

---


Electronic Theses and Dissertations, 2004-2019

---

2008

## Bi-directional Dcm Dc-to-dc Converter For Hybrid Electric Vehicles

Michael Pepper  
*University of Central Florida*

 Part of the [Electrical and Electronics Commons](#)  
Find similar works at: <https://stars.library.ucf.edu/etd>  
University of Central Florida Libraries <http://library.ucf.edu>

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact [STARS@ucf.edu](mailto:STARS@ucf.edu).

---

### STARS Citation

Pepper, Michael, "Bi-directional Dcm Dc-to-dc Converter For Hybrid Electric Vehicles" (2008). *Electronic Theses and Dissertations, 2004-2019*. 3587.  
<https://stars.library.ucf.edu/etd/3587>

# **BI-DIRECTIONAL DCM DC-TO-DC CONVERTER FOR HYBRID ELECTRIC VEHICLES**

by

MICHAEL JAMES PEPPER  
B.S. University of Central Florida, 2004

A thesis submitted in partial fulfillment of the requirements  
for the degree of Master of Science in Electrical Engineering  
in the School of Electrical Engineering and Computer Science  
in the College of Engineering and Computer Science  
at the University of Central Florida  
Orlando, Florida

Spring Term  
2009

Major Professor: Issa Batarseh

© 2008 Michael James Pepper

## ABSTRACT

With the recent revival of the hybrid vehicle much advancement in power management has been made. The most popular hybrid vehicle, the hybrid electric vehicle, has many topologies developed to realize this hybrid vehicle. From these topologies, a sub set was created to define a particular group of vehicles where the converter discussed in this thesis has the most advantage. This sub set is defined by two electric sources of power coupled together at a common bus. This set up presents many unique operating conditions which can be handled seamlessly by the DC-to-DC converter when designed properly.

The DC-to-DC converter discussed in this thesis is operated in Discontinuous Conduction Mode (DCM) of operation because of its unique advantages over the Continuous Conduction Mode (CCM) operated converter. The most relevant being the reduction of size of the magnetic components such as inductor, capacitor and transformers. However, the DC-to-DC converter operated in DCM does not have the inherent capability of bi-directional power flow. This problem can be overcome with a unique digital control technique developed here. The control is developed in a hierarchical fashion to separate the functions required for this sub set of hybrid electric vehicle topologies. This layered approach for the controller allows for the seamless integration of this converter into the vehicle.

The first and lowest level of control includes a group of voltage and controller regulators. The average and small signal model of these controllers were developed here

to be stable and have a relatively fast recovery time to handle the transient dynamics of the vehicle system.

The second level of control commands and organizes the regulators from the first level of control to perform high level task that is more specific to the operation of the vehicle. This level of control is divided into three modes called hybrid boost, hybrid buck and electric vehicle mode. These modes are developed to handle the specific operating conditions found when the vehicle is operated in the specific mode.

The third level of control is used to command the second level of control and is left opened via a communication area network (CAN) bus controller. This level of control is intended to come from the vehicle's system controller.

Because the DC-to-DC converter is operated in DCM, this introduces added voltage ripple on the output voltage as well as higher current ripple demand from the input voltage. Since this is generally undesirable, the converter is split into three phases and properly interleaved. The interleaving operation is used to counteract the effects of the added voltage and current ripple.

Finally, a level of protection is added to protect the converter and surrounding components from harm. All protection is designed and implemented digitally in DSP.

*Remember*

*Even when you forget to teach by example...you are still doing it.*

## ACKNOWLEDGMENTS

I would like to thank all of the people and organizations who made this work possible. I would like to thank my advisor Dr. Issa Batarseh, for all of his support through out the project. This work was a collaborative effort by APECOR and the John Deere Company. I would especially like to thank all of the team members at APECOR for all of their hard work and long hours on this project: Rene Kersten, Khalid Rustom, Keith Mansfield, and Sean Elmes. I would also like to thank my early mentor in power electronics Dr. Hussam Al-Atrash for all of his guidance and inspiring discussions. I am also very grateful to my committee members for their feedback on this work, Drs. Issa Batarseh, John Shen, and Michael Haralombous.

I am very thankful to my parents, Jim and Barbara Pepper, for their unwavering support though out all of my educational career. I would also like to thank my girlfriend Amanda Felton for her support and understanding during this work.

*Michael Pepper*

*August 2008*

# TABLE OF CONTENTS

LIST OF FIGURES .....	x
LIST OF TABLES .....	xii
CHAPTER ONE: INTRODUCTION.....	1
Typical Topologies .....	3
Series Hybrid .....	4
Parallel Hybrid.....	5
Combined Hybrid.....	6
CHAPTER TWO: LITERATURE REVIEW .....	8
Battery Voltage .....	8
Non Idealities .....	9
The DC-to-DC Converter Solution.....	10
CHAPTER THREE: SYSTEM DESIGN.....	11
System Overview .....	11
Level 1 Control .....	15
Average Model.....	16
Verify Average Model .....	21
Small Signal Analysis.....	24
Level 2 Control .....	29
Modes of Operation .....	32
Interleaving.....	39
Protection .....	47
DCM Explained .....	49



Communication Interface (Level 3 Control).....	51
CANRX.....	53
Command values (CMD) – Mailbox 1 .....	53
Data values (DATA) – Mailbox 1 .....	53
Data values (DATA) – Mailbox 2 .....	54
CANTX.....	55
State of the Converter (STATUS) – Mailbox 3 .....	55
Data values (DATA) – Mailbox 3 .....	55
Data values (DATA) – Mailbox 4 .....	57
User Interface.....	59
CHAPTER FOUR: EXPERIMENTAL RESULTS.....	62
Converter Prototype.....	62
Experimental Results .....	64
CHAPTER FIVE: CONCLUSION.....	73
APPENDIX A: FLOW CHARTS.....	76
Main .....	78
Initialize DSP .....	79
Call ADC .....	80
Control Manager.....	81
Load Values from CAN.....	82
Transmit Values to CAN .....	83
Check for Faults.....	84
Fetch State.....	85

Standby Handler.....	86
Set Polarity.....	87
Reset Handler.....	88
Run Handler.....	89
APPENDIX B: DSP CODE.....	90
Settings.h.....	92
Sys_fun.h.....	93
JD_PowerUnit.h.....	96
Sys_fun.c.....	96
JD_PowerUnit.c.....	113
APPENDIX C: EQUATIONS.....	120
LIST OF REFERENCES.....	122

## LIST OF FIGURES

Figure 1	Definition of Hybrid Vehicle.....	1
Figure 2	Series Hybrid Topology <sup>1</sup> .....	4
Figure 3	Parallel Hybrid Topology <sup>1</sup> .....	6
Figure 4	Combined Hybrid Topology <sup>1</sup> .....	7
Figure 5	Power Flow of Series Hybrid Topology.....	13
Figure 6	Power Flow of Combined Hybrid Topology.....	14
Figure 7	First Level of Control .....	15
Figure 8	Boost Mode Inductor Current.....	17
Figure 9	Boost Converter.....	17
Figure 10	Boost Converter Mode I .....	18
Figure 11	Boost Converter Mode II.....	19
Figure 12	Boost Converter Mode III .....	20
Figure 13	Boost Converter Average Model.....	21
Figure 14	Boost Converter Switching Model .....	22
Figure 15	Boost Converter Average and Switching Model.....	23
Figure 16	Verify Average Model Simulation Output .....	23
Figure 17	Control Loop .....	27
Figure 18	Calculated Frequency Responses .....	28
Figure 19	Measured Frequency Response .....	29
Figure 20	Level 2 Control.....	30
Figure 21	Single Switching Leg .....	31
Figure 22	Boost Operated Switching Leg.....	31
Figure 23	Buck Operated Switching Leg.....	32
Figure 24	Hybrid Boost Vehicle Mode Flow Chart.....	34
Figure 25	Hybrid Buck Vehicle Mode Flow Chart .....	35
Figure 26	Electric Vehicle Mode Flow Chart.....	37
Figure 27	Modified Output Voltage Regulator Operation.....	38
Figure 28	Semikron Module .....	39
Figure 29	Three Paralleled Buck Converters.....	41
Figure 30	Three Interleaved Inductor Current Waveforms .....	42
Figure 31	Capacitor Current of Three Phase Interleaved Converters.....	43
Figure 32	PWM Architecture.....	44
Figure 33	Three Phase PWM Module Diagram .....	45
Figure 34	Three Interleaved PWM Ramp Signals.....	47
Figure 35	Buck Converter Critical Inductance Plot.....	50
Figure 36	Boost Converter Critical Inductance Plot.....	51
Figure 37	User Interface .....	60
Figure 38	User Interface with Test Running Capabilities .....	61
Figure 39	3D-Design of Power Filter Board.....	62
Figure 40	Power Filter Board Prototype.....	63
Figure 41	Final Prototype .....	64
Figure 42	Power Sweep of Buck and Boost Modes .....	65
Figure 43	Bus Voltage Regulation During Load Transient.....	66
Figure 44	Inductor Current Regulation During Load Steps.....	67

Figure 45 Inductor Current Regulation During Input Voltage Steps.....	68
Figure 46 Commanded Mode Change Buck to Boost .....	69
Figure 47 Hybrid Boost Start Up.....	70
Figure 48 Experimental Inductor Current Interleaving .....	71
Figure 49 Over Voltage Fault Shut Down .....	72

## LIST OF TABLES

Table 1 System Specifications .....	14
Table 2 List of Implemented Fault Protections.....	48

## CHAPTER ONE: INTRODUCTION

A hybrid vehicle is defined as a vehicle that uses two or more distant power sources to propel it. While idea of hybrid vehicles might have been around for centuries, the first reported hybrid vehicles appeared in the Paris Salon in 1899 [10]. These hybrid vehicles had much different design goals than the hybrid vehicles we picture today. In many cases the, internal combustion engine was not powerful enough to propel the vehicle. So the electric motor was added, not to reduce fuel consumption, but to increase over the total vehicle power.

While the technology of hybrid vehicles has come a long way, the basic idea can still be seen. One of the most well known hybrid vehicles today is the Toyota Prius. The Prius utilizes power from a standard internal combustion engine and a battery pack. While the Prius uses only two distinct power sources, others may have more sources as shown in Figure 1.

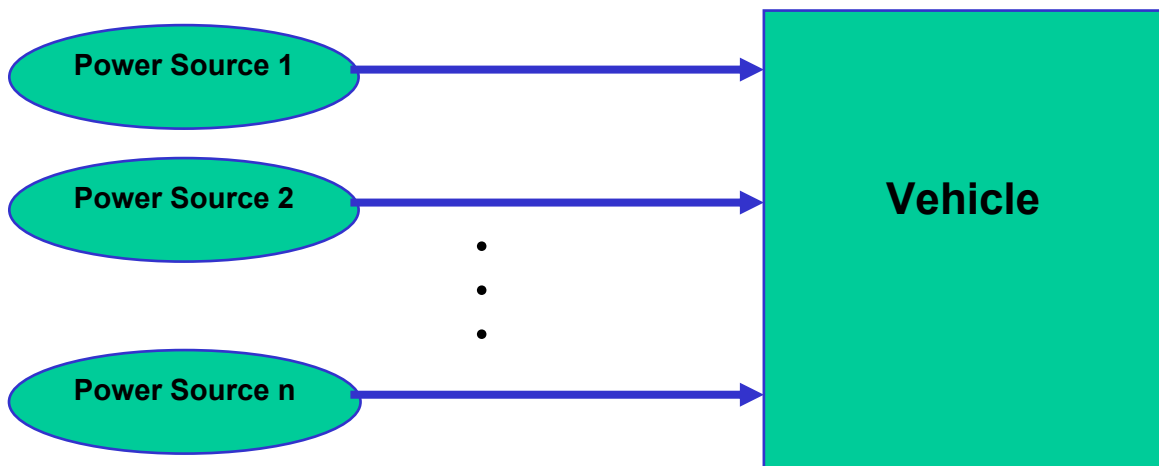


Figure 1 Definition of Hybrid Vehicle

There are many power sources that are utilized to propel today's hybrid vehicles including wind, compressed air, batteries, super capacitors, hydraulic, diesel, and gasoline. While any combination of these power sources would make a viable hybrid vehicle, the most common combination seen in the commercial market today is gasoline and batteries. These power sources, utilized for the purpose of reducing gasoline consumption, yield what is known as a hybrid electric vehicle (HEV).

Since their introduction, hybrid electric vehicles have become increasingly popular. One of the engineering challenges associated with these vehicles is increasing the efficiency of the motor drives and electronics. One method of achieving this goal is to replace the low voltage drive motors with higher voltage versions. This change reduces the high currents associated with driving the low voltage motors. Lower drive current reduces losses which results in improved efficiency.

Since hybrid electric vehicles are designed to have a certain battery capacity, adding more batteries to boost the voltage can adversely affect the overall design of the vehicle. For example, the Toyota Prius utilizes 228 Ni-MH cells to achieve the desired capacity. In order to obtain the highest possible voltage, all 1.2 V cells are strung in series. This results in a total battery voltage of 273.6 V [7]. While adding more cells in series will increase the battery voltage, it will also increase volume, weight, and cost of the battery pack. This is an unacceptable solution in most cases. An alternative solution to increase the voltage supplied to the motor drive electronics is to add in a DC-to-DC converter.

There are many configurations or topologies for hybrid electric vehicles. Some of the more common used in practice are known as series, parallel, and combined or series

parallel. Each topology yields advantages and disadvantages that make it more or less suitable for a given design. However, while these are very different topologies there are some distinct similarities between them.

In this thesis, some similarities will be shown and a sub group of HEVs will be created. The need for power electronics will be explored and shown how a DC-to-DC converter is needed in today's topologies. The converter and all controllers are designed to be universally acceptable for the sub group of topologies to be defined in the following chapters. Issues of tight space requirements in the vehicle will be discussed and designed accordingly. A layered controller architecture is designed to handle commonly encountered modes of operation. A level of protection is designed to protect the converter and its surrounding components. Finally, the highest level of control is designed to be open ended for easy integration into a HEV.

### Typical Topologies

While any realistic topology could be used to realize a hybrid electric vehicle, the three most used in today's vehicles are the series, parallel, and combined hybrid vehicle topologies. These topologies organize the multiple power sources in a specific manner that categorizes them into one of these three groups. From these groups a sub group will be classified where the DC-to-DC converter will have its specific application.



## Series Hybrid

There are a number of papers describing the topologies of hybrid electric vehicles [8]. The first topology to be examined is what is known as the series hybrid as depicted in Figure 2. The series hybrid is one of the first hybrid topologies to be used in commercial vehicles. This is because there is no mechanical coupling of the power sources. The commonly used power sources are gasoline internal combustion engine and batteries. The commonly used power sources are gasoline internal combustion engine and batteries.

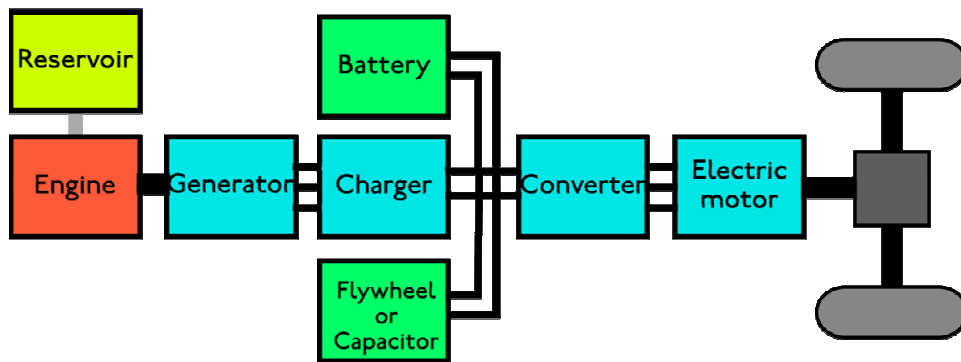


Figure 2 Series Hybrid Topology <sup>1</sup>

In Figure 2, the reservoir represents the gas tank which supplies the internal combustion engine to generate mechanical power. It is then used to drive an electric generator, which converts the mechanical power to electrical power. Traditionally, the charge unit is an electronic device that can be as simple as turning off the generator when the battery voltage reaches a set limit. The charger ensures the batteries maintain a certain level of energy. The battery power is then converted back to mechanical power via the converter and motor. The converter is typically referred to as a motor drive. The electric motor is then used to propel the vehicle.

It can be seen that there is no mechanical coupling of the internal combustion engine and the electric motor. This greatly simplifies the mechanical design of the vehicle. However, it suffers from major loss in efficiency. This is due to the number of conversions the power must go thru before reaching the wheels of the vehicle and the fact that no power conversion is completely ideal. This means there is some power loss each time the power is converted. In this case two conversions take place: mechanical to electrical and electrical back to mechanical.

The product of two or more sub-system efficiencies will always result in a number less than the original numbers. So, generally speaking, it is usually desired to reduce the number of conversion processes to help keep the efficiency high. However, if the product of the efficiencies is greater than the efficiency of a separate single conversion process it is still possible to achieve higher overall efficiency with a multi-stage process.

### Parallel Hybrid

Another hybrid electric vehicle topology is known as the parallel hybrid as seen in Figure 3. Similar to the series hybrid, the battery power is converted to mechanical power thru the converter and electric motor. The reservoir represents the gas tank and is used to supply the internal combustion engine. However, in a parallel hybrid topology the location for coupling of the power sources is done mechanically. This can be through a differential gear box or through a common drive shaft. Since the battery power only goes through one conversion process and the mechanical power of the internal

combustion engine does not go through a conversion, the overall efficiency of the system is potentially high. However, this is not always the case as described previously.

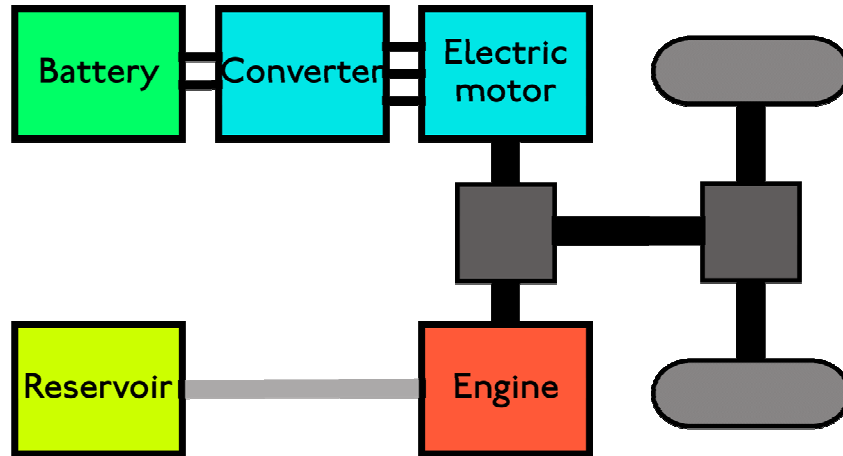


Figure 3 Parallel Hybrid Topology <sup>1</sup>

### Combined Hybrid

A third topology known as the combined hybrid topology or the series parallel hybrid topology is a combination of the series and the parallel hybrid topologies. An example of this topology can be seen in Figure 4. The combined hybrid topology reduces the complexity of the mechanical coupling but still has the double conversion losses when processing power thru the electric motor from the internal combustion engine. The generator in this case can simply be an oversized alternator which can be coupled to the engine thru a typical belt and flywheel. The generator is used to keep the charge of the batteries at a specified level. The batteries are then used to supply the converter and electric motor to produce mechanical power. The electric motor is usually directly coupled to the drive shaft but can be done thru a differential as well.

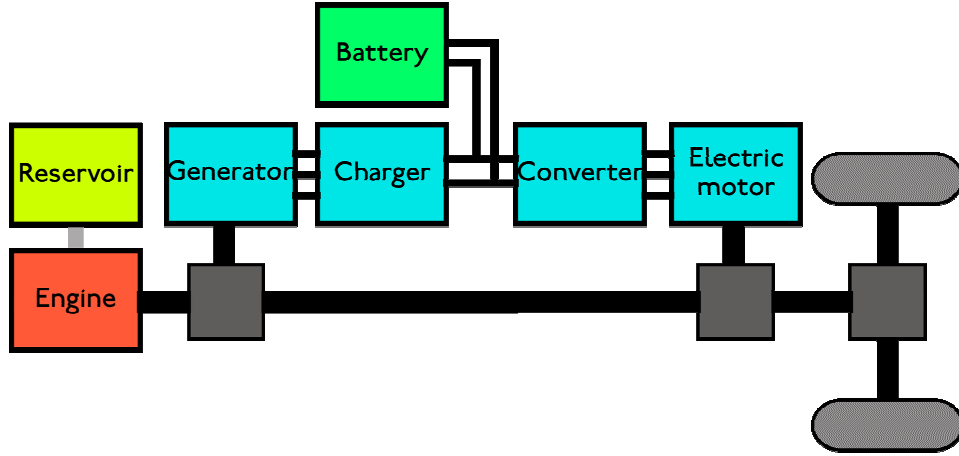


Figure 4 Combined Hybrid Topology<sup>1</sup>

This thesis will look into more detail of the combined hybrid topology and the series hybrid topology. These two topologies share a unique characteristic which creates a sub category of hybrid topologies. That is they both have two electric sources of power coupled together at a common bus. This can be seen in Figure 2 and 4 where the output of the charger is coupled with the terminals of the batteries. This presents many design considerations, which will lead to the definition of standard modes of operation. These modes of operation can be used for any topology which falls into this sub group.

## CHAPTER TWO: LITERATURE REVIEW

It is important to understand why the DC-to-DC converter is a good solution for this application. However, the DC-to-DC converter designed in this thesis has unique features designed specifically for the hybrid vehicle application.

