

**AUTOMOTIVE SUSPENSION PARAMETER ESTIMATION
USING SMART WIRELESS SENSOR TECHNOLOGY**

A Thesis
presented to
the Faculty of
California Polytechnic State University
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Mechanical Engineering

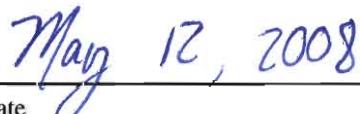
by
Samuel Chase Hoffman
May 2008

AUTHORIZATION FOR REPRODUCTION
OF MASTER'S THESIS

I grant permission for the reproduction of this thesis in its entirety or any of its parts,
without further authorization from me.



Signature



Date

APPROVAL PAGE

TITLE: AUTOMOTIVE SUSPENSION PARAMETER ESTIMATION
USING SMART WIRELESS SENSOR TECHNOLOGY

AUTHOR: Samuel Chase Hoffman

DATE SUBMITTED: May 2008

John Ridgely
Adviser or Committee Chair

John Ridgely
Signature

Joseph Mello
Committee Member

Joseph Mello
Signature

Jim Meagher
Committee Member

Jim Meagher
Signature

ABSTRACT

AUTOMOTIVE SUSPENSION PARAMETER ESTIMATION USING SMART WIRELESS SENSOR TECHNOLOGY

by

Samuel Chase Hoffman

This thesis project demonstrates the feasibility of using a smart sensor system to estimate vehicle parameters. It includes the development of the smart sensor system and the method for which vehicle parameters are estimated using this system. The smart sensor system is a wireless computer controlled sensor array that can be easily installed onto a vehicle. Parameter estimation was accomplished using grey box code in Matlab System Identification Toolbox, a software package from Mathworks. Front and rear suspension damping rates and pitch inertia were estimated on the 2008 Cal Poly SAE Baja Car with good accuracy during testing.

CONTENTS

<i>List of Tables</i>	x
<i>List of Figures</i>	xi
<i>Notation</i>	xiv
1. Introduction	1
1.1 Overview	1
1.2 Scope	2
2. Literature Review	3
2.1 Background	3
2.2 Common Vehicle Models	4
2.3 Active vs Passive Suspension	7
2.4 Parameter Estimation	8
3. Vehicle Model	10
4. Modal Analysis Approach	13
5. Controls Approach	16
5.1 Introduction to Grey Box	16
5.2 State Space Representation of the model	17
5.3 Grey Box Model Simulation	19

5.4	Grey Box Data Requirements	21
5.4.1	Input Parameters Required for method	21
5.4.2	Measured Data Required for method	22
6.	<i>Smart Sensor System</i>	23
6.1	Design Requirements	23
6.2	Components	23
6.2.1	Sensors	27
6.2.2	Microcontrollers	28
6.2.3	Radios	31
6.3	Operation	31
6.3.1	Event Triggering	33
6.3.2	Synchronization	34
6.3.3	Data Acquisition	36
6.3.4	Wireless Transmission	40
6.3.5	“Smarts”	41
7.	<i>Testing</i>	42
7.1	Test Vehicle	42
7.2	Test #1	45
7.3	Test #2	45
7.4	Test #3	46
7.5	Test #4	47
8.	<i>Results</i>	48
8.1	Data Processing	48
8.2	Grey Box Estimation	51
8.3	Sources of Error	53
8.3.1	Errors due to Presence of Non-Linearities	53

8.3.2	Errors due to insufficient model degrees of freedom	54
8.3.3	Errors due to Data Acquisition	57
9.	<i>Conclusion</i>	59
	<i>Bibliography</i>	61
	<i>Appendix</i>	62
A.	<i>Hardware</i>	63
B.	<i>Software</i>	68
B.1	ASV Hierarchical Index	70
B.2	ASV Class Hierarchy	70
B.3	ASV Class Index	70
B.4	ASV Class List	70
B.5	ASV File Index	70
B.6	ASV File List	70
B.7	ASV Class Documentation	71
B.8	array10bits Class Reference	71
B.8.1	Detailed Description	72
B.8.2	Constructor & Destructor Documentation	72
B.8.3	Member Function Documentation	72
B.9	avr_9xstream Class Reference	75
B.9.1	Detailed Description	76
B.9.2	Constructor & Destructor Documentation	76
B.9.3	Member Function Documentation	76
B.10	nb_uart Class Reference	82

B.10.1 Detailed Description	82
B.10.2 Constructor & Destructor Documentation	82
B.10.3 Member Function Documentation	82
B.11 packit_uart Class Reference	84
B.11.1 Detailed Description	84
B.11.2 Constructor & Destructor Documentation	84
B.11.3 Member Function Documentation	84
B.12 spi_bb_port Class Reference	85
B.12.1 Detailed Description	85
B.12.2 Constructor & Destructor Documentation	85
B.12.3 Member Function Documentation	86
B.13 uart Class Reference	88
B.13.1 Detailed Description	89
B.13.2 Constructor & Destructor Documentation	89
B.13.3 Member Function Documentation	89
B.14 ASV File Documentation	95
B.15 Desktop/ASV/Code/ASV.cpp File Reference	95
B.15.1 Detailed Description	96
B.15.2 Define Documentation	96
B.15.3 Function Documentation	97
B.15.4 Variable Documentation	97
B.16 Desktop/ASV/Code/avr_9xstream.h File Reference	98
B.16.1 Detailed Description	98
B.17 Desktop/ASV/Code/avr_a2d.h File Reference	99
B.17.1 Detailed Description	99
B.17.2 Function Documentation	100
B.18 Desktop/ASV/Code/avr_serial.h File Reference	102

B.18.1 Detailed Description	102
B.19 Desktop/ASV/Code/avr_spi_bb.h File Reference	103
B.19.1 Detailed Description	103
B.20 Desktop/ASV/Code/interval.cpp File Reference	104
B.20.1 Detailed Description	104
B.20.2 Function Documentation	105
B.21 Desktop/ASV/Code/interval.h File Reference	106
B.21.1 Detailed Description	106
B.21.2 Function Documentation	106
B.22 Desktop/ASV/Code/nRF24L01.h File Reference	108
B.22.1 Detailed Description	108
B.22.2 Define Documentation	108
B.23 Desktop/ASV/Code/packed_arrays.cpp File Reference	109
B.23.1 Detailed Description	109
B.24 Desktop/ASV/Code/packed_arrays.h File Reference	110
B.24.1 Detailed Description	110
B.25 Desktop/ASV/Code/packit.h File Reference	111
B.25.1 Detailed Description	111
B.25.2 Desktop/ASV/Code/synch_data.h File Reference	112
C. Matlab Code	113
C.1 FFT approach	114
C.2 Grey Box Theory Matlab Code	119

LIST OF TABLES

4.1	Car parameters used used to test FFT	14
4.2	Modal Analysis of using vehicle parameters presented in 4.1	15
5.1	Car parameters inputted into model then compared to grey box predicted parameters of same model	20
6.1	Status LED States	34
7.1	2008 Cal Poly SAE Baja Competition Car	43
8.1	Grey Box Estimated Parameters	53
A.1	Main hardware components of smart sensor systems 1 and 2	63

LIST OF FIGURES

2.1 Quarter car model with 2 degrees of freedom	5
2.2 7 degrees of freedom car model; After Holen [1]	6
2.3 Main vehicle modes using a 7 degree of freedom model; After Holen [1]	7
3.1 2 Degree of Freedom(θ, x_{car}) Half Car Vehicle Model Used	11
3.2 Free body diagram of vehicle model	11
3.3 Mass acceleration diagram of vehicle model	12
4.1 Magnitude of frequency content of car with 10% damping of critical .	15
5.1 Representation of the error function in PEM	17
5.2 Input and output data of simulation, Note that the left graphs refer to the input variables and the right graphs to the state variables AKA output variables	21
6.1 Smart System 1 hardware design	25
6.2 Smart System 2 hardware design	26
6.3 The accelerometer chip mounted on a board for easy wiring	29
6.4 Size comparison of 40 pin DIP to 44 pin surface mount package. Note: These chips are ATMEGA32 which are the same externally as the AT- MEGA644V.	30
6.5 Radios used in project	32

6.6	Test Graph of Data Acquisition System at 800 hz. All channels are connected to the same input.	39
7.1	The Vehicle used for testing the system: 2008 Cal Poly SAE Baja Competition vehicle	42
7.2	Engine noise data for the 2008 Baja Car, Data is taken at 200 Hz with car not moving and engine revving randomly	46
8.1	400 Hz Data Acceleration Data scaled with linear trend removed, taken at 400 hz with 800 points per channel	49
8.2	400 Hz Data State Space Input Output Actual Data;Output data compared to theoretical car model data generated from actual input data shown to the left	50
8.3	800 Hz Data Acceleration Data scaled with constant removed, taken at 800 hz with 800 points per channel	51
8.4	800 Hz Data State Space Input Output Actual Data;Output data compared to theoretical car model data generated from actual input data shown to the left	52
8.5	The estimated suspension travel from 400 Hz Data and 800 Hz Data data sets	55
A.1	Slave 1 MCU wiring diagram	64
A.2	Slave 2 MCU wiring diagram	65
A.3	Master MCU wiring diagram	66
A.4	Maxstream Radio interface with Laptop	67
C.1	2 Degree of freedom simulink model: "ridepitch.mdl" used with FFT2DOF.m	116

C.2	'ridepitch.mdl' Simulink file for integrating raw accelerometer data into displacement and velocity. Creates the input output data specified in section 5.4.2	125
C.3	'ridepitch_ss3' Simulink file for using actual input data to create a model output data. Uses 'state_space.m' for model structure	126

NOTATION

Units

lb	pounds force
in	inches
s	seconds

Terms

m	sprung mass of vehicle
CG	center of gravity of the vehicle's sprung mass
x_{car}	heave or bounce of vehicle CG
θ	pitch angle of vehicle
k_f	spring rate/stiffness of front suspension
k_r	spring rate/stiffness of rear suspension
c_f	damping rate of front suspension
c_{z_f}	damping ratio of front suspension
c_r	damping rate of rear suspension
c_{z_r}	damping ratio of rear suspension
l	wheel base of vehicle
wdf	weight ratio on the front axle
l_f	fore-aft distance from the CG to the front axle

l_r fore-aft distance from the CG to the rear axle

1. INTRODUCTION

1.1 Overview

This thesis project investigates estimation of vehicle parameters using a smart sensor system mounted on a vehicle. The idea of the system is to estimate vehicle parameters in a way transparent to the driver while the vehicle is operated normally. The parameters this system would estimate are ones that can only be dynamically measured such as inertia and damping. This parameter estimation could then be used to calibrate an on-board control system or simply alert the vehicle operator.

To be smart, the sensors are controlled by multiple microcontrollers. This allows them to do smart things such as self calibration, automatic triggering, and pre and post data processing. The smart sensor system is designed to be completely wireless for ease of integration into existing vehicles.

Parameter estimation is very similar to measuring properties of mechanical systems, except that parameter estimation generally applies to systems rather than individual components. Parameter estimation is used to define a model that will represent the system under a chosen set of conditions. The parameter estimation part of this thesis project was approached two different ways: the vibrations approach (chapter 4) and the controls approach (chapter 5).

The second part of this thesis project is the smart sensor system which falls under the category of mechatronics. Mechatronics is the combination of mechanics, electronics, software, and controls to accomplish a goal. In this project, the mechanical system is the vehicle. Then software and electronic hardware are used to provide the

information necessary for parameter estimation. Although it is hinted that this system could eventually be used to control the vehicle mechanical system, it is not developed in this thesis project.

1.2 Scope

The scope of this project is to develop a system to estimate vehicle suspension parameters with reasonable accuracy using an array of microcontroller based sensors or smart sensors. This system is designed to work on a passively suspended vehicle. The smart sensors will measure vehicle variables with respect to time and wirelessly transmit them to a laptop PC for final parameter estimation. The parameters that are estimated from this system are the pitch inertia, (i), and front and rear damping rates, (c_f and c_r). To estimate these parameters, the mass, weight distribution front and rear, and the front and rear spring rates of the vehicle must be known and inputted into the model.

2. LITERATURE REVIEW

2.1 Background

A challenge in designing passenger vehicles is huge variability of certain key parameters that could happen during the vehicles service life. Parameters such as vehicle mass and inertias can vary greatly with the variety of loads of both people and cargo. Shock absorbers wear out, leak, or get hot, resulting in loss of damping and possibly vehicle control. Because of the variability of many different vehicle parameters, big compromises for vehicle ride and handling are made to make sure the vehicle remains safe for the average consumer. Sometimes, no matter how the car is designed, improper loading or vehicle maintenance can lead to an unstable or poor riding vehicle. With the advent of sensor technologies that alert vehicle operators to unsafe operating conditions such as tire pressure monitoring systems and slipping tires, it is only logical for new sensor technologies to be developed that would alert the driver to improper vehicle loading, worn out shock absorbers etc.

Vehicle dynamic control technology is becoming more common on passenger vehicles every year. It started with anti-lock braking systems that were then developed into traction and stability control systems. Now suspension controllers are becoming more common and advanced enough to offer fully active damping control for vehicles. These control systems are all designed to handle the same variability of vehicle parameters mentioned previously, and therefore are compromised in performance because they have not been fully optimized for a static parameter set, but rather for a variable one. If these parameters were better known, there could be much improvement in the

performance of these control systems and the vehicles they are in.

2.2 *Common Vehicle Models*

Many different vehicle models are used for different applications. The common models used for vehicle ride have degrees of freedom that range from 2 to 7. Vehicle ride models focus mainly on vertical motions of vehicle components plus rotational axes that affect vertical motion such as pitch and roll. It is common to assume chassis stiffness is much greater than suspension, therefore resulting in the chassis being modeled as a rigid body. Most models also are linear for computation ease. Sometimes the models are not of entire cars, but of parts of one, like the quarter car and half car bicycle models. Since models are only estimations of actual vehicles the correct model must be chosen to accomplish the objective or task at hand.

The simplest common vehicle suspension model is the quarter car model shown in figure 2.1. It has 2 degrees of freedom; the sprung body, and intermediate suspension body. This model allows for two modes, the ride mode and wheel hop mode. Since it is only a 2 degrees of freedom linear model, all parameters of the real suspension must be lumped together and linearized to make equivalent parameters. The lumping together of parameters can introduce some inaccuracies that are explained in Kim and Ro [5]. To remedy the inaccuracies, the article introduces the process of starting with a complete model of the suspension and reducing the model order to a 2 degree of freedom model. According to Kim and Ro [5], the reduced order model is much more accurate than the lumped parameter model. Using the reduced order model, equivalent parameters can be extracted for comparison of lumped parameters.

Another common model is the 2 and 4 degree of freedom bicycle model. This model is also known as the half car model because it models the car in 2 dimensions by lumping left and right sides together. For suspension modeling it is the simplest model for seeing the relationship between the front and rear of the vehicle to find pitch

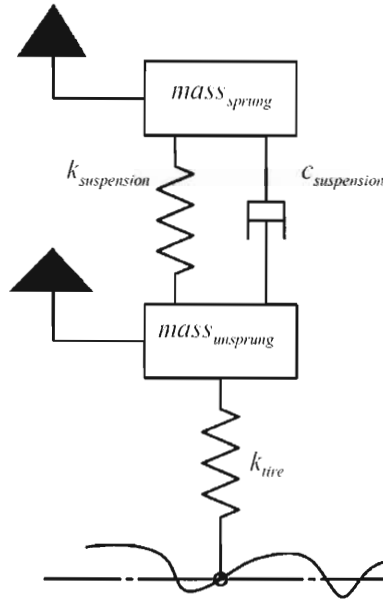


Fig. 2.1: Quarter car model with 2 degrees of freedom

reactions. In Ianncce [2] a 2 degree of freedom half car model is used to examine ride qualities of the vehicle. With 2 degrees of freedom there are two vehicle modes, usually considered the bounce mode and pitch mode, although both modes could have a combination of pitch and bounce. Using this model Ianncce [2] was able to identify important aspects of vehicle ride such as the center of percussion ratio which quantifies how decoupled the front and rear suspensions are. Also this model can illustrate the concept of level ride which reduces pitch of the vehicle when hitting a bump. With only 2 degrees of freedom, the model is limited because it does not include the tire stiffness and unsprung mass, which is why the 4 degree of freedom half car model would be used instead. The 4 degree of freedom half car model is a combination of 2 quarter car models.

Kasprzak and Floyd [3] uses the 4 degree of freedom half car model to tune race car suspension through damper selection. This model is used to predict performance of

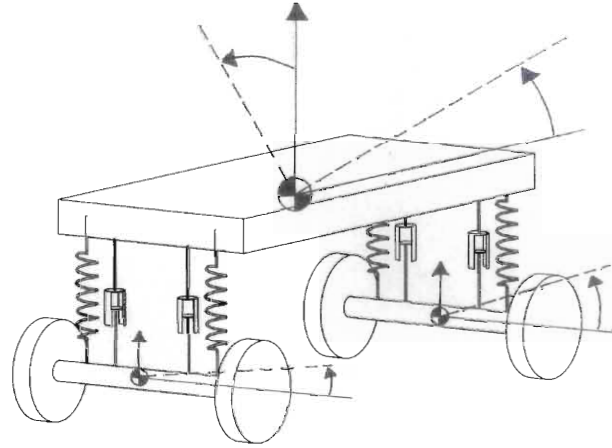


Fig. 2.2: 7 degrees of freedom car model; After Holen [1]

the vehicle before it is driven, or put on test stands. It is shown that through simulation, the vehicle performance can be estimated accurately and efficiently using a 4 degree of freedom model. This method greatly reduces the cost and time of tuning the vehicle by testing.

For ride quality, the 4 degree of freedom half car model does well modeling the vehicle for symmetrical terrain and straight line performance. Once the terrain becomes unsymmetrical, which is common, more degrees of freedom are needed to fully model the car. Holen [1] uses a 7 degree of freedoms model (Figure 2.2) to investigate ride quality in heavy trucks. The 7 degrees of freedom are bounce, pitch, and roll of the sprung mass plus the bounce and roll of the axle unsprung masses. For a ride model, this is the maximum degrees of freedom needed, assuming rigid chassis and suspension components. This model allows all 4 major chassis modes of the vehicle to be realized according to Holen [1]; Bounce, Pitch, Roll, Warp. The first three modes have mode shapes that are self explanatory, but the warp mode is where the front and rear unsprung masses are 180 degree out of phase with each other in roll. The modes can be seen in figure 2.3.

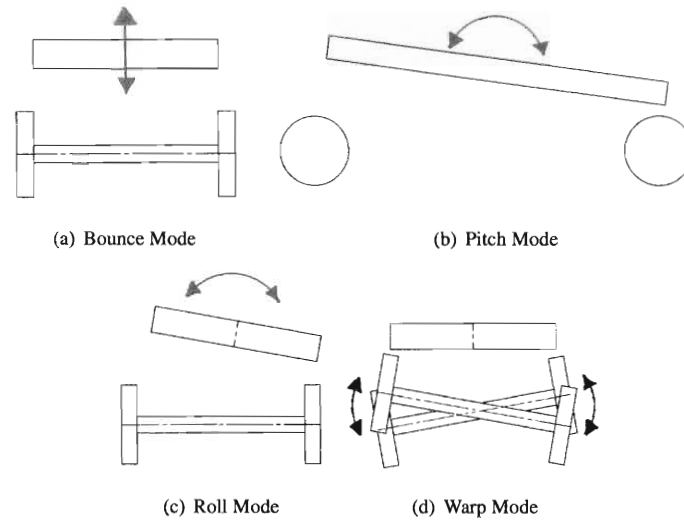


Fig. 2.3: Main vehicle modes using a 7 degree of freedom model; After Holen [1]

2.3 Active vs Passive Suspension

The difference between an active and passive system is that the active system can react to inputs where a passive system cannot. For suspension systems, the definition between passive and active systems still holds, although there can be many degrees between passive and fully active suspensions. The classic automotive passive suspension uses a spring and damper. The spring depends only on position, and the damper only on velocity. These suspension systems have to be tuned for a variety of responses and therefore are a compromise for many different performance goals. If the suspension system could react to input, it could be better optimized for that input. When a suspension system reacts, it starts becoming active.

For a suspension to be fully active, the position, velocity and acceleration of the suspension system must be controlled. This system could theoretically make the vehicle ride motions fully controllable no matter what the input is. Like most control systems, active suspension systems are subject to travel and power limits which can

prevent full control of suspensions to be realized. Another problem with fully active suspensions is that a complete failure can cause unsafe operating conditions for the entire vehicle. This system is also very complex and relies on many sensors. It is because of these obstacles that semi-active suspensions are much more common.

The most common semi-active suspension system today is a system where the damper or shock absorber damping coefficient is controlled. This system is semi-active because dampers can only exert force in direction opposite the direction of their velocity. According to Holen [1] there are primarily two ways the damping coefficient can be changed in a hydraulic damper. The first way is to change the valving in the shock by varying the size of the orifice that the hydraulic fluid flows through. The second is through a variable viscosity hydraulic fluid known as magnetorheological fluid or MR fluid. The MR fluid's viscosity can be varied by passing a magnetic field through it. The viscosity determines how the fluid flows through an orifice. In a damper the magnetic field is applied to the fluid at damper's piston orifice to vary damping characteristics.

There are also many other types of active and passive suspensions that are covered in detail in Holen [1]. A type of particular interest are connected suspensions systems. These systems connect the 4 wheels of the vehicle with hydraulics or springs and allow different vehicle modes to be more individually tuned. With the use many valves and levers these systems have performance characteristics of active suspensions.

2.4 Parameter Estimation

According to Keun [4], real-time parameter estimation is a mature study. But to apply parameter estimation to a vehicle requires a model that includes parameters that are being estimated and is representative of the actual vehicle. Keun [4] uses parameter estimation theory to measure tire cornering stiffnesses and the understeer gradient. After developing a model and equations Keun [4] uses VehSim, a vehicle dynamics software package, to simulate a car with sensor inputs. Using the simulation he was

successful at estimating the parameters.

Holen [1] used Matlab's System Identification Toolbox to estimate modal coordinates using a multiple input single output or MISO model. The modal coordinates describe the amplitude of their particular vehicle mode shape. This was done with previously collected heavy truck data using displacement sensors on the dampers, and a gyroscopic pitch sensor. Holen [1] also talks about difficulty measuring modal coordinates geometrically using only displacement sensors. The problem with displacement sensors is that they have no reference to ground, which why accelerometers need to be used in addition to the displacement sensors.

