibly damaging the engine or hardware. This mode was developed first in order to collect data on engine performance; we wanted to see what PWM settings corresponded to which engine outputs. This experimental data was later used to help engineer our throttling function.

Throttle Control
Since the throttle control unit plays with big currents, changing throttle by big amounts is destructive to the transistors of the H-bridge (which controls the motor). Therefore, whenever we corrected the throttle in the feedback loop, the new change was placed in small increments over a reasonable time interval in the time until the next feedback loop iteration.

Results

The results for the original challenge were shown in the graphs under the Software section. The evening before the demo, we lost the reverse throttle of the engine. Probably, one of the transistors in the H-bridge deteriorated. Without this capability, our algorithm cannot perform its feedback task properly since it can no longer issue corrective action.

However, we modified the challenge by using the PID theory for the steering and having the car approach the wall and converge in a movement parallel to the wall a certain distance from it. This has its own challenges and required a new algorithm which we crafted overnight. This is also what we demoed.
First of all, we had to keep the car far from the wall because if it is approaching at an angle, the sensor will face the wall at an angle and the readings will be incorrect due to partial reflection. Ultrasonic sensors work best with spherical surfaces or when facing a wall straight. This was easily fixed by having the car converge to a line 40 inches away from the wall.

The main difficulty was in turning the wheels in motion since it takes time to physically turn them and the lowest possible forward throttle we have is somewhat fast. Thus, we used the notion of a cycle. When we make a turn, the wheels move in the desired direction and stay like that for a certain number of cycles. The duration of the cycle is 50ms, Therefore, the sharpness of the turn is determined by the number of cycles over which it is executed. This approach makes the turning more predictable.

Although it is bad for the engine, we had to stifle the throttle – switch it on and off every few cycles so that the car doesn't gain speed too quickly.

The error is taken to be the distance to the line of convergence (this line is parallel to the wall). The car makes turns towards the line proportional to the distance from it. It monitors the rate of approach of the line to make sure that the turns resulted in the desired action (e.g. that we really turned towards the line sharply enough). If not it makes the car turn more. If no corrective action is required, the car maintains straight trajectory.

We can see the elements of PID feedback in this implementation. The GAIN equation looks at the error and makes the car turn proportional to this error. The DIFFERENTIATOR kicks in if the rate of approach towards the line is not satisfactory.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Responsibilities

PM module – Ben
Sensor module – Peter
Data filtering and smoothing – Ben
PID control – Peter
Software implementation – Thomas, Peter, Ben
Hardware interfacing and wiring – Thomas, Peter, Ben
Steering challenge – Peter and Ben
Debugging – Thomas, Peter, Ben

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image