Design Document

Design of Micro-Controller Systems for the Dagne Electric Vehicle

Leland Miller, Hemant Ramachandran, Nikolai Kallhovde, Wallace Luk, Navjot Singh, Aravind Sambamoorthy, Alejandro Aguilar

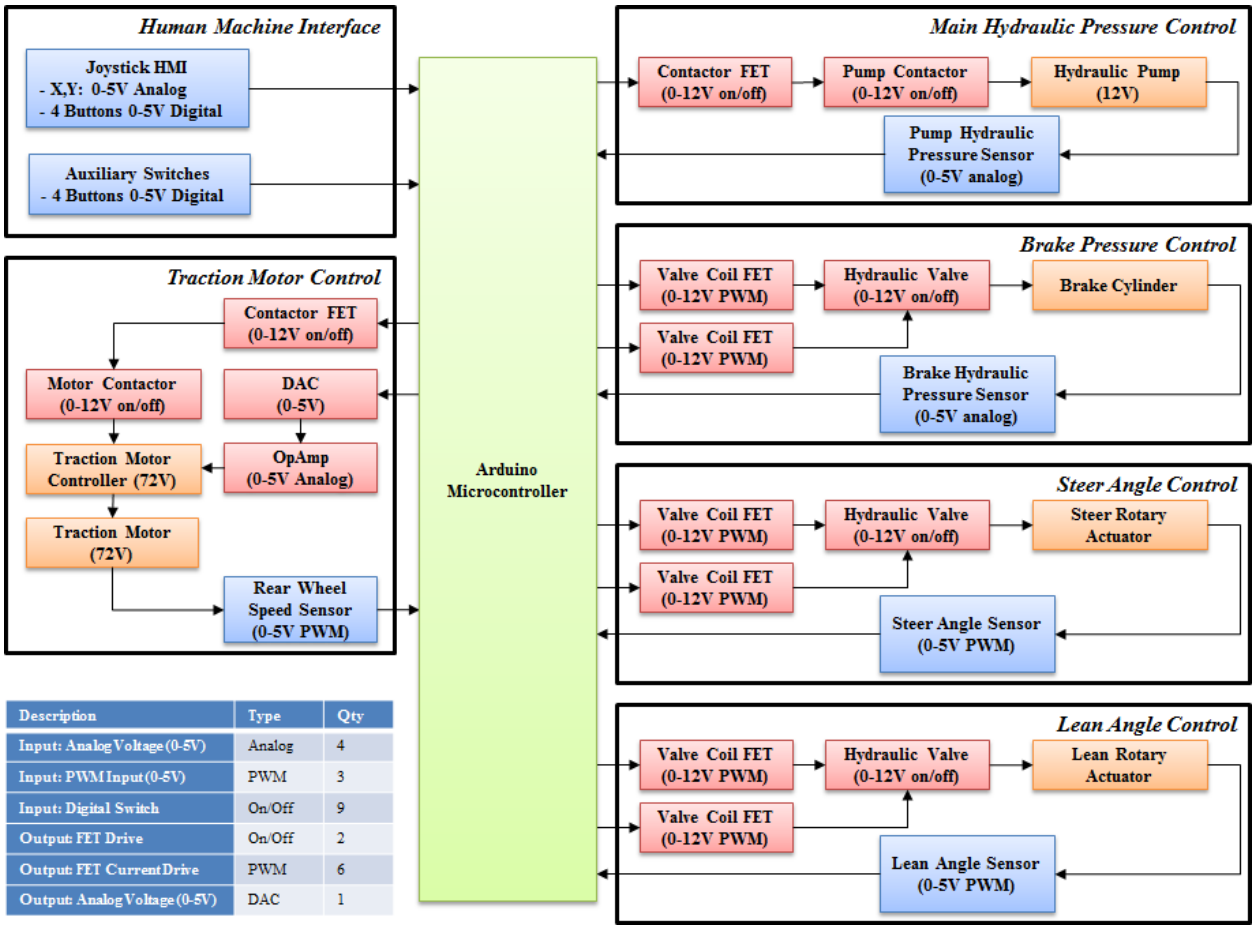
Electric Vehicle Design Document - Generated in second sprint, to address the lack of documentation for the 8051 microcontroller source.