Using things learned from authors of these papers allowed a more efficient approach to this thesis project. The model used in chapter 3 was used by Iannce [2] for vehicle ride. Holen [1] provided good overview of modeling vehicles for ride. Keun [4] attempted a very similar project that estimated parameters relating to vehicle handling and understeer, instead of vehicle ride. Kim and Ro [5] illustrated errors that can occur with lumped parameters in modeling. Kasprzak and Floyd [3] shows how vehicle simulation with a 4 degree of freedom half car model can predict the vehicle response. The literature review provided a good starting point for the direction of this thesis project.

3. VEHICLE MODEL

To determine vehicle parameters, a vehicle model needed to be developed first. The model's purpose is to show the relationship between the entire vehicle and the parameters being sought, which were specified in the scope(1.2). These parameters are front and rear damping rates (c_f and c_r), and the pitch inertia (i). This necessitated the use of a half car model which includes front and rear suspensions. Only linear models were considered for simplicity. A typical half car model has 4 degrees of freedom: bounce and pitch of chassis/body and bounce of unsprung mass¹ front and rear. Because the parameters being sought do not include tire stiffness and damping, front and rear unsprung masses were omitted in favor of a simpler 2 degree of freedom model shown in Fig 3.1.

This half car model uses the wheel position as input to the system rather than the road or terrain surface. The wheel position input are denoted as u_f and u_r in figure 3.1. By using the wheel displacement instead of road surface input, the unsprung suspension/wheel mass is eliminated from the system. Therefore, to properly use this model, the input to the system has to be wheel displacement and not the road surface. The parameters being estimated are already included in the 2 degree of freedom model so going to 4 degree of freedom half car model would needlessly complicate the project.

A number of assumptions have to be made to use this model. The first is that the chassis is reasonably stiff compared to the suspension, that way it can be modeled as a rigid body. Second, the suspension spring and damping rates are reasonably linear

¹ Unsprung mass is mass that is on the wheel side of the primary vehicle suspension but still suspended from the road through the tire stiffness and damping

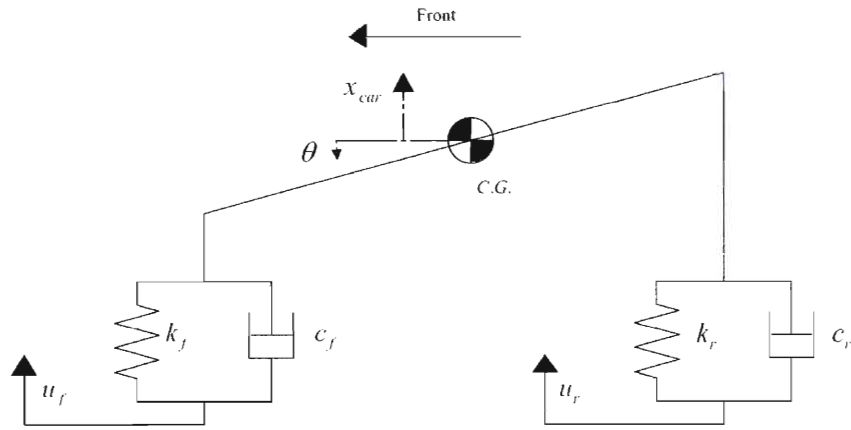
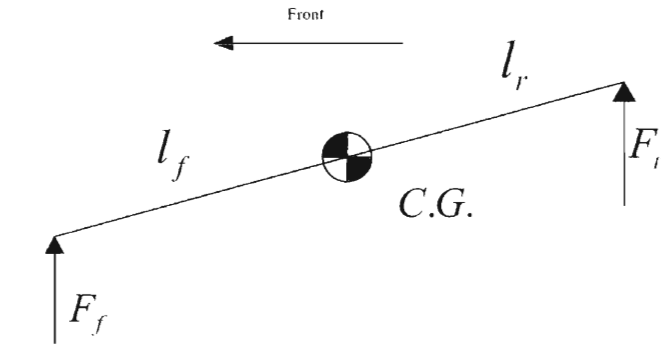


Fig. 3.1: 2 Degree of Freedom(θ, x_{car}) Half Car Vehicle Model Used

throughout the travel of the suspension. The third is that the pitch angle θ is small and therefore $\sin \theta = \theta$ and $\cos \theta = 1$. Finally, the left and right sides of the vehicle are symmetrical and can be lumped together. If these assumptions are reasonable for a vehicle, then this model should be able to accurately represent the vehicle.



$$F_f = -k_f(x_{car} - u_f - l_f \cdot \theta) - c_f(\dot{x}_{car} - \dot{u}_f - l_f \cdot \dot{\theta})$$

$$F_r = -k_r(x_{car} - u_r + l_f \cdot \theta) - c_r(\dot{x}_{car} - \dot{u}_r + l_f \cdot \dot{\theta})$$

Fig. 3.2: Free body diagram of vehicle model

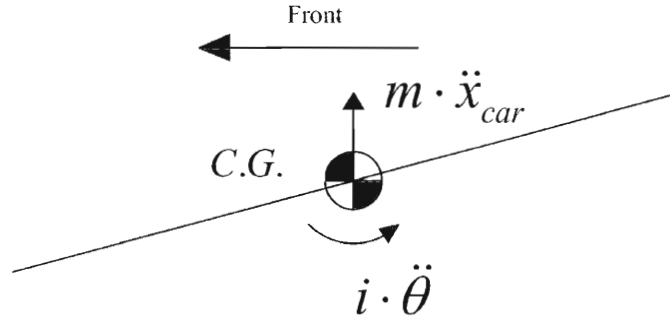


Fig. 3.3: Mass acceleration diagram of vehicle model

Using the free body diagram shown in figure 3.2, and mass acceleration diagram in figure 3.3 the equations of motion can be formulated in equation 3.1.

$$\begin{aligned}
 i \cdot \ddot{\theta} &= F_r \cdot l_r - F_f \cdot l_f \\
 m \cdot \ddot{x}_{car} &= F_f + F_r
 \end{aligned}
 \tag{3.1}$$

More complicated models can more accurately represent a vehicle for this project but would add a great amount of complexity. Therefore, the simplest car model was used to achieve the goals set forth in the scope, found in section 1.2. A 2 degree of freedom model proved to be sufficient throughout the project and did not need to be changed.

4. MODAL ANALYSIS APPROACH

The original approach attempted in this thesis used modal analysis to calculate vehicle parameters. It failed due to reasons explained below, so this section does not go into depth on mathematical details of this approach.

The chosen 2 degree of freedom model has 2 modal frequencies(eigenvalues) and 2 corresponding mode shapes(eigenvectors). If these modal frequencies and shapes could be found using experimental analysis, they could be used to estimate parameters of the vehicle.

The Fast Fourier Transform (FFT) is used to transform time domain vibrational data into the frequency domain. If an FFT is performed on vibrational data of an object, then the modal frequencies should show up as an increased amplitude at their respective frequencies. The object would have to be forced to vibrate equally at all frequencies so the displacements at the mode frequencies would be accurate. The relative magnitudes of the modal frequencies and the phase angles (also found with an FFT) can be used to estimate the mode shapes.

The problem with this approach is that if the vehicle has a large damping ratio, the FFT's effectiveness at identifying modes is significantly reduced. The damping ratio is the ratio of the system's damping coefficient to system's critical damping coefficient. When the system's damping coefficient is equal to or greater than its critical damping coefficient, (which is when the damping ratio is equal to ≥ 1), the system can be said to be critically or over damped. When the system is critically or over damped, it will not oscillate. cz_f and cz_r notate damping ratios for the front and rear respectively. Damp-

ing on vehicles can range close to and above critical damping. If a vehicle has a large damping ratio then the number of oscillations are reduced. The number of oscillations are critical for the FFT to pick out the frequencies of oscillations. A critically damped vehicle mode will not oscillate at all and therefore not show up on a FFT.

Tab. 4.1: Car parameters used used to test FFT

Parameter		Input Value
mass	m	2 lb-s ² /in
radius of gyration	k_g	20 in
weight distribution front	wdf	.4
wheel base	l	60 in
spring rate front	k_f	60 lb/in
spring rate rear	k_r	60 lb/in
damping ratio front	cz_f	.1
damping ratio rear	cz_r	.1

A simulation was done in Matlab and Simulink to test whether the FFT approach could be used to identify the vehicle modes despite their large damping ratios. The vehicle model from chapter 3 with parameters listed in table 4.1 was simulated going over terrain that caused white noise input¹ to the vehicle chassis due by movement of the wheels. The simulated vehicle is very lightly damped at 10% of critical to test if the FFT can easily identify modes. Shown in figure 4.1, there are not two easily identifiable modes, even on a vehicle with low damping. The first mode can possibly be estimated close to 1.1 Hz, but the second mode can be any of the three peaks shown from 1.3 to 1.7 Hz. This is most likely due to too few oscillations at the modal frequencies. Table 4.2 shows the actual modes obtained by solving the eigenvalue problem of the car. The values for damped natural frequency listed in table 4.2 correlate to peaks in figure 4.1 that are not well defined. Once the damping ratio is increased from only 10%(.1) to 30%(.3) for front and rear the mode frequencies are even harder to distinguish. If the damping ratio is increased to 70%(.7) for front and rear the second mode shape is

¹ White noise contains all frequencies equally

critically damped, which means it will never oscillate or show up in an FFT.

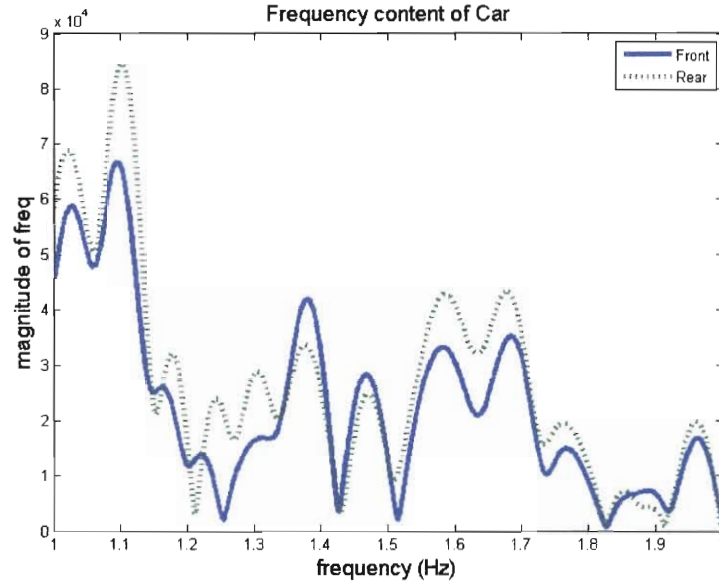


Fig. 4.1: Magnitude of frequency content of car with 10% damping of critical

Tab. 4.2: Modal Analysis of using vehicle parameters presented in 4.1

	1st Mode	2nd Mode	Units
Damped Natural Frequency	1.11	1.57	Hz
Damping Ratio	.099	.149	
Natural Frequency	1.23	1.85	Hz

In conclusion, the FFT is not effective on vehicles with damping ratios greater than 10%(.1) of critical damping. Therefore, a new approach is needed to estimate parameters. Control theory is the next logical option for this thesis project because it does not rely on vehicle oscillations or the FFT. An approach involving controls theory is examined in chapter 5.

5. CONTROLS APPROACH

5.1 *Introduction to Grey Box*

The next approach to parameter estimation involved grey box control theory. The grey box represents a transfer function with known internal structure that can be estimated with input and output data. Grey box is similar to black box except with a black box, the structure is not known. This advantage of grey box will allow a more accurate estimation of a model than black box simply because the structure is known. By using the chosen structure of the model described in chapter 3, grey box would allow estimation of the parameters in the model and allow the rest to be manually inputted.

Matlab System Identification Toolbox was planned to be used for grey box calculations from the beginning. It has functions that do all grey box calculations. The user is simply required to learn the notation to input the model and input/output (I/O) data. The Matlab Grey Box code is operated by first creating an 'idgrey' object using the state space form of control system. It is here that parameters are defined as being fixed or in need of estimation. Then I/O data is inputted into a 'iddata' object in the time domain. The objects 'idgrey' and 'iddata' are inputted into the Matlab PEM function for parameter estimation. Initial values are defined for the parameters being estimated and are the starting points for the optimization routine in PEM function. If the initial values are not close initially, the system may find a different set of optimized parameters

The PEM function stands for 'prediction error measuring' and is a function that minimizes the cost function in equation 5.1 by optimizing specified parameters in the

'idgrey' object. The optimum specified parameters that minimize equation 5.1 are the estimated parameters for the grey box system.

$$V_N(G, H) = \sum_{t=1}^N e^2(t) \quad (5.1)$$

The error function is defined by:

$$e(t) = H^{-1}(q)[y(t) - G(q)u(t)] \quad (5.2)$$

These equations are visualized by figure 5.1.

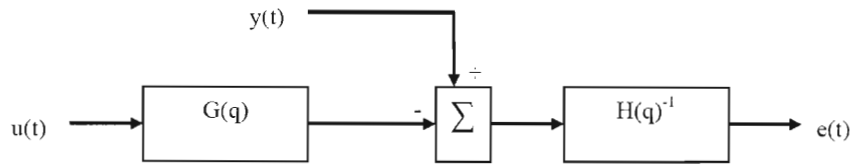


Fig. 5.1: Representation of the error function in PEM

$G(q)$ is the discretized version of $G(s)$, which is the system transfer function defined by the 'idgrey' object. $H(q)$ is a discretized version weighting function for the error and is controlled by Matlab. $u(t)$ and $y(t)$ are the input and output data functions respectively. The parameters being optimized are contained in $G(q)$ and the goal would be that $y(t) = G(q) \cdot u(t)$. The equations 5.1 and 5.2 are for a single input, single output (SISO) system which means that they will be more complex for a multiple input, multiple output (MIMO) system used in this thesis project.

5.2 State Space Representation of the model

Considering the vehicle model is a multiple input, multiple output(MIMO) system, representation in state space makes sense. Matlab System Identification Toolbox handles

MIMO state space described systems — another essential reason for choosing the grey box theory route.

To formulate a state space description of a model, a set of state variables need to be chosen that will completely describe the model and be linearly independent. The model has 2 degrees of freedom so a normal choice of state variable would be the 2 degrees of freedom and their derivatives. Therefore the state variables are shown in equation (5.3).

$$\bar{X} = \begin{bmatrix} \theta \\ \dot{\theta} \\ x_{car} \\ \dot{x}_{car} \end{bmatrix} \quad (5.3)$$

The inputs were decided upon by starting with the displacement inputs of the front and rear wheel suspension $u_{f \ wheel}, u_{r \ wheel}$. Because the damper depends on velocity, the derivatives of the road input $\dot{u}_{f \ wheel}, \dot{u}_{r \ wheel}$ are also needed. The input is therefore defined in equation (5.4).

$$\bar{U} = \begin{bmatrix} u_{f \ wheel} \\ \dot{u}_{f \ wheel} \\ u_{r \ wheel} \\ \dot{u}_{r \ wheel} \end{bmatrix} \quad (5.4)$$

Using the chosen state variables and the previously defined model, the state space description was formulated in equation (5.5). The output is all four state variables for simplicity and because they are easily measurable. All matrices are linearly indepen-

dent, which validates the chosen inputs and state variables.

$$\dot{\bar{X}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{-(k_r l_r^2 + k_f l_f^2)}{i} & \frac{-(c_r l_r^2 + c_f l_f^2)}{i} & \frac{k_r l_r - k_f l_f}{i} & \frac{c_r l_r - c_f l_f}{i} \\ 0 & 0 & 0 & 1 \\ \frac{k_r l_r - k_f l_f}{m} & \frac{c_r l_r - c_f l_f}{m} & \frac{-(k_r + k_f)}{m} & \frac{-(c_r + c_f)}{m} \end{bmatrix} \bar{X} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{k_f l_f}{i} & \frac{c_f l_f}{i} & \frac{-l_r k_r}{i} & \frac{-l_r c_r}{i} \\ 0 & 0 & 0 & 0 \\ \frac{k_f}{m} & \frac{c_f}{m} & \frac{k_r}{m} & \frac{c_r}{m} \end{bmatrix} \bar{U} \quad (5.5)$$

$$\bar{Y} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \bar{X} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \bar{U} \quad (5.6)$$

These state space matrices were defined in 'idgrey' object. All parameters were defined except the parameters specified for estimation: c_f , c_r , and i .

5.3 Grey Box Model Simulation

A simulation in Matlab was performed to evaluate the performance of the grey box Matlab code before actually testing its performance on a real car. The idea behind the simulation is to input terrain into the state space model and see if the grey box can predict the parameters used in the state space model. To do this simulation, vehicle parameters first needed to be defined. The parameters chosen are shown in table 5.1 and were loosely based on 2006 Cal Poly SAE Baja Car Parameters.

After inputting these parameters into the state space matrices the model was run with a simulated bump input shown in figure 5.2. Based on the input and output data, the grey box estimated the parameters shown in the third column of table 5.1. The other variables in the grey box were inputted as part of the grey box structure. This shows that the grey box estimation was accurate even with only 300 points of data, which is important considering limitations on the amount of data that can be collected and used in the mechatronics side of the project in chapter 6.

Tab. 5.1: Car parameters inputted into model then compared to grey box predicted parameters of same model

Parameter		Input Value	Predicted Value	Units
sprung mass	m	1.3587	inputted	lb-s ² /in
weight distribution front	wdf	.47	inputted	N/A
wheel base	l	66	inputted	in
spring rate front	k_f	100	inputted	lb/in
spring rate rear	k_r	100	inputted	lb/in
radius of gyration	k_g	10	9.985	in
damping ratio front	cz_f	.7	.7176	N/A
damping ratio rear	cz_r	.5	.5108	N/A

The parameters were inputted into a model with identical structure to the 'idgrey' object. This structure is listed in equations 5.5 and 5.6. Therefore, this simulation tested how well Matlab grey box code could identify the same parameters used in an identical model. This will not be the case in a real test because the vehicle being tested does not necessarily match the model structure defined in equations 5.5 and 5.6. Therefore, this simulation does not test how well the vehicle model used will actually match a vehicle. The assumptions made that allow an actual vehicle to be accurately represented by this model were presented in chapter 3 on page 10, and further explored in relation to actual testing in section 8.3 on page 53.

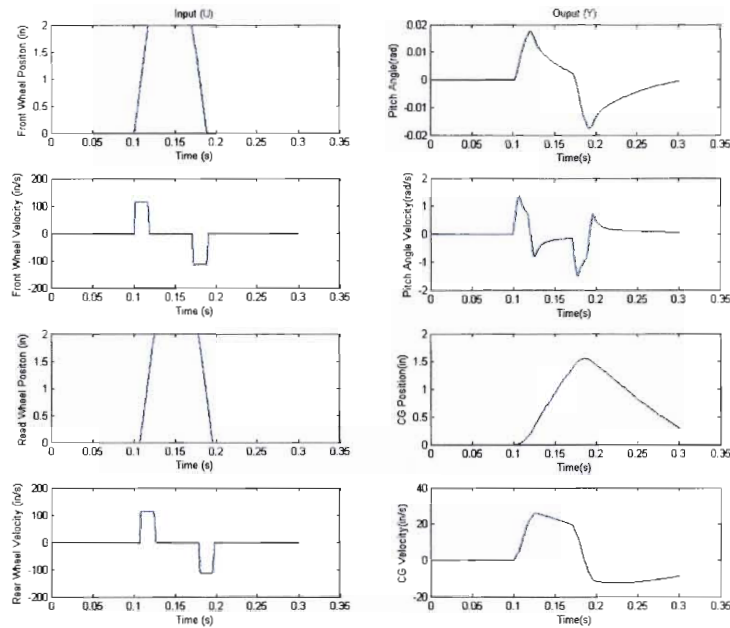


Fig. 5.2: Input and output data of simulation, Note that the left graphs refer to the input variables and the right graphs to the state variables AKA output variables

5.4 Grey Box Data Requirements

Due to the success of the grey box simulation, it was the chosen method to estimate parameters on an actual vehicle with real data. It is therefore important to make a set requirements for the sensor system described in chapter 6.

5.4.1 Input Parameters Required for method

The grey box method requires parameters to be known and inputted into the model prior to parameter estimation, these parameters are:

- m or sprung mass of vehicle
- l or wheel base of vehicle

- wdf or weight distribution front
- k_f or front spring rate
- k_r or rear spring rate

5.4.2 Measured Data Required for method

To estimate the parameters, the grey box method requires sensors to make multiple measurements of vehicle variables with respect to time. The data does not have to be taken at constant time intervals as long as the time is known. The required vehicle variables are:

- State Variables also known as Output Variables
 - x_{car} or heave displacement of car
 - \dot{x}_{car} or heave velocity of car
 - θ or pitch angle of car
 - $\dot{\theta}$ or angular pitch velocity of car
- Input Variables
 - $u_{f \text{ wheel}}$ or front wheel displacement
 - $\dot{u}_{f \text{ wheel}}$ or front wheel velocity
 - $u_{r \text{ wheel}}$ or rear wheel displacement
 - $\dot{u}_{r \text{ wheel}}$ or rear wheel velocity

6. SMART SENSOR SYSTEM

6.1 Design Requirements

The purpose of the smart system is to collect data at a specified time intervals from the vehicle. The design requirements for the smart system are listed below.

- Take measurements at known accurate time intervals of all output and input vehicle variables listed in section 5.4.2
- Ability to take data as fast as 800 hz
- Hold 800 points of data per channel
- Transfer data wirelessly for further analysis on laptop computer
- Be relatively easy to install with minimal vehicle modification
- Have the ability to be 'smart' by having logical controls and processing ability
- Be able to be controlled remotely

6.2 Components

There are three major components that make up this system: sensors, microcontrollers, (MCUs), and radios. They are gone over in detail in the following sections.

The smart system temporarily stores the data on board until it can be sent to a laptop computer over a wireless connection. To make this system easy to install, there were

two MCUs—one for the back of the vehicle and one for the front. The two MCUs would collect data from four total sensors, two per controller.

2 smart sensor systems were developed each with their own strengths and weaknesses. These are labeled System 1 and System 2 and are represented in the figures 6.1 and 6.2. The operation of these systems are explained in detail in section 6.3.