### Battery Voltage

Traditionally, the battery voltage is designed to supply the motor drive converter with the necessary voltage in order to operate properly. However, in later designs this has not been the case. This is due to the fact that the motor drive converters and the electric motors used in new designs are of higher voltage. One reason for doing this is to reduce the losses in the motor at high loads. Given a certain power level, if the voltage is increased the current can be decreased. This can be seen in Equation 1.

$$\begin{aligned} Power &= V_{in} \cdot I_{in} \\ Power &= V_{new} \cdot I_{new} \mid V_{new} = k \cdot V_{in} \\ \therefore Power &= (k \cdot V_{in}) \cdot \left( \frac{I_{in}}{k} \right) \end{aligned}$$

**Equation 1**

Where  $V_{in}$  and  $I_{in}$  are the initial voltage and current applied to the input of a system.  $V_{new}$  and  $I_{new}$  are the new voltage and current values. If the power is to be kept constant and If  $V_{new}$  is proportional to  $V_{in}$  by a factor of  $k$  then it can be seen to maintain a constant power,  $I_{in}$  must be modified by a factor of  $k$  as well.

### Non Idealities

Since there is no such thing as an ideal wire, the wires used to build the motor will always have some resistance. Hence, resulting in a reduced overall efficiency of the system as given by Equation 2.

$$P_{loss} = i_R^2 \cdot R_w$$

**Equation 2**

Where,  $P_{loss}$ , is the power loss defines a cross the wire resistor,  $R_w$ , with  $v_R$  and  $i_R$  represent the resistor voltage and current, respectively. By substituting  $v_R$  from the second equation into the first equation the power loss as a function of  $i_R$  and  $R_w$  is revealed. Therefore, the power loss in the motor will increase exponentially with higher currents. This leads to the need to bring the current in the motor down but still maintaining the power output of the motor. This reveals the need for high bus voltages for the electric motor.

The battery pack of a hybrid electric vehicle is typically designed to have a certain energy capacity. To maximize the output voltage of the battery pack, all voltage cells are generally connected in series. A higher bus voltage can be achieved by increasing the voltage of the batteries. However, this is typically not a suitable solution. This means to increase the output voltage of the battery pack, more cells would be needed to connect in series. This would result in a larger physical size and larger energy storage size of the battery. While larger energy storage capacity is not a negative side affect, it is not an acceptable solution if it increases the physical size of the battery pack. This is because in any vehicle application, size is a scarce commodity. Minimizing the size of the battery

pack as well as any other mechanical and electrical sub-systems in the vehicle is always a major design criterion.

### The DC-to-DC Converter Solution

A better solution would be to add a DC-to-DC converter in between the existing battery pack and the motor drive converter to boost the voltage to the necessary level. Traditionally, the non-isolated bi-directional DC-to-DC converter has been used to perform the task mentioned above. This configuration can achieve higher conversion efficiencies than other common non-isolated DC-to-DC converters such as the Ćuk and SEPIC converters [1, 5].

Operating as a synchronous buck/boost converter, the bi-directional power flow is an inherent property of this topology [4]. Since the converter is typically operated in continuous conduction mode (CCM), its design requires a larger valued filter inductor. A larger inductance can result in an increase in physical size of the inductor, which is not desirable. This large filter inductor can also slow down the transient response of the converter as well as slow down any type of mode transitioning.

If the converter is designed to operate in discontinuous conduction mode (DCM) the value of the inductor can be greatly reduced. Also, the efficiency of the converter at very light load can be increased since there is no negative current in the inductor which produces more conduction losses.

## CHAPTER THREE: SYSTEM DESIGN

After reviewing the existing topologies for hybrid electric vehicles it can be seen that there is a clear need for power electronics conversion and motor drive sub-systems. In most popular designs, this power converter is designed as a synchronously switched buck or boost converter that operates in CCM. However, in the vehicle application there is a need to reduce the size of all components to be integrated into the system. A DCM converter has the advantage of reducing the inductance value of the power filter, which helps to improve power density. DCM operated converters are typically not chosen for this application since they are not inherently bi-directional.

A DC-to-DC converter intended for this application must have a series of controllers which are derived from the needs of the system. Since the DC-to-DC converter interfaces the battery to the motor drive's bus voltage, bi-directional capabilities are a necessity. The battery must be able to supply power to the motor drive system and it must be able to take power back to the battery for charging. The DC-to-DC converter will be controlled by the vehicle system controller (VSC) which is not very fast relative to the dynamics of the system. Because of this, a layered approach for the controllers is appealing since the DC-to-DC converter can safely operate without the intervention from the VSC.

### System Overview

In order to properly design the controllers and system modes, it is necessary to take a closer look at the system power flow for the sub set of HEV's discussed in Chapters one and two. This sub group includes the series hybrid and the combined hybrid topology but is not limited to them. It applies to any hybrid topology that couples

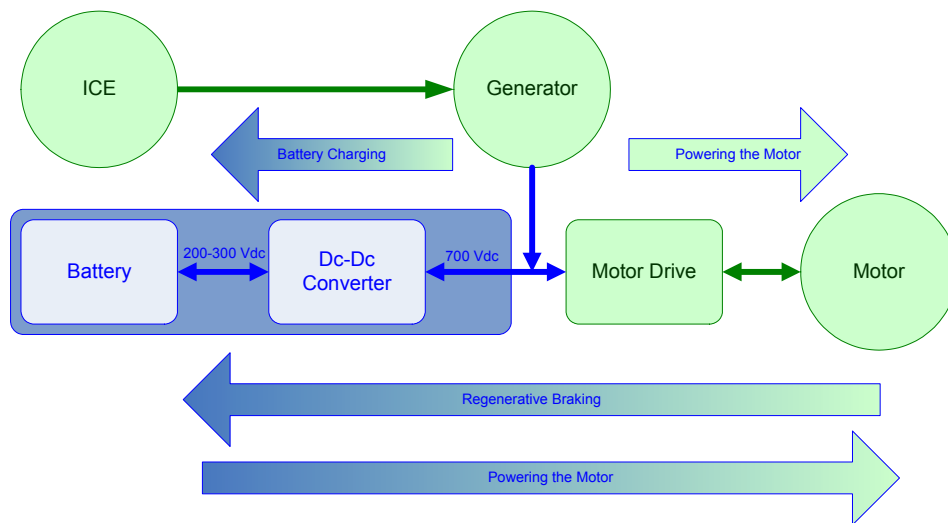


the DC-to-DC converter with a voltage source. This voltage source is usually a generator but could be any other voltage source including another DC-to-DC converter operated in voltage controlled mode, ultra capacitor, or fuel cell [6].

The generator in Figure 5 is driven by an internal combustion engine (ICE) and has a regulated output voltage. During hybrid operation, the DC-to-DC converter must supplement the generator's power to the motor drive system. Since the bus voltage is regulated by the generator, one method to regulate the power from the DC-to-DC converter is to regulate the output current. This reveals that the first parameter the VSC will want to command: output current. However, since the VSC controller is relatively slow and the dynamics of the motor drive system can change suddenly, the DC-to-DC converter might want to change its regulation mode to prevent the bus voltage from increasing to unsafe levels. This can be easily illustrated as follows: If the VSC knows there is a high demand for power to the motor drive system, it will command a large current command from the DC-to-DC converter. Now, if that load suddenly is not there any more (ie. The vehicle stops accelerating), the DC-to-DC converter acts like a current source into the bus capacitance with very low load. This will cause the bus voltage to increase to an unsafe level. In this case, the DC-to-DC converter would switch to a bus voltage regulation mode at a voltage set point slightly above the voltage set point of the generator. This is the second parameter the VSC would need to control. Alternately, if the VSC would like to charge the battery pack, either from the generator or from regenerative braking operation, the power flow with respect to the DC-to-DC converter has now flipped. In order to charge the battery with the proper charging algorithm, the DC-to-DC converter will need to regulate the battery current. Once the batteries are fully

charged the DC-to-DC converter enters what is known as float charge. This means the batteries have reached their fully charged voltage and the DC-to-DC converter then regulates this voltage. Once the converter has entered float charge the amount of current flowing to the battery is dependent on the battery and is typically smaller than the initial current commanded by the VSC. This mode reveals the next two parameters the VSC will want to control: battery current and voltage.

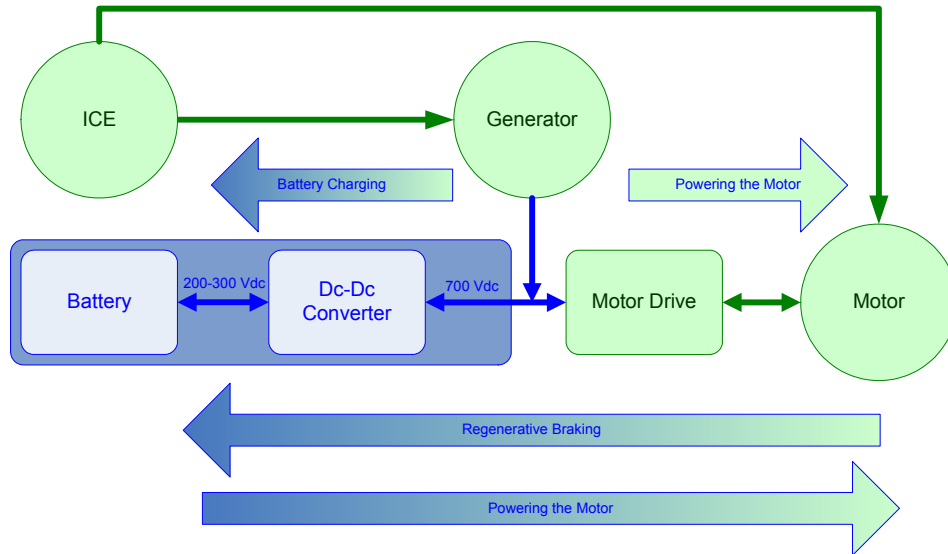
One feature that many strong hybrid vehicles have is the ability to shut down the ICE to conserve fuel. This means all of the power delivered to the motor drive system must come from the batteries through the DC-to-DC converter. It also means the generator is no longer regulating the bus voltage to the motor drive system, so it is now the responsibility of the DC-to-DC converter to regulate the bus voltage.



**Figure 5 Power Flow of Series Hybrid Topology**

The electrical behavior of the DC-to-DC converter in the combined hybrid topology is identical to the series hybrid topology discussed above. Hence, all of the modes and

control parameters are assumed to be the same. This can be seen from the power flow diagram of the combined hybrid topology in Figure 6.



**Figure 6 Power Flow of Combined Hybrid Topology**

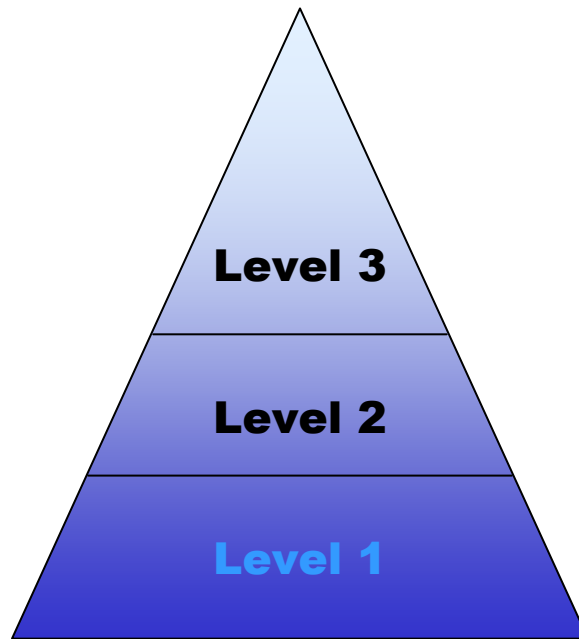
The specifications for the DC-to-DC converter designed here are taken from typical specifications for a system of this nature. The battery voltage will reside between 200V and 300V dependent on state of charge and loading or charging conditions. The bus voltage will fall between 650V and 725V. The maximum power to be pushed to the motor drive system (boost direction) and to the batteries (buck direction) will be 12kW and 6kW, respectively. These specifications are provided in Table 1.

**Table 1 System Specifications**

Criteria	Minimum	Typical	Maximum	Unit
Battery Voltage Range	200	270	300	Volts
Main Bus Voltage Range	650	700	725	Volts
Boost Power Rating	0	6	12	kW
Buck Power Rating	0	4	6	kW

## Level 1 Control

The first level of control is actually the lowest level and acts as a base of controllers for the system, where the level above issues commands to the level below it. This concept is illustrated in Figure 7. Since the control loops are implemented in the DSP, it is important to sample the control parameters and execute the controller code at a fixed frequency. This allows the use of standard digital control theory and sampling theory. The control loops were designed using a direct digital design method which can be found in [3].



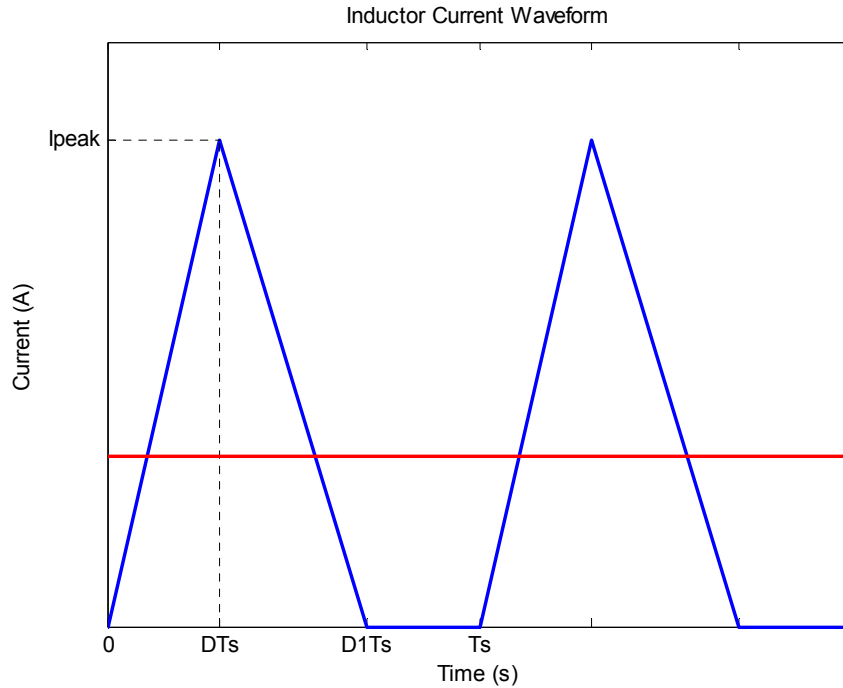
**Figure 7 First Level of Control**

In the first level of control, there are three parameters that need to be controlled: the average inductor current, the input voltage, and the output voltage (bus voltage). By controlling the inductor current it is possible to regulate the current to the battery and the current to the motor drive system. Since the inductor is connected to the low side

voltage, the average inductor current is the battery current. When the average current in the inductor is supplying the motor drive system, this current is proportional to the motor drive current. This allows the inductor current regulation to be used for the motor drive current as well. Since the direction of the current is dependent on which mode of operation the converter is set, it can be assumed that if the mode is known then the direction of the current is known. Because of this fact, the absolute value of the current measurement is used as the controlling parameter. Since all loops are designed digitally and implemented in a DSP, an offset is added to the average inductor current measurement. This value is then sampled by the analog to digital converter (ADC). The offset is then removed in the DSP by subtracting the digital value that corresponds to the offset added to the current measurement.

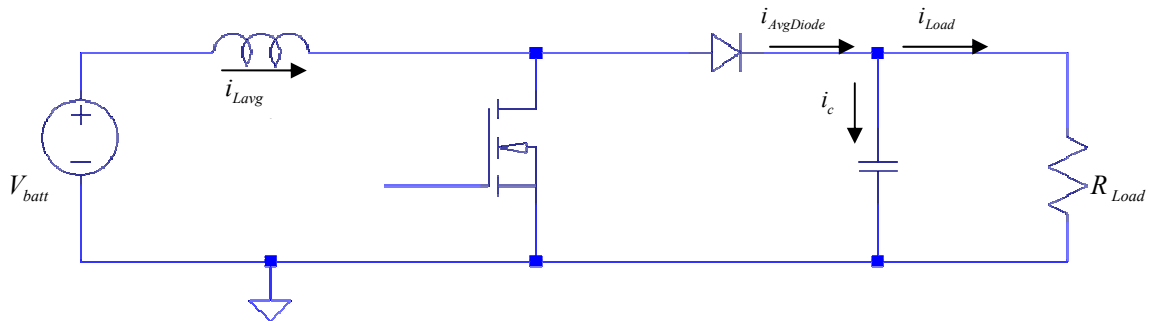
### Average Model

When the converter is supplying power to the motor drive system, it must operate in the boost mode to meet the bus voltage specifications. To help design the controller the equations for the average inductor current are developed here. By looking at the waveform of the inductor current in a converter that is operating in DCM, a few variables can be defined. The peak current ( $I_{\text{peak}}$ ) is the value of the current in the inductor after it has been connected across the input voltage.  $D$  is the duty cycle and  $D_1$  is the percentage of the period ( $T_s$ ) when current is flowing in the inductor Mode I and II.



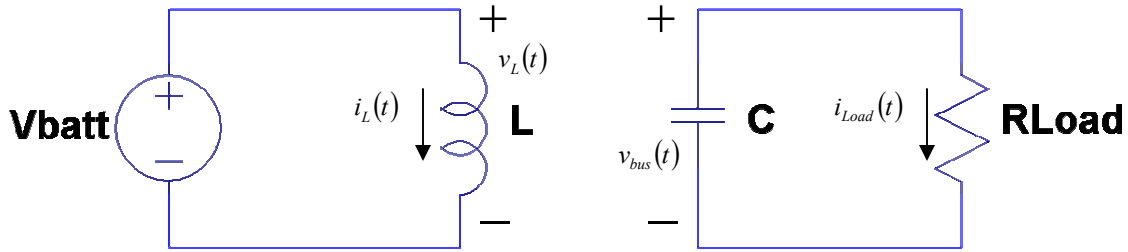
**Figure 8 Boost Mode Inductor Current**

The boost converter circuit can be found in Figure 9. In order to find the average values for the inductor current and output voltage it is helpful to break the circuit up into its different modes as can be seen in Figure 10 thru Figure 12.



**Figure 9 Boost Converter**

This is known as mode I of the converter and can be seen in Figure 10. Mode one occurs when the lower switch is on and the time duration lasts for  $DT_s$  of the switching period,  $T_s$ .



**Figure 10 Boost Converter Mode I**

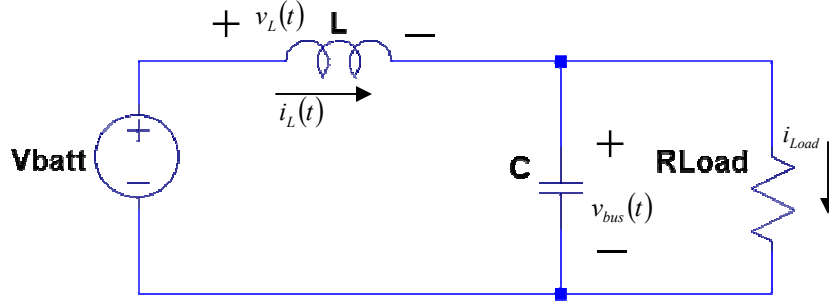
Therefore the peak inductor current can be derived as follows.

$$L \frac{di_L(t)}{dt} = v_L(t)$$

$$I_{peak} = \frac{V_{batt} DT_s}{L}$$

**Equation 3**

At the end of mode I, the inductor is switched across the input and output voltages, and since the output voltage is greater than the input voltage, the inductor current starts discharging into the output capacitor and load. Also, since the converter is operating in DCM, when the inductor current reaches zero it remains there and is not allowed to go in the negative direction. The duration of mode two lasts for a period of  $(D_1-D)T_s$ , as illustrated in Figure 8.



**Figure 11 Boost Converter Mode II**

From here the equations for mode two can be defined as followed.

$$L \frac{di_L(t)}{dt} = v_L(t)$$

$$I_{peak} = \frac{-(V_{batt} - V_{bus})T_s(D_1 - D)}{L}$$

$$D' = (D_1 - D) = \frac{V_{batt}D}{(V_{bus} - V_{batt})}$$

**Equation 4**

By substituting  $I_{peak}$  and  $D'$ , the equation for the average inductor current can be derived as follows.

$$I_{avg} = \frac{I_{peak}(D + D')}{2}$$

$$I_{avg} = \frac{V_{batt}D^2T_s \left( \frac{V_{bus}}{V_{bus} - V_{batt}} \right)}{2L}$$

**Equation 5**



Where  $I_{avg}$  is the average inductor current over one switching cycle.  $V_{bus}$  is the average capacitor voltage over one switching cycle.  $V_{batt}$  is the input voltage to the boost converter

Now that the equation for the average inductor current has been derived, it can be used to help derive the equations for the output voltage. The output voltage is defined by the voltage of the output capacitor. The current into the capacitor,  $i_c$  is the difference between the diode and load current as seen from Figure 9.