Overview

The Dagne has currently decided to use an Arduino to replace the 8051 microcontroller previously used. We have been tasked with converting this legacy code. In the process we have decided to develop a comprehensive design document to cover how this code functions and the mathematics and concepts used to develop it.

I/O Configuration

Below is a diagram provided generously by Eric Sandoz - one of the co-founders of Revolution Motors and creator of the Dagne. It specifies input/output configuration of the arduino device with respect to the whole machine.

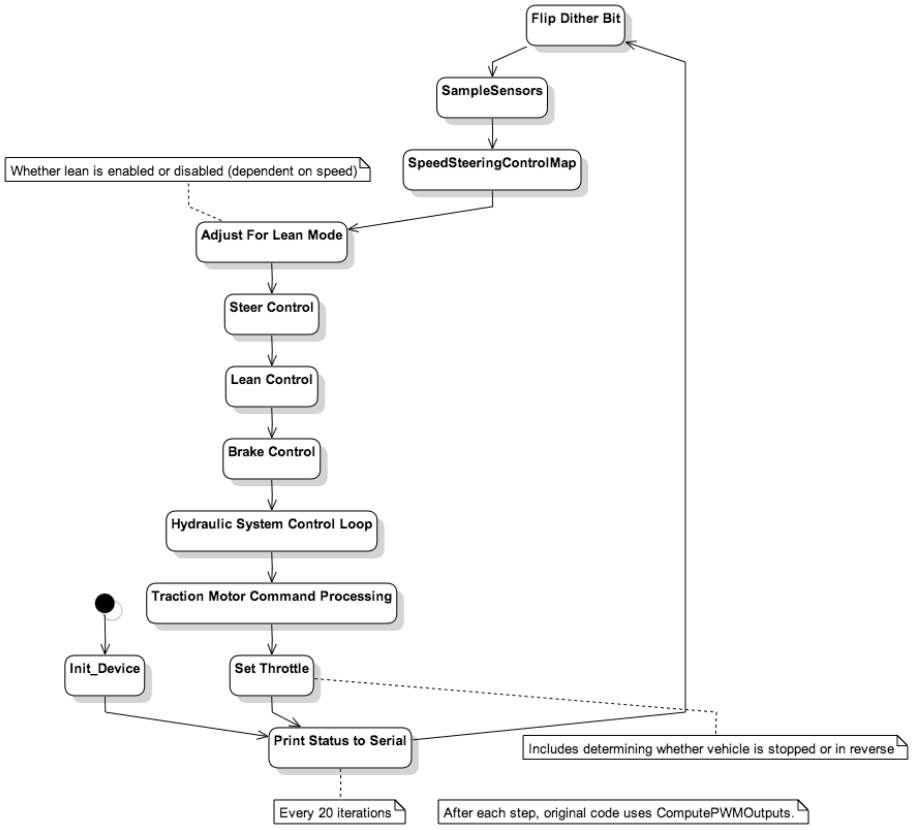


Description	Type	Qty
Input: Analog Voltage (0-5V)	Analog	4
Input: PWM Input (0-5V)	PWM	3
Input: Digital Switch	On/Off	9
Output: FET Drive	On/Off	2
Output: FET Current Drive	PWM	6
Output: Analog Voltage (0-5V)	DAC	1

The version of arduino hardware he recommends us use to implement this is the Panucatt Azteeg X3 Pro which is a 3D printer controller. It has a lot of the material we want to use for I/O already built in. Notably, the FETs for driving coils etc. are already included and screw terminals for easy wiring.

Analysis of Existing Code

The division of functions in this document was determined by an analysis of the original code. A UML activity diagram was generated to overview the main functions in the original code, and these functions were split and run from within the main loop.



Understanding the timing sequence

SampleRate is used to only sample once at the beginning of a PWM output cycle

SampleRateHigh is used to slow the calling of ComputePWMOutput() using a hardware timer

PID_Controller()

lines: 653-673

A PID Controller (proportional-integral-derivative controller), for those who don't know is something that measures a PV (process variable) which is an output value you want to change, compares it to a SP (setpoint) which is the value you want PV to be, and makes the adequate adjustments to try to correct for future PV values.

A suitable example from the internet is that of the automated thermostat. Say that it is too hot your 31° C room and you want to cool down to 22° C. Your setpoint is then 22° C and your process variable is 31° C. The PID controller in the thermostat will then continually calculate the difference between the current temperature (PV) and the target temperature (SP) and will turn on the air conditioning accordingly until PV = SP.