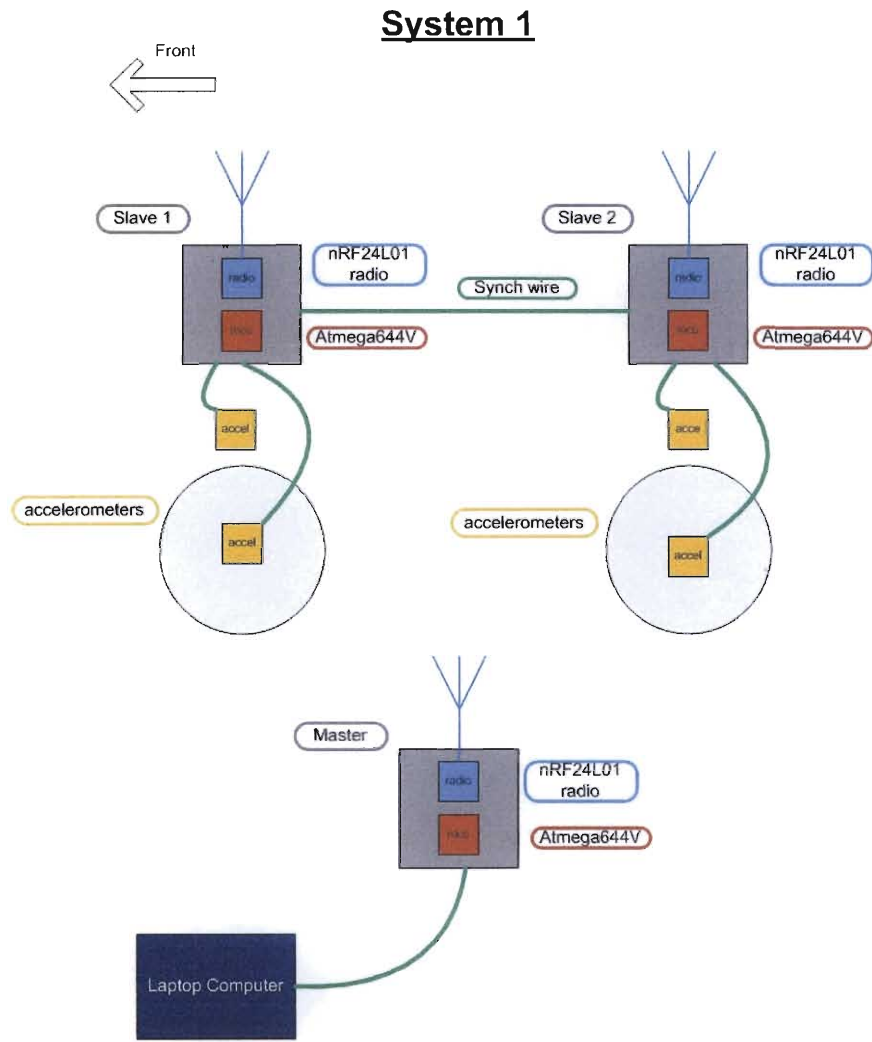


Fig. 6.1: Smart System 1 hardware design

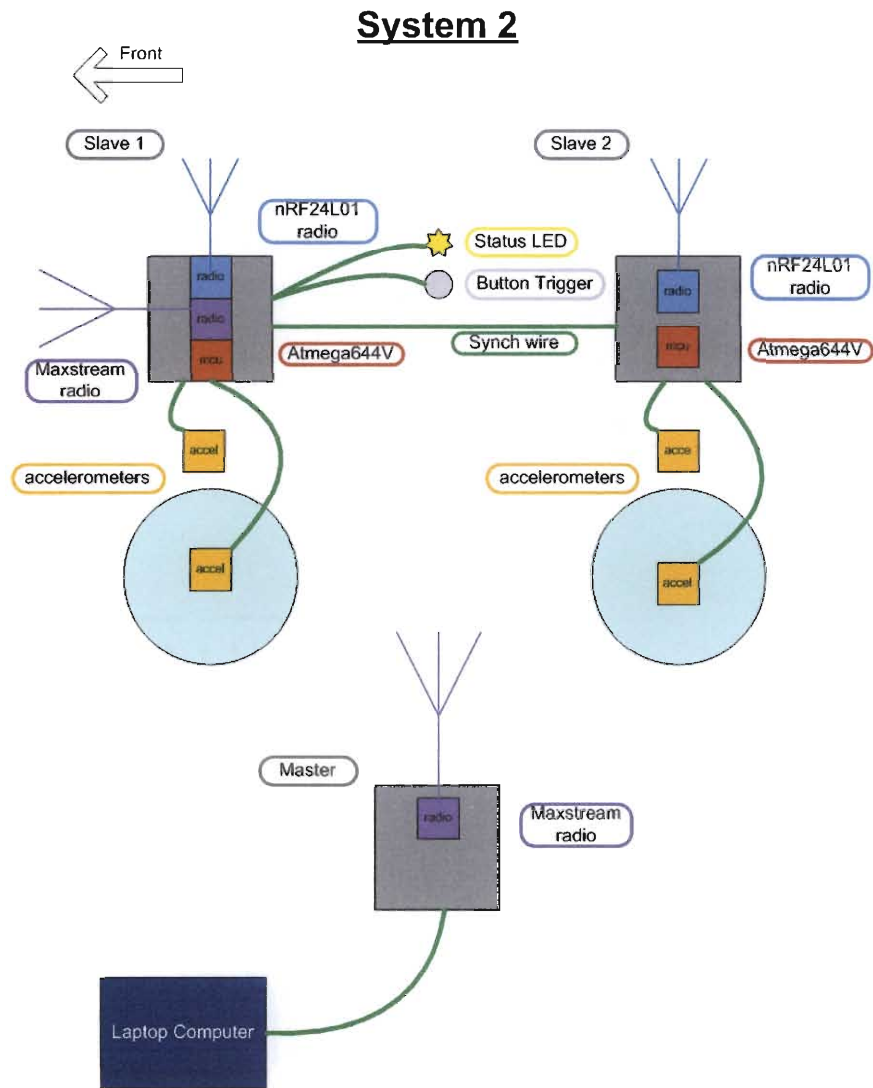


Fig. 6.2: Smart System 2 hardware design

6.2.1 Sensors

In chapter 5 the variables that needed to be collected were decided upon and labeled the input and output variables. The output variables, (also known as the state variables), are the pitch (θ), heave (x_{car}) and the first derivatives ($\dot{\theta}, \dot{x}_{car}$), of the vehicle's *CG*, and are shown in equation 5.3. The output variables are the displacement and velocity of the front and rear axles, ($u_{f\ wheel}, \dot{u}_{f\ wheel}, u_{r\ wheel}, \dot{u}_{r\ wheel}$). To measure these variables, only four sensors are needed because derivatives or integrals of the data can be found in post process. In order to fulfill the requirement that this sensor system is easy to install, the option of a displacement sensitive sensors such as a potentiometer or global positioning signal are ruled out because they require additional components and hardware to install and operate. A common velocity sensor is the vibrometer, also known as a velocity coil, but these were ruled out due to the fact they need to be as long as the displacement that they measure, which would make them bulky for installing on a vehicle that will displace more than a couple inches. This leaves accelerometers, which currently are smaller and cheaper than a comparable displacement or velocity sensor. Accelerometers measure acceleration and are not displacement limited. A downside of accelerometers is that their output must be integrated to get the measurement data in the required form of displacement and velocity. This integration can introduce drift errors into the system which is when the displacement and velocity become offset from actual. Drift is caused by numerical integration of signals, and can be equated to integration of an error. Drift error can be compounded through multiple integrations.

The two accelerometers that would measure the state variables were mounted on the chassis at the points of the axles. Using equation 6.1 derived from geometry and assuming small angles, the pitch angle (θ), can be estimated.

$$\theta = \frac{x_f - x_r}{l} \quad (6.1)$$

Where θ is the pitch angle and x_f and x_r are the front and rear displacements respectively. The same is true for heave (x_{car}), which is shown in equation 6.2.

$$x_{car} = \frac{wdf \cdot x_f + (1 - wdf) \cdot x_r}{l} \quad (6.2)$$

Where wdf is the weight distribution in the front. Both equations 6.1 and 6.2 can be differentiated to find the derivatives of θ and x_{car} required for the state variables.

To measure input variables, the accelerometers were mounted to the front and rear suspensions. Given that the model in chapter 3 lumps the left and right sides of the car together, the side of the car that the accelerometers are mounted on does not matter. This will be accounted for by testing the vehicle on symmetrical terrain.

The accelerometers chosen are shown in figure 6.3. They are a 2 axis accelerometer mounted on a break out board, allowing them to be easily connected to wires. The ADXL322 version, which is a two axis $\pm 2g$ version, was mounted on the chassis while the ADXL320 two axis $\pm 5g$ was mounted on the suspension at the wheel. This was done because higher acceleration amplitudes were expected on the suspension compared to the chassis. These chips are powered on voltage ranging from 2.4-5.25 volts and will output an analog voltage signal, per axis, proportional to acceleration in the range of 0 volts to supply voltage. Only one channel or axis of the accelerometer needed to be used per accelerometer. The accelerometer chips are set up for a 50 hz analog bandwidth according to the data sheet and boards they are mounted on. The highest mode of the vehicle model in section 7.1 on page 42 is not greater than half the accelerometer bandwidth, making the bandwidth sufficient for most cases.

6.2.2 Microcontrollers

The microcontrollers (MCUs) used for this project are ATMEGA644V which are manufactured by Atmel. They operate on a 8 bit RISC, (Reduced Instruction Set Computer), architecture. They were picked for their low cost and large memory. They were

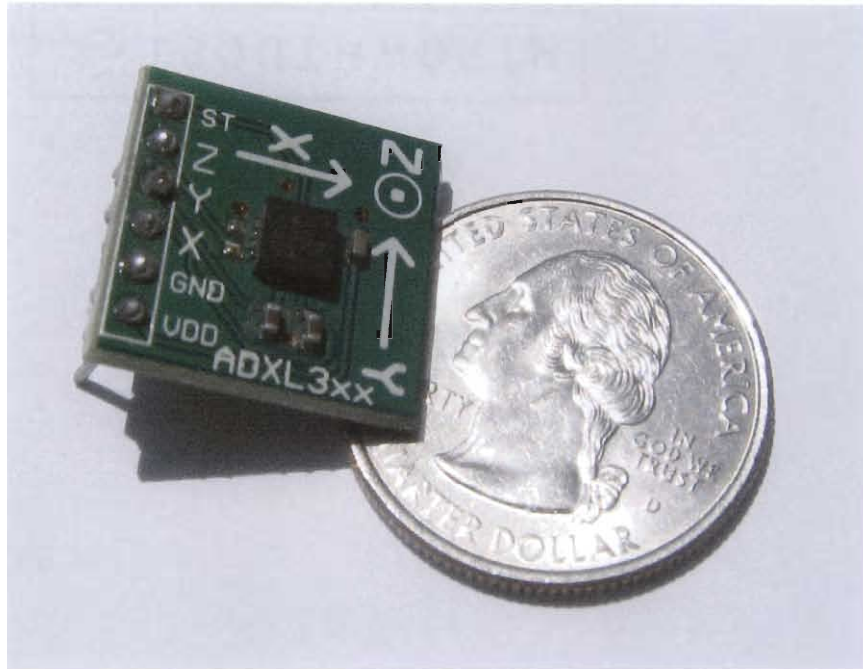


Fig. 6.3: The accelerometer chip mounted on a board for easy wiring

clocked at 4mhz, but could be clocked up to 20mhz¹ if needed. They have 32 digital I/O pins that have special functions such as serial ports, analog to digital (ADC) conversion, input capture etc. They are also available in a 40 pin DIP package which has .1 inch pitch between pins that make it able to fit on a prototyping bread board. Smaller surface mount packages can be used for permanent circuit boards and offer significantly reduced size as shown in figure 6.4.

The ADC function of the chip is important for measuring voltage signals from the chosen accelerometers. This chip has one channel ADC circuitry that can be connected to 8 I/O pins through internal hardware. Once the voltage is read in to the chip, it is

¹ The high power ATMEGA644 can be clocked to 20mhz with same architecture and similar cost

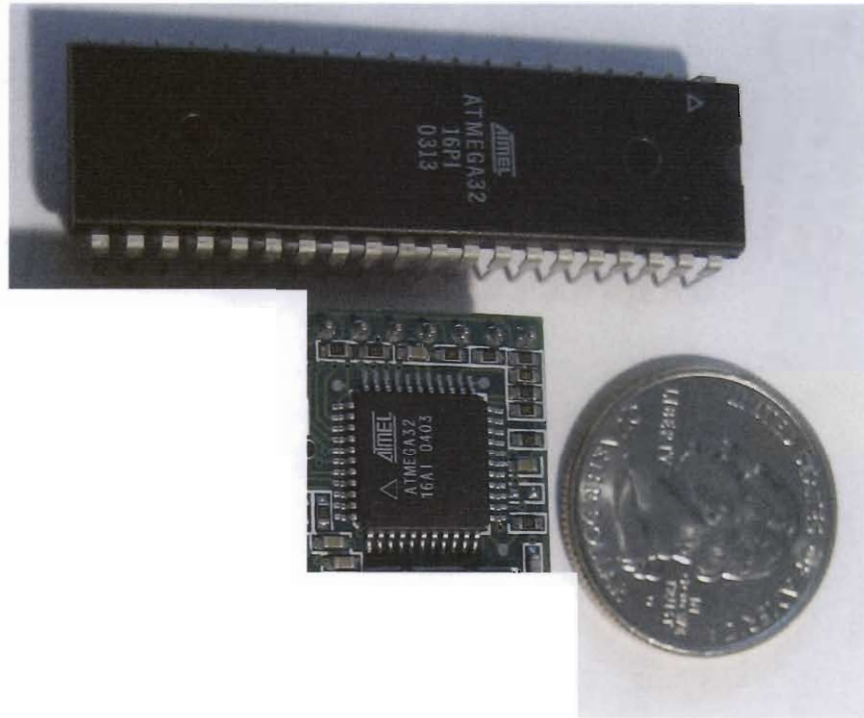


Fig. 6.4: Size comparison of 40 pin DIP to 44 pin surface mount package. Note: These chips are ATMEGA32 which are the same externally as the ATMEGA644V.

converted to a 10 bit analog number based on the voltage reference by using

$$N = \frac{V_{in} \cdot 1024}{V_{ref}}$$

where N is the 10 bit reading number and V_{in} and V_{ref} are the voltage in and voltage reference respectively. The voltage reference is inputted through an external pin or selected from supply voltages. The ADC clock runs off the CPU clock and has options to be prescaled to run slower.

Because there is only one ADC channel, there cannot be truly simultaneous measuring of two voltage sources connected to the MCU. This can cause data calculation errors if not accounted for.

6.2.3 Radios

Radio communication allowed this project to be wireless, which was important for installation and ease of use. The project started out using exclusively the nRF24L01 radios that were mounted on a break out board for easy prototyping connections. These radios operated on the 2.4Ghz frequency and, with high gain antennas, should have a range of 100 meters by independent tests. Their data rates of 2Mbps, (2 Mega bits per second), made them very fast for the relative sizes of data being sent. They also offered hardware error checking, which was used for extra reliability on top of a software based error checking. These radios required a MCU to handle data processing to a known interface such as RS232 serial data.

Due to range issues explained in chapter 7 with the nRF24L01 radio, Maxstream radio modems were eventually used for communication from vehicle to laptop PC. Maxstream radios are stand alone RS232 serial modems and do not require a separate controller; i.e. they can be directly connected to a laptop PC. They operate on a 900mhz band and offer miles of range depending on the antenna used. The reason they were not used from the start was because they are based on a 9600 baud, (9600 bits per second), asynchronous serial interface, (RS232), which made them significantly slower than the nRF24L01 radios. Maxstream can be used as standalone without a controller while nRF24L01 radios cannot. Maxstream radios are also over 7 times more expensive than the nRF24L01 radios.

6.3 Operation

The operation of the MCU sensor system is governed by software design which is designed for a specific component setup. Two different component setups were used—labeled System 1 and System 2 for ease of reference. System 1 was the first system design, using all short range nRF24L01 radios and remote triggering. System 2 was a branch of system 1 to use the long range Maxstream radios and vehicle operator

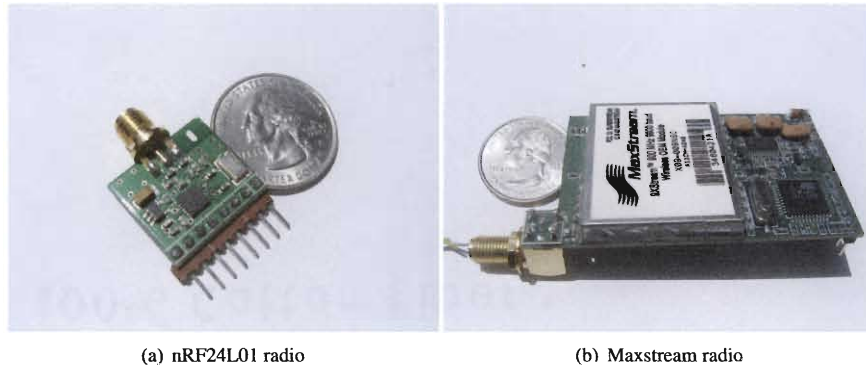


Fig. 6.5: Radios used in project

triggering. System 2 is the one used for all data acquisition of this project. It is also important to note that the main reason for the transition from System 1 to System 2 was poor range of the nRF24L01 radios, which might have been remedied with high gain antennas which were never tested with System 1 due to success of System 2. These systems are illustrated in figures 6.1 and 6.2.

System 1 has three different MCUs: Master, Slave 1, Slave 2. The Master MCU controls Slave 1 and Slave 2 and is connected to the laptop computer. It does not take data, merely creates the interface between the laptop operator and the slave chips mounted on the vehicle. Slave 1 and Slave 2 MCUs are mounted on the vehicle and take data from the accelerometers.

System 2 does not use a Master MCU and instead adds user controls on board the vehicle connected to Slave 1. The link to the laptop is handled by the more powerful and longer range Maxstream radio. A frequency controller is also on board Slave 1 but is not shown in figure 6.2, it is further explained in section 6.3.3.

All systems were assembled on breadboard for rapid prototyping ability.

6.3.1 Event Triggering

The triggering of the system is defined in this report as how the data taking event is triggered for data acquisition. Three different types of triggering were considered for this project: remote, on-board, smart/auto. Smart/auto triggering were not actually used during this project but are further discussed in section 6.3.5.

System 1 used remote triggering for data taking event. An advantage to remote triggering is that the vehicle operator is left out of the picture and someone watching the vehicle could remotely trigger the system over different terrain. Due to range issues, this proved to be very difficult. The laptop with remote controller had to be within 10 feet of the car to work, and the vehicle operator proved too unpredictable for the person operating the remote trigger. Because of range issues the trigger was not very instant which made it more impossible to trigger at the correct time.

It is because of these downfalls that the triggering of the system was changed from remote to on-board triggering in System 2. On-board triggering means that the vehicle operator will trigger the data acquisition on-board the vehicle. The triggering interface for the vehicle operator uses a tactile pushbutton switch with a status led light mounted within reach and sight. To trigger the data acquisition event the vehicle operator would push the button trigger with the status led off or status led state 1 in table 6.1. When data started recording the status led would progress to state 2. If the process did not self terminate due to accelerometer saturation², the system would hang in state 3 until the vehicle operator would hit the trigger button again to trigger a download. Then the unit will cycle from state 4 to state 5 and finally back to state 4.

Ideally, the trigger would be close to instantaneous for starting the data acquisition. In reality, there was fraction of a second delay as preparations were made for synchronizing and data acquisition on Slave 1 and Slave 2 MCUs, which are explained in detail in sections 6.3.2 and 6.3.3. For future versions it would be important to reduce

² see section 6.3.5 for details on saturation checking

the time of this delay when taking data in the 400 hz + range because of the short time the system actually takes data. The total time data is taken for 400 hz is 2 seconds to fill up the designed 800 data point memory. With a delay of .5 second a quarter of the data could be missed if the delay is not estimated correctly by the vehicle operator. The best way to reduce the delay is to have the system continually update and synchronize Slave 1 and Slave 2 MCUs so that when triggered they would be ready to start taking data.

Tab. 6.1: Status LED States

State No.	Status LED State	State of System
1	Off	Standby, ready to take data
2	Constant On	Taking Data
3	Short Long	Waiting for user to trigger data transfer
4	Slow Flash	Transferring data from Slave 1 to laptop over Maxstream Radio
5	Fast Flash	Transferring data from Slave 2 to Slave 1 over nRF24L01 Radios

6.3.2 Synchronization

Synchronization of Slave 1 and Slave 2 is important for getting data from both of them that will correlate together. As mentioned previously in section 6.2.1, a sensor front and back will measure state or output variables, and another set of sensors front and back will measure input variables. Since in both cases the sensors are connected to separate MCUs, they have to be synchronized. The synchronization process is identical for System 1 and System 2.

The synchronization software class is written to be expandable to multiple MCUs though only two are used for this system. The two MCUs used are Slave 1 and Slave 2, which are specified Master Slave and Slave Slave respectively. Master Slave, (Slave 1), is the controller of the synchronization process. The synchronization process is started

by the trigger and will automatically trigger the data acquisition. It is written to utilize the nRF24L01 radios and a Synch-wire³ connected between the MCUs. Though it was the original intention to keep the system completely wireless, the extra wire simplified the synchronization process. More testing would need to be done to synchronize using only the nRF24L01 radios. The synchronization is accomplished multiple steps:

1. The synchronization is triggered and Slave 1 sends Slave 2 a radio command to get ready for a high pulse on the Synch-wire to reset the data acquisition system's timer clock.
2. Slave 2 responds to Slave 1 command to say it is ready for the pulse. Slave 1 will then pulse the Synch-wire, which through hardware copies the timer clock to a hardware register using the input capture of the MCU. Both Slave 1 and Slave 2 capture the timer during the pulse, this capture is then subtracted from the both timer clocks, effectively resetting and synchronizing the timer clocks on both Slave 1 and Slave 2 MCUs.
3. Slave 1 sends a radio command to check the time difference between the MCUs using the Synch-wire.
4. Slave 2 responds signifying it is ready to receive the command and Slave 1 pulses the Synch-wire. The time is store on both MCUs and Slave 2 sends the time back to Slave 1. Slave 1 will find the difference in time between the two MCUs and store it for later retrieval.
5. Slave 1 then issues another radio command telling Slave 2 to get ready for another high pulse on the Synch-wire, which will trigger the data acquisition system.
6. Slave 2 responds to Slave 1 with a radio command signifying it is ready to accept the pulse. Slave 1 pulses the Synch-wire triggering the data acquisition on both

³ See figures 6.1 and 6.2 for Synch-wire

Slave 1 and Slave 2.