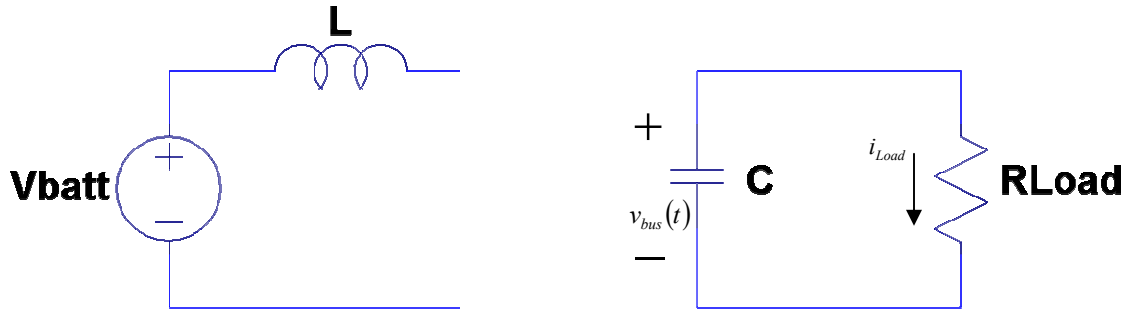


Figure 12 Boost Converter Mode III

Using the properties of the Laplace transform, the equations for the output voltage can then be defined as follows.

$$\begin{aligned}
 i_c(t) &= C \frac{dV_c(t)}{dt} \\
 I_c(s) &= C(sV_c(s) - V_{ini}) \rightarrow V_{ini} = 0 \\
 V_c(s) &= \frac{I_c(s)}{sC}
 \end{aligned}$$

Equation 6

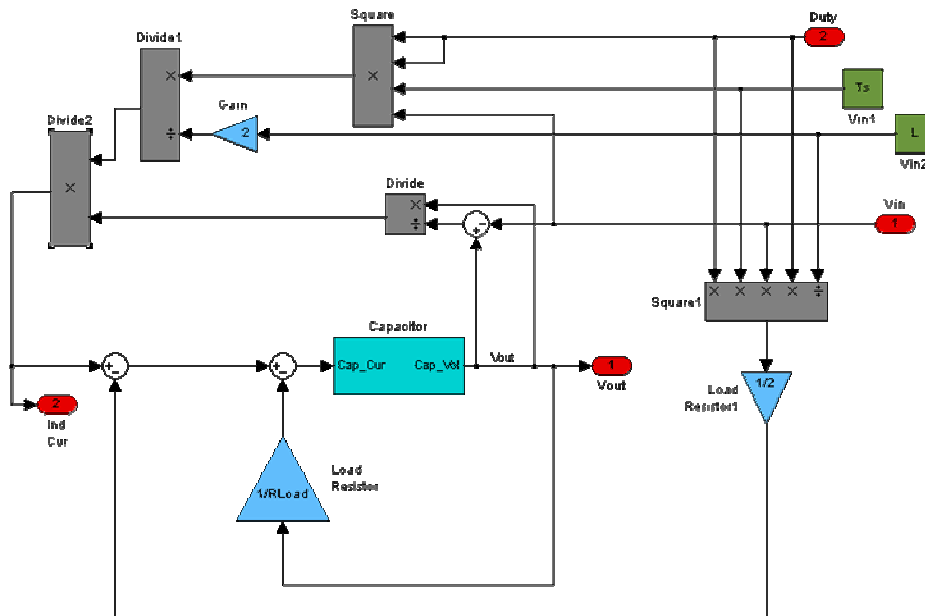
## Verify Average Model

Now that the average model equations for a boost converters inductor current and capacitor voltage have been derived as seen in Equation 7, have been derived, they can be used in simulation software to help design and verify the controllers. To verify the average model, the equations were implemented in a Simulink Matlab simulation. The simulation blocks for the average model can be seen in Figure 13.

$$I_{avg} = \frac{V_{batt} D^2 T_s \left( \frac{V_{bus}(s)}{V_{bus}(s) - V_{batt}} \right)}{2L}$$

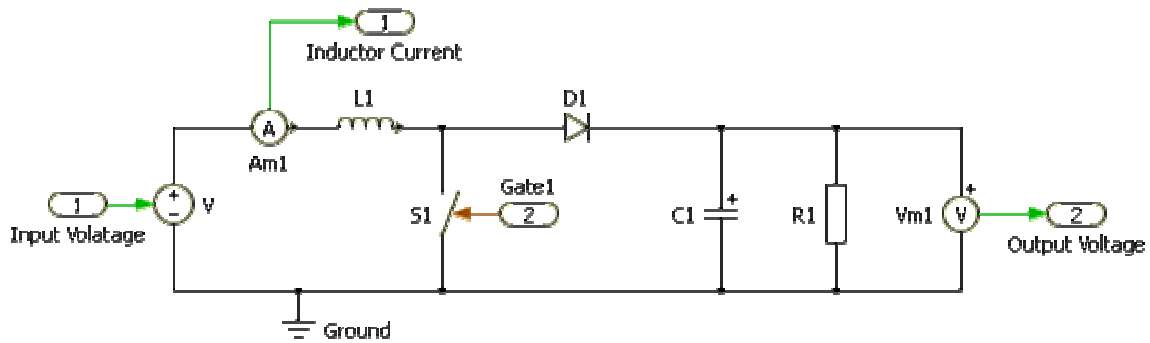
$$V_{bus}(s) = \frac{V_{batt}^2 T_s D^2(s)}{s2CL(V_{bus}(s) - V_{batt})} - \frac{V_{bus}(s)}{sR_{Load}C}$$

**Equation 7**



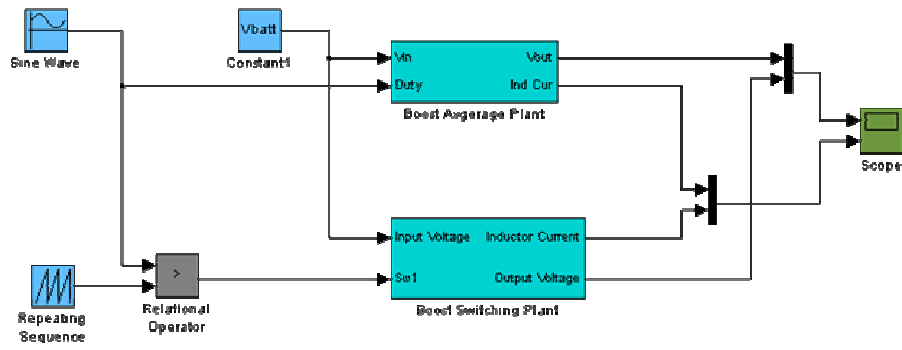
**Figure 13 Boost Converter Average Model**

By using a switching model toolbox known as PLECS in Simulink Matlab, the switching model of the boost converter is added to the simulation. The switching model can be seen in Figure 14 and can be used to simulate the controller response of the system. However, it is very time consuming since the time constants of the controller are much larger than the switching frequency.



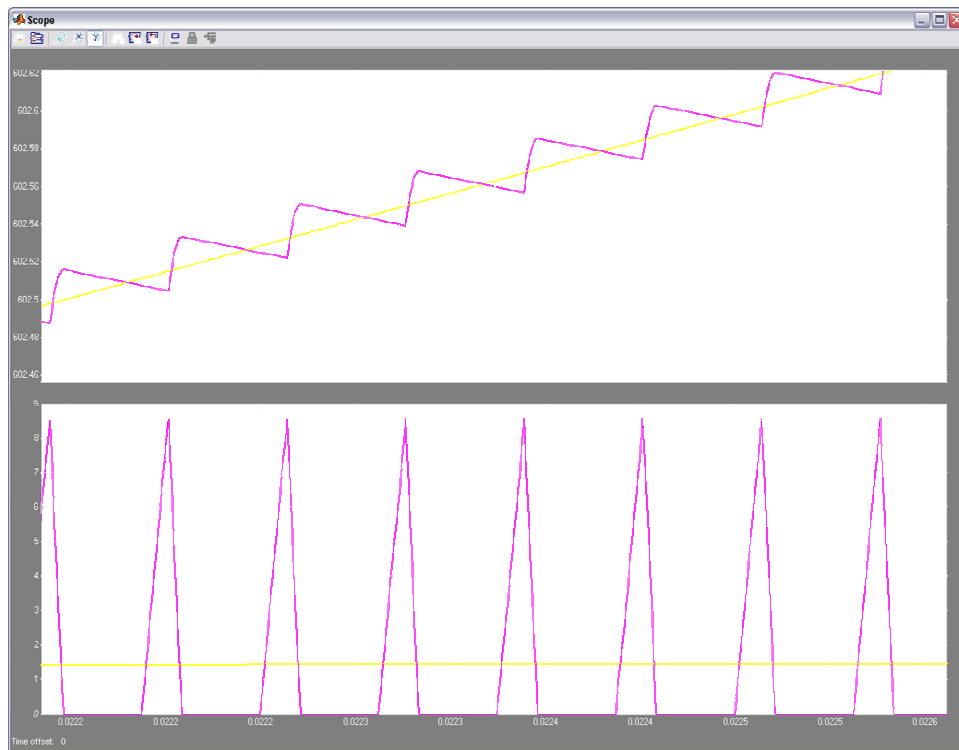
**Figure 14 Boost Converter Switching Model**

The switching model and the average model were then put into the same simulation and run with the same inputs. The output voltage and the inductor current were then fed to the same scope capture output to verify that the average model correctly models the average characteristics of the switching model. This simulation can be seen in Figure 15.



**Figure 15 Boost Converter Average and Switching Model**

It can be seen in Figure 16 that the average output voltage and the average inductor current correctly model the average characteristics of the switching model since the average model signal in yellow seems to take on the average value of the switching simulation signal in red.



**Figure 16 Verify Average Model Simulation Output**

## Small Signal Analysis

One method to design the control loops is to obtain frequency and phase response plots for the entire loop and adjust the phase and gain margin to desired values in order to make the loop stable. The average model derived in the previous section is used as the base model of the plant. However, to find the frequency response the circuit model must be linear. Since the average model has higher order terms, the average model must be linearized around given operating conditions. This is done by perturbing the ac time varying signals and then small signal assumptions around steady state operating conditions are made to eliminate the higher order terms. It can be shown that the small signal model is found in Equation 8. See APPENDIX C for further details.

$$\begin{aligned}
 V_{bus}(s) &= \frac{V_{batt}^2 T_s D^2(s)}{s 2CL(V_{bus}(s) - V_{batt})} - \frac{V_{bus}(s)}{s R_{Load} C} \\
 V_{bus}(s) &= V_{bus} + \tilde{v}_{bus} \\
 D(s) &= D + \tilde{d} \\
 \frac{\tilde{v}_{bus}}{\tilde{d}} &= \frac{\left( \frac{V_{batt}^2 T_s D R_{Load}}{2V_{bus} - V_{batt}} \right)}{s R_{Load} CL + L}
 \end{aligned}$$

**Equation 8**

Once the small signal transfer functions between the duty-cycle to output are found, then the frequency responses of the entire closed loop can be obtained.

In the feedback path there is a voltage divider which scales the output voltage to a range which is suitable to be sampled by the ADC. Since the maximum output voltage for this system is 725V, the maximum voltage seen at the input of the ADC should be set to a safe level above this value. In this system, the voltage divider is designed so that an

output of 1000V results in 3V at the input to the ADC, where 3V is the maximum voltage the ADC can convert.

There is also a low pass filter at the input of the ADC. The actual filter implemented is a simple RC filter designed with a cutoff frequency around the switching frequency. This filter is sometimes referred to as an anti-aliasing filter. The filter helps to filter out some of the switching frequency ripple seen on the output voltage. This is important since the output voltage is only sampled once every switching cycle. If the output voltage is large, then these frequencies would be aliased into the lower frequency components after the signal is sampled. Also, because of the high switching frequency radiated and conducted noise, a carefully placed filter very close to the input of the ADC pin on the DSP can be used to minimize this noise.

After the ADC samples the voltage, the value is converted to a digital value between 0 and the maximum resolution of the ADC. The ADC used in this system has a maximum resolution of 12 bits. This means the maximum value converted by the ADC has a digital value given by Equation 9.

$$ADC_{\max} = 2^{12} - 1 = 4095$$

**Equation 9**

In this system, the ADC was set to left adjust the ADC measurement into a 16 bit register. This means the converted value from the ADC is stored in the upper 12 bits of this 16 bit register. Therefore the ADC gain can be calculated as in Equation 10.

$$ADC_{\text{gain}} = \frac{ADC_{\max} \cdot 2^{(16-12)}}{3} = \frac{ADC_{\max} \cdot 2^4}{3} = \frac{65520}{3} = 21840$$

**Equation 10**

The controller must hold the value for the duty cycle once it reaches steady state. When the control loop has reached steady state the output voltage is equal to the reference voltage. This means the loop error signal at steady state is zero. In order to hold the output value at the proper value, a digital integrator is used. This is one of the simplest but very effective methods to design an initial control loop. Any digital integrator can be used. In this system the backwards Euler integrator is used because the difference equation implemented in the DSP is in a form which has a small number of computations. This can be seen from the difference equation in Equation 11 since the difference equation can be directly transferred into C code for the DSP.

$$H_{EI}(z) = \frac{Y(z)}{X(z)} = \frac{1}{1-z^{-1}}$$

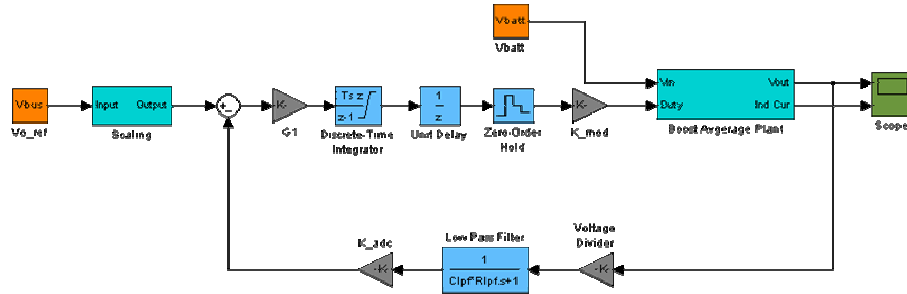
$$Y(z)(1-z^{-1}) = X(z) \Rightarrow Y(z) = X(z) + z^{-1} \cdot Y(z)$$

**Equation 11**

The unit delay and zero order hold are added to model the characteristics of the PWM module. Since the DSP takes some time to calculate the value for the duty cycle, the value is held until the start of the next switching cycle before it takes affect. This is the reason for added unit delay. The zero hold is added since the analog signal for the duty cycle does not change over one switching cycle, effectively bringing the sampled digital signal back to the analog domain.

Finally, the modulator and the controller gains are added. The modulator gain transforms the digital value to the actual duty cycle value. This gain is found to be the one over the maximum counter value. The maximum counter value is discussed in more detail in the interleaving section. The controller gain is used to design the cross over

frequency of the control loop in order to make the system stable. The entire control loop is illustrated in Figure 17.



**Figure 17 Control Loop**

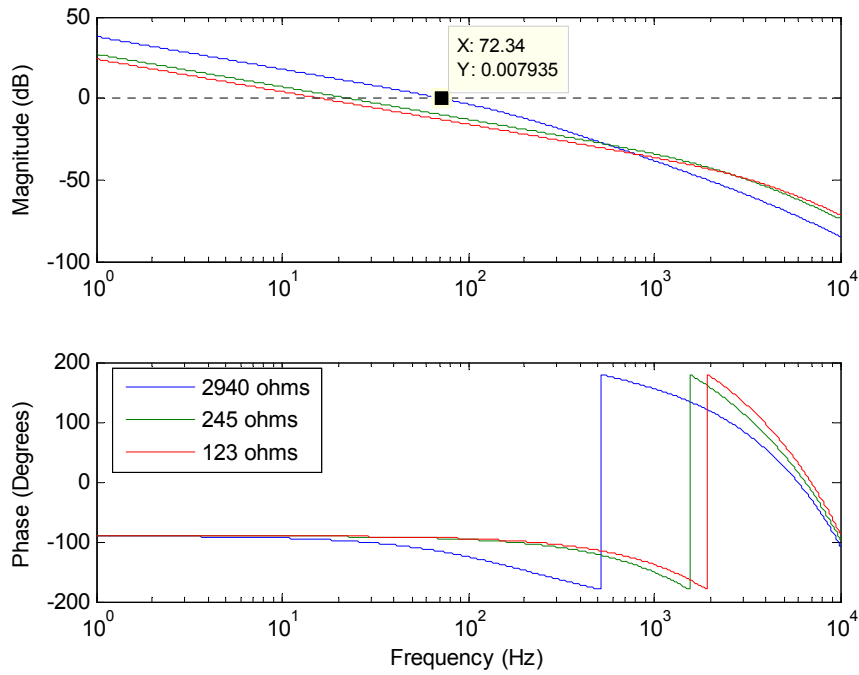
All components are then added to a Matlab program. To find the frequency response of both components in the S domain and the Z domain, the following relationships to frequency are used, where  $T_s$  is the sampling period. In this case the sampling period is equal to the period that the control loop runs at. The controller gain is then designed to give a stable control loop response, which can be seen in Figure 18.

$$s = j2\pi f$$

$$z = e^{j2\pi f T_s}$$

**Equation 12**





**Figure 18 Calculated Frequency Responses**

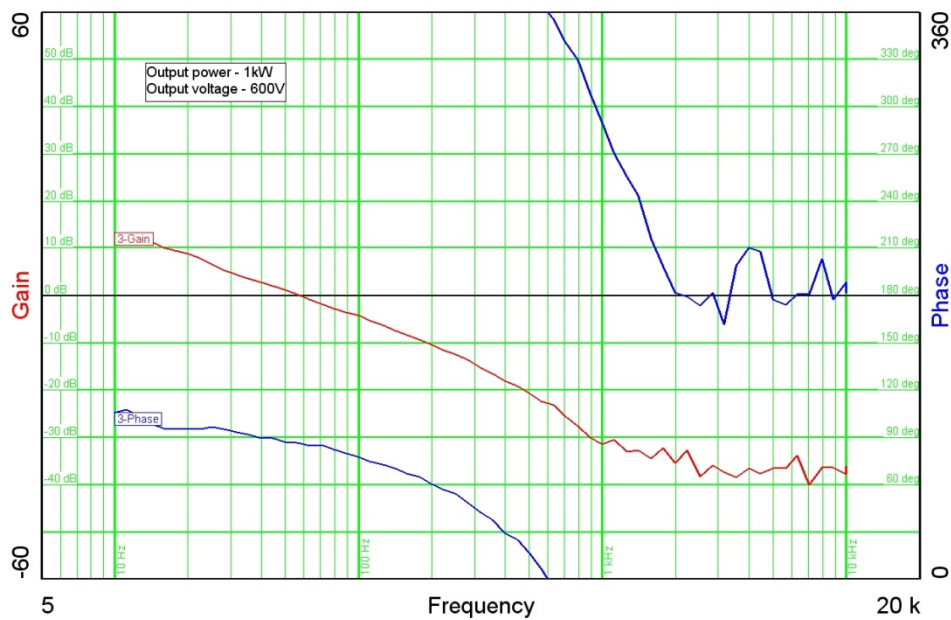
All control loop parameters are then programmed into the DSP and run in the actual system. Then a frequency response analyzer is added to the control loop to perturb it with small signals. This effectively produces the bode plots of the actual systems control loop. This plot was then used to fine tune the frequency response using digital zeros. The form of the digital zero was also chosen to minimize calculation time and can be found in Equation 13.

$$H_{Zero}(z) = 1 - \left(1 - \frac{1}{a}\right)z^{-1}$$

$$H_{ComplexZero}(z) = 1 - \left(2 - \frac{1}{b}\right)z^{-1} + z^{-2}$$

**Equation 13**

The complete controller function is simply a combination of the digital zero, digital integrator, and gain. Since the controller design was done entirely in the z domain, there is no need to convert any of the controller functions into the s domain. This is one of the major advantages of the direct digital design approach. The final frequency response can be seen in Figure 19. The control was optimized to maximize overshoot and undershoot without increasing the settling time by too much.



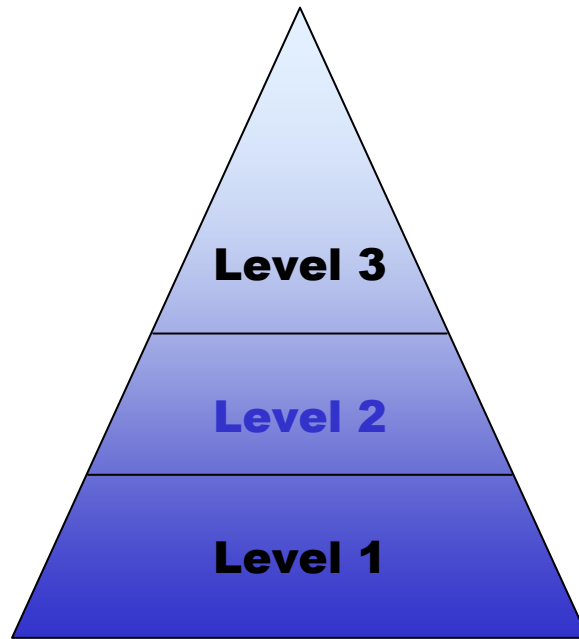
**Figure 19 Measured Frequency Response**

It can be seen that the cross over frequency of the calculated and measured results match very well, crossing over around 50Hz. The phase plots match fairly well with a phase margin of about 90 degrees and a phase cross over around 600Hz.

### Level 2 Control

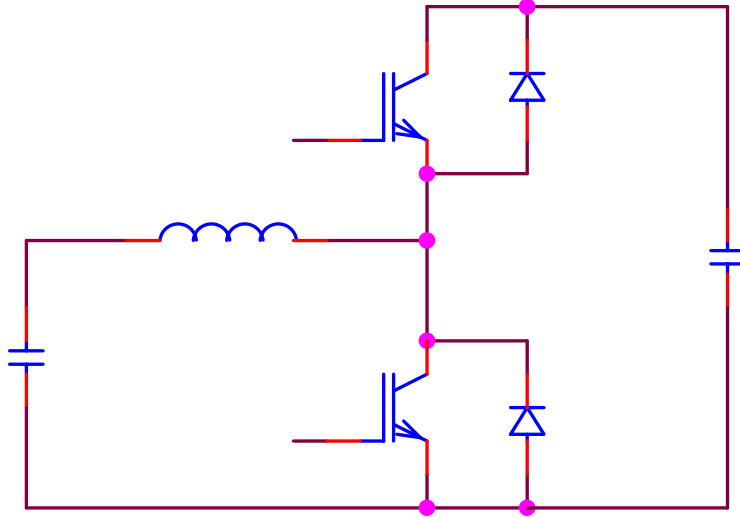
The second level of control is used to set up and control the first level of control. This is important because it alleviates some of the functions and time demand for the

vehicle system control. This means that the vehicle system controller does not have to be very involved or react very fast to changes in the system, which is usually the case. The vehicle system controller can then command modes of operation with regulation set points and let the converter run in a safe manner. It also provides the mechanism that allows the converter to operate in DCM mode while still maintaining bi-directional power flow.