The way the PID controller decides to how to adjust certain settings to obtain acceptable PV values comes in three modes, P - proportional, I - integral, and D - derivative. Some machines don't even use all three modes leaving them to be P controllers, or PI controllers etc.

For P controllers in particular, the adjustments that the controller will make based on the incoming PV will depend on a value proportional to the difference between the SP and the PV, which is called the error. The error is multiplied by some constant Kp and the adjustments are made based on that value. To be clear, Kp is a constant you define yourself - note that if the controller makes adjustments that are too big, you'll end up with fluctuations i.e. your thermostat makes the room too cold, realizes it, then makes your room too hot, and adjusts again, only for your room to be too cold again and continues in a loop. Kp will be the value to control the strength of your adjustments.

On a separate plane, integral and derivative controllers work similarly, except instead of being proportional to the error itself, the adjustment value is calculated based on the *rate of change* of the error in the case of the derivative and the *accumulation of past* errors in the case of the integral. I.e. adjustments to your room temperature could be made based on whether or not your room temperature is falling or rising in the case of the derivative. Kd and Ki are constants you multiply with the integral and derivative functions of the error respectively.

In combination, the adjustments your PID controller makes could be based on the weighted sum of values where P depends on the present error, I on the accumulation of past errors, and D is a prediction of future errors.

Our function in particular takes three arguments: PIDInfo * p, float r, and float y; where p is the struct holding the previous SP and PV values, as well as the Kp, Ki, Kd info, and r is the new SP, and y is the new PV. The float value it returns after being called is something I assume represents in some form, the kind of adjustments that should be made.

Function: the return value is set to a float variable ControlVal used in the Lean Control and Steer Control part of main.

There are two more variables: an integer output and a float E. E is assigned to r-y, which stands for the new SP minus the new PV. As is the function of the PID, this is a way to try to assign the PV to the SP, thereby changing the output.

As p is a pointer, the r that p is pointing to is equal to the current value of r. It is the same thing for y.

ComputePWMOutputs()

line: 689
global constants used:

STEER_DITHER = 170
STEER_OFFSET, BRAKE_OFFSET, LEAN_OFFSET = 30

function: sets the values of pSteerValveS1, pSteerValveS2, pLeanValveS1, pLeanValveS2, pBrakeValveS1, and pBrakeValveS2 all of which represent output pins set to high or low to create PWM output similar to that one schematic he emailed us.

This function will only do something when the SampleFlagHighRate bit is set to high by the Timer3_ISR(), the following is how it will function when it is:

Add one to a PWMCounter variable which helps track of how long the pins are set high or low, this float goes from 0 to 256 and back to zero again, incrementing once each time the function is called. Upon hitting zero, the flag: SampleFlag is set to high which will help flip the Dither bit on the next iteration of the main loop. Afterwards, the next line of execution depends on the float SteerValveCmd a value calculated using a PID controller which is set in the Steer Control.

When SteerValveCmd is equal to or greater than zero, a high Dither sets pSteerValveS2 to 0:

if Dither is set to high:

```
if (PWMCounter <= SteerValveCmd + STEER_OFFSET + STEER_DITHER)  pSteerValveS1 = 1;
else pSteerValveS1 = 0;

pSteerValveS2 = 0;
```

Else if the Dither is set to low:

```
if (PWMCounter <= SteerValveCmd + STEER_OFFSET) pSteerValveS1 = 1;
else pSteerValveS1 = 0;

if (PWMCounter <= STEER_OFFSET + STEER_DITHER) pSteerValveS2 = 1;
else pSteerValveS2 = 0;
```

When SteerValveCmd is negative, a low Dither sets pSteerValveS1 to 0:

if Dither is set to high:

```
if(PWMCounter <= STEER_OFFSET + STEER_DITHER) pSteerValveS1 = 1;
else pSteerValveS1 = 0;
```

```
if(PWMCounter <= -SteerValveCmd + STEER_OFFSET) pSteerValveS2 = 1;
else pSteerValveS2 = 0;
```

Else if the Dither is set to low:

```
if(PWMCounter <= -SteerValveCmd + STEER_OFFSET + STEER_DITHER) pSteerValveS2 = 1;
else pSteerValveS2 = 0;

pSteerValveS1 = 0;
```

pBrakeValveS1/S2 and pLeanValveS1/2 are not dependent on dither bit. Their logic is the same and as as follows:

When the LeanValveCmd or BrakeValveCmd is equal to or greater than zero:

```
if(PWMcounter <= [Lean|Brake]ValveCmd + [LEAN|BRAKE]_OFFSET) p[Lean|Brake]ValveS1 = 1;
else p[Lean|Brake]ValveS1 = 0;

p[Lean|Brake]ValveS2 = 0;
```

When the LeanValveCmd or BrakeValveCmd is negative, the logic is the same except S1 and S2 is swapped and you set the ValveCmd to be negative so you can compare it to PWMCounter:

```
if(PWMcounter <= -[Lean|Brake]ValveCmd + [LEAN|BRAKE]_OFFSET) p[Lean|Brake]ValveS2 = 1;
else p[Lean|Brake]ValveS2 = 0;

p[Lean|Brake]ValveS1 = 0;
```

Both LeanValveCmd and BrakeValveCmd are calculated in Lean Control and Brake Control.

SampleVoltage()

line: 219
Takes an unsigned char this specifies which input we are sampling. (Ex. SampleVoltage(0) returns the voltage outputted by the joystick along the y axis) .
SampleVoltage(0): Joystick Y-axis Sensor.
SampleVoltage(1): Joystick X-axis Sensor.
SampleVoltage(6): Brake Pressure Sensor.
SampleVoltage(7): Hydraulic Pressure Sensor.

Init_Device()

line:1379
preconditions: Timer_Init(), PCA_Init(), ADC_Init(), DAC_Init(), Voltage_Reference_Init(), Port_IO_Init(), Oscillator_Init(), Interrupts_Init(), UART_Init()
function: calls initialization functions.

SampleSensors()

line: 862

Global constant used:

> // RPP - Rev per Pulse (60), eta - belt drive ratio = 11.25"/3.25", r - Tire Rad = 0.3 m,
> // Meters in a mile = 1609.344, DT - Time of one PCA cycle SYSCLK/4 = 8.03755e-008
> SPEED2MPH_CNST = 252586.7479 // RPP*eta*2*pi*r*3600/1609.344/DT

Precondition: float variables: SteerDuty, SteerPeriod, LeanDuty, LeanPeriod, long variable: SpeedDT, unsigned char variable: CntOvrFlow_Speed, and bit variables: SteerSensFlag, LeanSensFlag, NewSampleSpeedFlag are all set to adequate values using the PCA_ISR() (program counter array interrupt service routine).

Function: calculates floats SpeedState, SteerAngleState and LeanAngleState using the values described above. SteerAngleState and LeanAngleState are only adjusted if the SteerSensFlag or LeanSensFlag is set to high.

> **if SteerSensFlag is set to high:**
> SteerAngleState = 360.0 * SteerDuty / SteerPeriod
>
> **if LeanSensFlag is set to high:**
> LeanAngleState = 360.0 * LeanDuty / LeanPeriod - 1.80 //to correct for a 1.8 deg offset

SpeedState is then calculated based on the following conditions:

> **if the NewSampleSpeedFlag is set to low and CntOvrFlow_Speed >= 100:**
> SpeedState = 0.0
>
> **if the NewSampleSpeedFlag is high:**
> SpeedState = 0.0 if SpeedDT = 0.0
> **otherwise**
> SpeedState = SPEED2MPH_CNST / (float) SpeedDT
> **and then we set the NewSampleSpeedFlag back down to low**

If neither of these conditions are matched, we do nothing.

SpeedSteeringControlMap()

list of global constants used:

LEAN_MAX_ANGLE = 29.0
LEAN_MIN_ANGLE = 2.0
STEER_MAX_ANGLE = 22.0
STEER_MIN_ANGLE = 15.0
SPEED_SENS_MAX = 30.0
SPEED_SENS_MIN = 4.0

Precondition: SampleSensors() needs to be called to get the correct value for SpeedState function: compute floats LeanAngleLimit, SteerAngleLimit, Joystick_LR_Ref, Joystick_FB_Ref values used to **Adjust for lean mode, Joystick input**

Joystick_LR_Ref and Joystick_FB_ref is the float value that represents the left-right input and the front-back input from the joystick respectively, they're calculated with the following equation:

Joystick_LR_Ref = ((float) tmpV) / 1642.0
where tmpV = 1729 - (int) **SampleVoltage(0)**