In reality, steps 1 and 2 could be combined with steps 5 and 6 to create a much faster synchronization process. They were simply kept separate to allow steps 3 and 4, which would allow some feedback on to how well the synchronization process was working. As mentioned previously in section 6.3.1, the trigger delay was too long, so one of the first things to do is eliminate steps 3 and 4 and combine steps 1,2 with steps 5,6 for a significant reduction in the delay.

6.3.3 Data Acquisition

The data acquisition system is designed to take data at different intervals or frequencies set by the user. The system also remained the same in the transition from System 1 to System 2. The data acquisition runs off Timer 1 of the ATMEGA644v MCU. This timer is 16 bit and is clocked or prescaled from the MCU clock which runs at 4 million hertz or Mhz. The data acquisition also uses the ADC function of the chip to convert an analog voltage signal from the accelerometers into a digital signal that can be stored by the MCUs.

The data acquisition system as mentioned in section 6.3.2 is triggered by the synchronization function of the controller to make sure that both Slave 1 and Slave 2 start data acquisition close to the same time. Prior to the actual trigger, Slave 1 sends Slave 2 the required frequency of the data to be taken then both Slave 1 and 2 update their data taking intervals. The frequency in System 1 is set by the Master MCU which is set by the laptop user. Due to the trigger control moving from laptop to vehicle operator in System 2, the frequency control moved as well. The frequency control of System 2 has 5 preset levels indicated by an LED array on board Slave 1. The frequency was set by the rotating a potentiometer mounted on Slave 1 until the corresponding frequency light would light up.

It is important that the interval remain constant and accurate in order for the col-

lected data to have meaning. Therefore, the data acquisition uses an interrupt function of the ATMEGA644v chip. The interrupt function allows the software execution of the MCU to be interrupted by a programmed hardware state change. The advantage of this method is that hardware state changes have repeatable response times and do not depend on where in the execution of code the MCU is. When the interrupt happens, the processor is interrupted and the code is run to save the state of the processor at the time of interruption. The processor will then run code in the interrupt, then reload the state and continue on execution of code that was interrupted. What this means is that when an interrupt happens, there is a repeatable and predictable time for execution of the interrupt code, which is what is needed for consistent data. An interrupt is not instantaneous or does not necessarily offer the fastest response time; it only allows for repeatability and predictability. This means that measurements will be taken at the same accurate time apart which were the goals of the data acquisition program.

The hardware that triggers the interrupt is caused by the compare match of Timer 1 to specified number in OCR1A hardware register on the MCU. After the timer matches, it is reset and starts from zero. The number of clock cycles Timer 1 processes between interrupts is equal to OCR1A minus 1 because of the extra transition from OCR1A to zero. Therefore the interval time can be calculated using the following equation.

$$\text{Interval Time} = \frac{\text{OCR1A} - 1}{\text{freq}_{\text{timer}}}$$

The interval is not affected by the interrupt action time because it remains constant for reasons explained above.

By using a 16 bit timer, higher precision of the interval time is possible, compared to an 8 bit timer. But if a period is selected that is not a multiple of the MCU clock period, the interval will never be an exact number of MCU cycles. The software uses a simple algorithm that minimizes Timer 1's periods using different prescale values of the MCU clock. By minimizing Timer 1's periods, the closer Timer 1 can match a

desired frequency. To account for precision errors, the software prints out the actual frequency when data is downloaded so the actual frequency can be used instead of the desired frequency. It is for the end user to decide whether the desired frequency is close enough to the actual frequency, considering all tolerances involved.

The purpose of the interrupt is to trigger the ADC reading of the accelerometers. As mentioned previously in section 6.2.2, the ADC can only read one thing at a time. Therefore, two accelerometer readings are read one after another. Since the readings cannot happen simultaneously, they should be closer together than the interval. Therefore, the ADC clock is run fast and the estimated time difference is kept well below 1 percent of the period at 1000 Hz. For accuracy, multiple ADC readings are taken and averaged for every one reading. This approach was chosen over running the ADC at slower frequencies for redundancy.

The memory of the MCU where the measured data would be stored is 4000 bytes in size. This memory is used for program operation so all of it cannot be used for storing data. 2000 bytes were chosen to be set aside for storing measurement data. With two channels and measurements that are 10 bits long, it works out to 800 data points per channel of data that could be stored. The grey box simulation conducted in section 5.3 indicated that 800 points would be sufficient which is why it is listed as a requirement in section 6.1. More memory could possibly be set aside for storing data if the exact amount of memory needed for processor operation was calculated, but it was not deemed necessary.

The maximum speed of the data acquisition has not been tested but the system has been successfully tested at 800 Hz, which was specified by the smart system design requirements in section 6.1. To test the data acquisition system at 800 Hz the same accelerometer was connected to both channels of Slave 1 and Slave 2. Then the system was triggered while the accelerometer was subjected to arbitrary motion to see how well all 4 channels matched together. The 4 channels are graphed in figure 6.6 with a

very small vertical scale for detail. The total vertical scale for the ADC ranges from 0 to 1023, which is from zero volts to voltage supply respectively. Ideally, the points should be right on top of each other, but due to the time difference between measuring channel 1 and channel 2 on both MCU slaves, the points deviate a little. The synchronization of both slaves is accurate, shown by channel 1 of Slave 1 and Slave 2 being the same for or very close to the same. The same can be said for both channel 2s. The system is capable of more speed, especially with faster MCU clocks, but no attempt was made at testing the maximum speed. Even though a faster MCU chip could make the system faster, accuracy of accelerometer measurements may be reduced due to running the ADC too fast.

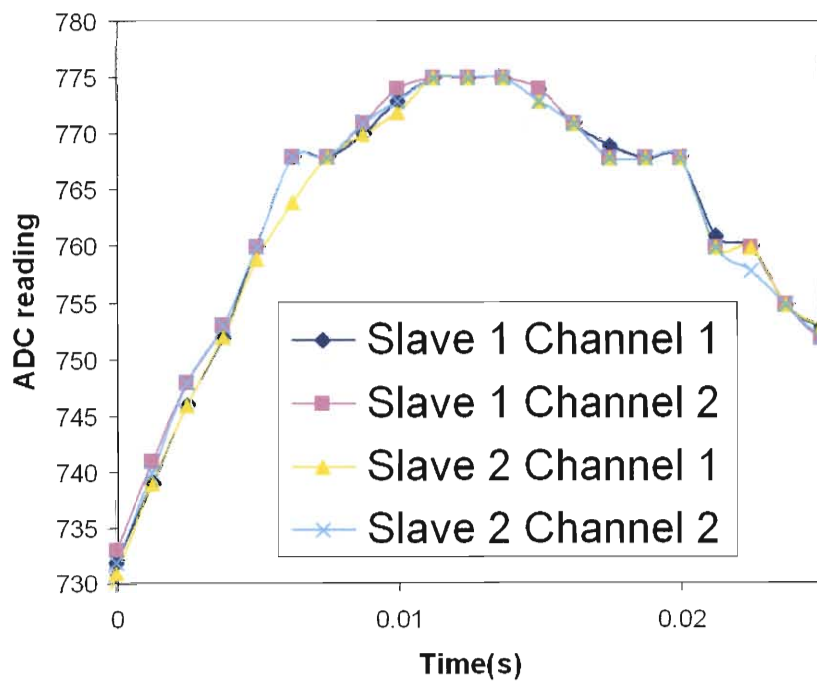


Fig. 6.6: Test Graph of Data Acquisition System at 800 hz. All channels are connected to the same input.

6.3.4 *Wireless Transmission*

The smart sensor systems transfer all data wirelessly using radios. In order to make the data transfer reliable, error checking is needed. The nRF24L01 radios themselves have hardware error checking, but another level of control is needed to make sure the data is transferred accurately in the system. Without this level of control, data could be frequently be lost in transmission because the hardware error checking will automatically reset after so many transmission errors. The software level of control also has checksum error checking for extra consistency, but this function is secondary to controlling data transmission of the system.

The radio control software was originally developed by the author during ME507 in Spring Quarter of 2007 at Cal Poly. This software was tweaked for performance in this project and became very important to the success of this project. This software, once set up, would allow data to be written by one MCU onto another that was connected to radio interface. The software would try until successful or until it was reset. It allowed very easy integration into the data acquisition system once setup properly. It was only used for the nRF24L01 radios in System 1 and System 2 and was not adapted or tested with the Maxstream radios.

The Maxstream radios were used to fix the range issue occurring with the nRF24L01 radios between the laptop and vehicle. Control software was not developed for them because they act as standalone asynchronous serial data or RS232 modems. The Maxstream radios solved range problem in System 2 because they were much more powerful than the nRF24L01 radios. But while they were more powerful, after testing with them in Test #2, (section 7.3), it was found out that they skipped lines of data due to transmission errors. Instead of developing the same software control for them as used with the nRF23L01 radios, a quick tweak to software made them hold transmission until the radio on the vehicle and the laptop were in close proximity of each other. This extra state is shown in table 6.1 and required the vehicle operator to

press the trigger button to start the transmission. This process, though not ideal, made the system successful in data transmission from vehicle to laptop.

6.3.5 “Smarts”

This system can be considered smart because of all aspects explained under section 6.3. One aspect that was not previously mentioned that is considered smart is the system’s ability to detect accelerometer saturation and throw out the data. This works by checking accelerometer readings—if the accelerometer reaches max or min value, (1023 and 0 respectively), 10 times, the data is thrown out as saturated. It then sends a message to the laptop saying the data is saturated and on which MCU. This was an important time saving device that would let you know of bad data and throw it out before time is wasted analyzing it.

An obvious smart feature not implemented is a smart trigger. This trigger could be configured to find what it thinks is good data and store it. It might work by having the MCU looking for signs of good data while taking data all the time and storing it in a looped buffer. It might look for consistent large variations in amplitude as an indication of good data. If it found what it thought to be good data it would decide how far back in the data buffer to set aside as actual data for transmission to the laptop.

More smart features are easily integrated into the system. The MCUs were not maxed out in this project, and a simple increase in clock speed could more than double processing power. Smart features such as data pre-processing could provide very useful information as to the quality of the data before actual parameter estimation is performed. The ease of use and performance of parameter estimation can be improved by addition of more smart features.

7. TESTING



Fig. 7.1: The Vehicle used for testing the system: 2008 Cal Poly SAE Baja Competition vehicle

7.1 Test Vehicle

To test this project, the 2008 Mini Baja Competition Vehicle pictured in figure 7.1 was used. This vehicle is a small single occupant off-road car made for the SAE Baja Collegiate Competition. This vehicle has a full roll cage and durable long travel suspension. This vehicle was ideally suited for this test because most parameters were

already known. In addition, the vehicles small scale and tube frame allowed easy integration of sensor and computers. The known parameters that relate to this project are listed in table 7.1. The parameter data listed for the baja car is not guaranteed to be accurate, and in most cases, is the best educated guess. The mass for the car includes the test driver but the estimated unsprung mass has been subtracted to get the sprung mass. The spring rates should be within 5% because they are cataloged and calculation is based merely on geometry. Damping rates were estimated using manufactured supplied numbers for bound and rebound in the front, which were 5 and 15 lb-s/in respectively. These numbers were averaged to 10 lb-s/in, then made effective through the suspension motion ratio. The manufacturer did not supply numbers for the rear damper but they were known to use the next greater damping rate setup compared to the front damper.

Tab. 7.1: 2008 Cal Poly SAE Baja Competition Car

Parameter		2008 Baja Car	
sprung mass	m	1.294	lb-s ² /in
radius of gyration	k_g	22	in
pitch inertia	i	626	lb-s ² -in
weight distribution front	wdf	.47	
wheel base	l	68	in
spring rate front	k_f	100	lb/in
spring rate rear	k_r	100	lb/in
damping rate front	c_f	7.2	lb-s/in
damping rate rear	c_r	10	lb-s/in
damping ratio front	c_{z_f}	.461	
damping ration rear	c_{z_r}	.603	

It should be noted that the baja car parameters are not necessarily accurate for a few reasons listed below, but the goal of this project was not necessarily absolute parameter accuracy. It was merely to see if it might be possible to use a system such as this to estimate parameters with reasonable accuracy. Therefore, not very much time was spent measuring the car and perfecting the lumped parameters of the model like that

which was done in Kim and Ro [5].

The cars mass and inertial properties changed with each driver and many different setups so it was not static and changed from day to day as the team added and subtracted components. These changed the mass and inertial properties, which necessitated the weighing of the car with driver before every run.

Dampers were never put on a shock dynamometer and the only numbers the team has are numbers for are the lowest damping setting from the manufacturer. This is unfortunate because the non-linearity or curvature of the damping curves was not known. The actual estimated damping rate should depend on how the damper is cycled up and down during the test.

The pitch inertia of the car was estimated using known and estimated masses of components. A proper swinging pendulum setup or some other apparatus would need to be constructed to actually measure the inertia. A 4 post testing rig was not available to optimize lumped parameters. How the parameters are lumped, such as what mass is unsprung versus sprung, can affect how close the model will match the vehicle.

The 2008 Baja Car was instrumented with 4 accelerometers in the way described in section 6.2.1. Small aluminum brackets were made and mounted on the car in the prescribed locations to make vertical surfaces for accelerometers. The sensors were mounted on the car using Scotch Mounting Squares, which are double stick foam tape squares. Foam mounting of accelerometers was used in the hopes that the foam would damp high frequency vibrations. The accelerometers were aligned vertical by hand which means that they were within 5 degrees from vertical. Shielded wire was used to minimize electrical noise between the accelerometer and MCU. The MCUs were powered by two 9 volt batteries in parallel that were regulated to 5 volts with a voltage regulator.

7.2 Test #1

The first test used sensor System 1 with very limited success. The test was conducted on a relatively smooth farm field used for tractor training at Cal Poly University. Even though the field was smooth, it had berms and elevation changes that separated the field from the adjoining road and farm field. Due to the model and this project's design, symmetrical inputs to both left and right sides of the vehicle were required. The only clearly symmetrical terrain would be hitting the berms head on. But due to the failure of System 1 to get data, the terrain input at this field was not tested.

System 1 as mentioned previously in section 6.3 has range issues that caused the remote trigger and data transfer function to be useless. The system would work with the car off and the laptop within a few feet, but once the engine was running it was impossible to control the system. Causes for poor range could be magneto ignition system of the 2008 Baja Car's Engine, and to low gain antennas. Due to these issues, System 1 was modified to System 2 which moved the trigger on board the car and used more powerful radios for the wireless link to the laptop computer.

7.3 Test #2

Trying to remedy the failures of test # 1, this test had many changes. The course was now made up parking stops that made 2 bumps wide enough for both left and right sides of the car to contact at the same time. The 2 bumps were set approximately 12 inches apart and the car was driven over them at speed. This test was the first to use an early version of System 2. This system did not include the feature that delayed transmission to the laptop, so the data collected was missing measurements. The data also had significant amount of noise due to engine vibration. The engine in the vehicle is a single cylinder 4 stroke engine with no counter balancer and is mounted directly to the frame with no built-in compliance such as rubber mounts. Therefore by itself it would vibrate the chassis mounted accelerometer as shown in figure 7.2.

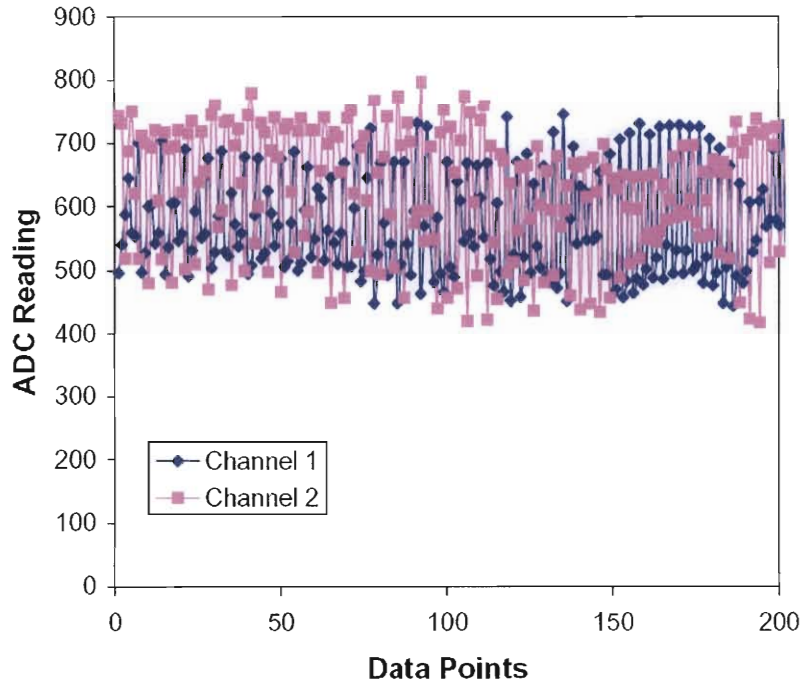


Fig. 7.2: Engine noise data for the 2008 Baja Car, Data is taken at 200 Hz with car not moving and engine revving randomly

7.4 Test #3

This was the first test from which successful data was collected. To eliminate the engine vibration the car was manually pushed over the same parking stop bumps at around 7-12mph. Before the car hit the bumps, the pushing of the car stopped to avoid extra chassis forces. This speed was just slow enough to not saturate or max out the accelerometers.

Though the data taking was successful, problems arose during analyzing the data afterwards. Due to the trigger delay the data was taken mid bump which means that the initial states of the suspension and chassis are unknown and non zero. Another problem was the car was setup with multiple rate springs and inconsistent damping settings. The

grey box code was not able to estimate realistic parameters using this data, which may have been due to the unknown initial states and highly nonlinear suspension.

7.5 Test #4

This final test gave successful data listed in chapter 8. The issues in test #3 were taken care of by fitting linear single rate springs to the suspension and having the driver predict the bump and hit the trigger early to get a steady state data before the bump. The adjustable dampers were softened to their lowest settings for which the damping was specified but not verified for the front dampers. The car was still manually pushed over parking stops as described in test #3.

8. RESULTS

8.1 Data Processing

The results for this project are from data obtained in the fourth and final test which is in section 7.5. Two successful data sets were taken. The data was determined successful if the system was triggered early enough to get steady state data but not too early that the session was terminated before hitting the obstacles. For ease of reference the data sets will be named 400 Hz Data and 800 Hz Data. 400 Hz Data was taken at 400hz and 800 Hz Data was at 800hz.

The 400 Hz Data raw data was first scaled using calibration data for each of the accelerometers. Each sensor had its static value subtracted and was multiplied by a constant to convert the units to in/s^2 . The linear trend was also subtracted from the now scaled 400 Hz Data to remove possible accelerometer drift. The 400 Hz Data accelerometer data is shown in figure 8.1.

Using Matlab and Simulink, 400 Hz Data was then integrated twice to get velocity and displacement input output data listed in section 5.4.2. This data is graphed in figure 8.2 which shows the input data on the left side, and the output data on the right. It is important to note that the input data is not the road position, but the wheel hub movement. This model does not include the tire and the tires interface with the ground, which is explained in chapter 3. The front and rear wheel displacements indicate that there were two bumps. Figure 8.2 compares the model with estimated parameters by running the input data through the state space car model (equation 5.5) to get output data which is then graphed along side the actual output data. The model fits what is

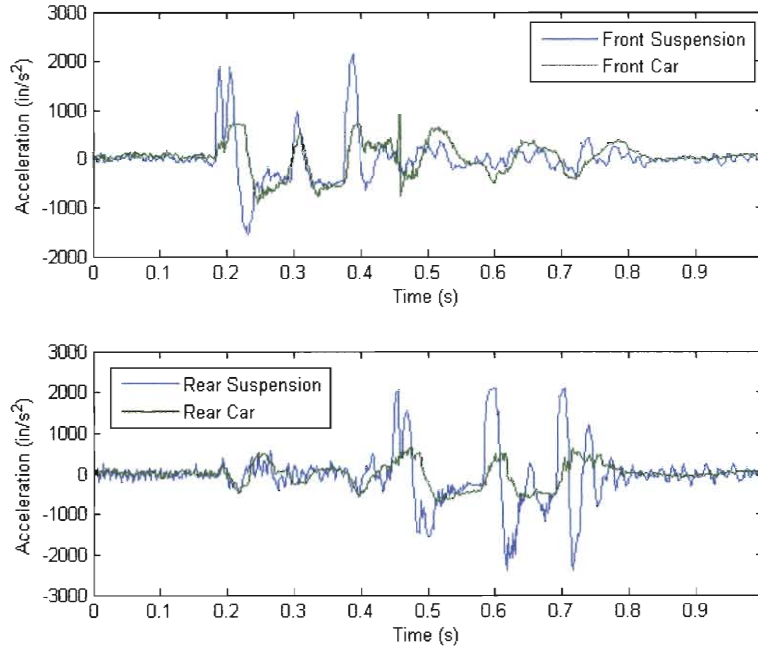


Fig. 8.1: 400 Hz Data Acceleration Data scaled with linear trend removed, taken at 400 hz with 800 points per channel

observed fairly well in this case, although reasons are offered for the inaccuracies in section 8.3.

800 Hz Data was handled the same way 400 Hz Data was handled with the scaling and subtracting the linear trend from the data. The problems arose when 800 Hz Data was run through the grey box parameter estimation explained in section 8.2. The parameters estimated using 800 Hz Data were many orders of magnitude off from the estimated values. 800 Hz Data was processed again, but the linear trend was not subtracted. This resulted in the accelerometer drift not being filtered out, but gave better grey box parameter estimation values. Therefore, subtracting the linear trend from 800 Hz Data corrupted the data, which did not happen in 400 Hz Data.