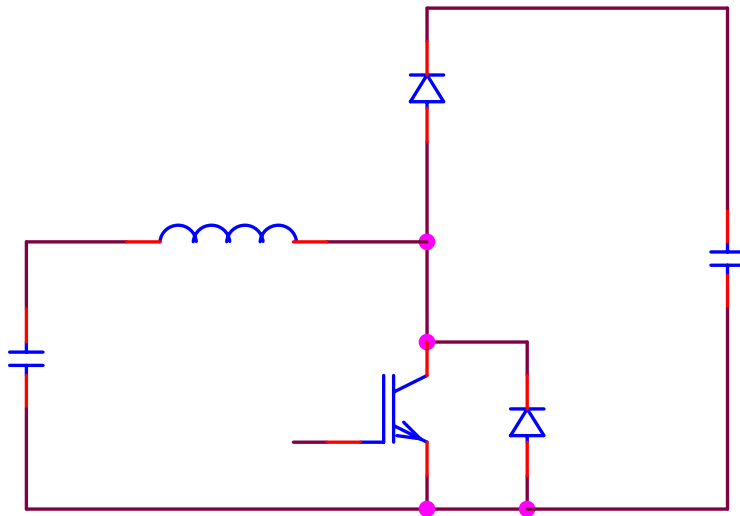
**Figure 20 Level 2 Control**

In this system, the IGBT is used as the switching device. Each IGBT package is also equipped with an anti-parallel diode which makes the overall package behave like a MOSFET (bidirectional conduction, unidirectional blocking). The schematic for a single phase of the system can be seen in Figure 21.



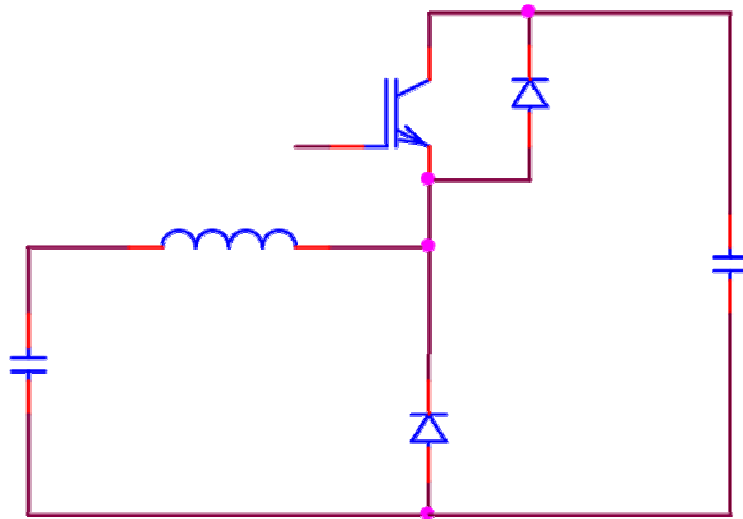
**Figure 21 Single Switching Leg**

However, if these switches are synchronously switched, the converter will operate in CCM, which is not desired in this design. In order to ensure the converter operates in DCM in boost mode the inductor current must be blocked from going in the negative boost direction. To do this, the driving of the upper switch is deactivated when the converter is operated in boost most. The anti-paralleling diode is then utilized to block the negative boost current in the inductor. This can be seen in Figure 22.



**Figure 22 Boost Operated Switching Leg**

Similarly, when the converter is desired to operate in the buck direction the negative buck current must be blocked to ensure DCM operation. By deactivating the driving of the lower switch and utilizing the anti-parallel diode the converter can now be operated in DCM in the buck direction. This schematic is illustrated in Figure 23.



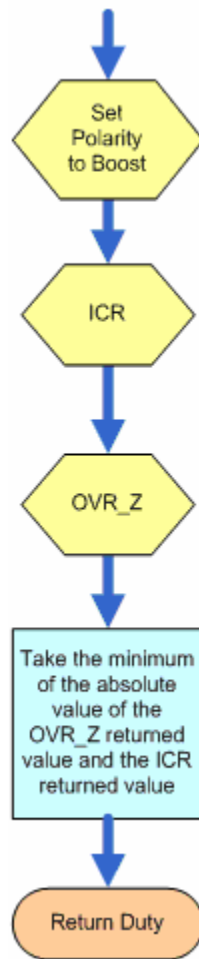
**Figure 23 Buck Operated Switching Leg**

### Modes of Operation

Now that the mechanism for bidirectional and DCM operation has been defined, the modes of operation, that the vehicle system controller would normally command, will be defined. All modes are defined by analyzing situations that will be present in the actual system. The modes include Hybrid Boost Vehicle Mode, Hybrid Buck Vehicle Mode, and Electric Vehicle Mode. To help design the code in the DSP for the second level of control an interactive system of flow charts was designed in Visio. Each function block which contains not trivial functions can be double clicked to reveal the underlying function. By double clicking the return block in the function it redirects the view back to

the place where the function was called. This emulates the way the code would actually be designed in the DSP and helps making an easy transition from the flow chart to the actual DSP code. The flow chart in its entirety can be found in APPENDIX A. The DSP code generated from the flow charts can be found in APPENDIX B.

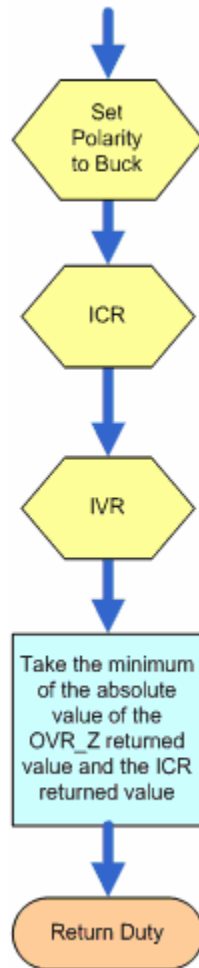
During hybrid boost vehicle mode the high side bus voltage is regulated by the generator and the DC-to-DC converter is used to supplement the power to the motor drive system. The power to be delivered to the motor drive system is proportional to the average inductor current. This means, under normal operating conditions the converter will operate in inductor current regulation mode. The set point is commanded by the vehicle system controller along with an over voltage regulation set point. This is important if the dynamics of the vehicle change before the vehicle system controller has time to respond. To put this into an example: if the motor drive system requires a large amount of power, the DC-to-DC converter will receive a large inductor current reference to regulate to. If the load suddenly drops out (ie. someone takes their foot off the gas pedal) then the converter acts as a current source into the output capacitors with little to no load. This means the output voltage will increase very fast. If it reaches the over voltage regulation set point the converter will then switch to output voltage regulation mode at that set point. The converter will remain at this safe operating condition until the vehicle system controller has time to respond to the situation. This mode is exemplified in Figure 24.



**Figure 24 Hybrid Boost Vehicle Mode Flow Chart**

In hybrid buck vehicle mode the converter is used to regulate the current to the battery in order to perform the proper charging algorithm. So in normal operation the converter is operated in inductor current regulation in the buck direction. The high side bus voltage is regulated by the generator. The vehicle system controller commands a current regulation set point with an over voltage regulation set point. Since the batteries are connected to the low side voltage of the converter, voltage over shoot is not too much of a concern. However, what is known as the float charge voltage can be programmed as the over voltage regulation set point. This means once the batteries are fully charged and

they have reached the float voltage, the converter switches to input voltage regulation mode. During this mode the current supplied to the battery is smaller than the commanded current. The flow chart describing this mode can be found in Figure 25.

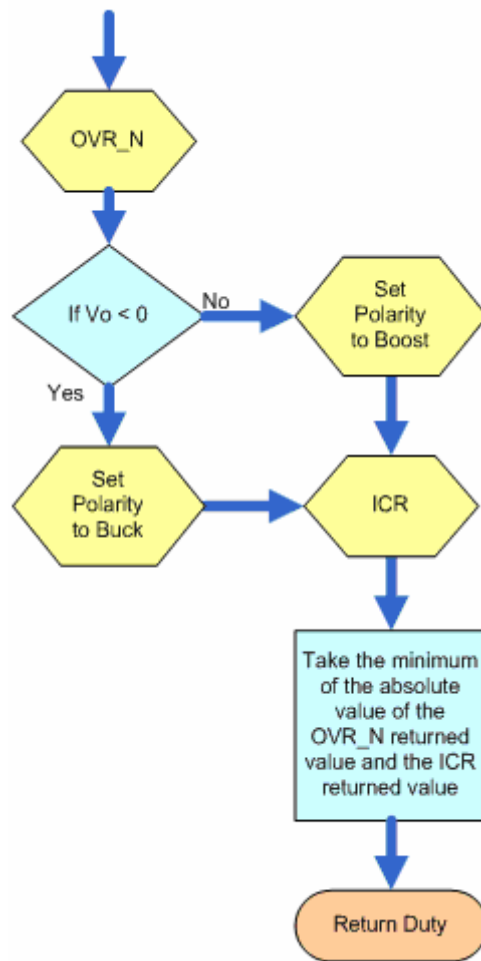


**Figure 25 Hybrid Buck Vehicle Mode Flow Chart**

Electric vehicle mode is the most complex out of the three modes. In this mode the generator is disabled, so the converter is now responsible for regulating the high side bus voltage. This means the converter is operated in output voltage regulation during normal operations. If the motor drive system requires an unsafe current from the converter it will switch to inductor current regulation mode at the commanded over

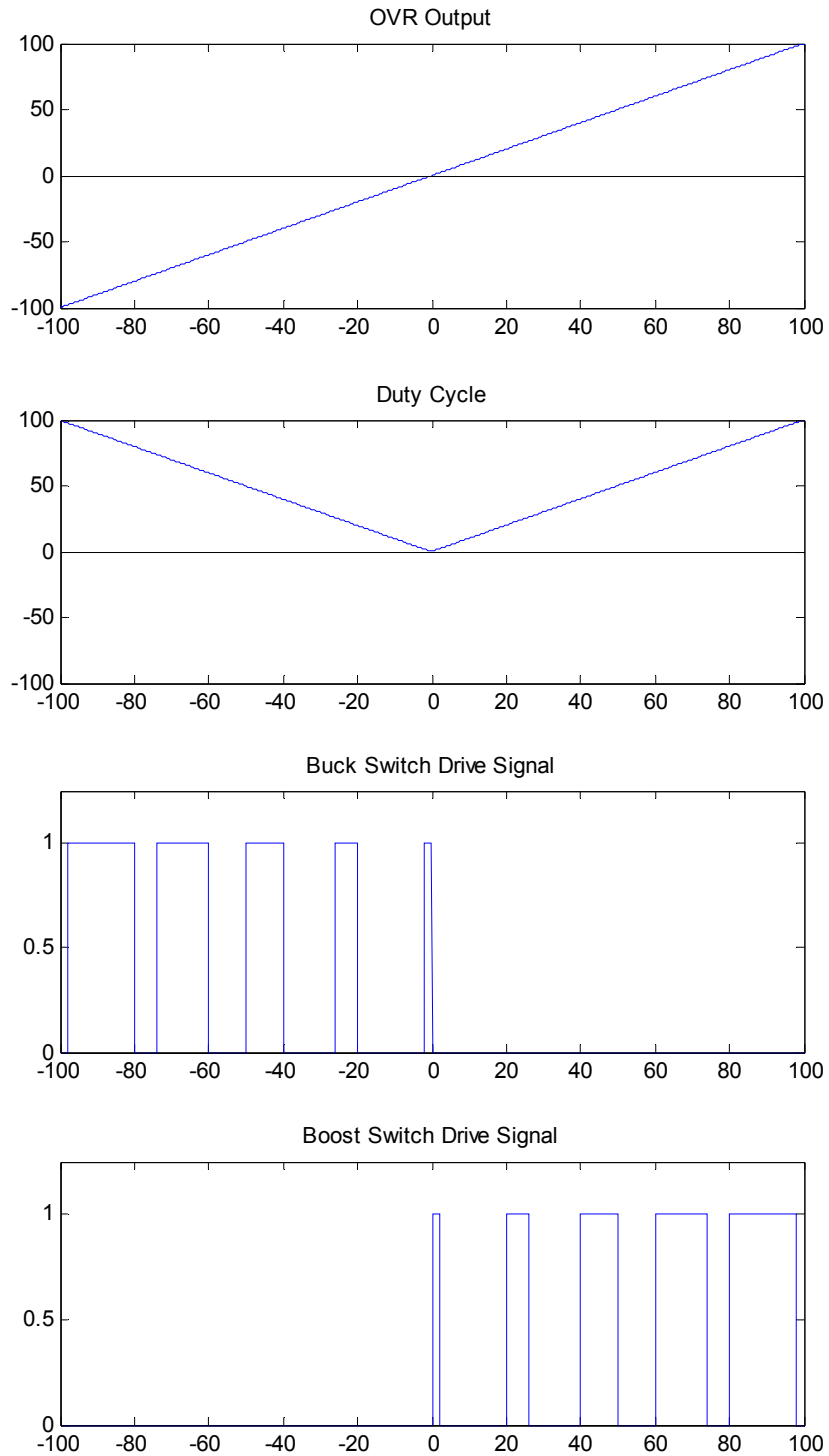


current regulation set point. Also, if a regenerative braking operation is performed on the motor drive system, the resulting power must be processed by the converter back to the batteries. The current from the motor drive system back to the bus capacitors will try to increase the voltage on the high side bus. The output voltage regulator will reduce the duty cycle in order to maintain regulation of the high side bus voltage. If the duty cycle reaches zero and the bus voltage continues to increase, the converter will switch the operation of the converter to buck mode and continue to increase the duty cycle in the buck direction. Similarly, if the current to the batteries reaches an unsafe level the converter will switch to inductor current regulation in the buck direction at the commanded over current regulation set point. The flow chart for this mode of operation can be found in Figure 26. To perform this task in a seamless manner a modified version of the output voltage regulator was created. The output of the modified controller is allowed to go to negative values. Normally negative values on the output of the controller would not make sense since the output of the controller is used to command the duty cycle. However, by using the sign of the output of the controller to dictate which direction the converter should set the switches, it can be used to provide a smooth transition between buck and boost mode.



**Figure 26 Electric Vehicle Mode Flow Chart**

To elaborate more on the modified output voltage regulator, refer to Figure 27. When the output of the regulator is negative the converter is set to operate in buck mode, deactivating the driving for the lower switch, and the absolute value of the output of the regulator is passed to the driving of the duty cycle of the active switch. Similarly, when the output of the regulator is positive, the driving of the upper switch is deactivated, setting the converter to run in boost mode. The output of the regulator is then sent to the duty cycle of the active switch.



**Figure 27 Modified Output Voltage Regulator Operation**

The actual hardware used to realize the converter was a modified Semikron SKAI3001GD12-1452W module. The module was modified to take the TI DSP

TMS320F2808. This module can be seen in the red box in Figure 28. This module is originally intended to be used as a three phase motor controller. However, by attaching the three power filter inductors to the three phases and to the filter capacitors, the converter can now be controlled as three paralleled buck or boost converters, using the techniques discussed here.

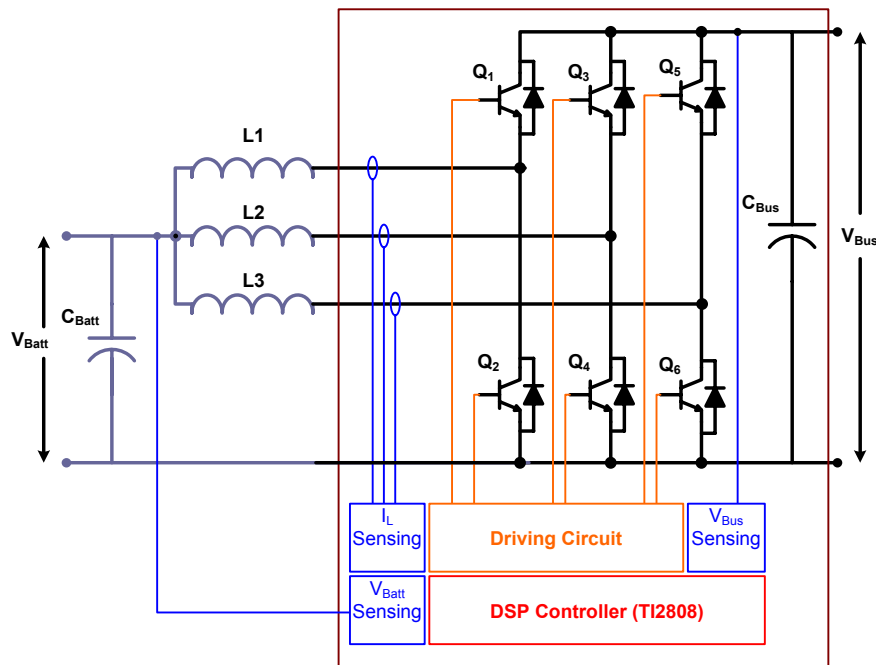


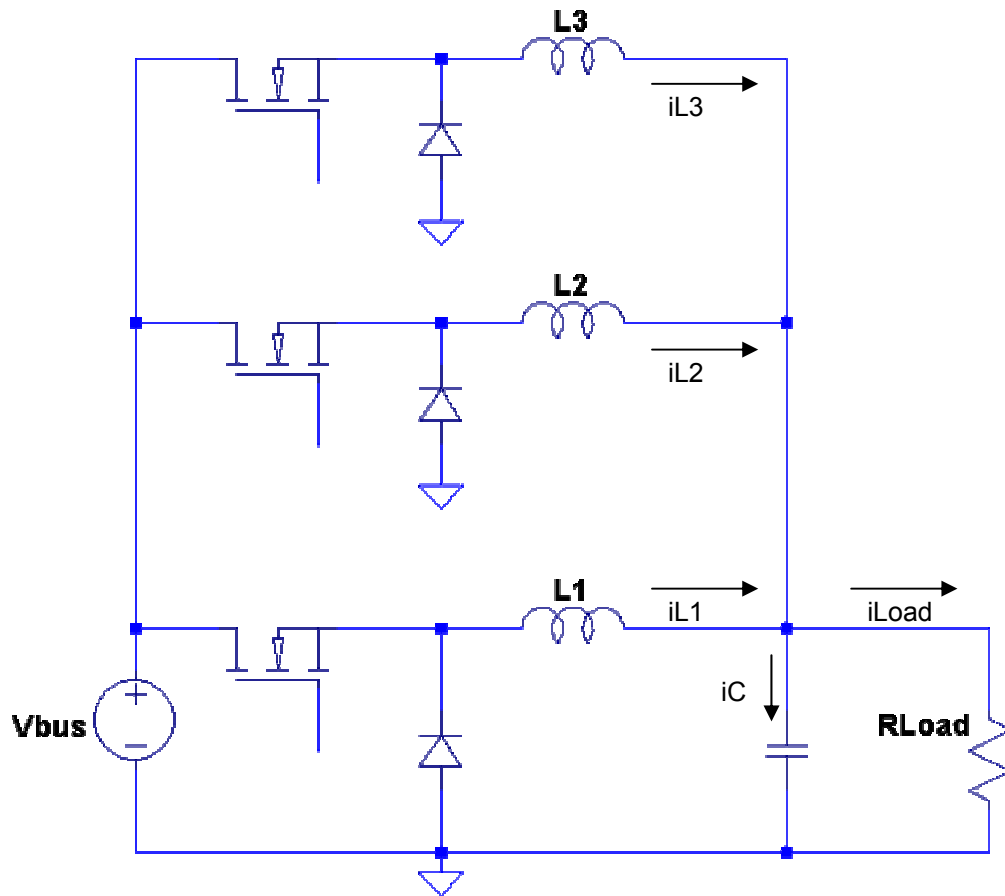
Figure 28 Semikron Module

### Interleaving

Since the converter designed here is running in DCM the output voltage ripple is expected to be greater than that of a converter designed to run in CCM. This is a known phenomenon in DC-to-DC converter design. One method to counteract the additional voltage ripple is to interleave multiple converters. Interleaving is the process of aligning the inductor current waveforms in a way to minimize the ripple current into the output

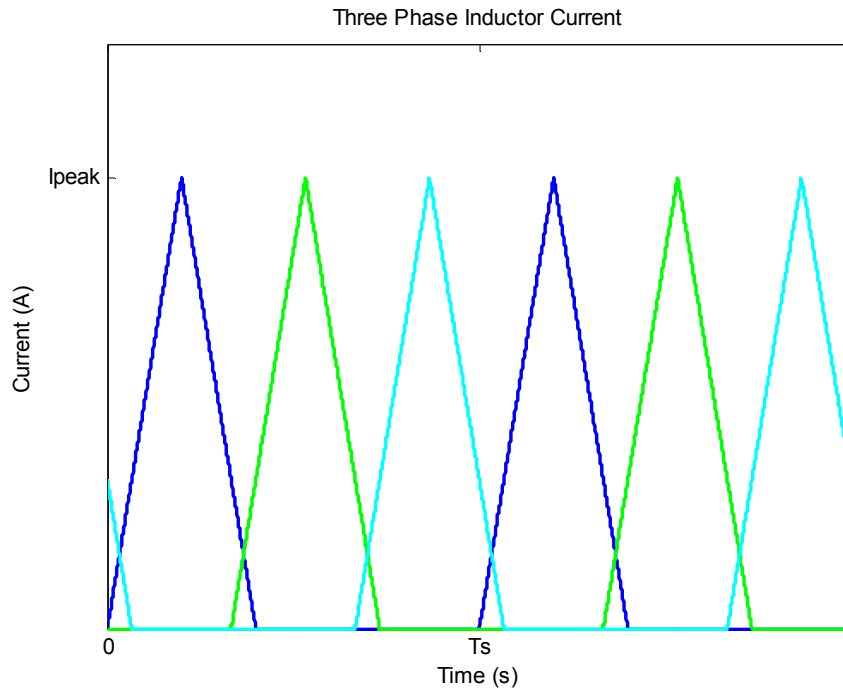
capacitor [4]. Since the input current ripple to the output capacitor is reduced, the losses in this component are also reduced [2].