Joystick_FB_Ref = ((float) tmpV) / 1623.0
where tmpV = 1672 - (int) **SampleVoltage(0)**

Both Joystick references are then adjusted to comply with the adequate deadzones. I.e. when the joystick isn't receiving any input from us, the user it shouldn't be doing anything. But the state of "not doing anything" shouldn't be represented by just a single value but rather a range of values. The deadzones: LR_DEADZONE = 0.02, AB_DEADZONE = 0.08 are used to setup the range in where the Joystick reference should be in a null state. For left right input, any Joystick_LR_Ref value between -0.02 and 0.02 is set to zero. Accerleration brake input is treated in the same way.

LeanAngleLimit and SteerAngleLimit is calculated using the following equation:

LeanAngleLimit = SPEED_LIMIT_LEAN_SLOPE * SpeedState + LEAN_MIN_ANGLE
SteerAngleLimit = SPEED_LIMIT_STEER_SLOPE * SpeedState + STEER_MAX_ANGLE

where SPEED_LIMIT_LEAN_SLOPE =
(LEAN_MAX_ANGLE - LEAN_MIN_ANGLE) / (SPEED_SENS_MAX - SPEED_SENS_MIN)
where SPEED_LIMIT_STEER_SLOPE =
(STEER_MIN_ANGLE - STEER_MAX_ANGLE) / (SPEED_SENS_MAX - SPEED_SENS_MIN)
and SpeedState is calculated in SampleSensors()

Both values are then clamped to their maxima, i.e. if it is greater than it's max, it is set to the max, if it is less than its minimum, it's set to its minimum.

Adjust For Lean Mode

line: 414

list of global constants used:

```
Aux_Pin = P5^0
LAMBDA_GAIN = 0.05            //Map a lean error of 10 deg to max tolerable level
LAMBDA_SPEED_GAIN = 1/60  //Map our counter steer only point to 20mph
STEER_REF_GAIN = 0.05         //Gain for carving!
```

list of variables used:

ControlMode - Uses inverted value of Aux_Pin to register lean or no lean mode
SteerAngleLimit - Calculated in SpeedSteeringControlMap(), max steer angle allowed at curr speed
LeanAngleLimit - Calculated in SpeedSteeringControlMap(), max lean angle allowed at curr speed
LeanRef - The angle of lean that will be applied, calculated from lean limit and joystick position
SteerRef - The steer angle to be applied, calculated from steer limit & joystick but also countersteer
Joystick_LR_Ref - Left-right output of joystick
LeanError - Difference between ideal and actual lean
SpeedState - Calculated in SampleSensors(), current detected speed of car
LambdaSens - Used to derive lambda; does not factor in lean error or counter steer only point
Lambda - Used for countersteer, converges to 1 (full countersteer) as car gets faster and remains there if car is travelling faster than 20mph. Non-negative

preconditions: None
function: This function calculates the current lean and steer angle that is to be applied to the car, based on joystick input, predefined limits for safety, and countersteer considerations.

This function starts by setting ControlMode to the bitwise negation of Aux_Pin, indicating whether the car is in Lean or No Lean mode. If ControlMode is 0, No Lean mode is specified, which means LeanRef (angle of lean applied) is set to 0.0, and SteerRef (angle of steer applied) is set to some fraction of the maximum steer angle based on the joystick input, with no consideration for countersteer.

If ControlMode is not 0, Lean mode is specified, and LeanRef is set to some fraction of the maximum steer angle based on the joystick input. LeanError is set to the difference between ideal and the actual lean (LeanRef - LeanAngleState). Then comes some countersteer calculation to determine SteerRef. First, LambdaSens is determined using the following formula:

```
LambdaSens = (SpeedState - 2.0)/20.0
```

LambdaSens is then rounded back if it is outside the bounds of 0 and 1. After that, the following formula is used to compute Lambda:

```
Lambda = LeanError*LambdaSens*LAMBDA_GAIN + SpeedState*LAMBDA_SPEED_GAIN
```

Note that this formula effectively uses the absolute value of LeanError -- if it is below 0, then negative LeanError is used in its stead. Afterwards, Lambda is rounded back if it does not fall between the values of 0 and 1. Now SteerRef can be calculated with the following formula:

SteerRef = (-STEER_REF_GAIN*SteerAngleLimit*LeanError)*Lambda
+ (1-Lambda)*(-Joystick_LR_Ref * SteerAngleLimit)

Lastly, SteerRef is rounded to ensure it falls within the SteerAngleLimit.