Without subtracting the linear trend from 800 Hz Data, there was a lot of accelerom-

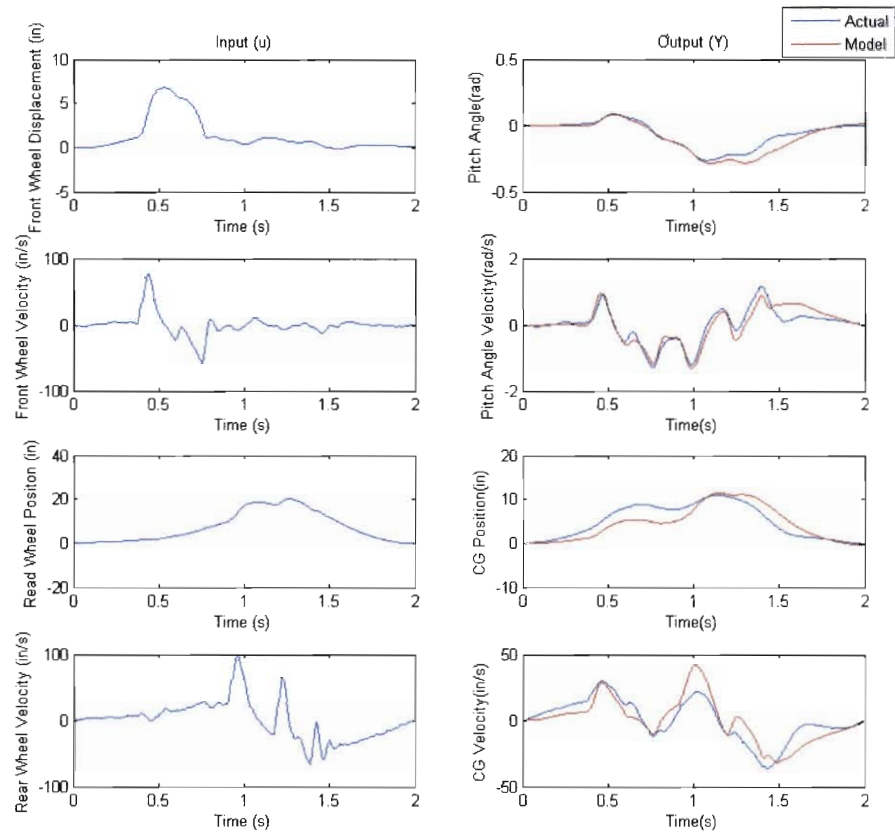


Fig. 8.2: 400 Hz Data State Space Input Output Actual Data; Output data compared to theoretical car model data generated from actual input data shown to the left

eter drift or constant errors that caused the second derivative (position) to go out of reasonable bounds. To get rid of this error, a constant was manually subtracted from 800 Hz Data acceleration data until the second anti derivative of the data ended on zero. Figure 8.3 shows 800 Hz Data acceleration data after the constant was subtracted.

800 Hz Data was integrated the same as 400 Hz Data using Simulink to get input and output data needed for grey box parameter estimation. Figure 8.4 shows 800 Hz Data after being integrated. The same vehicle model that was compared to 400 Hz Data

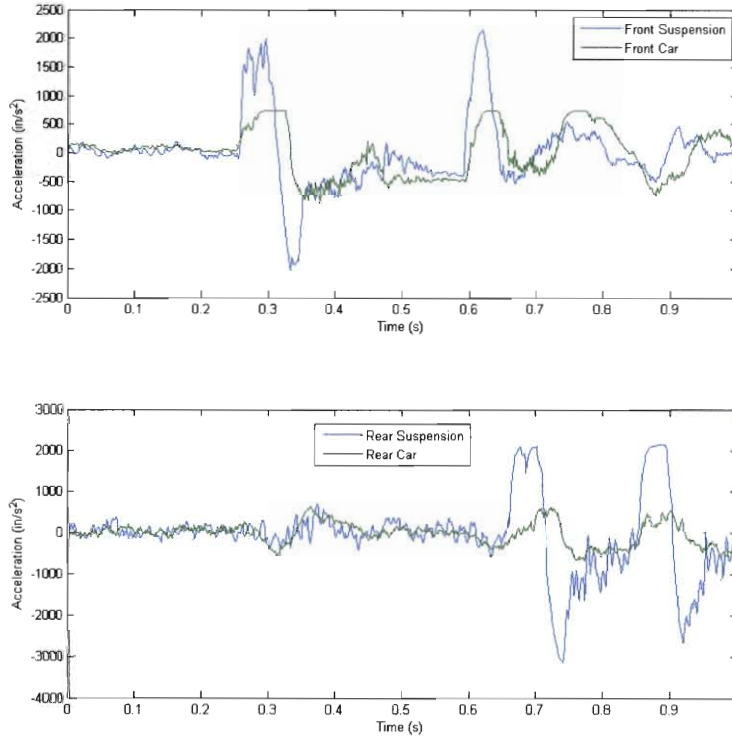


Fig. 8.3: 800 Hz Data Acceleration Data scaled with constant removed, taken at 800 Hz with 800 points per channel

in figure 8.2, is compared to 800 Hz Data in figure 8.4 as well.

8.2 Grey Box Estimation

The input output data is then run through the grey box system identification tool function in Matlab to identify i , c_f and c_r using both 400 Hz Data and 800 Hz Data. The results of this simulation are in table 8.1 and are compared to the 2008 parameters first presented in section 7.1 on page 42. These parameters are also the same used for the comparison model in the right sides of figures 8.2 and 8.4. As seen in the table, the

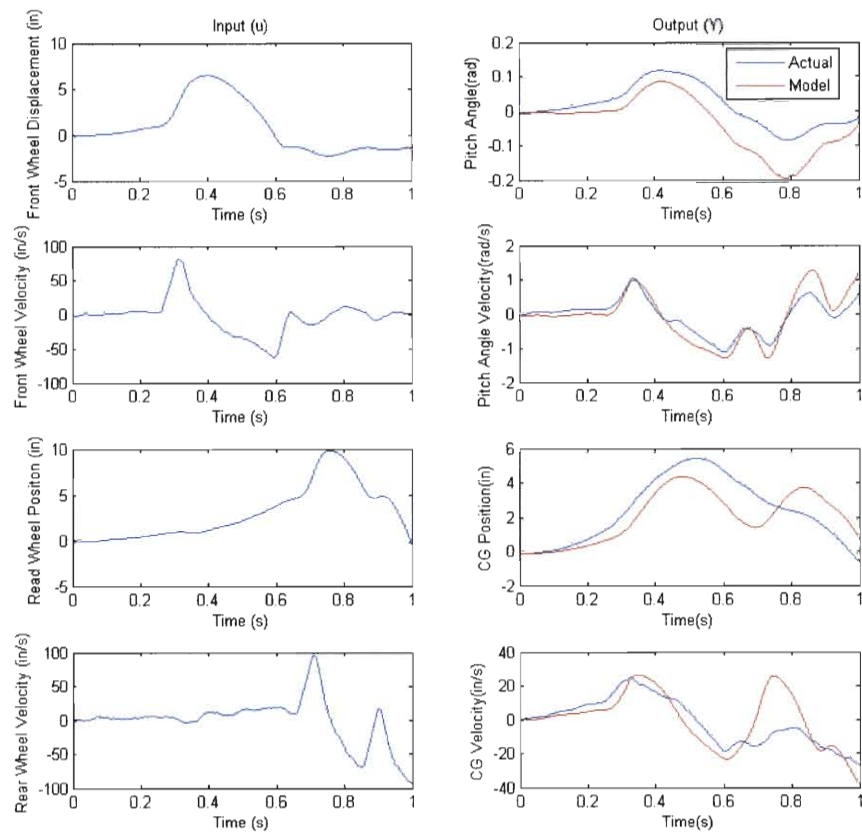


Fig. 8.4: 800 Hz Data State Space Input Output Actual Data; Output data compared to theoretical car model data generated from actual input data shown to the left

parameters match fairly well. The vehicle pitch inertia seems to be off by the most but if you examine the radius of gyration, (k_g), you will find the smaller variations from a very roughly estimated car inertia explained in section 7.1. The variations are smaller because inertia (i) depends on k_g^2 .

Tab. 8.1: Grey Box Estimated Parameters

Parameter	Estimated Grey Box Using		2008 Baja Car	Units
	400 Hz Data	800 Hz Data		
i	796	667	626	lb-s ² -in
c_f	9.2	8.3	7.2	lb-s/in
c_r	10.4	12.4	10	lb-s/in
k_g	24.8	22.7	22	in

8.3 Sources of Error

A major source of error could be related to the model used to describe the vehicle. The errors due to this model can be broken up into two categories, errors due to non-linearities of the system and errors due to inadequate degrees of freedom. These two sources of error are examined in this section separately under in this section.

Other sources of error are signal processing and sensor errors which are also examined separately in this section.

8.3.1 Errors due to Presence of Non-Linearities

Errors due to non-linearities are one of the downsides of using a linear model. The known non-linearities present during the testing phase are the different rebound and bound damping rates of the dampers, and possible suspension travel limits of the vehicle. For the damper settings used during test #4, the rebound rate was three times the rate as mentioned in section 7.1. The reason that this was not remedied is because the damping rates were only known for these settings.

The suspension travel limits of the suspension are non-linear because the spring or stiffness characteristics change when the suspension contact the travel limiting devices. This non-linearity can be avoided by not going over rough enough terrain to bottom out the suspension. Using the second integral of 400 Hz Data and 800 Hz Data, the suspension travel can be estimated and is shown in figure 8.5. Upon inspection of the estimated suspension travel it shows that the suspension did hit the limits of its travel for 400 Hz Data data set in the rear shown in figure 8.5(a). The limits for the rear suspension are close to 4 up and 6 down. This is the only travel limitation suggested by figure 8.5. The suspension travel shown is very fragile data because it depends on the second integral of 400 Hz Data and 800 Hz Data. The second integral depends greatly on how accelerometer drift¹ was filtered out of the raw acceleration data.

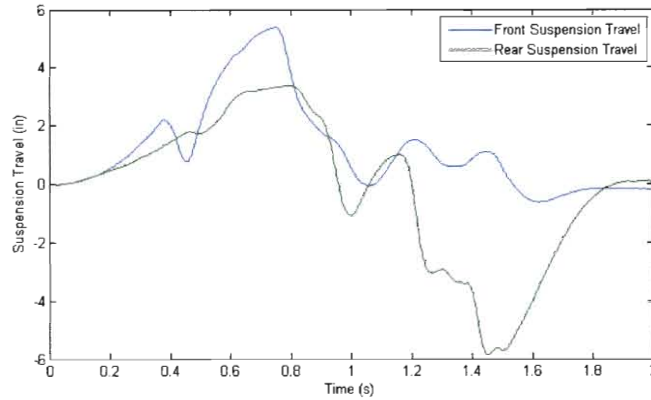
Using a non-linear model adds many complexity issues and greatly reduces the amount of engineering tools available for this project, which is why the model is linear. The grey box parameter estimation could not handle a non-linear model for the functions used.

Since the model does not include the tire and the tire's interface to the ground, this linear system should be robust from non-linearities such as when the tire comes off the ground. The motion ratio variation is known to be small on the test vehicle, so it is not a major source of non-linearity.

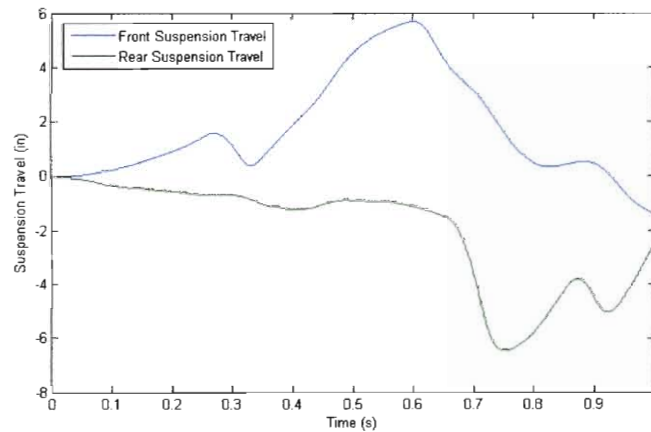
8.3.2 *Errors due to insufficient model degrees of freedom*

The most obvious crux of the 2 degree of freedom bicycle type model is that it depends on symmetry of both the vehicle and the inputs. This symmetry is not always present and never perfect, so when using a model that depends on symmetry, there will always be sources of error. If left or right weight transfer occurs, the vehicle is not symmetrical for that test. 400 Hz Data and 800 Hz Data data sets were both collected by running the vehicle over very symmetrical terrain, which was required by the model choice. To

¹ See sections 6.2.1 and 8.3.3 for further explanation of accelerometer drift



(a) 400 Hz Data



(b) 800 Hz Data

Fig. 8.5: The estimated suspension travel from 400 Hz Data and 800 Hz Data data sets

eliminate the symmetry constraint for more variety of inputs, the model would have to be expanded to something similar to the 7 degree of model used by Holen [1]. The 7 degree of freedom model includes degrees of freedom for all wheels, so the input of the system is at the tire road interface, which is hard to measure. A better model would be to add roll of the vehicle chassis to the current model, making it 3 degree of freedom. With the addition of roll there would now be 8 possible inputs—the positions

and velocities of all 4 wheels. This adds a great deal more complexity to the system, like increasing the accelerometer sensor count from 4 to 7, but would eliminate the symmetrical constraint of this project.

The vehicle chassis is modeled as a rigid body even though theoretically it has infinite degrees of freedom. The assumption that allows this is that the chassis is very stiff compared to the stiffness of vehicle suspension. This assumption also has to be made for suspension links in the system. Vehicle chassis and suspension flex or vibration are probably not a major source of error during testing of this project because the assumptions apply to the test vehicle.

Another issue of concern is how the suspension geometry relates to this model. The spring and damper of the suspension exert most of the vertical force that this project is concerned with, but the suspension links can also exert force in the vertical direction. This can cause discrepancies in the estimation of the pitch inertia because the model assumes that point of rotation of the vehicle is at the *CG* location, but if the vehicle suspension geometry forces it to rotate about a different point, then pitch inertia is artificially increased due by $m \cdot r^2$ where m is the sprung mass and r the distance from the point of rotation to the *CG*. This error might have easily contributed to the variation of i and hence k_g in table 8.1.

How the suspension geometry affects pitch of vehicle is similar to how suspension geometry affects roll of the vehicle. Suspension geometry determines the roll center, which is the point about which a vehicle will roll and a common design aspect in vehicle dynamics. There is also a pitch center caused by suspension geometry that can force the vehicle to rotate about a pitch center in the same way a vehicle will roll about a roll center.

The suspension geometry design of the 2008 Baja car causes its wheel movement to not be vertical with respect to ground and in the case of the rear the wheel movement is in an arc. During testing the front tires would leave the ground therefore the pitch

center would likely move closer to the rear of the car. When the rear tires left the ground, the pitch center would likely move toward the front of the car. This means that the pitch center can vary depending on the state of the vehicle similar to the roll center.

Another area where suspension can affect the model is when a lateral load is applied at the tire, causing a jacking force to be applied to the vehicle body through suspension links. This lateral force can be generated even when the vehicle is not cornering through tire scrub, which is the lateral movement of the tire on the ground when the suspension travels up and down. This force can affect the entire model results and may act as a coulomb friction damping source.

This model only accounts for vertical accelerations and rotations of the chassis—no horizontal or lateral accelerations are included. Instead of adding many more degrees of freedom to account for these motions, this source of error can be reduced by making sure that the vehicle is not experiencing very high horizontal and lateral accelerations when data is taken. This is likely a small source of error because care was made to make sure the vehicle was not braking, accelerating and going straight when 400 Hz Data and 800 Hz Data was gathered in test #4.

8.3.3 Errors due to Data Acquisition

Data acquisition errors are present because the type of data filtering done before grey box parameter estimation greatly affected the estimated parameters. The major error that required filtering is accelerometer drift. This happens with integration of accelerometer errors that will compound over time, especially if multiple integrations are taken. Accuracy of estimated parameters therefore can be improved if accelerometer drift errors can be reduced using more sophisticated filters or different sensors.

The accelerometer analog bandwidth is 50 hz, which might not be sufficient for very fast suspension motions even though the suspension modal frequencies are generally less than half of 50 hz. The data was sampled in 400 Hz Data and 800 Hz Data at

much higher frequencies, so aliasing is not a major contributor.

9. CONCLUSION

The methods presented in this project are shown to estimate vehicle parameters with reasonable accuracy. A smart data acquisition system has been developed and tested to be accurate and easy to install on a common vehicle for parameter estimation. Since the smart system is almost completely wireless, the installation and use of the system is simplified. The grey box approach is a good option for parameter estimation for a vehicle. Using Matlab's System Identification Toolbox the grey box estimation code was easily implemented and tested for viability in estimating vehicle parameters.

This thesis project is an investigation, or proof of concept, that suspension parameter estimation is possible using methods discussed in this report. There are, however, many recommendations for how to build on this thesis project to make this system more robust.

All the sources of error listed in section 8.3 need to be further researched and tested so they can be minimized and possibly eliminated. Depending on the application of this project, some errors may not need to be eliminated. For example if this system is used to calibrate a vehicle dynamics active control system, then estimated pitch might be more valuable with the effects of suspension geometry explained in section 8.3.2, because that is the how the vehicle reacts. The pitch inertia about the CG might only be important to an active control system if the vehicle rotates about the CG which might not necessarily be the case.

The electronics packaging for the mechatronics system needs to be further developed from a bread board to a printed circuit board (PCB) with enclosure. It might be

best to integrate the accelerometer with each MCU unit so wires to accelerometers are limited and mounting of components is minimized. The synchronization of the system could be further developed to not use any wires for a completely wireless system.

The grey box estimation code could be further expanded to use a non-linear model. This would make it more applicable to the non-linearities present in most vehicles. The integration of grey box code on-board the vehicle would allow parameter estimation to happen real-time. Real-time parameter estimation could be coupled with a vehicle control system allowing the control system to react to varying suspension parameters.

This thesis project provides a good start for a vehicle suspension parameter estimation using a smart sensor system. Using the recommendations listed above, this thesis project might be expanded to be used for a wider variety of applications with more robust results and performance.

BIBLIOGRAPHY

- [1] Holen, P. (2006). *On modally distributed damping in heavy vehicles*. PhD thesis, KTH, Aeronautics and Vehicle Engineering.
- [2] Iannce, P. (2002). *ME 416 Class Notes, California Polytechnic State University San Luis Obispo*. El Corral Publications.
- [3] Kasprzak, J. L. and Floyd, R. S. (1994). Use of simulation to tune race car dampers.
- [4] Keun, K. (2005). Real-time vehicle dynamics parameter estimation. *Proceedings of the ASME Dynamic Systems and Control Division-2005; presented at 2005 ASME International Mechanical Engineering Congress and Expositi*, pages 313–318.
- [5] Kim, C. and Ro, P. I. (2000). Reduced-order modeling and parameter estimation for a quarter-car suspension system. *Journal of Automobile Engineering*, 214(8):851–864.

APPENDIX

A. HARDWARE

The main hardware parts and locations of data sheets are listed in table A. Some hardware components were mounted on breakout boards by an independent company for easier prototyping. The independent company will be listed in table A under assembler. Breakout boards are printed circuit boards designed for mounting of small surface mount chips for ease of electrically connecting wires and pins to the chip.

All smaller common hardware components such as resistors, capacitors, LEDs will not be listed but are required for this project.

Tab. A.1: Main hardware components of smart sensor systems 1 and 2

Name	Part Number	Manufacturer	Assembler	Vendor	Cost
Microcontroller	ATMEGA644V-10PU	Atmel	N/A	digkey.com	\$7.87
Accelerometer	ADXL322	Analog Devices	sparkfun.com	sparkfun.com	\$29.95
Audio	MAX9814L01	Nordic Semiconductor	sparkfun.com	sparkfun.com	\$19.95
Radio	X09-009WNC	MaxStream Inc	N/A	digkey.com	\$166.94
RS232 to USB	FT232RL	FTDI Chip	sparkfun.com	sparkfun.com	\$14.95

Figures A.1 through A.4 show the how the systems are wired. They do not include wiring for programming which is covered in the ATMEGA644V data sheet. All prototyping was done on a breadboard which is a solder-less prototyping board. More info can be found <http://en.wikipedia.org/wiki/Breadboard>