This is seen easiest in the example of a buck converter. Since the inductor is connected directly to the output capacitor the capacitor current is equal to the sum of the inductor current and the load current tied in parallel with the output capacitor. Since the output voltage is assumed to be in steady state, the load current is also assumed to be relatively constant. In this example the load is defined as an output resistor. The current thru the resistor is considered to be at DC because the output voltage is at steady state. This means the majority of the ripple current comes from the inductor. If three converters are connected in parallel the capacitor current is equal to the sum of the three inductor currents over time and the load current which is assumed to be DC. This schematic can be seen in Figure 29.



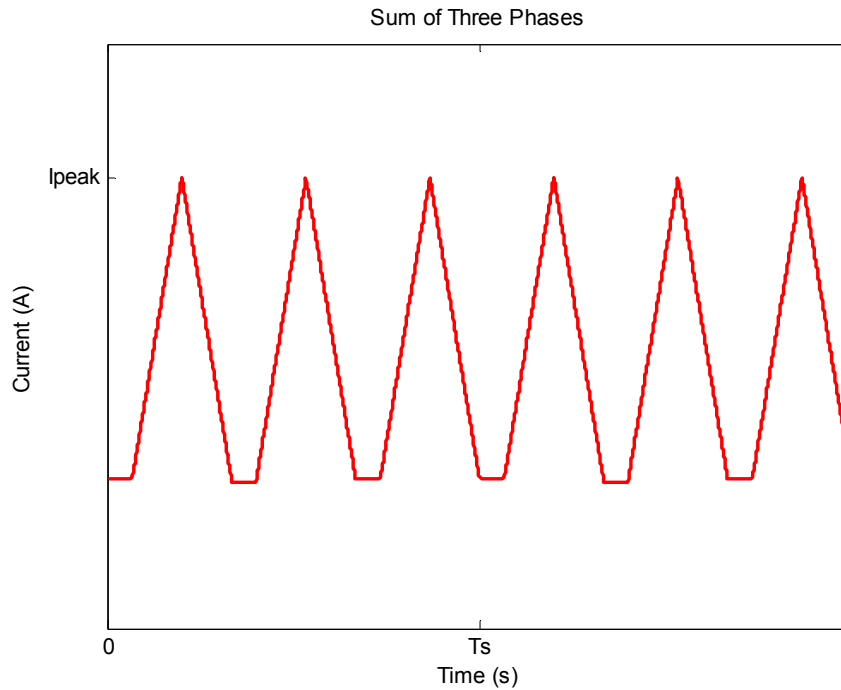
**Figure 29 Three Paralleled Buck Converters**

If the inductor current waveforms are synchronized in a way to distribute the ripple current evenly over the switching period, the effect of the ripple current into the output capacitor is minimized. Since the output capacitor voltage is a function of the capacitor current, minimizing the capacitor current ripple also minimizes the output voltage ripple. The three inductor currents, properly interleaved, can be seen in Figure 30.



**Figure 30 Three Interleaved Inductor Current Waveforms**

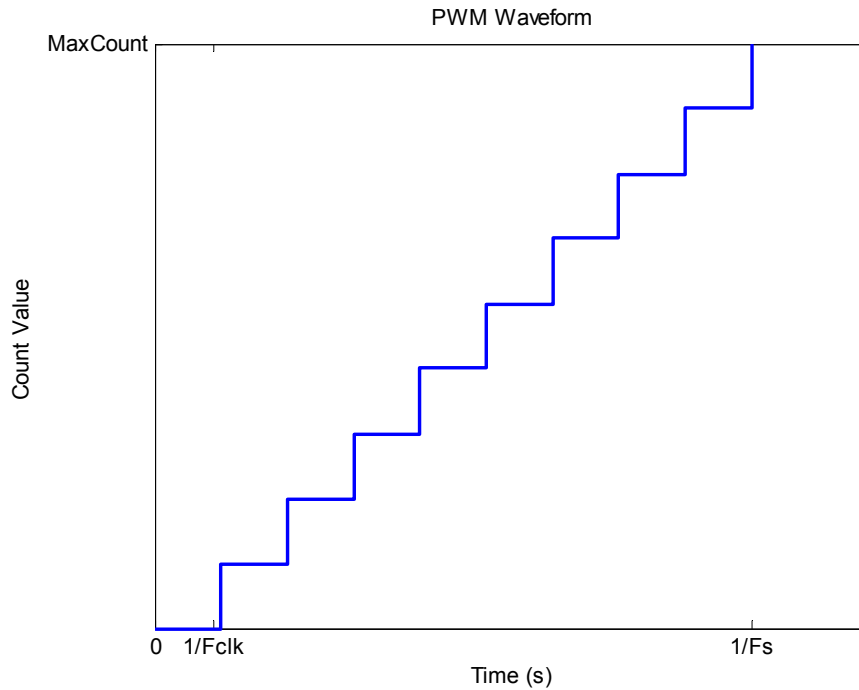
Ignoring the DC load current, the capacitor current waveform can be seen in Figure 31. It is important to note that the frequency of the capacitor current has increased to 3 times the switching frequency. In addition, the peak to peak current values have been reduced since the capacitor current does not go to zero, like the inductor current does.



**Figure 31 Capacitor Current of Three Phase Interleaved Converters**

To properly interleave the inductor current, it is important to understand in more detail how the pulse width modulated (PWM) signals are generated in the DSP. These are the signals that are generated from the duty cycle command from the controllers. The DSP used for this converter is the TMS320F2808 DSP from Texas Instruments. The PWM architecture used in this DSP utilizes a high frequency counter to generate what is known as the ramp signal. The ramp is counted up to a maximum counter value where it is reset to zero and the counting starts again. A very low resolution version of this PWM architecture can be seen in Figure 32.





**Figure 32 PWM Architecture**

Since the counter is incremented once every clock period, the clock period should be greater than the switching period of the PWM signals. In addition there should also be a sufficient number of steps between zero and the maximum counter value to provide a good resolution for the duty cycle. The duty cycle value is compared to the ramp signal. If the value of the current value of the ramp is less than the compare value the PWM module outputs a high signal to drive the active switch. If the current value of the ramp is greater than the compare value the PWM module outputs a low signal to turn off the active switch. This is how the duty cycle command generates the PWM signals. The maximum counter value must then be designed to produce the proper switching period. The value is designed using Equation 14.

$$T_{period} = \frac{1}{F_{switching}}$$

$$T_{clk} = \frac{1}{F_{clk}}$$

$$T_{period} = T_{clk} MaxCount$$

$$MaxCount = \frac{T_{period}}{T_{clk}} = \frac{F_{clk}}{F_{switching}}$$

Equation 14

There are multiple PWM modules in this DSP and they are utilized to generate the six PWM signals for the three interleaved converters. Each module has its own ramp signal which is also called a timer. It is important to note that all timers are driven from the same clock. Because of this, once the timers are initialized and running, the ramp signal will not drift from one another. Therefore the assigned initial phase shift will be kept at all times throughout their uninterrupted operation. A diagram of this set up can be seen in Figure 33.

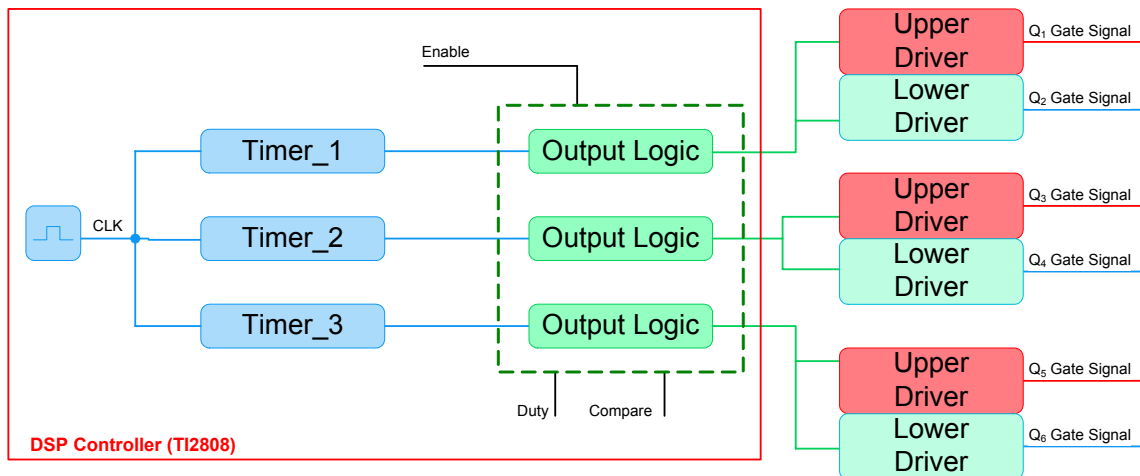


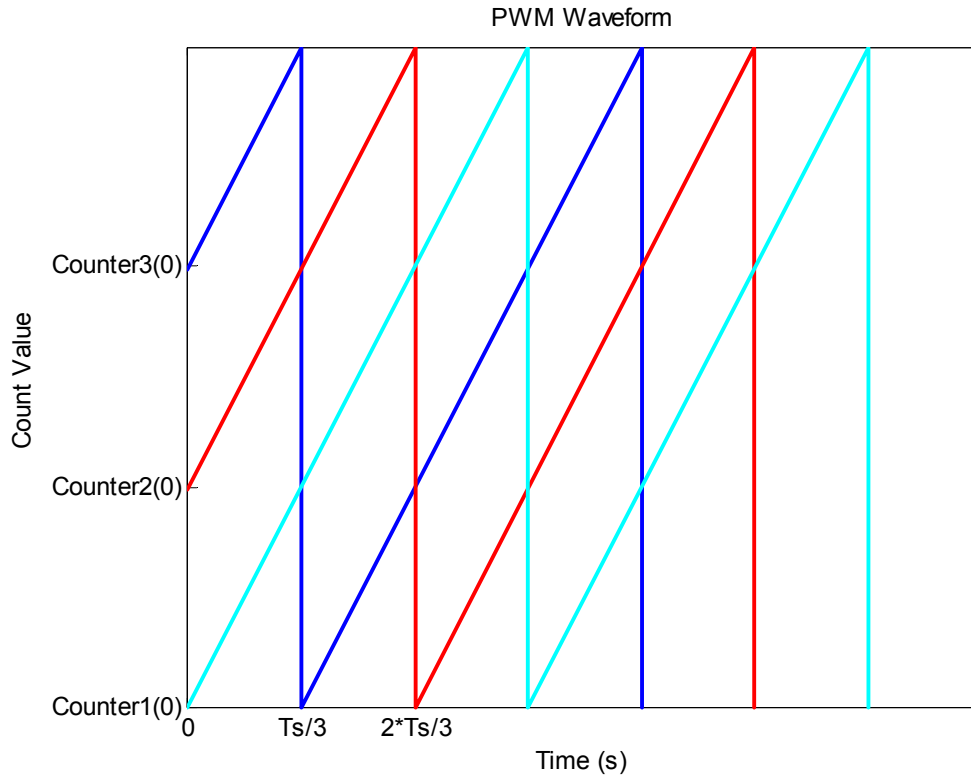
Figure 33 Three Phase PWM Module Diagram

The inductor current waveforms will be properly interleaved since they are directly related to the properly interleaved PWM signals. Since the initial phase shift of the ramp signal that generates the PWM signal is maintained, it is only necessary to design the proper initial conditions of the ramp to properly interleave the inductor current throughout their operation. The initial values of the ramp signals are calculated using Equation 15.

$$\begin{aligned}Counter_1(0) &= 0 \\Counter_2(0) &= \frac{MaxCount}{3} \\Counter_3(0) &= \frac{2MaxCount}{3}\end{aligned}$$

**Equation 15**

It can be seen that the three ramp signals are properly interleaved in Figure 34 by setting the initial conditions appropriately.



**Figure 34 Three Interleaved PWM Ramp Signals**

### Protection

It is important to protect the converter in the event of an unavoidable situation. All protection functions were programmed in code in the DSP. The location of the fault checking can be found in APPENDIX A. If any fault condition is detected, the converter will enter a standby state. The standby is used as an intermediate state between normal run and normal shut down. This is important because it allows the converter to remain in standby until the vehicle system controller can respond properly to the fault and reset it. The fault is reset by commanding the converter to a normal shutdown. All PWM signals are disabled during standby and shutdown.

A list of the fault conditions implemented in this converter can be found in Table 2. If for some reason the voltage on the low side exceeds an unsafe level for the batteries the converter will enter standby to prevent damaging the batteries. Similarly, if the voltage on the bus reaches an unsafe level the converter will go to standby to prevent damaging the bus capacitors, motor drive system, and generator. This could happen if the load dynamics are too fast for the over voltage regulation set point to regulate. If the inductor current reaches an unsafe level because of a short on the output, the converter will enter standby to protect the converter. There are multiple temperature sensors in the hardware module. These sensors are used to monitor the temperature of the IGBTs. If the IGBT temperature reaches an unsafe level the converter will enter standby in order to protect the converter. Finally, if the communication to the vehicle system controller is lost for more than 250ms the converter enters standby and waits for the communication link to be reestablished.

**Table 2 List of Implemented Fault Protections**

Over voltage low side
Over voltage high side
Over current in the inductor
Over temperature
Loss of communication

### DCM Explained

In level 2 control the mechanism for allowing the converter to operate in DCM was discussed. Dependent on which direction the converter is set to push power, a switch is deactivated, utilizing the anti paralleling diode to block the negative current. However, if the inductance value and switching frequency are not designed properly, the converter could still operate in CCM.

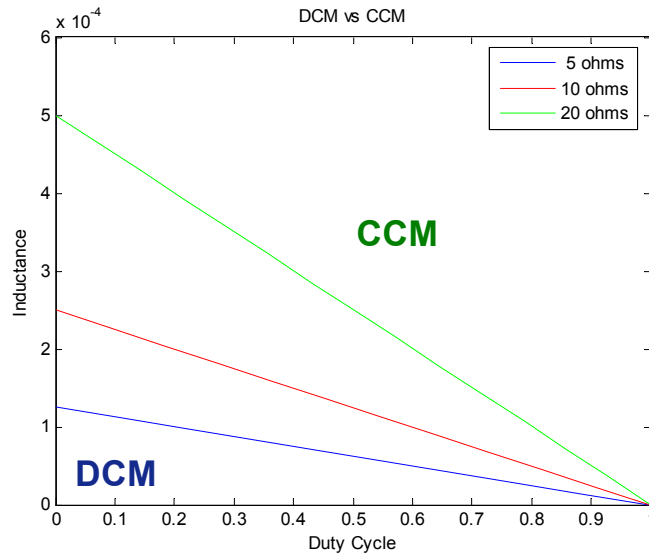
The minimum inductance value needed to insure the converter operates in CCM is known as the critical inductance value. For the buck and boost converter the critical inductance value is dependent on the steady state duty cycle, switching period and the load resistance. The equation for the critical inductance for the buck converter can be found in Equation 16.

$$L_{critical} = \frac{1 - D}{2} \cdot T_s \cdot R_{Load}$$

**Equation 16**

The switching frequency of the converter is set to 20 kHz. This is mainly do to the maximum frequency driving capabilities of the driving circuit on the physical hardware. Assuming a constant switching frequency of 20 kHz, the critical inductance value is plotted over the full range of duty cycles for different load resistors, Figure 35. It is important to note that the DCM region for the lower valued resistors is always included in the DCM region for higher valued resistors. This means if the inductance value is designed to operate in DCM at the highest possible load, then the converter will also

operate in DCM for any load lighter load condition. This is how the inductance value is designed for this converter.

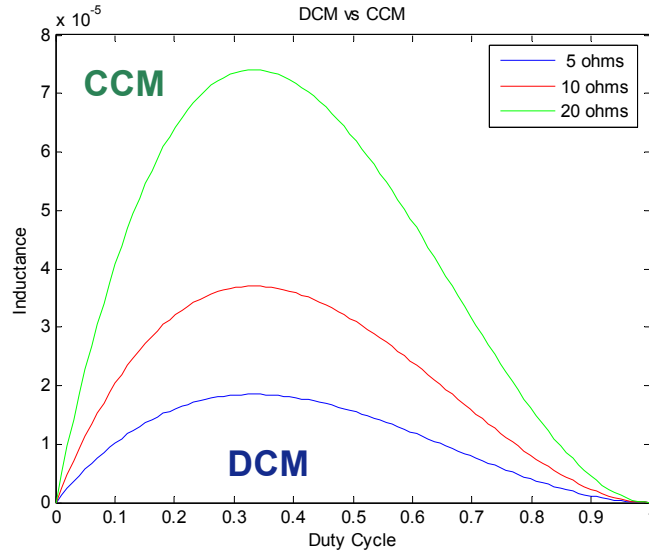


**Figure 35 Buck Converter Critical Inductance Plot**

Similarly, the equation for the boost converter’s critical inductance is examined, Equation 17. The plot of the critical inductance value for the boost converter shows similar characteristics in Figure 36. The DCM region for higher load conditions is included in the region for lighter load conditions. The minimum inductance value from the two methods is then chosen to insure the converter operates in DCM during all modes of operation.

$$L_{critical} = \frac{T_s \cdot R_{Load}}{2} \cdot (1 - D)^2 \cdot D$$

**Equation 17**



**Figure 36 Boost Converter Critical Inductance Plot**

The calculation of the inductance value was based on the fact that the inductor current must always operate in DCM. The final value for the three experimental inductors was 140  $\mu\text{H}$ . The battery side capacitor value was 160  $\mu\text{F}$  with a low ESR. The high side bus capacitor was 1000 $\mu\text{F}$ .

### Communication Interface (Level 3 Control)

The third level of control is assumed to be the vehicle system controller. While this controller is not designed here, it is necessary to simulate inputs from the controller in order to properly send commands to the second level of control. To do this, an interface was developed to send these commands and reference values through the CAN bus. CAN stands for communication area network.

The CAN module in the DSP was implemented to retrieve the information. The module was set up for a communication rate for 250 kbps. The module utilizes a number of mailboxes to transmit and receive information with little intervention from the central



processing unit (CPU). This is important, because the CPU is responsible for a number of time critical operations such as the level one and level two controls. The mailboxes are can transmit or receive 8 bytes at a time. In order to transmit all of the information it was split up into 4 mailboxes: 2 for sending and 2 for receiving. One byte of information is used to transmit the current status of the converter. This contains information about the mode of operation and running state of the converter. Five more bytes in that same mailbox are used to transmit the measurement for the temperature, high side bus voltage, and the status of all the faults. Each fault is assigned a bit location. If that bit is high, it indicates the fault has occurred, if it is low it means the fault has not occurred. The measurements for the three inductor currents and the total average current are sent in eight bytes of another mailbox. Similarly, the commands received by the converter occupy one byte of another mailbox. Four more bytes in the same mailbox are used to transmit the low side and high side bus voltage set points. The remaining byte is left empty. In another mailbox the current set point for the average buck and boost current is sent in four bytes. The break down of all the communication can be found in the following tables.



High side bus voltage (HSVBUS)

15 8



**Bits Name Description**

15:0 HSVBUS High Side Bus Voltage Regulation/Limit: 0-725V  
 Regulation set point in EV  
 Voltage limit for Hybrid Boost

Data values (DATA) – Mailbox 2

Average Current Command Buck (AVGBUCK)

15 8



**Bits Name Description**

15:0 AVGBUCK Average Current Command Buck (Low side): 100A  
 Current Commands are Current LIMITS when in EV  
 mode

Average Current Command Boost (AVGBOOST)

15 8



HSTEMP

**Bits Name      Description**

7:0 HSTEMP Heat sink temperature: -40C-120C

High side DC Bus voltage (HSVBUS)

15 8

HSVBUS

7 0

HSVBUS

**Bits Name      Description**

15:0 HSVBUS High Side DC Bus voltage: 0-900V

Faults (FAULTS.ALL)

15 8

FAULTS.ALL

7 0

FAULTS.ALL

**Bits Name              Description**

15:0 FAULTS.ALL A list of all faults masked into 16 bits

Data values (DATA) – Mailbox 4

Average low side current (AVGCURLOW)

15

8



7

0



<b>Bits</b>	<b>Name</b>	<b>Description</b>
15:0	AVGCURLOW	Average current determined by internal sensors (low side): -100/+100A

Average current Phase 1 (AVGCURPH1)

15

8



7

0



<b>Bits</b>	<b>Name</b>	<b>Description</b>
15:0	AVGCURPH1	Average Phase 1 current: -100/+100A

Average current Phase 2 (AVGCURPH2)

15

8



7

0



<b>Bits</b>	<b>Name</b>	<b>Description</b>
15:0	AVGCURPH2	Average Phase 2 current: -100/+100A

Average current Phase 3 (AVGCURPH3)

15

8



7

0



<b>Bits</b>	<b>Name</b>	<b>Description</b>
15:0	AVGCURPH3	Average Phase 3 current: -100/+100A

To help code and decode the mailboxes in the DSP, custom data types were designed to fill the proper memory space locations with the corresponding information. The data types consist of a struct of all relevant bits. This struct allowed the access of single bits without affecting other bits in the memory location. This is important to speed up coding time and execution time. The struct memory space was then joined in a union with a memory location that takes covers all of the bit locations. This allows the ability to affect all of the bits with a single instruction. For instance, the resultant data type could be used to set the fault bits individually with a single instruction, while still being able to clear all bits with a single instruction. These data types can be found in APPENDIX B.

### User Interface

The user interface allows the simulation of the third level of control. The third level of control is intended to be the vehicle system controller. Since the vehicle system controller is not designed here, there must be a way to simulate inputs and test the converter under different operating conditions.

Two versions of the interface were developed. The first version allows direct control of all of the parameters seen in Figure 37. These parameters include low voltage side set point, high voltage side set point, average inductor current set point in the buck direction, average inductor current set point in the boost direction, state, and mode. The command state of the converter includes run, standby, and reset. The modes include electric vehicle mode, hybrid boost vehicle mode, and hybrid buck vehicle mode. All commanded values are set on the left side of the interface. The data received back from



the converter is displayed on the right side of the interface. This includes the current state and mode of the converter. As well as the high voltage side, temperature, average inductor current measurements.