Steer Control

line: 450

list of variables used:

SteerValveCmd - used in ComputePWMOutputs to control steer valves
PWM_LIMIT - upper limit = (PWM_RESOLUTION - #_OFFSET)
SteerRef - value you want
SteerAngleState - the value you have

preconditions: PID_Controller()

function: This section of code calls PID_Controller and sets the value of SteerValveCmd. Then it updates PWM outputs.

Lean Control

line: 460

list of variables used:

LeanValveCmd - used in ComputePWMOutputs to control lean valves
PWM_LIMIT - upper limit = (PWM_RESOLUTION - #_OFFSET)
LeanRef - value you want
LeanAngleState - the value you have

preconditions: PID_Controller()

function: This section of code calls PID_Controler and sets the value of LeanValveCmd. Then it updates PWM outputs.

Brake Control

line:470

list of variables used:

BRAKE_GAIN - a constant used to specify by how much to amplify the power by.(10)
Code2VoltageCnst - I think this stans for code to voltage constaint. Which was derived from measured values.
PRESS_VSLOPE - Pressure Voltage sensor characteristic slope
PRESS_VOFFSET - Pressure Voltage sensor characteristic y-offset
BrakeRef - Brake pressure you have.
BrakePressState - Brake pressure you want.

preconditions: SampleVoltage(),

function: This function computes a number for ControlVal. Using is value BrakeValveCmd is then set which is then used in PWMOutputs() to set the brakes.

Hydraulic System Control loop

line:480

list of variables used:

Code2VoltageCnst - I think this stans for code to voltage constaint. Which was derived from measured values.

PRESS_VSLOPE - Pressure Voltage sensor characteristic slope

PRESS_VOFFSET - Pressure Voltage sensor characteristic y-offset

Code2VoltageCnst - I think this stans for code to voltage constaint. Which was derived from measured values.

SFRPAGE - Used to specify the current page.

HydPumpEn - Hydraulic pump control pin.

preconditions: SampleVoltage(),
functions: sets SysHydPressState variable which is then use to decide whether the next pump state. HydPumpEn is the pin that controls the hydraulic.

Traction Motor Command Processing

line:505
list of global constants used:

```
#define PARK_STATE_SPEED   0.18   // Speed below which we throw the brakes in park state
```

list of variables used:

SFRPAGE - Used to specify the current page.

KeySwState - Binary motor control is either on or off.

pKeySwitch - This is a pin. Its the throttle key switch to enable motor controller. This is only active when the motor is active (ie when KeySwState == 1 its active).

AccelRef - It's a global variable that has the acceleration you want.

BrakeRef - It's a global variable that has the amount of "brake" you want. It applies limited and controlled braking when coming to a stop from the acceleration stage. Currently it seems BrakeRef=0 is when the brakes are not applied, and BrakeRef>0 apply the brakes. It directly controls the amount of brake pressure.

DriveEn - A global variable that tells whether the drive state is to set to be set (go forward) or to be put in reverse. .

ParkBrakeEn - Determines when the parking brake is engaged seems to be handled physically. If actively engaged the BrakeRef is set to 0 so as to not accidently 'double' brake which helps identify which brakes are actually applied.

JoystickBtn1/2 - pins to the joystick buttons. The current setting seems to only allow XOR operation.

FwdRevState - Dependent on DriveEn, and Joystick_FB_Ref. See related SpeedSteeringControlMap()

Preconditions: DriveEn=0, as we initially we want to start the car in the park state.
Function: It basically decides whether the motor controller is on or off. Additionally if braking force is applied or throttle is not applied, the motor is not needed to generate acceleration and is then set to the off state. Only when DriveEn is engaged and the throttle is moved forward/backward does the car have forward acceleration capabilities, and only then does the motor apply power. Current code adds double checks when not in forward or reverse to apply full parking brake as a safety measure. Both forward and

reverse states have an extra check in them to make sure that parking brake are not slammed on at a improper speed (the car has to be very slow in order to apply parking brake); refer to global constants. Currently, the joystick sensitivity is handled in SpeedSteeringControlMap(), however in this section it is linearly proportional to acceleration. Double check physical operation (brake pedals available?), but for now it seems that when in the acceleration state, pulling back on the throttle applies specific braking pressure calculated by:

Note the addition of the negative sign:
BrakeRef = -Joystick_FB_Ref*MAX_BRAKE_PRESSURE;

Additionally pushing forward on the throttle when in the reverse state also applies braking pressure:

Note the lack of the negative sign as we are already moving backward:
BrakeRef = Joystick_FB_Ref*MAX_BRAKE_PRESSURE;

Post condition: the current code does not allow the car to enter the park state after having been in either drive or reverse. It handles the problem by having a manual parking switch, which when compressed actively applies braking forces to not allow for movement.