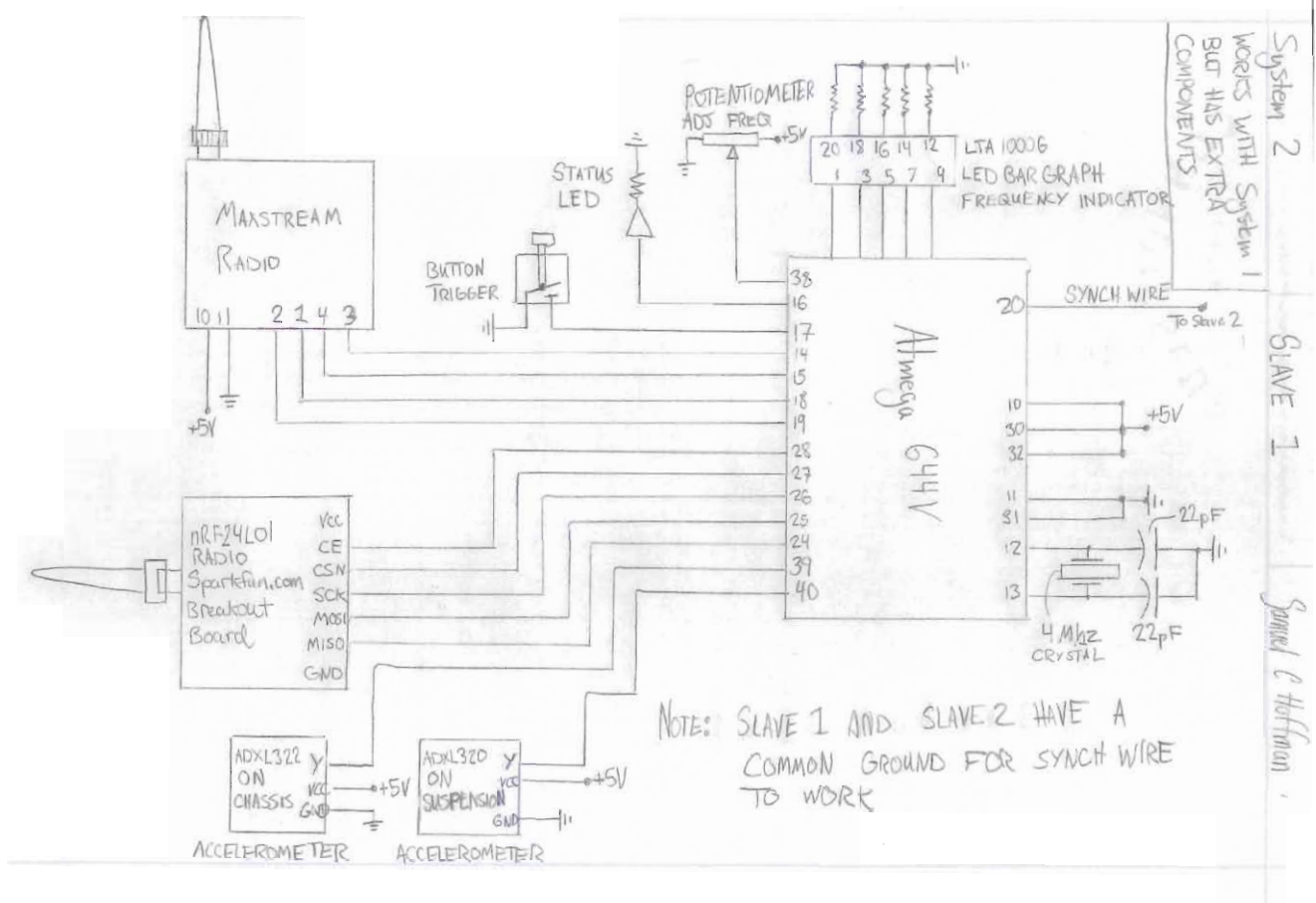


Fig. A.1: Slave 1 MCU wiring diagram

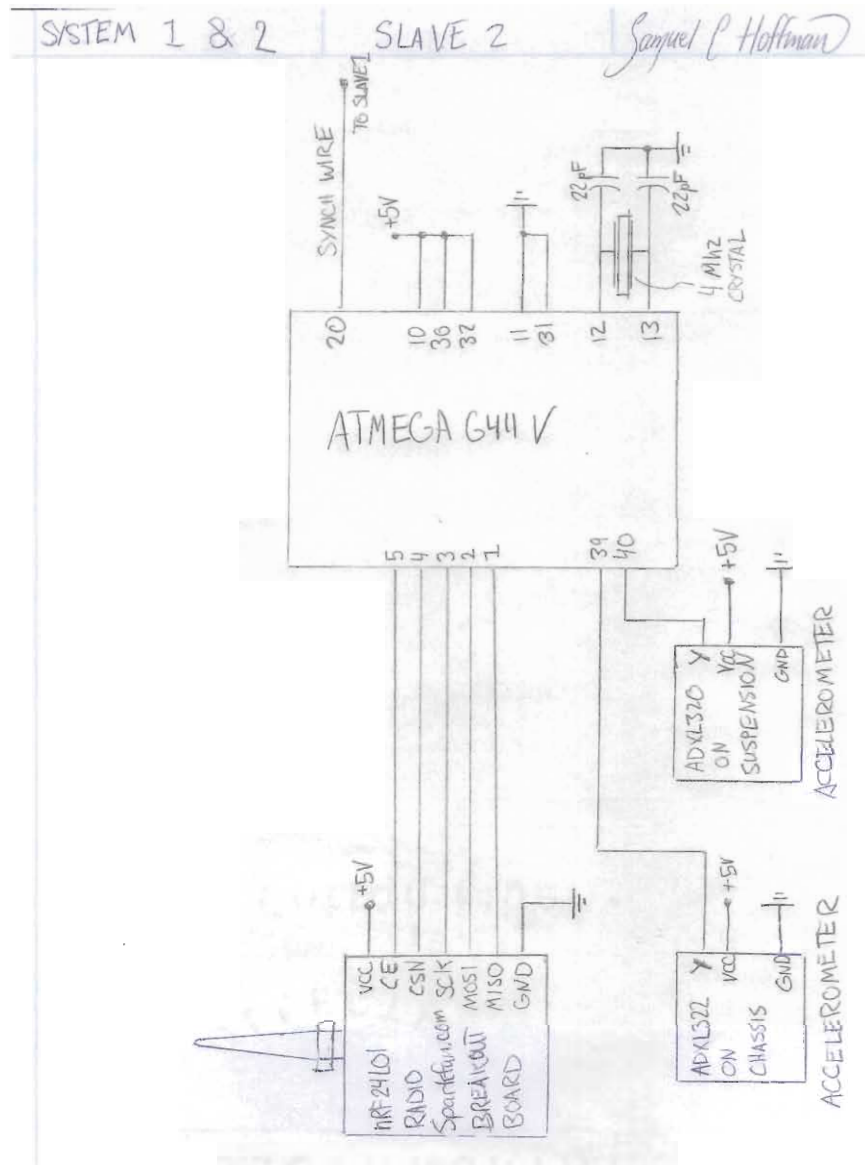


Fig. A.2: Slave 2 MCU wiring diagram

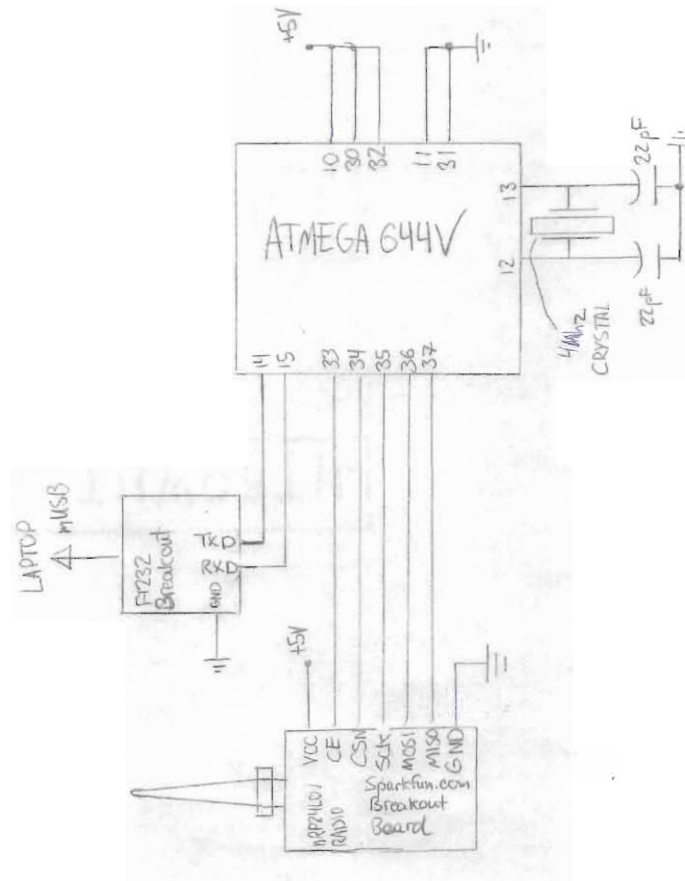


Fig. A.3: Master MCU wiring diagram

SYSTEM 2

LAPTOP RADIO

Samuel C. Hoffman

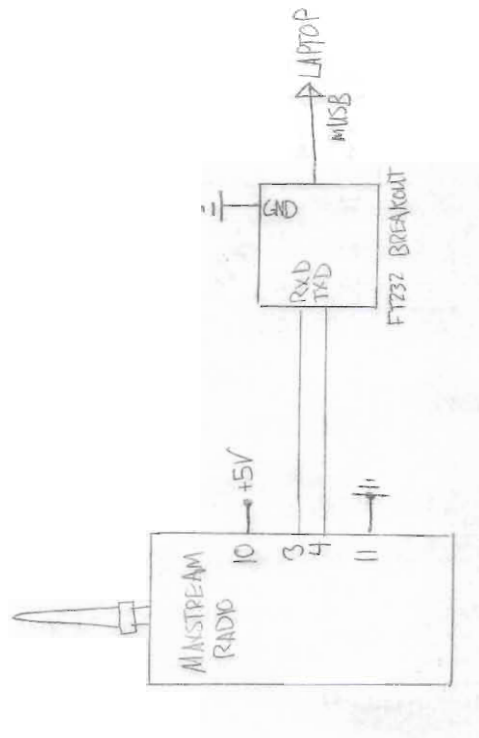


Fig. A.4: Maxstream Radio interface with Laptop

B. SOFTWARE

All software files can be found here:

<http://me.me.calpoly.edu:1280/svn/suspension/tags/>

or on the supplemental compact-disc (CD) under directory “Smart System Control”.

The documentation is on the following pages for ASV, or Automotive Suspension Vibration, which is name of the software used in this thesis.

ASV Reference Manual

v.8

Generated by Doxygen 1.5.2

Fri Apr 25 12:03:55 2008

B.1 ASV Hierarchical Index

B.2 ASV Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

<code>array10bits</code>	71
<code>packit_uart</code>	84
<code>spi_bb_port</code>	85
<code>uart</code>	88
<code>avr_9xstream</code>	75
<code>nb_uart</code>	82

B.3 ASV Class Index

B.4 ASV Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<code>array10bits</code>	71
<code>avr_9xstream</code>	75
<code>nb_uart</code>	82
<code>packit_uart</code>	84
<code>spi_bb_port</code> (Define this to enable debugging features)	85
<code>uart</code>	88

B.5 ASV File Index

B.6 ASV File List

Here is a list of all documented files with brief descriptions:

Desktop/ASV/Code/ASV.cpp (Main run file for all MCU devices)	95
------------------------------------------------------------------------	----

Desktop/ASV/Code/ avr_9xstream.h	98
Desktop/ASV/Code/ avr_a2d.h (This file is the header for using the A/D converters on an AVR processor. It is designed to contain code which can be used on several processor versions:)	99
Desktop/ASV/Code/ avr_serial.h	102
Desktop/ASV/Code/ avr_spi_bb.h	103
Desktop/ASV/Code/ interval.cpp	104
Desktop/ASV/Code/ interval.h	106
Desktop/ASV/Code/ nRF24L01.h	108
Desktop/ASV/Code/ packed_arrays.cpp	109
Desktop/ASV/Code/ packed_arrays.h	110
Desktop/ASV/Code/ packit.h	111
Desktop/ASV/Code/ synch_data.h (Files used to synchronize one micorcontrollers data taking with another)	112

B.7 ASV Class Documentation

B.8 array10bits Class Reference

```
#include <packed_arrays.h>
```

Public Member Functions

- **array10bits** (int)
- int **get** (int)
- bool **put** (int index, int data)
- unsigned char **get_data_byte** (int idx)
- int **size** (void)
- int **bytes** (void)

B.8.1 Detailed Description

This class implements an array of 10-bit numbers, storing them by "packing" them into memory so as to not waste bits. Putting each 10-bit number into a 16-bit int would waste about 37% of the memory used. The numbers which are given to, and taken from, the array are in integers (the next convenient size up). This class implements very crude bounds checking, making sure nobody tries to manipulate data from outside the array's bounds. As this is C++, indexes begin at zero.

B.8.2 Constructor & Destructor Documentation

B.8.2.1 array10bits::array10bits (int new_size)

This constructor creates an array object to efficiently hold a bunch of 10-bit numbers in memory.

Parameters:

`new_size` This is the number of 10-bit numbers which the new array can store.

B.8.3 Member Function Documentation

B.8.3.1 int array10bits::get (int index)

This method returns a 10-bit number from the given index location in the array. If an out-of-range index is given, the number 0x8000 is returned. This is a reliable error code because it's not a valid 10-bit number. The error code can be conveniently checked by looking at the highest bit in the returned number.

Parameters:

`index` The location from which the data is to be retrieved

Returns:

The number from the array, or 0x8000 if that data can't be retrieved

B.8.3.2 bool array10bits::put (int index, int data)

This method inserts a 10-bit number into the array at the given index location. If the index is out of bounds, the function returns false; if things work out okay, it returns true.

Parameters:

index The location from which the data is to be retrieved

Returns:

True if the data was stored, false if not

B.8.3.3 unsigned char array10bits::get_data_byte (int idx) [inline]

This method returns a byte from the data storage array. It is intended to be used for testing purposes, but it will probably be useful when sending data over a link such as serial or radio, or when storing data in some mass storage device.

Parameters:

byte_num The index into the array of bytes from which to get data

Returns:

The data byte from the array

B.8.3.4 int array10bits::size (void) [inline]

This method returns the number of 10-bit items in the array.

Returns:

The number of items in the array

B.8.3.5 int array10bits::bytes (void) [inline]

This method returns the number of bytes used to store the data.

Returns:

How many bytes are used to store the data

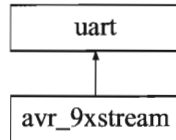
The documentation for this class was generated from the following files:

- Desktop/ASV/Code/**packed_arrays.h**
- Desktop/ASV/Code/**packed_arrays.cpp**

B.9 *avr_9xstream* Class Reference

```
#include <avr_9xstream.h>
```

Inheritance diagram for `avr_9xstream`::



Public Member Functions

- `avr_9xstream` (bool)
- void `sleep` (void)
- void `wake_up` (void)
- bool `putchar` (char)
- void `puts` (char *)
- unsigned char `timeouts` (void)
- void `write_bin` (unsigned char)
- void `write_hex` (unsigned char)
- void `write` (unsigned char)
- void `write` (char num)
- void `write_bin` (unsigned int)
- void `write_hex` (unsigned int)
- void `write` (unsigned int)
- void `write` (int)
- void `write_hex` (unsigned long)
- void `write` (unsigned long)
- void `write` (long)

Protected Member Functions

- **bool wait_for_CTS** (void)

B.9.1 Detailed Description

This class communicates with the 9XStream radio. It is a descendent of the **uart** (p. 88) class from `avr_serial.*`. The radio adds support for the CTS and sleep pins, and it operates the serial port with a baud rate which is set in **avr_9xstream.h** (p. 98).

B.9.2 Constructor & Destructor Documentation

B.9.2.1 avr_9xstream::avr_9xstream (bool setup)

This constructor creates a radio modem object by calling the serial port constructor setting up the input and output pins for CTS and sleep mode, and sending the correct setup codes to the radio modem. Due to the need for dumb delay loops to satisfy the radio's timing requirements, this constructor takes several seconds to execute.

B.9.3 Member Function Documentation

B.9.3.1 bool avr_9xstream::wait_for_CTS (void) [protected]

This method waits for the modem's CTS line to become low, indicating that the modem is ready to send data. There's a timeout in case the modem is never ready.

Returns:

True if the radio is ready to send, false if there was a timeout

B.9.3.2 void avr_9xstream::sleep (void)

This method causes the radio to enter sleep mode. The sleep mode used is "pin sleep", in which the radio consumes about 25 uA of current and wakes up only when its sleep pin (pin 2) is set to 0.

B.9.3.3 void avr_9xstream::wake_up (void)

This method awakens the radio from sleep mode by de-asserting the sleep pin.

B.9.3.4 bool avr_9xstream::putchar (char the_char)

This method writes one character to the radio modem through the associated serial port. Before doing so, it checks to ensure that the Clear To Send line is active. If the CTS line doesn't become active and times out, the character is not sent and this function returns false.

Parameters:

`the_char` The character which is to be sent

Returns:

True if the character was successfully sent out, false otherwise

Reimplemented from **uart** (p. 89).

*B.9.3.5 void avr_9xstream::puts (char * str)*

This method writes a string to the radio modem. It writes one character at a time until the null character at the end of the string is reached. Note that this method can block the processor's execution and take an awful lot of time.

Reimplemented from **uart** (p. 90).

B.9.3.6 unsigned char avr_9xstream::timeouts (void) [inline]

This method returns the number of timeout errors which have occurred.

B.9.3.7 void avr_9xstream::write_bin (unsigned char num)

This method writes an unsigned character to the radio modem in binary format. It overrides the serial port method of the same name.

Parameters:

num The number to be written

Reimplemented from **uart** (p.91).

B.9.3.8 void avr_9xstream::write_hex (unsigned char num)

This method writes a character to the serial port as a text string showing the 8-bit unsigned number in that character in hexadecimal form.

Parameters:

num The 8-bit number to be sent out

Reimplemented from **uart** (p.91).

B.9.3.9 void avr_9xstream::write (unsigned char num)

This method writes a character to the serial port as a text string showing the 8-bit unsigned number in that character.

Parameters:

num The 8-bit number to be sent out

Reimplemented from **uart** (p.92).

B.9.3.10 void avr_9xstream::write (char num)

This method writes a character to the serial port as a text string showing the 8-bit signed number in that character.

Parameters:

num The 8-bit number to be sent out

Reimplemented from **uart** (p.92).

B.9.3.11 void avr_9xstream::write_bin (unsigned int num)

This method writes an integer to the serial port as a text string showing the 16-bit unsigned number in that character in binary form.

Parameters:

num The 16-bit number to be sent out

Reimplemented from **uart** (p. 92).

B.9.3.12 void avr_9xstream::write_hex (unsigned int num)

This method writes an integer to the serial port as a text string showing the 16-bit unsigned number in that integer in hexadecimal notation.

Parameters:

num The 16-bit number to be sent out

Reimplemented from **uart** (p. 92).

B.9.3.13 void avr_9xstream::write (unsigned int num)

This method writes an integer to the serial port as a text string showing the 16-bit unsigned number in that integer.

Parameters:

num The 16-bit number to be sent out

Reimplemented from **uart** (p. 93).

B.9.3.14 void avr_9xstream::write (int num)

This method writes an integer to the serial port as a text string showing the 16-bit signed number in that integer.

Parameters:

num The 16-bit number to be sent out

Reimplemented from **uart** (p. 93).

B.9.3.15 void avr_9xstream::write_hex (unsigned long num)

This method writes a long integer to the serial port as a text string showing the 32-bit unsigned number in that long integer.

Parameters:

num The 32-bit number to be sent out

Reimplemented from **uart** (p. 93).

B.9.3.16 void avr_9xstream::write (unsigned long num)

This method writes an unsigned long integer to the serial port as a text string showing the 32-bit unsigned number in that long integer.

Parameters:

num The 32-bit number to be sent out

B.9.3.17 void avr_9xstream::write (long num)

This method writes a long integer to the serial port as a text string showing the 32-bit signed number in that long integer.

Parameters:

num The 32-bit number to be sent out

Reimplemented from **uart** (p. 94).

The documentation for this class was generated from the following files:

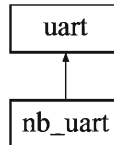
- Desktop/ASV/Code/**avr_9xstream.h**

- Desktop/ASV/Code/avr_9xstream.cpp

B.10 nb_uart Class Reference

```
#include <avr_serialnb.h>
```

Inheritance diagram for nb_uart::



Public Member Functions

- **nb_uart** (unsigned char)
- char **run** (void)
- void **puts_nb** (char *)

B.10.1 Detailed Description

File avr_serial_nb.h Inherited from avr_serial driver by JRR Expands functionality of class avr_serial to make the puts function non_blocking By Samuel Hoffman 6-3-07

B.10.2 Constructor & Destructor Documentation

B.10.2.1 nb_uart::nb_uart (unsigned char divisor)

Constructor no setup except starting the serial port using original avr_serial class object "uart"

B.10.3 Member Function Documentation

B.10.3.1 char nb_uart::run (void)

run service function, must be called with no particular time constraints until string has been sent returns 0 when still string to be sent, function requires the null (0x00) at the

end of string

*B.10.3.2 void nb_uart::puts_nb (char * string_data2)*

function call to send string of data over the serial port in a non blocking way string must be followed by the null character (0x00)

The documentation for this class was generated from the following files:

- Desktop/ASV/Code/avr_serialnb.h
- Desktop/ASV/Code/avr_serialnb.cpp

B.11 *packit_uart* Class Reference

```
#include <packit.h>
```

Public Member Functions

- **packit_uart** (**avr_9xstream** *, unsigned char)
- void **packit_clear** (void)

B.11.1 *Detailed Description*

Not tested or operational

B.11.2 *Constructor & Destructor Documentation*

B.11.2.1 *packit_uart::packit_uart* (**avr_9xstream** * ser_ptr2, unsigned char dev_addr)

Constructor for child class that uses the serial port -This class uses the usart to transmit data packets with its own checksum -Can't transmit at the same time as other micro-controllers, not tested

B.11.3 *Member Function Documentation*

B.11.3.1 *void packit_uart::packit_clear* (void)

This function clears the packit job list to stop it from sending data, use this function call, to clear all command and data jobs.

The documentation for this class was generated from the following files:

- Desktop/ASV/Code/**packit.h**
- Desktop/ASV/Code/packit.cpp

B.12 *spi_bb_port* Class Reference

Define this to enable debugging features.

```
#include <avr_spi_bb.h>
```

Public Member Functions

- **spi_bb_port** (volatile unsigned char *, volatile unsigned char *, volatile unsigned char *, char, char, char)
- void **add_slave** (char, char)
- char **transfer_bytes** (char *, char, char)

B.12.1 *Detailed Description*

Define this to enable debugging features.

This class holds the parameters and methods necessary to operate a bit-banged SPI port. The parameters include the addresses of the input, output, and data direction registers used as well as bitmasks that allow manipulation of the I/O pins which are used to communicate with the SPI chip(s) to which the bit-banged SPI port is attached.

B.12.2 *Constructor & Destructor Documentation*

*B.12.2.1 spi_bb_port::spi_bb_port (volatile unsigned char * inpt, volatile unsigned char * outpt, volatile unsigned char * ddrpt, char sck_msk, char mosi_msk, char miso_msk)*

This constructor sets up a bit-banged SPI port. Such a port uses regular I/O pins, manipulated by software, to communicate with SPI peripherals. Each port needs the pins SCK, MOSI, and MISO (clock, master out, master in) as well as one or more slave select pins, which are configured in **add_slave()** (p. 86).

Parameters:

`inpt` The address of the input port, such as `&PIND`

`outpt` Address of the output port, as `&PORTD` (must be same port as `inpt`)

`ddrpt` The address of the data direction register, such as `&DDRD`

`sck_msk` Bitmask for the SCK serial clock pin

`miso_msk` Bitmask for the MISO data pin

`mosi_msk` Bitmask for the MOSI data pin

B.12.3 Member Function Documentation

B.12.3.1 void spi_bb_port::add_slave (char ss_msk, char s_num)

This method adds an SPI slave to a bit-banged SPI port. It does so by adding an entry into the array of slave masks. This entry should be one unique bitmask that identifies one pin connected to the I/O port which is used for the other SPI pins (SCK, MISO, and MOSI). That pin is connected to the CS' (or SS') pin of the slave chip.

Parameters:

`aport` The bit-banged port data structure

`ss_msk` The mask for the Slave Select (aka Chip Select) pin on slave chip

`s_num` The number of the slave whose bitmask is to be set

*B.12.3.2 char spi_bb_port::transfer_bytes (char * bytes, char size, char slave_num)*

This method transfers bytes to and from a chip which is attached to a bit-banged SPI port. The bytes in the given array are sent to the receiving chip, and at the same time bytes are received from the other chip. The received bytes are put into the array which held the bytes that were sent out.

Parameters:

bytes A pointer to an array holding bytes sent to and received from the device

size The number of bytes to be sent and received

slave_num Which slave is to be selected and data exchanged with

Returns:

A result code - zero for success, nonzero for failure such as timeout

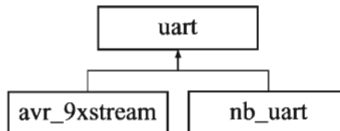
The documentation for this class was generated from the following files:

- Desktop/ASV/Code/**avr_spi_bb.h**
- Desktop/ASV/Code/avr_spi_bb.cpp

B.13 *uart Class Reference*

```
#include <avr_serial.h>
```

Inheritance diagram for `uart`:



Public Member Functions

- **uart** (unsigned char)
- char **putchar** (char)
- void **puts** (char *)
- char **check_for_char** (void)
- char **getchar** (void)
- char **transmitter_empty** (void)
- void **write_bin** (unsigned char)
- void **write_hex** (unsigned char)
- void **write** (unsigned char)
- void **write** (char num)
- void **write_bin** (unsigned int)
- void **write_hex** (unsigned int)
- void **write** (unsigned int)
- void **write** (int)
- void **write_hex** (unsigned long)
- void **write** (long)

B.13.1 Detailed Description

This class controls a UART (Universal Asynchronous Receiver Transmitter), a common serial interface. It talks to old-style RS232 serial ports (through a voltage converter chip such as a MAX232) or through a USB to serial converter such as a FT232RL chip. The UART is also sometimes used to communicate directly with other microcontrollers, sensors, or wireless modems.