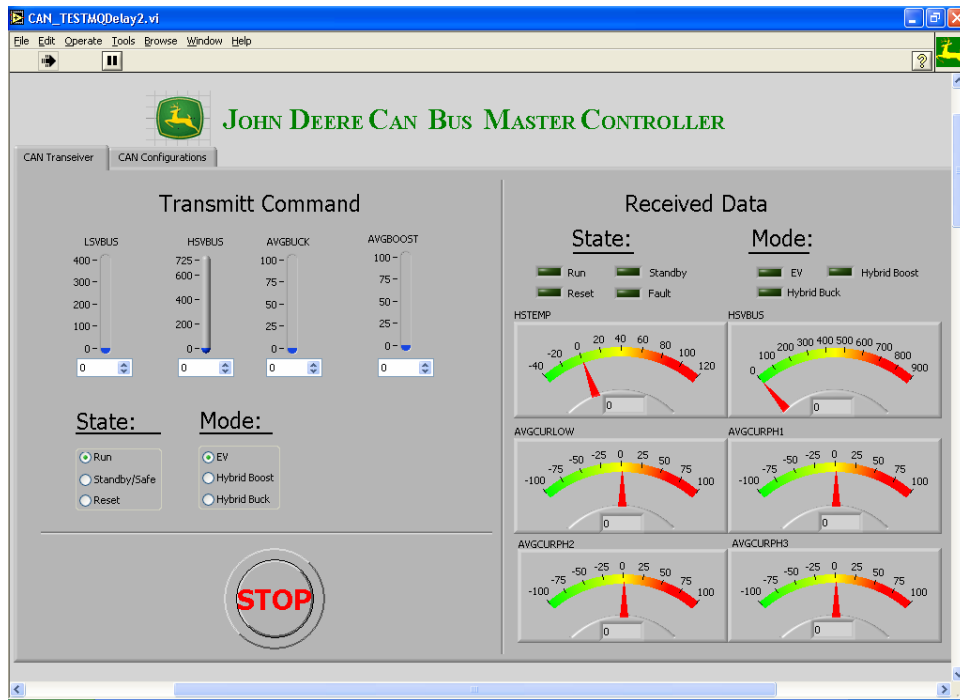


Figure 37 User Interface

The second version of the user interface is almost identical to the first versions in appearance. All received data is displayed in the same way as the first version, however the commanded values can not be set in the user interface directly. A pre compiled tab delimited file is generated prior to running the converter. The file contains all of the commanded values for a given time step size. In this case all commanded values are updated every 100ms, since the converter expects to receive new information on this time interval. This is very important if it is desired to test the converter when the input

commands from the level three control is updated much faster than a human could enter them into the interface. The test running capable interface can be seen in Figure 38.

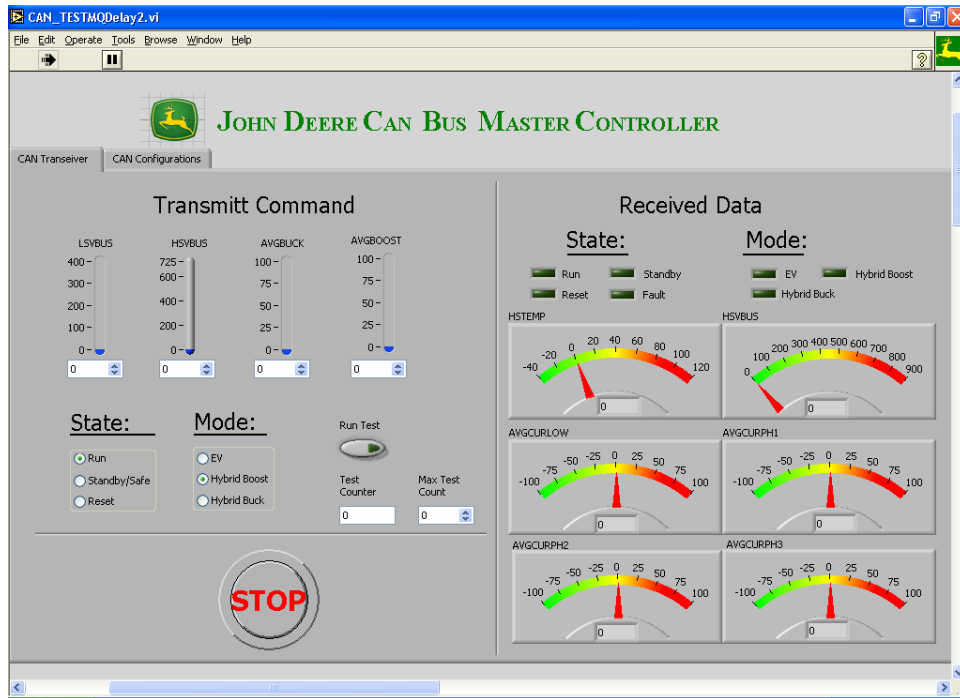
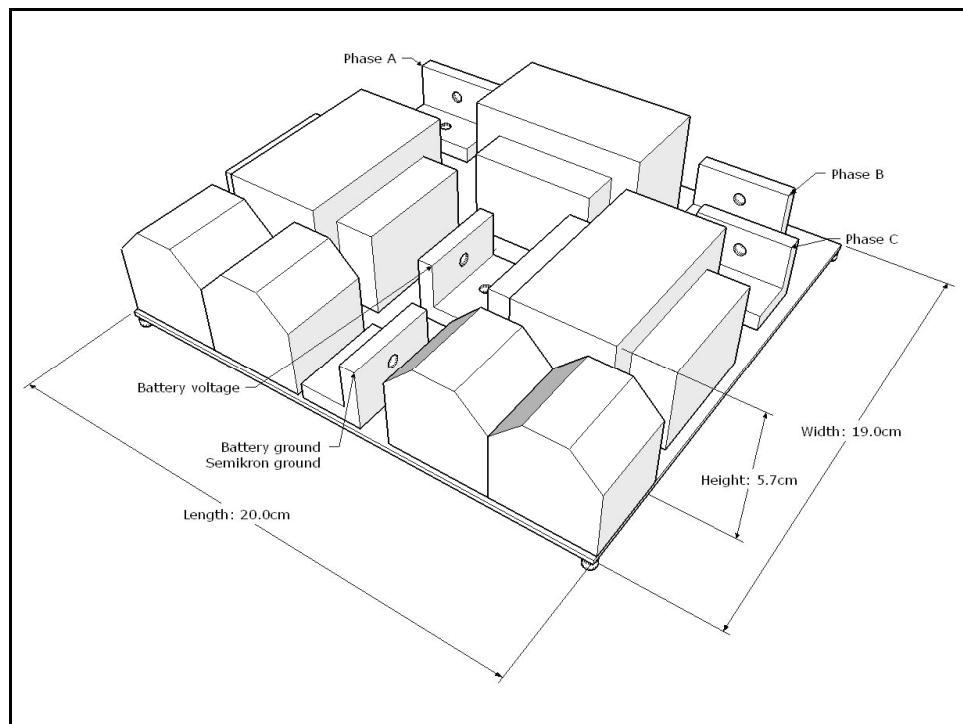


Figure 38 User Interface with Test Running Capabilities

## CHAPTER FOUR: EXPERIMENTAL RESULTS

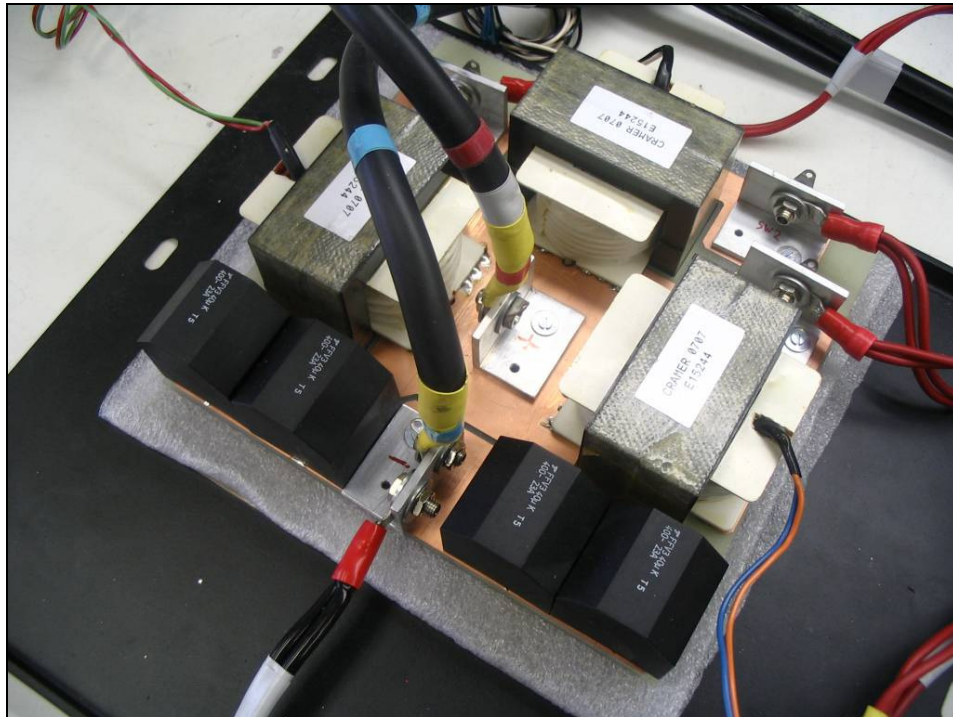
### Converter Prototype

A 3D-model was constructed of the input filter and used to help build the entire board assembly including, components, devices, battery connections and the rest of the DC-to-DC Converter. The power filter board was designed to minimize the space requirements. The fact that the power filter components are designed on a separate board allows the ability to store them in a separate physical location than the Semikron unit. This allows for the separation of the size requirement for the entire converter. Since heat dissipation is always an issue in the automotive world, the power filter board was also designed with notches cut in the PCB material to allow the inductor core material to protrude through. This allows for direct cooling of the core material on the top and bottom. The 3D-model and prototype is shown in Figure 39.



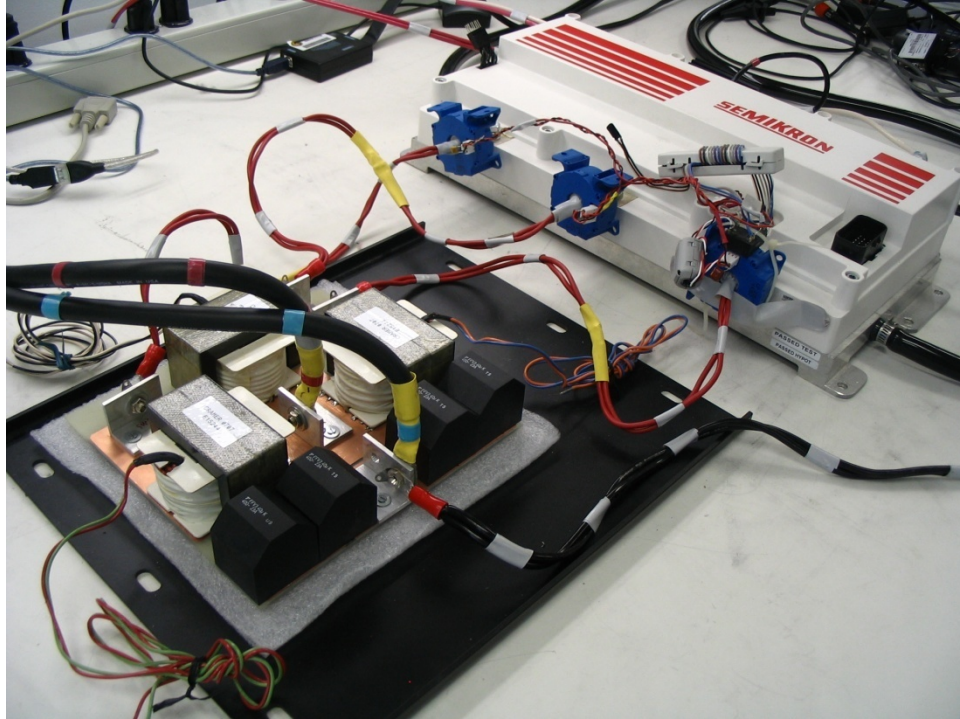
**Figure 39 3D-Design of Power Filter Board**

It can be seen in that the actual prototype very closely resembles the 3D-model as shown in Fig. 39. The two large black wires are connected to the battery pack at the low voltage side. The small black wire at the bottom of the Figure 40 is the common ground connection shared with the DC-to-DC converter. Finally, the three red wires seen in the top of the figure are connected to the three phases on the Semikron unit.



**Figure 40 Power Filter Board Prototype**

The final prototype can be seen in Figure 41. The Semikron unit can be found in top right hand side of the figure. The Semikron unit is equipped with three internal hall effect current sensors for each phase. However, these sensors were bypassed by the three blue external hall effect sensors seen in the Figure 41. This was done to increase the resolution of the current measurement for the inductor current regulation loop discussed in Chapter 3.



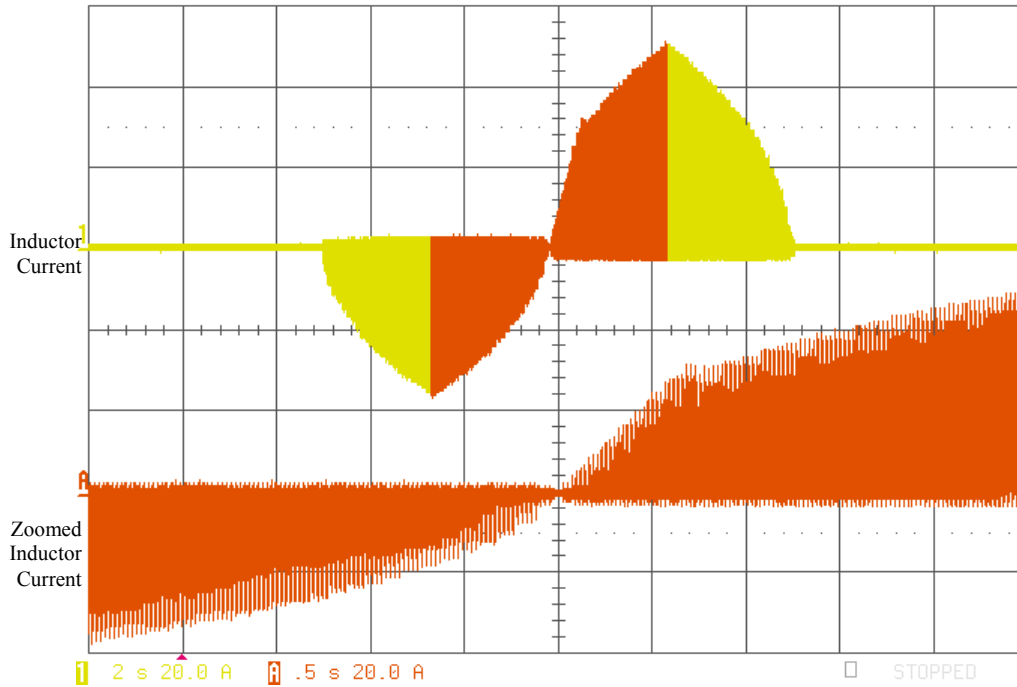
**Figure 41 Final Prototype**

### Experimental Results

All control loops and modes of operation were implemented on the prototype mentioned above. The bi-directional capabilities were tested in both Hybrid Mode and Electric Vehicle Mode. All control loops were tested under a range of load conditions and voltage configurations.

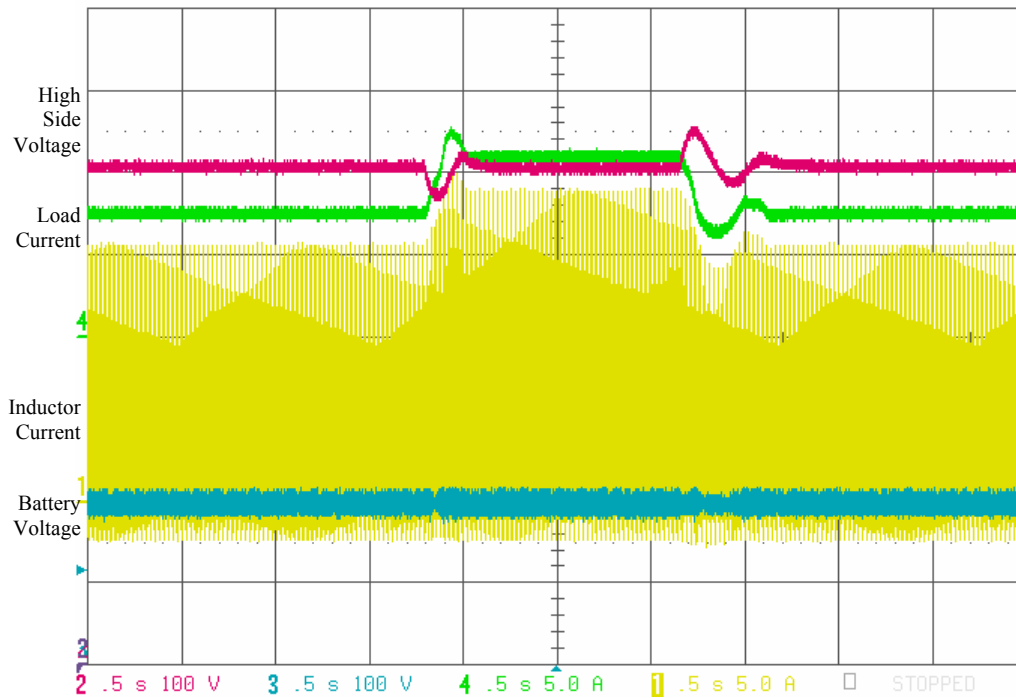
In Figure 42, the converter is first held in standby and then ramped up to full power in the buck direction by increasing the current reference set point in the buck direction. Next the reference is ramped back down to zero. The mode then changes to the boost direction and the reference is ramped to full power and back down again. This tests an extreme case for the converter that is usually not seen in normal operation. One instance that the converter could experience is during a massive regenerative braking

action followed immediately by a full throttle command. The yellow signal Figure 42 represents the inductor current of one of the phases, whereas, the orange signal is a zoomed in version of the inductor current to show how the transition between buck and boost is done in a seamless manner.



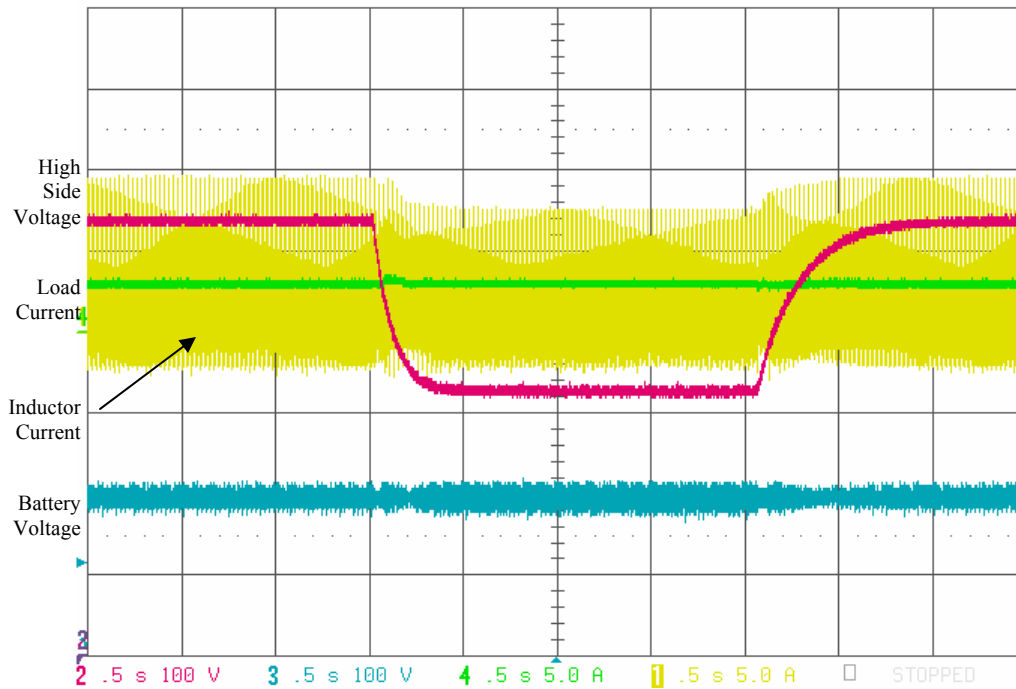
**Figure 42 Power Sweep of Buck and Boost Modes**

In Figure 43, the converter is operated in electric vehicle mode where the voltage on the high side is regulated to 700V, represented by the red signal. Then the load was increased to represent a higher demand from the motor drive system. This effect can be seen in the inductor current in yellow and the battery current seen in green. It can be seen how the voltage controller returns to the regulated set point after a transient period. The load was then reduced to simulate less of a demand from the motor drive system. The blue signal represents the input voltage from the batteries



**Figure 43 Bus Voltage Regulation During Load Transient**

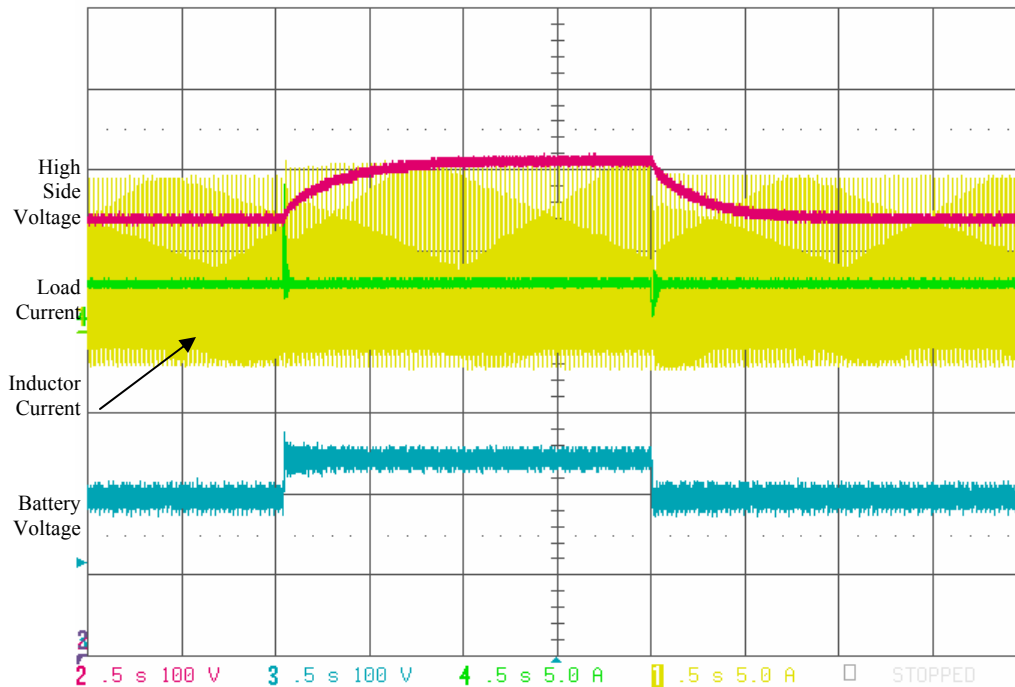
In Figure 44, the converter is running in hybrid boost mode and the average inductor current is regulated. The battery current is seen in green is representative of the average inductor current since it is the filtered inductor current. The load is then increased representing a higher demand from the motor drive system. In the actual system the high side voltage, seen in red, would be regulated by the generator discussed in previous chapters. However, in this test the voltage was left unregulated, so it can be seen how the voltage drops because of the increased load. After a short time interval, the load is returned to the previous value, simulating a reduction in demand from the motor drive system. It can be seen how average inductor current is regulated during the load steps. The blue signal represents the battery voltage.