Set Throttle

line: 532
list of global constants used:

THROTTLE_FWD_GAIN = 0.040 *//Address sat limits of throttle circuit*
THROTTLE_REV_GAIN = 0.0990
THTOTTLE_OFFSET = 280.0

list of variables used:

FwdRevState - Indicates car 'state'; 0 is park, 1 is direction choose, 2 is forward, and 3 is reverse
ControlVal - Value representing new car state
AccelRef - Acceleration amount as requested by driver

preconditions: None
function: Changes control value based on the state of the car (fwd, reverse, etc.) and the amount of acceleration that the driver requests.

If the driver is in reverse or forward, the control value is set via these formulae:

ControlVal = 600.0* THROTTLE_REV_GAIN*AccelRef + THTOTTLE_OFFSET *//Reverse*
ControlVal = 4095.0* THROTTLE_FWD_GAIN*AccelRef + THTOTTLE_OFFSET *//Forward*

The constants at the start are saturation control values. The value of ControlVal is affected by these constants and is also limited to a maximum of the constants at the beginning of each formula (600.0 and 4095.0, for reverse and forward respectively) and a minimum of 0. The constant for the reverse formula is lower because, generally, you don't want to accelerate as quickly if you're backing out. Note the difference is approximately a factor of 6.8. The throttle

rev gain, throttle forward gain, and throttle offset are global constants that compensate for saturation limits. From what I understand, these have to do with staying within the maximum and minimum operating limits of the actuator. If the car is in neither forward nor reverse mode, or no acceleration is applied, ControlVal defaults to 0.0.

Timer2_ISR ()

line:885
preconditions: none

Timer3_ISR()

line: 940
From code: This routine divides the clock further for sampling whenever Timer2 overflows.
Precondition: Triggered by interrupt 14 (timer 3)
Postcondition: SampleFlagHighRate = 1; TMR3CN = 0x00

UART0_ISR()

line: 958

This interrupt handles the UART serial connection, both sending and receiving data over the serial line. We will not have to implement this as it is handled by the Arduino Serial library for us.

PCA_ISR()

line: 992

This interrupt routine is responsible for decoding PWM input into the system. Both the speed, lean, and steering sensors are detected by this interrupt. It seems that we will be using the analog input on the arduino, which would mean that we would not need to implement this code. Basically this interrupt is responsible for determining which sensor caused the interrupt, whether the we are at a rising or falling edge, and using the counter to calculate the frequency and subsequent values.

SampleEncoders()

line:783
preconditions: none

InitGlobalVariables()

line:495
preconditions: none

SendUartBuf()

line: 904

Development of Arduino Code

- 3 Input sensors will need to accept PWM readings
    - Rear Wheel Speed Sensor (0-5V PWM)
    - Lean Angle Sensor (0-5V PWM)
    - Steer Angle Sensor (0-5V PWM)

Why We Don't Need Any of the Interrupts in the Original Code

Simply because all of the original interrupts were for PWM handling and serial communication and these functions are built into the Arduino standard library.

How to Read/Write I/O

In the original code PWM I/O had to be handled manually. This led to a great complication of this piece of code, and much of our work has been in separating what code handles this versus code that is performing necessary mathematics and useful transformations of data. Two of the interrupt handlers in the original code were for handling PWM signals. Fortunately, the Arduino has built in support for PWM and analog signals.

True analog output can occur on pins DAC0 and DAC1 (Digital to Analog Converter pins) and PWM output in the rest of the pins using analogWrite (see http://arduino.cc/en/Reference/analogWrite). Analog input is done with analogRead()

PWM output in the original code is 8-bit, matching that of the Arduino by default. This means that as long as the old chip ran on 5V we can match it to the Arduino that is currently being used.