This class has originally been written for AVR processors which have only one UART, but it should be extendable for use with processors which have dual UARTs.

B.13.2 Constructor & Destructor Documentation

B.13.2.1 `uart::uart (unsigned char divisor)`

This method sets up the AVR UART for communications. It enables the appropriate inputs and outputs and sets the baud rate divisor.

Parameters:

`divisor` The baud rate divisor to be used for controlling the rate of communication.

See the *.h file in which various values of the divisor are defined as macros.

B.13.3 Member Function Documentation

B.13.3.1 `char uart::putchar (char chout)`

This method sends one character to the serial port. It waits until the port is ready, so it can hold up the system for a while. It times out if it waits too long to send the character; you can check the return value to see if the character was successfully sent, or just cross your fingers and ignore the return value. Note: It's possible that at slower baud rates and/or higher processor speeds, this routine might time out even when the port is working fine. A solution would be to change the count variable to an integer and use a larger starting number. Note 2: Fixed! The count is now an integer and it works

at lower baud rates.

Parameters:

`chout` The character to be sent out

Returns:

0 if everything was OK and `(char)(-1)` if there was a timeout

Reimplemented in **avr_9xstream** (p. 77).

*B.13.3.2 void uart::puts (char * str)*

This is the usual...it just writes all the characters in a string until it gets to the `'\0'` at the end. Warning: By repeatedly calling **putchar()** (p. 89), this method can hold up the program while it's running, and so it shouldn't be used when the program has to meet timing constraints.

Parameters:

`str` The string to be written

Reimplemented in **avr_9xstream** (p. 77).

B.13.3.3 char uart::check_for_char (void)

This function checks if there is a character in the serial port's receiver buffer. It returns 1 if there's a character available, and 0 if not.

Returns:

1 for character available, 0 for no character available

B.13.3.4 char uart::getchar (void)

This method gets one character from the serial port, if one is there. If not, it waits until there is a character available. This can sometimes take a long time (even forever),

so use this function carefully. It's generally safer to use `check_for_char()` (p. 90) to ensure that there's data available first.

Returns:

The character which was found in the serial port receive buffer

B.13.3.5 char uart::transmitter_empty (void)

This function checks if the serial port transmitter is ready to send data. It simply tests the bit in the serial port status register which indicates that the transmitter buffer is empty.

Returns:

0 if the transmitter is empty and ready to send, and 1 if not

B.13.3.6 void uart::write_bin (unsigned char num)

This method writes a character to the serial port as a text string showing the 8-bit unsigned number in that character in binary form.

Parameters:

num The 8-bit number to be sent out

Reimplemented in `avr_9xstream` (p. 77).

B.13.3.7 void uart::write_hex (unsigned char num)

This method writes a character to the serial port as a text string showing the 8-bit unsigned number in that character in hexadecimal form.

Parameters:

num The 8-bit number to be sent out

Reimplemented in `avr_9xstream` (p. 78).

B.13.3.8 void uart::write (unsigned char num)

This method writes a character to the serial port as a text string showing the 8-bit unsigned number in that character.

Parameters:

num The 8-bit number to be sent out

Reimplemented in **avr_9xstream** (p. 78).

B.13.3.9 void uart::write (char num)

This method writes a character to the serial port as a text string showing the 8-bit signed number in that character.

Parameters:

num The 8-bit number to be sent out

Reimplemented in **avr_9xstream** (p. 78).

B.13.3.10 void uart::write_bin (unsigned int num)

This method writes an integer to the serial port as a text string showing the 16-bit unsigned number in that character in binary form.

Parameters:

num The 16-bit number to be sent out

Reimplemented in **avr_9xstream** (p. 79).

B.13.3.11 void uart::write_hex (unsigned int num)

This method writes an integer to the serial port as a text string showing the 16-bit unsigned number in that integer in hexadecimal notation.

Parameters:

num The 16-bit number to be sent out

Reimplemented in **avr_9xstream** (p. 79).

B.13.3.12 void uart::write (unsigned int num)

This method writes an integer to the serial port as a text string showing the 16-bit unsigned number in that integer.

Parameters:

num The 16-bit number to be sent out

Reimplemented in **avr_9xstream** (p. 79).

B.13.3.13 void uart::write (int num)

This method writes an integer to the serial port as a text string showing the 16-bit signed number in that integer.

Parameters:

num The 16-bit number to be sent out

Reimplemented in **avr_9xstream** (p. 79).

B.13.3.14 void uart::write_hex (unsigned long num)

This method writes a long integer to the serial port as a text string showing the 32-bit unsigned number in that long integer.

Parameters:

num The 32-bit number to be sent out

Reimplemented in **avr_9xstream** (p. 80).

B.13.3.15 void uart::write (long num)

This method writes a long integer to the serial port as a text string showing the 32-bit signed number in that long integer.

Parameters:

num The 32-bit number to be sent out

Reimplemented in **avr_9xstream** (p. 80).

The documentation for this class was generated from the following files:

- Desktop/ASV/Code/**avr_serial.h**
- Desktop/ASV/Code/**avr_serial.cpp**

B.14 ASV File Documentation

B.15 Desktop/ASV/Code/ASV.cpp File Reference

Main run file for all MCU devices.

```
#include <stdlib.h>
#include <avr/io.h>
#include <stdint.h>
#include <avr/interrupt.h>
#include "avr_serial.h"
#include "avr_serialnb.h"
#include "avr_9xstream.h"
#include "avr_spi_bb.h"
#include "nRF24L01.h"
#include "packit.h"
#include "avr_a2d.h"
#include "packed_arrays.h"
#include "interval.h"
#include "synch_data.h"
```

Defines

- **#define MY_DIVISOR 26**
- **#define _SLAVE_1**
- **#define SLAVE_DATA_FREQ -4**

Functions

- **int main (void)**
- **ISR (TIM1_COMP_VECT)**

Variables

- unsigned int **data_counter**
- synch * **synch_ptr**
- char **check_saturation**

B.15.1 Detailed Description

Main run file for all MCU devices.

Revisions:

- 10-13-07 SCH Original file
- 02-15-08 SCH Revised for System 2

This File using defines can be compiled to give three different program codes for the different MCUs

License: This file is released under the Lesser GNU Public License.

B.15.2 Define Documentation

B.15.2.1 #define _SLAVE_1

Device Selection Pick Slave 1, Slave 2, or Master to program each device

Master is not used in System 2 while both slaves are

B.15.2.2 #define MY_DIVISOR 26

This is the baud rate divisor for the UART. Values which have worked: 26: 9600 baud, 4 MHz crystal osc., works with FT232RL RS232-USB chip

B.15.2.3 #define SLAVE_DATA_FREQ -4

For packit class data label

B.15.3 Function Documentation

B.15.3.1 ISR (TIM1_COMP_VECT)

The interrupt routine for taking data Frequency of interval is controlled by interval class

B.15.3.2 int main (void)

Main Run Function for Slave 1

This is where the preset frequencies need to be set

Where saturation is checked for in the data, if found will terminate data taking

B.15.4 Variable Documentation

*B.15.4.1 char **check_saturation***

Variable to allow **ISR()** (p.97) to tell **main()** (p.97) to check for saturation on the readings that were just taken

*B.15.4.2 unsigned int **data_counter***

Variable to hold how many data points have been taken

*B.15.4.3 **synch*** **synch_ptr***

Code needed for synch class to work

B.16 Desktop/ASV/Code/avr_9xstream.h File Reference

Classes

- class **avr_9xstream**

B.16.1 Detailed Description

This file contains a class which extends the AVR serial port object to operate a MaxStream 9XStream(tm) radio modem connected to the serial port.

Revised:

- 07-19-07 JRR Created this file

B.17 Desktop/ASV/Code/avr_a2d.h File Reference

This file is the header for using the A/D converters on an AVR processor. It is designed to contain code which can be used on several processor versions:.

Functions

- void **A2D_init_default** (void)
- short int **A2D_read_once** (unsigned char channel)
- short int **A2D_read_oversampled** (unsigned char channel, unsigned char samples)
- void **A2D_off** (void)

B.17.1 Detailed Description

This file is the header for using the A/D converters on an AVR processor. It is designed to contain code which can be used on several processor versions:.

- ATmega8535 (not supported yet)
- ATmega8 (tested and works)
- ATmega32 (tested and works)
- ATmega644 (working on it – using 645 for compatibility)

Not all versions are supported yet.

Revisions:

- 06-18-06 JRR Original program
- 03-02-07 JRR Compatibility code for mega644/645

This file released under the Lesser GNU Public License. The program is for educational use only.

B.17.2 Function Documentation

B.17.2.1 void A2D_init_default (void)

This function initializes the A/D converter for single readings.

Power on off pin for turning on resistive load for measuring

B.17.2.2 void A2D_off (void)

This function turns off the A2D to conserve power It will have to be turned on again to get conversions

B.17.2.3 short int A2D_read_once (unsigned char channel)

This function takes one A/D reading from the given channel. It sets the A/D multiplexer, then reads that channel once and returns the result in an integer. The function waits for the A/D conversion to be complete before returning.

Parameters:

channel The A/D channel which is being read must be from 0 to 7

Returns:

The result of the A/D conversion, or -1 if there was a timeout

B.17.2.4 short int A2D_read_oversampled (unsigned char channel, unsigned char samples)

This function sets the A/D multiplexer to read from the given channel, then reads that channel the given number of times (up to a maximum of 32) and computes the average of the readings. This can help reduce noise. Note that there are many ways to digitally filter a signal; this is just one very crude, simple way.

Parameters:

channel The A/D channel which is being read must be from 0 to 7

samples The number of samples to be taken and averaged

Returns:

The averaged result of the A/D conversions, or -1 if a timeout occurred

B.18 Desktop/ASV/Code/avr_serial.h File Reference

Classes

- class **uart**

B.18.1 Detailed Description

This file contains functions which allow the use of a serial port on an AVR microcontroller. Compatibility macros are provided to isolate the names of various registers from the many specific AVR device types.

This code is designed to work for low-performance applications without requiring the use of interrupts. Interrupt based receiving code has not been completed or tested.

Revised:

- 04-03-06 JRR For updated version of compiler
- 06-10-06 JRR Ported from C++ to C for use with some C-only projects; also the serial_avr.h header has been stuffed with defines for compatibility among lots of AVR variants
- 08-11-06 JRR Some bug fixes
- 03-02-07 JRR Ported back to C++. I've had it with the limitations of C.

B.19 Desktop/ASV/Code/avr_spi_bb.h File Reference

Classes

- class **spi_bb_port**

Define this to enable debugging features.

B.19.1 Detailed Description

This file contains a class which allows the use of a bit-banged SPI port on an AVR microcontroller. This allows several SPI ports on one chip, none of which has to be shared with the SPI port which is used for in-system program downloading. Compatibility macros are provided to isolate the names of the various registers from the many specific AVR device types.

This code is designed to work for low-performance applications without requiring the use of interrupts. Interrupt based SPI port code has not been completed or tested.

Revisions:

- 03-23-07 JRR Original file
- 04-23-07 MNL Added functions to get I/O ports from the SPI_BB_PORT object

B.20 Desktop/ASV/Code/interval.cpp File Reference

```
#include <stdlib.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "avr_serial.h"
#include "avr_serialnb.h"
#include "avr_9xstream.h"
#include "interval.h"
```

Functions

- void **interval_setup** (int freq)
- unsigned int **interval_timer_cycles** (void)
- unsigned int **interval_timer_prescale** (void)
- void **interval_interrupt_on** (void)
- void **interval_interrupt_off** (void)
- void **print_freq** (int freq, **avr_9xstream** *ser_ptr)
- bool **interval_interrupt_status** (void)

B.20.1 Detailed Description

This file sets up the timer compare routine to take A2D data at certain intervals and packs it into an array external array

Revisions:

- 7-13-07 SCH initial file

License This file released under the Lesser GNU Public License. Use it as you like; the author(s) cannot be responsible for any use to which this file is put by others.

B.20.2 Function Documentation

B.20.2.1 `void interval_interrupt_off (void)`

This function disable the interrupt on the output compare

B.20.2.2 `void interval_interrupt_on (void)`

This function enables the interrupt on the output compare

B.20.2.3 `bool interval_interrupt_status (void)`

This function checks to see if the interval interrupt is on returns false if it is on; and true if it is off

B.20.2.4 `void interval_setup (int freq)`

This function setups up the interval frequency and time on timer 0 It doesn't enable the interrupt

B.20.2.5 `unsigned int interval_timer_cycles (void)`

This function returns the output compare value in which the frequency can be used to calculate the actual freq It would be nice to actually print floats out on the serial port

B.20.2.6 `unsigned int interval_timer_prescale (void)`

This function returns prescale value of timer 1

B.20.2.7 `void print_freq (int freq, avr_ostream * ser_ptr)`

This function prints the freq to the serial port

B.21 Desktop/ASV/Code/interval.h File Reference

Functions

- void **interval_setup** (int freq)
- unsigned int **interval_timer_cycles** (void)
- unsigned int **interval_timer_prescale** (void)
- void **interval_interrupt_on** (void)
- void **interval_interrupt_off** (void)
- void **print_freq** (int freq, **avr_9xstream ***)
- bool **interval_interrupt_status** (void)

B.21.1 Detailed Description

This file contains classes which set allow data to be taken at certain intervals using an interrupt service routine. These set of functions merely set up Timer0 for taking data.

Revisions:

- 07-13-07 SCH initial file
- 10-30-07 SCH made to run on 16 bit timer

License This file released under the Lesser GNU Public License. This program is for educational use only.

B.21.2 Function Documentation

B.21.2.1 void interval_interrupt_off (void)

This function disable the interrupt on the output compare

B.21.2.2 void interval_interrupt_on (void)

This function enables the interrupt on the output compare

B.21.2.3 bool interval_interrupt_status (void)

This function checks to see if the interval interrupt is on returns false if it is on; and true if it is off

B.21.2.4 void interval_setup (int freq)

This function setups up the interval frequency and time on timer 0 It doesn't enable the interrupt

B.21.2.5 unsigned int interval_timer_cycles (void)

This function returns the output compare value in which the frequency can be used to calculate the actual freq It would be nice to actually print floats out on the serial port

B.21.2.6 unsigned int interval_timer_prescale (void)

This function returns prescale value of timer 1

*B.21.2.7 void print_freq (int freq, **avr_9xstream** * ser_ptr)*

This function prints the freq to the serial port

B.22 Desktop/ASV/Code/nRF24L01.h File Reference

Defines

- #define `MAX_PACKET_SIZE` 32

B.22.1 Detailed Description

This file contains a class that interfaces to the nRF24L01 radio module. It uses the bit banded SPI class to simplify communications and to encapsulate the objects. It also uses a serial port for debugging information. This will likely to stay in future revisions unless there is an absolute need to remove it from memory for some reason. The definitions for the MCU's pins and ports connected to the radio are mostly through the SPI port with a few others set through here. See the constructor for more details.

Revisions:

- 04-22-07 MNL Original file
- 04-26-07 MNL Added a bunch of functions. Transfer verified working
- 05-09-07 MNL Added virtual interrupt handler
- 05-14-07 MNL Removed virtual interrupt handler and implemented setup re-transmit function

B.22.2 Define Documentation

B.22.2.1 #define MAX_PACKET_SIZE 32

These defines are macros for the various settings and such relevant to the radio unit. This allows a transparency for the user so they don't have to enter binary numbers all the time, and the code doesn't have to waste precious processing power trying to decipher what the heck it is the user wants.

B.23 Desktop/ASV/Code/packed_arrays.cpp File Reference

```
#include <stdlib.h>
#include "packed_arrays.h"
```

B.23.1 Detailed Description

This file contains classes which efficiently maintain arrays of data which aren't of the standard sizes (8, 16, or 32 bits).

Supported formats:

- 10-bit unsigned integers stored in arrays of bytes

Revisions:

- 07-08-07 JRR Original file stores 10-bit A/D conversion results

License This file released under the Lesser GNU Public License. Use it as you like; the author(s) cannot be responsible for any use to which this file is put by others.

B.24 Desktop/ASV/Code/packed_arrays.h File Reference

Classes

- class **array10bits**

B.24.1 Detailed Description

This file contains classes which efficiently maintain arrays of data which aren't of the standard sizes (8, 16, or 32 bits).

Revisions:

- 07-08-07 JRR Original of this file
- 10-15-07 SCH Modified to add function to return a pointer to the data

License This file released under the Lesser GNU Public License. This program is for educational use only.

B.25 Desktop/ASV/Code/packit.h File Reference

Classes

- class **packit_uart**

B.25.1 Detailed Description

This file contains functions which allow the user to transfer large amounts of data over a serial port or radio link.

This code is designed to work for low-performance applications without requiring the use of interrupts. Interrupt based receiving code has not been completed or tested.

B.25.2 Desktop/ASV/Code/synch_data.h File Reference

Files used to synchronize one micorcontrollers data taking with another.

Defines

- #define **SY_PUTS(y)**

B.25.2.1 Detailed Description

Files used to synchronize one micorcontrollers data taking with another.

This Method Relies on the Input Capture Pin of the Microcontroller for timing It uses 1 wire connected between microcontrollers to pulse the input capture pin

Revisions:

- 10-21-07 SCH Initial file created

License: This file is released under the Lesser GNU Public License.

B.25.2.2 Define Documentation

#define SY_PUTS(y) Used to turn on or off debugging mode define to turn on and undef to turn off

C. MATLAB CODE

This section displays Matlab code written for this project that can be found on the supplemental compact-disc (CD) under “Matlab”.

List of files included on CD:

- Matlab

- FFT_Approach
 - * FFT2DOF.m
 - * ridepitch.mdl
 - * ModeFinder_Lagrange.m
- Grey_Box_Approach
 - * car2dof.m
 - * integrate_data.mdl
 - * state_space.m
 - * use_real_data.m
 - * ridepitch_ss3.mdl

C.1 FFT approach

The fft approach was investigated using Matlab and Simulink. The FFT was performed on data created in Matlab. The first file is FFT2DOF which performs the FFT on data created with a Simulink simulation “ridepitch.mdl” shown in figure C.1. The second file ‘ModeFinder_Lagrange’ solves the eigenvalue problem for the 2 degree of freedom model from chapter 3.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%FFT2DOF.m
%Matlab file
%This code performs and FFT on the data created from the ridepitch
%simulink model
%The ride pitch uses simulates a two degree of freedom car and is where all
%car parameters are inputted
%The fft is then plotted

clear      %Clears the workspace
sim('ridepitch') %Runs the simulation

timestep=tout(2)-tout(1); %finds the time step of the data
timedivider=20; %decreases the number of points sent to the FFT
FFT_Freq=1/(timestep*timedivider); %Finds new freq of data
Data_Freq=1/timestep; %Finds original freq of data
Graph_Start=1; %Hz start value
Graph_Range=2; %Hz high value
Range_Value=Graph_Range/FFT_Freq; %Ratio to know how many terms of FFT to look at
Range_Start=Graph_Start/FFT_Freq; %Ratio to know how many terms of
                                %The FFT ranges from 0 to .5*FFT Frequency,
                                %only half the terms are needed since
                                %they are repeated

t_i=.1; %Start Time
t_f=20; %End time

front=Car(Data_Freq*t_i:Data_Freq*t_f,1); %grab data from the simulation
s=size(front)/timedivider;
%Loop to build the data at the FFT_freq rather then the Data_Freq
for n=1:s
    front2(n)=front(n*timedivider);
end

k=100000; %number of FFT points
F = fft(front2,k); %Fast fourier transform with k terms

f = FFT_Freq*(k*Range_Start:k*Range_Value)/k; %frequency array

%Finds the power by squaring the absolute value
magF = abs(F(k*Range_Start+1:k*Range_Value+1))./f.^2;
phF = unwrap(angle(F(k*Range_Start+1:k*Range_Value+1)));
```

```

rear=Car(Data_Freq*t_i:Data_Freq*t_f,4); %grab data from sim

for n=1:s %Loop to build the data at the FFT_freq rather than the Data_Freq
    rear2(n)=rear(n*timedivider);
end

R = fft(rear2,k); %fft for rear
magR = abs(R(k*Range_Start+1:k*Range_Value+1))./f.^2;% R.* conj(R) / k; %power for rear
pnR = unwrap(angle(R(k*Range_Start+1:k*Range_Value+1)));

phase_diff=(phF-phR)./3.14159;
figure(1);
%subplot(2,1,1);
plot(f,magF,'-',f,magR,'--','linewidth',3); %plots power and freq in the range wanted
title('Frequency content of Car','fontsize',14)
xlabel('frequency (Hz)','fontsize',14)
ylabel('magnitude of freq','fontsize',14)
legend('Front','Rear','fontsize',14)

subplot(2,1,2);
plot(f,phase_diff);%plots power and freq in the range wanted
title('Frequency content of Car')
xlabel('frequency (Hz)')
ylabel('phase difference (front-rear (rads))')