**Figure 44 Inductor Current Regulation During Load Steps**

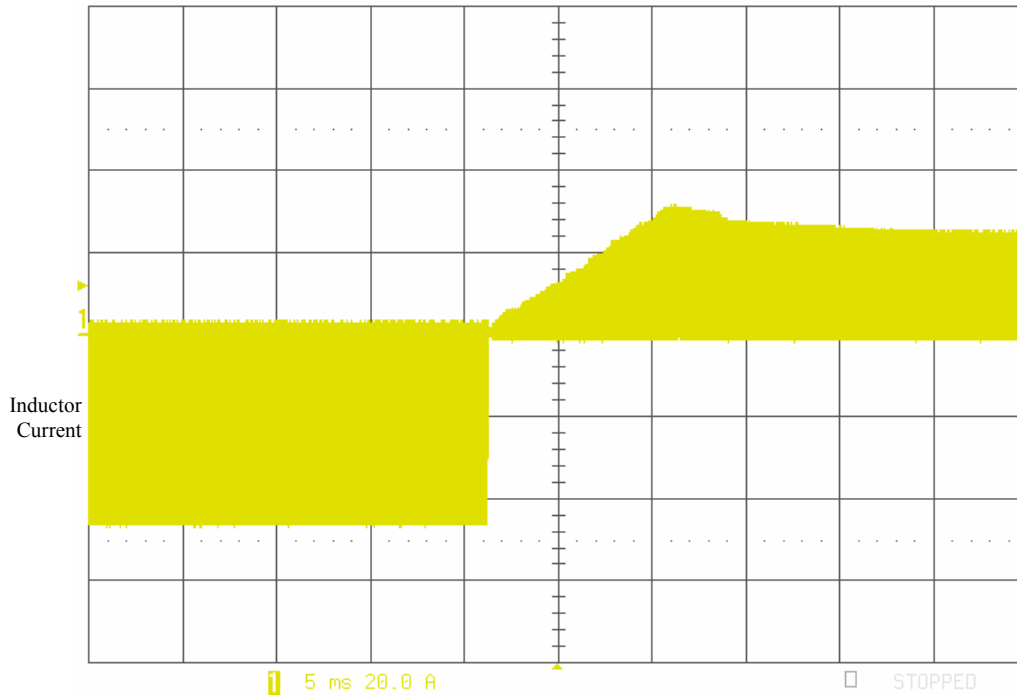
In Figure 45, the converter is running in hybrid boost mode, where the average inductor current is regulated seen in green. Next, the input voltage, seen in blue, is stepped from 200V to 250V to simulate fluctuations in the battery voltage. In the real system the fluctuations would never be this dramatic however testing under this extreme case verifies the converter can maintain regulation during changes in the input voltage.





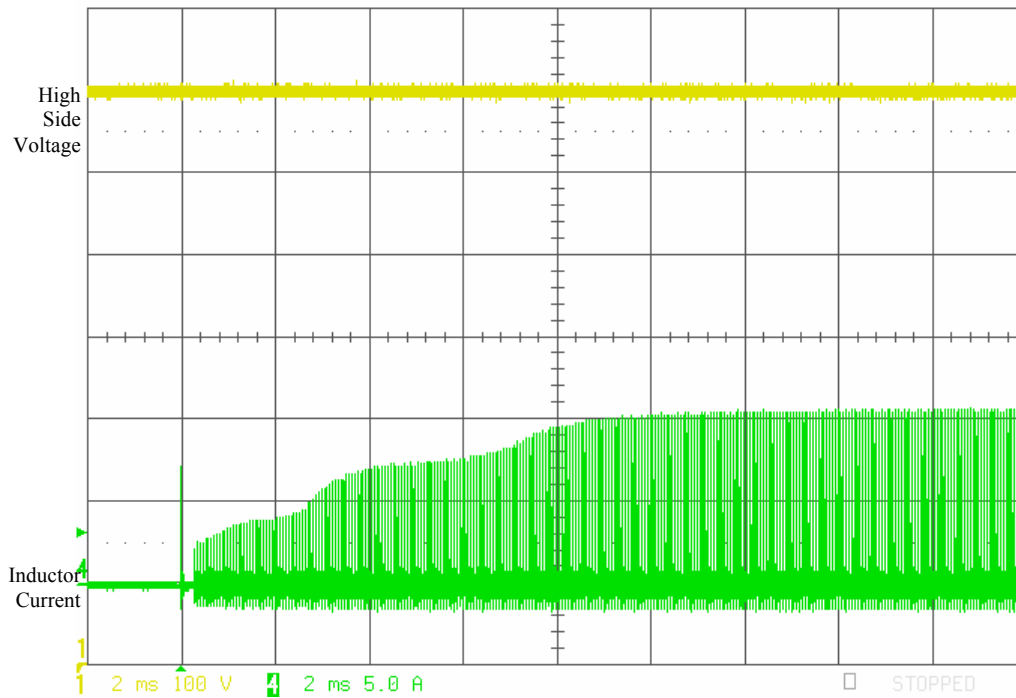
**Figure 45 Inductor Current Regulation During Input Voltage Steps**

In Figure 46, the converter is initially running in hybrid buck mode. The average inductor current is regulated as shown by the yellow waveform, the inductor current of one phase, since the peak current is held constant. Then, from the user interface, the converter is commanded to hybrid boost mode. After the command is received the converter switches to hybrid boost mode and ramps the current reference up to the commanded value in the boost direction. This simulates the vehicle system controller commanding the converter to stop charging the batteries and start pushing power to the motor drive system. It can be seen that in inductor current does indeed operate in DCM since the inductor current never goes negative with respect to the mode.



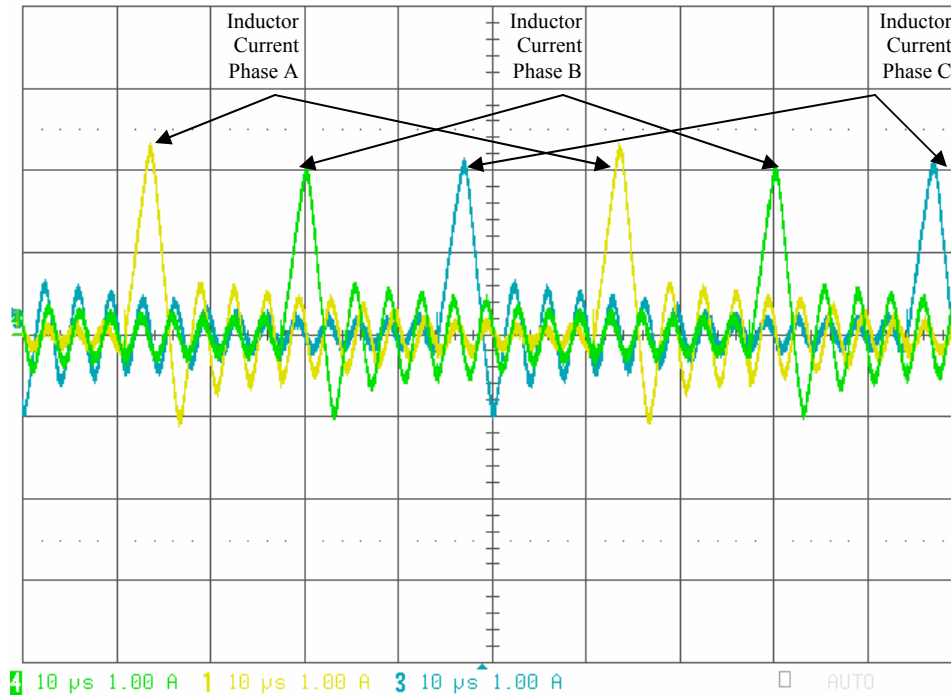
**Figure 46 Commanded Mode Change Buck to Boost**

In Figure 47, the converter is initially in standby. From the interface the converter is command to hybrid boost mode. The average inductor current set point is ramped up to the regulation set point, seen in the green waveform. In this test the voltage on the high side is regulated by an external voltage source tied in parallel with a resistive load bank, seen in the yellow signal. The method of ramping the reference up to the set point is known as soft starting. This helps to reduce high peak demands from the converter during start up and makes the start up a much smoother process.



**Figure 47 Hybrid Boost Start Up**

The inductor current of each of the three phases can be seen in Figure 48. It is shown how the inductor current is properly interleaved by spacing out the inductor current waveforms equally over the switching period. The converter is running in hybrid boost mode with a small reference for the inductor current regulator. The additional ringing seen when the inductor current should be zero is due to the added parasitic capacitance in the IGBT module. This is common and seen in all converters that operate in DCM.



**Figure 48 Experimental Inductor Current Interleaving**

In Figure 49, the converter is running in hybrid boost mode with a high set point for the inductor current regulator. The load seen on the high side is initially sufficient enough to make the output voltage less than 800V. Then the load is taken away. This simulates an initial high demand from the motor drive system and then a sudden loss of load. The converter continues to try and regulate the current until the voltage reaches the over voltage regulation set point. However, in this case the rise in voltage is so dramatic, seen in red, the output voltage regulator is not fast enough to respond and regulate a safe over voltage value. The voltage then reaches the protection level of 800V where the converter detects this and shuts the converter down, putting it in standby.



## CHAPTER FIVE: CONCLUSION

Since the introduction of modern hybrid electric vehicles in 1899 [10], there have been many major advances in technology in terms of devices, ICs, DPS and circuits. A number of topologies have been discovered and implemented in today's hybrid electric vehicles. From these current topologies a subset of topologies has been classified and evaluated in this work. It has been found that an addition of a bi-directional DC-to-DC converter has much to benefit for this subset of topologies.

The advantages of a DCM converter were discussed to reduce the physical size of the inductors. This is an important design criterion since space is a limited commodity in the vehicle world. The converter design presented in this work is applicable for any subset of the topologies discussed here in which DCM operation is required.

A series of operational modes were discussed, derived from real world operating conditions present in the subset of hybrid electric vehicle topologies. Together, these modes along with the requirement for DCM operation define functions necessary for the level two controller. Complex functions are used to control the driving of the switches in a way that guaranties DCM operation while still maintaining the bi-directional power flow of the converter. The second level of control was designed to be open-ended, so the vehicle system controller would be able to send generalized commands to the converter. This allows the converter to operate in safe manner without to much intervention from the vehicle system controller. This is important since the vehicle system controller is relatively slow compared to the system dynamics.

Keeping true to the hierarchical design of the controllers, the first level of control accepts commands from the second level of control. The first level of control is comprised of all control loops which directly control the sensed parameters. This includes inductor current regulation in buck and boost modes, high voltage side regulation, and low voltage side regulation. All controllers were designed using a direct digital designed and implemented in a DSP.

Two versions of the computer interface were developed to simulate inputs from the third level of control. This allowed the test of the converter under all real world cases. The user interface also provided a means to acquirer and display digital data already present in the DSP. While the first version of the interface allows the direct control over all system parameters in real time, the second version of the interface allowed for the use of precompiled test scenarios to be run.

A system of fault protections were identified and implemented into the digital controller. While hierarchical design is intended to protect the converter from unsafe operations, sometimes these situations are unavoidable. In this case, it is important to have a series of protections to prevent damage to the converter and surrounding components.

The use of interleaving was investigated and shown how it can alleviate the added voltage ripple present in DCM operated converters. After analyzing the PWM modules preset in the implemented DSP, a method was developed to properly interleave the inductor currents in the three paralleled converters. The interleaved inductor currents were then verified experimentally

The system as a whole was designed as a drop in unit for today's hybrid electric vehicles. Since a converter with these advanced controller techniques is not available for the automotive world, this converter is a major advancement for this application.



## **APPENDIX A: FLOW CHARTS**

Main \_\_\_\_\_ This is where the code first starts, initializes modules, and shows the branch for the main controller interrupt.

Initialize DSP \_\_\_\_\_ Steps to initialize the DSP

Call ADC \_\_\_\_\_ Function used to trigger ADC and average results.

Control Manager \_\_\_\_\_ Controller structure

Load Values from CAN \_\_\_\_\_ Fetches current results from the CAN

Transmit Values to CAN \_\_\_\_\_ Sends updated values to the CAN

Check for Faults \_\_\_\_\_ Checks any new fault conditions

Fetch State \_\_\_\_\_ Determines the appropriate state of the converter.

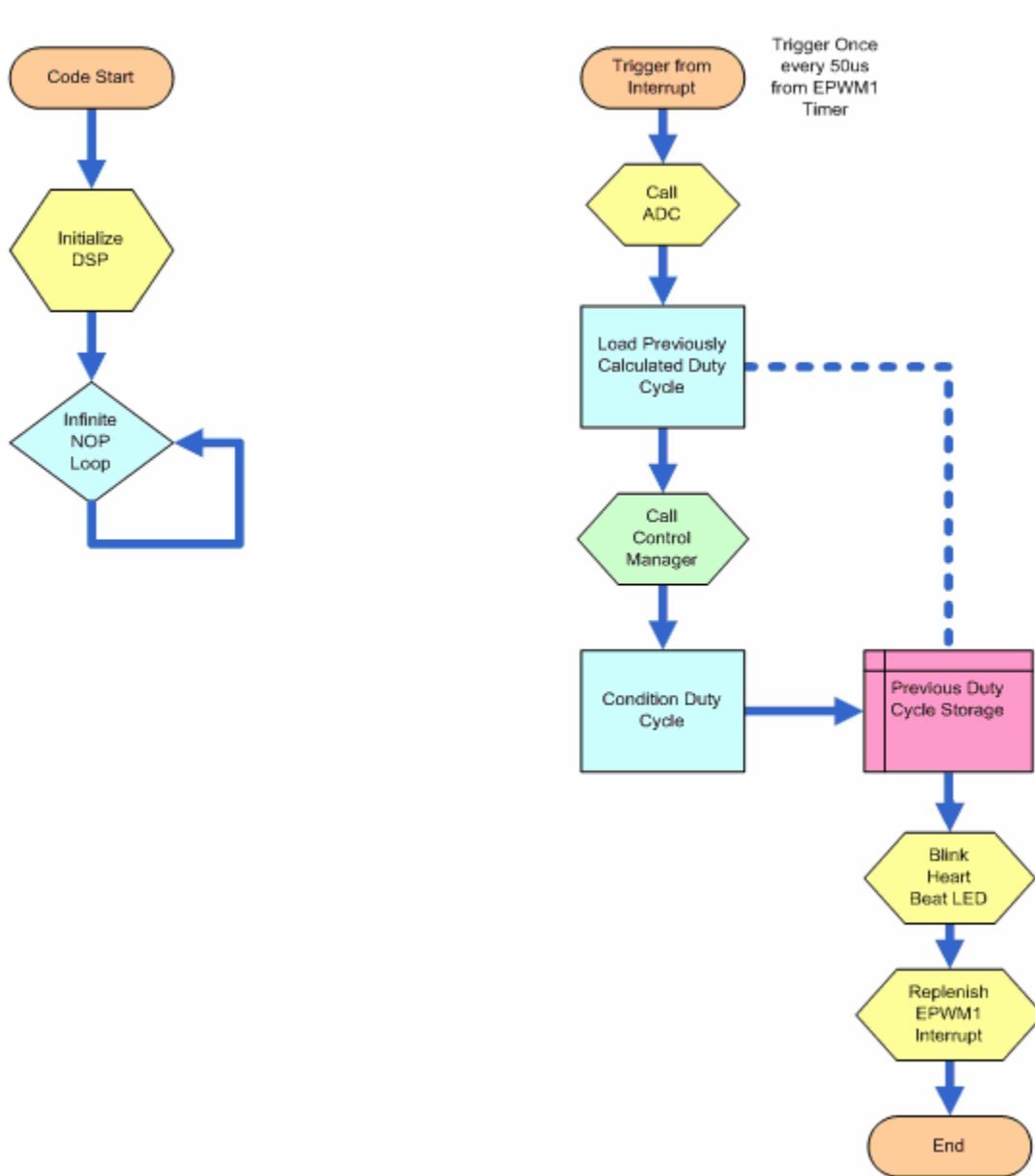
Standby Handler \_\_\_\_\_ Functions need to hold converter in standby mode

Set Polarity \_\_\_\_\_ Sets the polarity of the switches

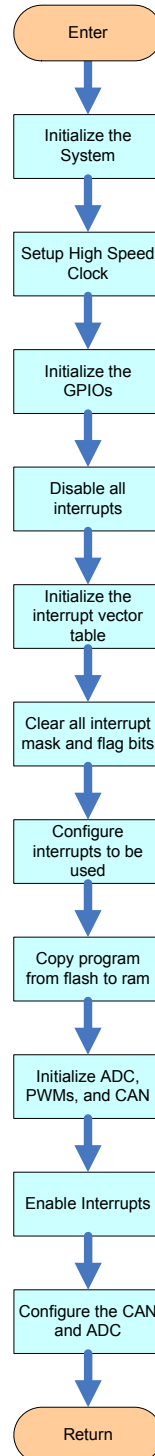
Reset Handler \_\_\_\_\_ Functions needed to hold the converter in reset mode