%End Matlab File
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

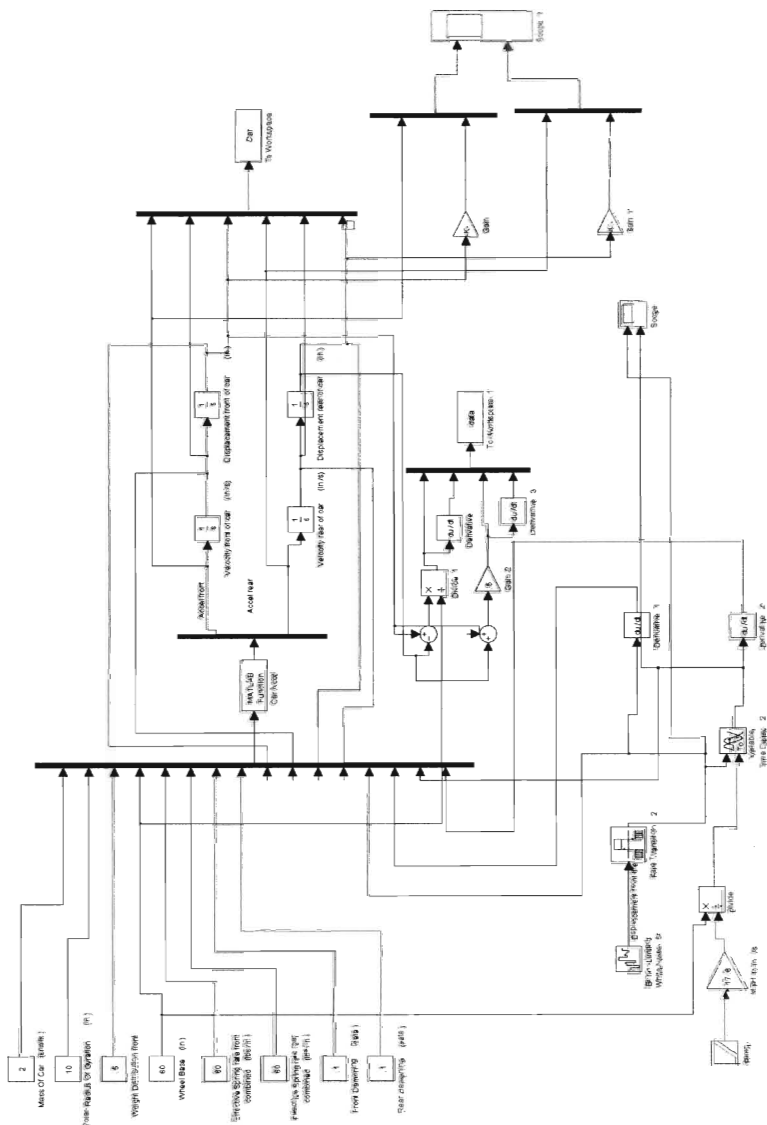


Fig. C.1: 2 Degree of freedom simulink model: "ridepitch.mdl" used with FFT2DOF.m It simulates the 2 degree of freedom car from 3 with a band limited white noise input. This is random noise that should contain all frequencies equally up to the user specified band limited frequency.

```

%File: ModeFinder_Lagrange.m
%Solves the eigenvalue problem for a 2 degree of freedom halfcar
%Finds natural frequencies, damped natural frequencies and the damping
%ratio for both vehicle modes
%Equations of motion are derived using a Lagrange energy method

format short

m=2;% mass of car (snails)
k_g=20;% radius of gyration (in)
wdf=.4;% weight distribution front
l=60;% wheel base (in)
k_f=60;% effective spring rate front (both sides combined) (lb/in)
k_r=60;% effective spring rate rear (both sides combines) (lb/in)
cz_f=.1;% damping percentage front (zeta) (%)
cz_r=.1;% dampning percentage rear (zeat) (%)
x_f=0;% displacement front car (in)
x_fdot=0;% displacement velocity front car (in/s)
x_r=0;% displacement rear car(in)
x_rdot=0;% displacement velocity rear car (in/s)
z_f=0;% displacement front wheel (in)
z_fdot=0;% displacement velocity front wheel (in/s)
z_r=0;% displacement rear wheel(in)
z_rdot=0;% displacement velocity rear wheel (in/s)

l_r=l*wdf; %length from CG to rear
l_f=l-l_r; %length from CG to front
m_f=wdf*m; %mass on front of the car
m_r=m-m_f; %mass on rear of the car

c_f=2*cz_f*sqrt(k_f*m_f); %damping on the front
c_r=2*cz_r*sqrt(k_r*m_r); %damping on the rear

i=m*k_g^2;

mm=[i/l^2+m/4,m/4-i/l^2;m/4-i/l^2,i/l^2+m/4];

cc=[c_f,0;0,c_r];

%kk=[(k_r+k_f)/4+(l_f*k_f+l_r*k_r)/l^2,(k_r+k_f)/4-(l_f*k_f+l_r*k_r)/l^2;
%      (k_r+k_f)/4-(l_f*k_f+l_r*k_r)/l^2,(k_r+k_f)/4+(l_f*k_f+l_r*k_r)/l^2];
kk=[k_f,0;0,k_r];

[v,d]=eig(kk,mm);

%Normalizing
v(1:2,1)=v(1:2,1)/sqrt(v(1,1)^2+v(2,1)^2);
v(1:2,2)=v(1:2,2)/sqrt(v(1,2)^2+v(2,2)^2);
v=-1*v;
M=v'*mm*v;
K=v'*kk*v;
C=v'*cc*v;

%Normalizing
K(2,2)=K(2,2)/M(2,2);
C(2,2)=C(2,2)/M(2,2);
M(2,2)=1;

```

```

z1=C(1,1)/(2*sqrt(K(1,1)));
z2=C(2,2)/(2*sqrt(K(2,2)));

%New seperate EOM
disp('seperate DOF EOM')
M=M
C=C
K=K

%Frequencies
w1=sqrt(K(1,1));
w2=sqrt(K(2,2));
wd1=sqrt(1-z1^2)*w1;
wd2=sqrt(1-z2^2)*w2;

f1=w1/(2*pi());
fd1=wd1/(2*pi());
f2=w2/(2*pi());
fd2=wd2/(2*pi());
%-----
%
%Complex modes

%Sets up state space eigenvalue problem
A=[cc,mm;mm,zeros(2,2)];
B=[kk,zeros(2,2);zeros(2,2),-mm];
F=[zeros(4,1)];
[w,e]=eig(B,-A);

%Using code from example problem in Fundamentals of Structural Dynamics
alp=real(diag(e));
bet=imag(diag(e));
omega=sqrt(alp.*alp+bet.*bet); %Finds magnitude of natural freq
zeta=-alp./omega;

disp('Angle difference of first mode (real and angle)')
natural_freq=omega(3)/2/pi
damping_zeta=zeta(3)
damped_freq=omega(3)/2/pi*sqrt(1-zeta(3))^2
dof1=w(1,3)/w(1,3)
dof2=w(2,3)/w(1,3)
andlediff=angle(dof1)-angle(dof2)

disp('Angle difference of 2nd mode (real and angle)')
natural_freq=omega(1)/2/pi
damping_zeta=zeta(1)
damped_freq=omega(1)/2/pi*sqrt(1-zeta(1))^2
dof1=w(1,1)/w(1,1)
dof2=w(2,1)/w(1,1)
andlediff=angle(dof1)-angle(dof2)
%End ModeFinder_Lagrange.m

```

C.2 Grey Box Theory Matlab Code

To use the following code requires Matlab System Identification Tool Box. All files found on CD under 'Grey_Box_Approach' are required to run this code. Data needs to be imported into Matlab using a text file of the following form arranged in columns. The columns contained data point number, front suspension accelerometer reading, front Chassis accelerometer reading, rear suspension accelerometer reading, and rear chassis accelerometer reading from left to right.

```
%Use_real_data.m

%By Samuel Hoffman
%Takes data from workspace and prepares it for grey box identification
%Uses integrate_data.mdl to integrate the data
%Uses car2dof.m for grey box 'idgrey' function
%Uses state_space.m in conjunction with ridepitch_ss3.mdl to compare to
%actual data

%Puts results of grey box estimation in structures: 'actual' and 'model' on
%the workspace

%=====
%=====
%Converts the data to in/s^2 and subtracts static value
actual.real_data=CS; % The Data
actual.sample_frequency=800; % Data Property
actual.sample_period=1/actual.sample_frequency;

simul_time=800/actual.sample_frequency;

actual.time=(actual.real_data(:,1)-1)/actual.sample_frequency;

%Constant, includes accel due to gravity in inches
actual.scaling=32.2*2*12/1023;

%Front Suspension (FS)
actual.real_data(:,2) = (actual.real_data(:,2)-569.5)*7.87*actual.scaling;

%Front Chassis (FC)
actual.real_data(:,3) = (actual.real_data(:,3)-641)*3.27*actual.scaling;

%Rear Suspension (RS)
actual.real_data(:,4) = (actual.real_data(:,4)-581.5)*7.93*actual.scaling;

%Rear Chassis (RC)
actual.real_data(:,5) = (actual.real_data(:,5)-636)*3.25*actual.scaling;
```

```

% Following Code Subtracts the linear trend from the data
%actual.real_data(:,2) = detrend(actual.real_data(:,2)); %Front Suspension (FS)
%actual.real_data(:,3) = detrend(actual.real_data(:,3)); %Front Chassis
(FC)
%actual.real_data(:,4) = detrend(actual.real_data(:,4)); %Rear
Suspension (RS)
%actual.real_data(:,5) = detrend(actual.real_data(:,5)); %Rear
Chassis (RC)

=====
%
=====

%Adds time difference in measuring the points
front_car=[actual.time+.000675 actual.real_data(:,3)];
rear_car=[actual.time+.000675 actual.real_data(:,5)];
front_susp=[actual.time actual.real_data(:,2)];
rear_susp=[actual.time actual.real_data(:,4)];

% front_car(:,2)=detrend(front_car(:,2));
% rear_car(:,2)=detrend(rear_car(:,2));
% front_susp(:,2)=detrend(front_susp(:,2));
% rear_susp(:,2)=detrend(rear_susp(:,2));

=====
%
=====

%Runs Simulations
state_space;
clear y_m u_m y u;
sim('integrate_data');
sim('ridepitch_ss3');

=====
%
=====

%Plots the input output variables for the State Space Model
figure(1)

subplot(4,2,2);
plot(y(:,1),y(:,2),'b',y_m(:,1),y_m(:,2),'r');
ylabel('Pitch Angle(rad)');
xlabel('Time(s)');
legend('Actual','Model');
title('Output (Y)');

subplot(4,2,4);
plot(y(:,1),y(:,3),'b',y_m(:,1),y_m(:,3),'r');
ylabel('Pitch Angle Velocity(rad/s)');
xlabel('Time(s)');

subplot(4,2,6);
plot(y(:,1),y(:,4),'b',y_m(:,1),y_m(:,4),'r');
ylabel('CG Position(in)');
xlabel('Time(s)');

```

```

subplot(4,2,8);
plot(y(:,1),y(:,5),'b',y_m(:,1),y_m(:,5),'r');
ylabel('CG Velocity(in/s)');
xlabel('Time(s)');

subplot(4,2,1)
plot(y_m(:,1),u_m(:,1));
ylabel('Front Wheel Displacement (in)');
xlabel('Time (s)');
title('Input (u)');

subplot(4,2,3)
plot(y_m(:,1),u_m(:,2));
ylabel('Front Wheel Velocity (in/s)');
xlabel('Time (s)');

subplot(4,2,5)
plot(y_m(:,1),u_m(:,3));
ylabel('Rear Wheel Positon (in)');
xlabel('Time (s)');

subplot(4,2,7)
plot(y_m(:,1),u_m(:,4));
ylabel('Rear Wheel Velocity (in/s)');
xlabel('Time (s)');
%=====
%=====
model.timestep=y_m(2,1) - y_m(1,1); %finds the time step of the data
model.timedivider=20; %decreases the number of points sent to the FFT

model.s=size(y_m)/model.timedivider;

%if(mdlp.s(1) > 800)
%   mdlp.s(1)=800;
%end;
clear yy uu;
%Loop to build the data at the FFT_freq rather than the Data_Freq
for n=1:model.s(1)
    for c=1:4
        yy(n,c)=y_m(n*model.timedivider,c+1);
        uu(n,c)=u_m(n*model.timedivider,c);
    end
end

model.Ts=model.timestep*model.timedivider;
data = iddata(yy,uu,model.Ts); %model input data
%data.int= {'foh','foh','foh','foh'};

data2=iddata(y(:,2:5),u,(y(10,1)-y(1,1))/9); %measured data input
%data2.int= {'foh','foh','foh','foh'};

% % The idgrey object definition, used for estimation model and actual
% parameters
%Has the initial estimated parameters as second option in []'s, three parameters

```



```

%correspond to par(1), par(2), and par(3) respectively
mm = idgrey('car2dof',[10 14 800],'c',[]);
% %
%Estimating Parameters from state space, note state_space.m parameters have
%to agree with car2dof.m for correct estimation
mdl=pem(data,mm);

%Iterative function for Estimating parameters for actual data
act=pem(data2,mm);

%Outputs selected parameters on Matlab
model.k_f=mdl.b(4,1)*ss.m;
model.k_r=mdl.b(4,3)*ss.m;
model.c_f=mdl.b(4,2)*ss.m;
model.c_r=mdl.b(4,4)*ss.m;
model.cz_f=model.c_f/2/sqrt(model.k_f*ss.m_f);
model.cz_r=model.c_r/2/sqrt(model.k_r*ss.m_r);
model.I=model.k_f*ss.l_f/mdl.b(2,1);
model.k_g=sqrt(model.I/ss.m)

actual.k_f=act.b(4,1)*ss.m;
actual.k_r=act.b(4,3)*ss.m;
actual.c_f=act.b(4,2)*ss.m;
actual.c_r=act.b(4,4)*ss.m;
actual.cz_f=actual.c_f/2/sqrt(actual.k_f*ss.m_f);
actual.cz_r=actual.c_r/2/sqrt(actual.k_r*ss.m_r);
actual.I=actual.k_f*ss.l_f/act.b(2,1);
actual.k_g=sqrt(actual.I/ss.m)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% state_space.m
% By Samuel Hoffman
% Automotive Suspension Vibration Thesis Project
% Description: State space form of car parameters used in ridepitch_ss3
% state space block

ss.m=500/32.2/12;% mass of car (snails)
ss.k_g=22;% radius of gyration (in)
ss.wdf=.47;% weight distribution front
ss.l=68;% wheel base (in)
ss.k_f= 100;% effective spring rate front (both sides combined) (lb/in)
ss.k_r= 100;% effective spring rate rear (both sides combines) (lb/in)
ss.cz_f=.7182;% damping percentage front (zeta) (%)
ss.cz_r=.4919;% damping percentage rear (zeat) (%)
ss.x_f=0;% displacement front car (in)
ss.x_fdot=0;% displacement velocity front car (in/s)
ss.x_r=0;% displacement rear car(in)
ss.x_rdot=0;% displacement velocity rear car (in/s)
ss.z_f=0;% displacement front wheel (in)
ss.z_fdot=0;% displacement velocity front wheel (in/s)
ss.z_r=0;% displacement rear wheel(in)

```

```

ss.z_rdot=0;% displacement velocity rear wheel (in/s)

ss.l_r=ss.l*ss.wdf; %length from CG to rear
ss.l_f=ss.l-ss.l_r; %length from CG to front
ss.m_f=ss.wdf*ss.m; %mass on front of the car
ss.m_r=ss.m-ss.m_f; %mass on rear of the car

ss.c_f=7.2;%2*ss.cz_f*sqrt(ss.k_f+ss.m_f); %damping on the front
ss.c_r=10;%2*ss.cz_r*sqrt(ss.k_r+ss.m_r); %damping on the rear

ss.i=ss.m*ss.k_g^2;
ss.A = [0 1 0 0;
        -(ss.k_r*ss.l_r^2+ss.k_f*ss.l_f^2)/ss.i -(ss.c_r*ss.l_r^2+ss.c_f*ss.l_f^2)/ss.i
        (ss.k_r*ss.l_r-ss.k_f*ss.l_f)/ss.i (ss.c_r*ss.l_r-ss.c_f*ss.l_f)/ss.i;

        0 0 0 1;
        (ss.k_r*ss.l_r-ss.k_f*ss.l_f)/ss.m (ss.c_r*ss.l_r-ss.c_f*ss.l_f)/ss.m
        -(ss.k_r+ss.k_f)/ss.m -(ss.c_r+ss.c_f)/ss.m];

ss.B = [0 0 0 0;
        ss.l_f*ss.k_f/ss.i ss.l_f*ss.c_f/ss.i
        -ss.l_r*ss.k_r/ss.i -ss.l_r*ss.c_r/ss.i;
        0 0 0 0;
        ss.k_f/ss.m ss.c_f/ss.m ss.k_r/ss.m ss.c_r/ss.m];

ss.C = eye(4);

ss.D = zeros(4,4);

ss.K = zeros(4,4);

ss.x0 = zeros(4,1);

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

% car2dof.m
% By Samuel Hoffman
% Automotive Suspension Vibration Thesis Project
% Description: Defines the Grey Box model

```

```

function [A,B,C,D,K,x0] = car2dof(par,T,aux)

```

```

m=500/32.2/12;% mass of car (snails)
k_g=10;% radius of gyration (in)
wdf=.47;% weight distribution front
l=68;% wheel base (in)
k_f= 100;% effective spring rate front (both sides combined) (lb/in)
k_r= 100;% effective spring rate rear (both sides combines) (lb/in)
cz_f=0.1;% damping percentage front (zeta) (%)
cz_r=0.1;% damping percentage rear (zeat) (%)

```

```

x_f=0;% displacement front car (in)
x_fdot=0;% displacement velocity front car (in/s)
x_r=0;% displacement rear car(in)
x_rdot=0;% displacement velocity rear car (in/s)
z_f=0;% displacement front wheel (in)
z_fdot=0;% displacement velocity front wheel (in/s)
z_r=0;% displacement rear wheel(in)
z_rdot=0;% displacement velocity rear wheel (in/s)

l_r=l*wdf; %length from CG to rear
l_f=l-l_r; %length from CG to front
m_f=wdf*m; %mass on front of the car
m_r=m-m_f; %mass on rear of the car

c_f= par(1);%2*cz_f*sqrt(k_f*m_f);%par(2); %damping on the front
c_r= par(2);%2*cz_r*sqrt(k_r*m_r);%par(3); %damping on the rear

i= par(3);%m*k_g^2;
A = [0 1 0 0;
      -(k_r*l_r^2+k_f*l_f^2)/i -(c_r*l_r^2+c_f*l_f^2)/i
      (k_r*l_r-k_f*l_f)/i (c_r*l_r-c_f*l_f)/i;
      0 0 0 1;
      ((k_r*l_r/m)-(k_f*l_f/m)) ((c_r*l_r)-(c_f*l_f))/m
      -(k_r+k_f)/m -(c_r+c_f)/m];

B = [0 0 0 0;
      l_f*k_f/i l_f*c_f/i -l_r*k_r/i -l_r*c_r/i;
      0 0 0 0;
      k_f/m c_f/m k_r/m c_r/m];

C = eye(4);

D = zeros(4,4);

K = zeros(4,4);

x0 = zeros(4,1);

```

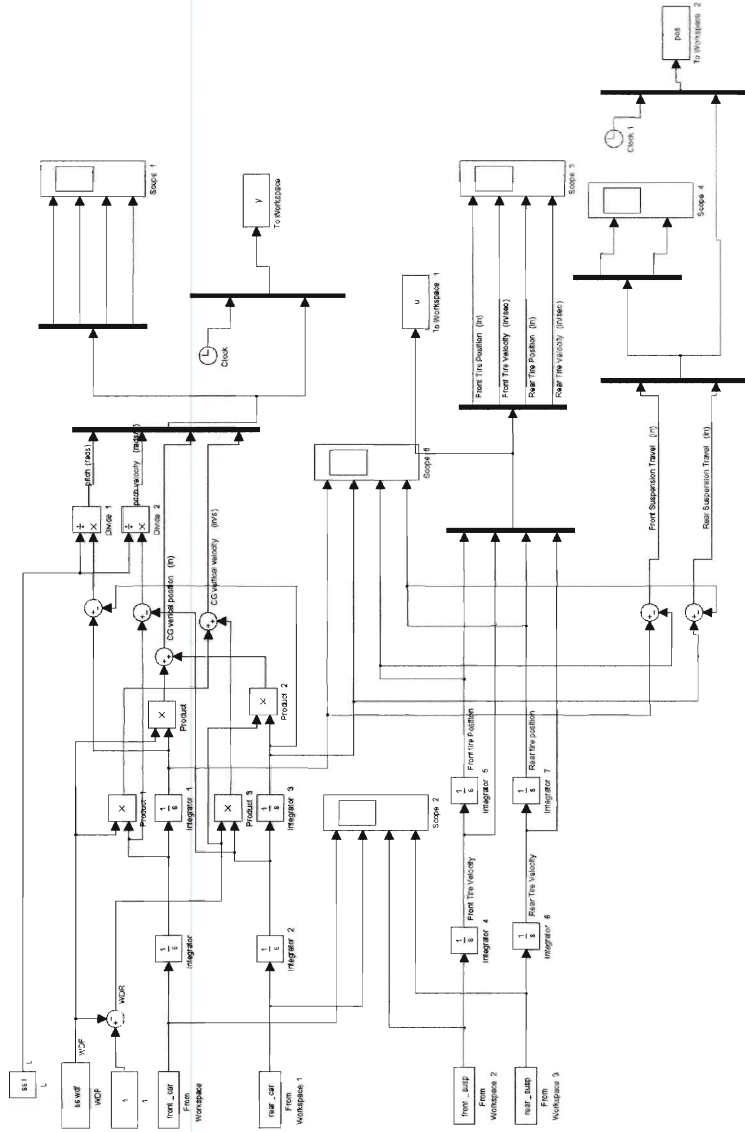


Fig. C.2: 'ridepitch.mdl' Simulink file for integrating raw accelerometer data into displacement and velocity. Creates the input output data specified in section 5.4.2

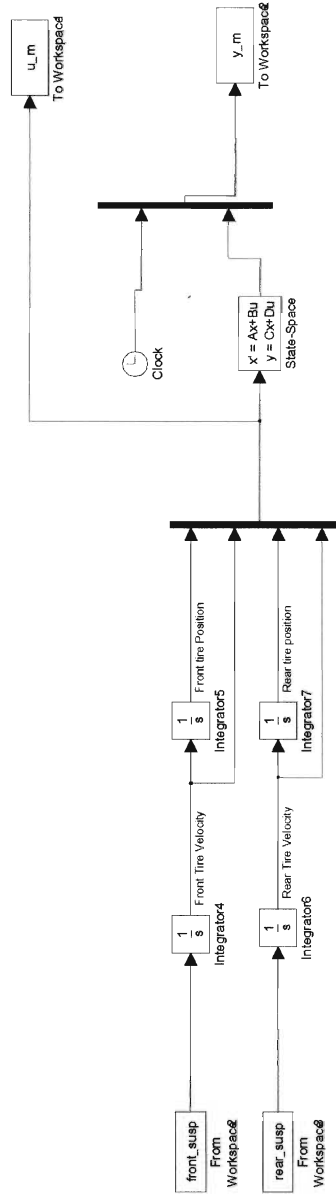


Fig. C.3: 'ridepitch_ss3' Simulink file for using actual input data to create a model output data. Uses 'state_space.m' for model structure