Run Handler \_\_\_\_\_ Functions needed to hold the converter in run mode

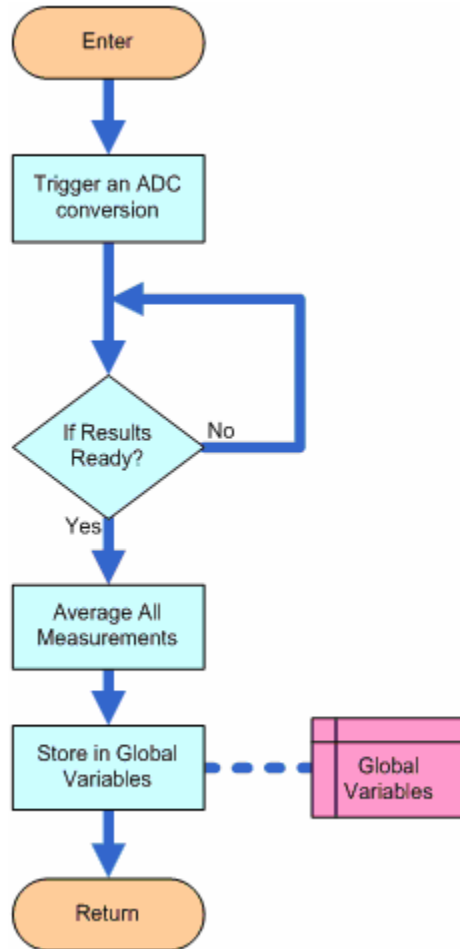
# Main



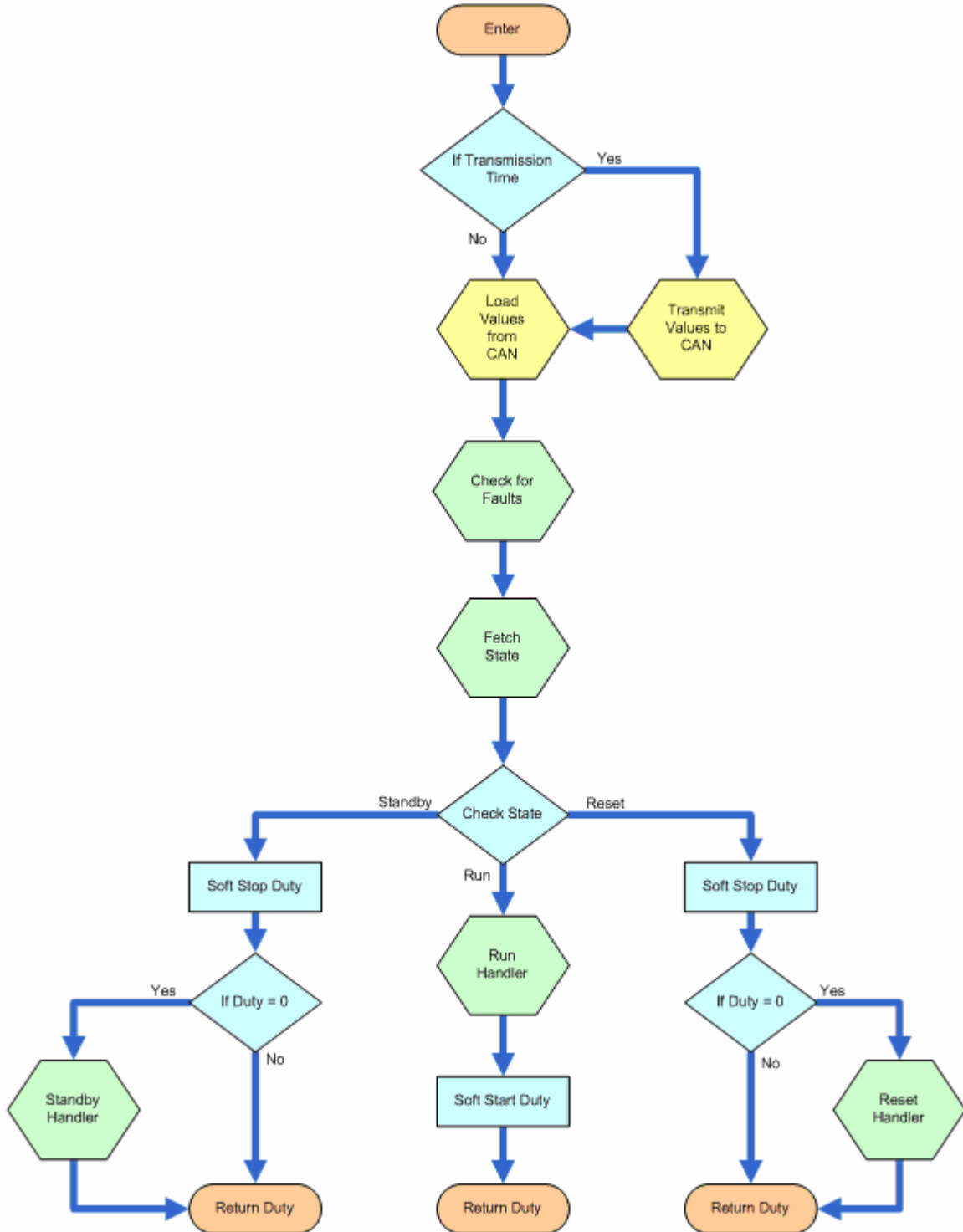
## Initialize DSP



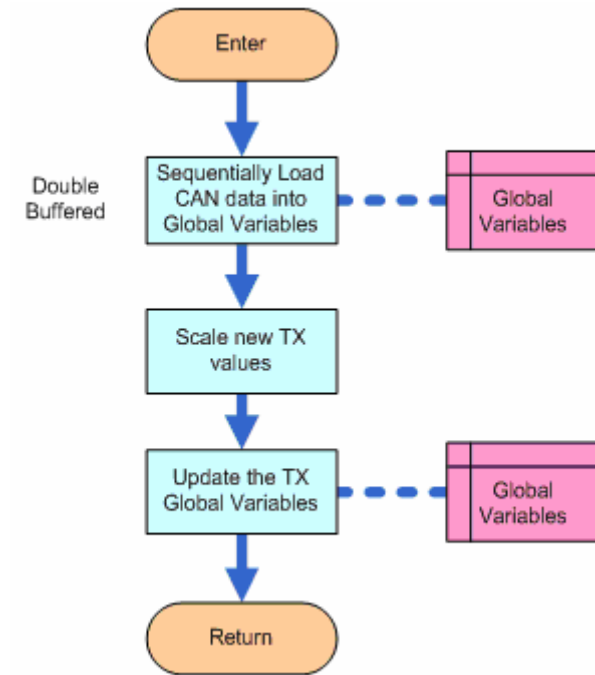
## Call ADC



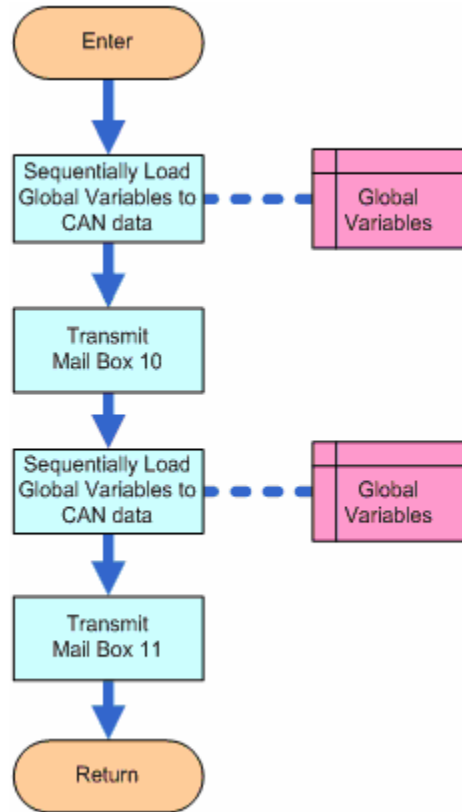
## Control Manager



## Load Values from CAN

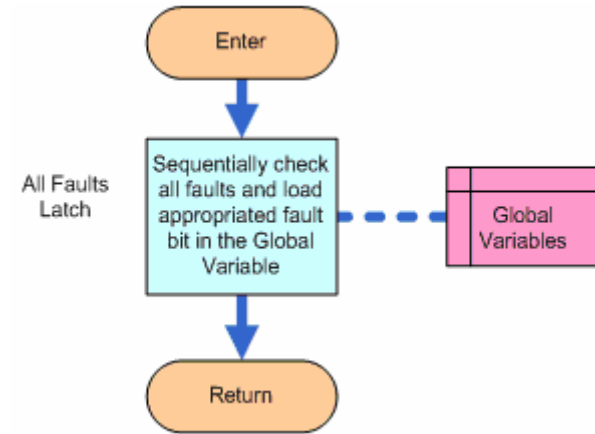


## Transmit Values to CAN

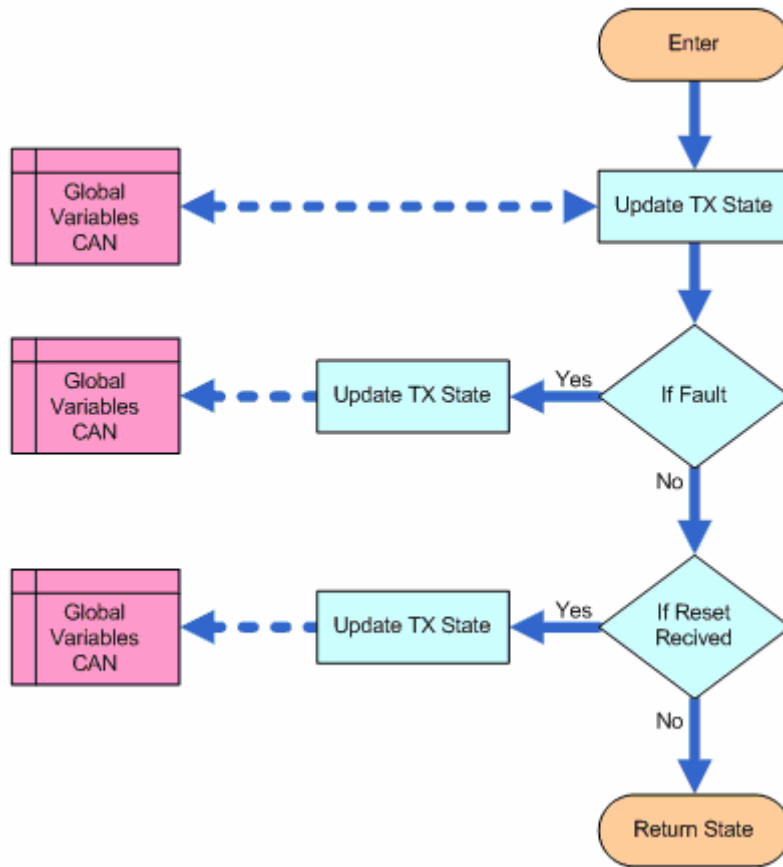




## Check for Faults



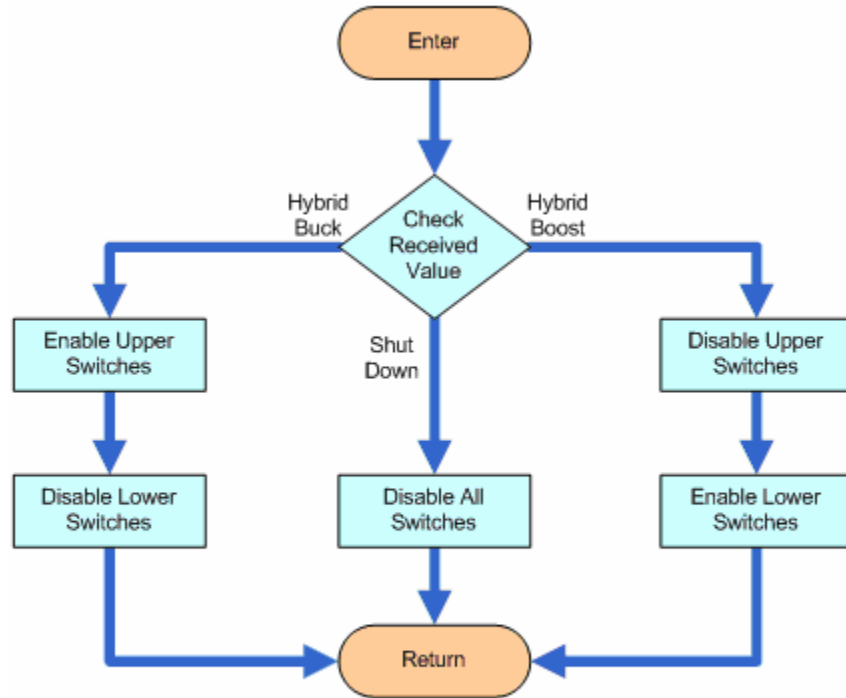
## Fetch State



## Standby Handler

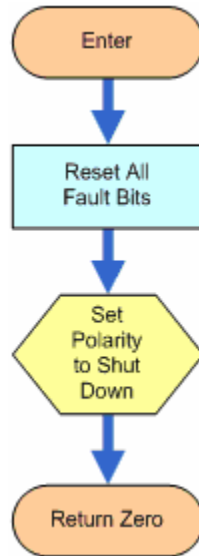


## Set Polarity

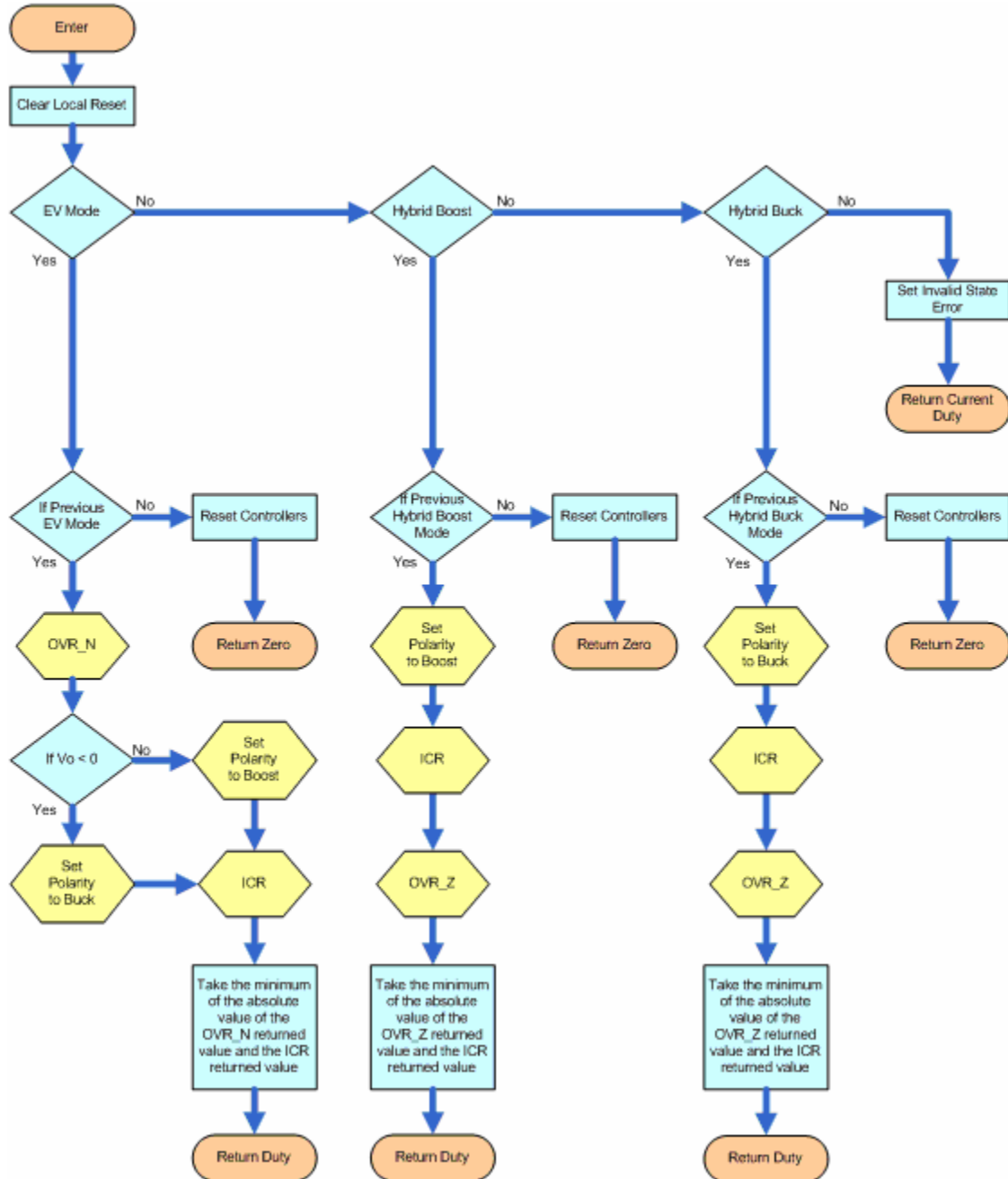


This function was called from multiple locations, therefore it will return you to the Code Start page

## Reset Handler



## Run Handler



## **APPENDIX B: DSP CODE**

Settings.h \_\_\_\_\_ Contains defines used to easily adjust parameters in  
the code

Sys\_fun.h \_\_\_\_\_ Function prototypes for DSP specific functions

JD\_PowerUnit.h \_\_\_\_\_ Function prototypes for application specific  
functions

Sys\_fun.c \_\_\_\_\_ Function definitions for DSP specific functions

JD\_PowerUnit.c \_\_\_\_\_ Function definitions for application specific  
functions



## Settings.h

```
#ifndef SETTINGS_H
#define SETTINGS_H

// adc pin definitions
#define HS_VBUS_PIN          0x7
#define LS_VBUS_PIN          0xF
#define IL_AVG1_PIN          0x0
#define IL_AVG2_PIN          0x1
#define IL_AVG3_PIN          0x2
#define HS_TEMP_PIN          0xA
#define SPARE_ADC_PIN        0xD

// define maximum values for faults
800 V #define HS_VBUS_MAX          (const long)800*59          //
300 V #define LS_VBUS_MAX          (const long)300*20          //
#define IL_BUCK_AVG_MAX      (const long)35*1200          // 35 A
#define IL_BOOST_AVG_MAX     (const long)65*1200          // 65 A
#define HS_TEMP_MAX          (const long)100*143+32476
//100 C
#define SPARE_ADC_MAX        1000

// timer definitions
#define TPERIOD                2500 //2500=20kHz //2940 = 17kHz
.. clk 50MHz
#define ADC_PERIOD              100 //500
// 100MHz/1MHZ = 1000/2 ? why need /2
#define DUTY_MAX                1875 // 1875 = 75% duty
#define DUTY_MIN                 5

// controller settings
#define VB_REF_LIM                (long)300*20
#define VO_REF_LIM                (long)725*20*3
#define IO_REF_LIM                (long)65*20*60

#define VSET                      0x7ffff
#define ISET                      0x7ffff

#define IGAIN                      1
#define VGAIN                      3

#define INT_SAT_UPPER              (long)DUTY_MAX<<8 //629146

// CAN addresses
#define CAN_ADD_1                  10
#define CAN_ADD_2                  11
#define CAN_ADD_3                  10
#define CAN_ADD_4                  11

// miscellaneous
#define true                        1
#define false                       0
```

```

// Offsets
#define HS_VBUS_OFFSET          90
#define HS_TEMP_OFFSET         27655
#define CUR_OFFSET1            32980
#define CUR_OFFSET2            32960
#define CUR_OFFSET3            32620
#define CUR_OFFSET              (long) (CUR_OFFSET1+CUR_OFFSET2+CUR_OFFSET3)

#endif

```

### Sys\_fun.h

```

// header file for sys_fun.c
#ifndef SYS_FUN_H
#define SYS_FUN_H

#define FLASH // comment this line and use RAM linker
file to program to RAM

// mask adc conversion results
#define HS_VBUS_R1 AdcRegs.ADCRESULT0
#define HS_TEMP_R1 AdcRegs.ADCRESULT1
#define HS_VBUS_R2 AdcRegs.ADCRESULT2
#define HS_TEMP_R2 AdcRegs.ADCRESULT3

#define IL_AVG1_R1 AdcRegs.ADCRESULT4
#define IL_AVG2_R1 AdcRegs.ADCRESULT5
#define IL_AVG3_R1 AdcRegs.ADCRESULT6
#define IL_AVG1_R2 AdcRegs.ADCRESULT7

#define IL_AVG2_R2 AdcRegs.ADCRESULT8
#define IL_AVG3_R2 AdcRegs.ADCRESULT9
#define IL_AVG1_R3 AdcRegs.ADCRESULT10
#define IL_AVG2_R3 AdcRegs.ADCRESULT11

#define IL_AVG3_R3 AdcRegs.ADCRESULT12
#define IL_AVG1_R4 AdcRegs.ADCRESULT13
#define IL_AVG2_R4 AdcRegs.ADCRESULT14
#define IL_AVG3_R4 AdcRegs.ADCRESULT15

#define min(a,b) (a<b?a:b)
#define max(a,b) (a>b?a:b)
#define limit(a,b,c) min(max(a,b),c)

// function prototypes for system related functions
void sys_init();
void adc_replenish();
void epwm_replenish();
void InitEPwm1Example(void);
void InitEPwm2Example(void);
void InitEPwm3Example(void);
void InitEPwm4Example(void);
unsigned polarity(unsigned int);
void load_can_data();

```

```

void trans_can_data();
unsigned int current_duty(void);
signed long OVR_Z(signed long, long, int);
signed long OVR_N(signed long, int);
signed long IVR(signed long, int);
signed long ICR(signed long int, int);
void heartBeat_LED(int);
void call_ADC();
// unsigned int freq_resp(unsigned int);

// redirect interrupt service routines
interrupt void adc_isr(void);
interrupt void epwm1_timer_isr(void);
interrupt void ecan0inta_isr(void);
interrupt void ecanlinta_isr(void);
interrupt void ecan0intb_isr(void);
interrupt void ecanlintb_isr(void);

// CAN stuff
extern void DSP280x_ECanConfig(void); //MQ
extern void DSP280x_ECanaConfig(void); //MQ
extern void DSP280x_InterruptsConfig(void); //MQ
extern void DSP280x_CANA_RX(int MBOXnumber); //MQ
extern void DSP280x_CANB_RX(int MBOXnumber); //MQ
extern void DSP280x_CANA_TX(unsigned char *Message, int
MBOXnumber); //MQ
extern void DSP280x_CANB_TX(unsigned char *Message, int
MBOXnumber); //MQ
extern void DSP280x_ECanbConfig(void); //MQ

// Custom structers ////////////////
////////////////////////Faults////////////////////////
struct FAULT_FLAGS_BITS { // bits description

    Uint16 soft_fault_0:1;
    Uint16 soft_fault_1:1;
    Uint16 soft_fault_2:1;
    Uint16 soft_fault_3:1;

    Uint16 COMM_ERR:1;
    Uint16 LS_OV:1;
    Uint16 TEMP:1;
    Uint16 IL_OC:1;

    Uint16 crit_fault_0:1;
    Uint16 crit_fault_1:1;
    Uint16 crit_fault_2:1;
    Uint16 crit_fault_3:1;

    Uint16 crit_fault_4:1;
    Uint16 crit_fault_5:1;
    Uint16 START_UP:1;
    Uint16 HS_OV:1;
};

```

```

union FAULT_FLAGS {
    Uint16          all;
    struct FAULT_FLAGS_BITS bit;
};
extern volatile union FAULT_FLAGS fault;

//////////CAN DATA//////////
struct CAN1_RX_CMD_BITS {

    unsigned char state:2;
    unsigned char mode:2;
};

union CAN1_RX_CMD {

    unsigned char all;
    struct CAN1_RX_CMD_BITS bit;
};

struct STATS_BITS {

    unsigned char STATE:2;
    unsigned char MODE:2;
};

union STATS {

    unsigned char all;
    struct STATS_BITS bit;
};

struct TEMP_STATUS {

    unsigned char HSTEMP;
    union STATS STATUS;
};

struct CAN_TX {

    Uint16 FAULTS;
    Uint16 HSVBUS;
    struct TEMP_STATUS other;
    Uint16 AVGCURLLOW;
    Uint16 AVGCURPH1;
    Uint16 AVGCURPH2;
    Uint16 AVGCURPH3;
};
extern volatile struct CAN_TX tx;

struct CAN_RX {

    Uint16 HSVBUS;
    Uint16 LSVBUS;
    union CAN1_RX_CMD cmd;
    int16 AVGCURBOOST;
    int16 AVGCURBUCK;
};

```

```

extern volatile struct CAN_RX rx;

extern volatile unsigned int comm_cnt;

// pototype
extern volatile unsigned int HS_VBUS;
extern volatile unsigned int LS_VBUS;
extern volatile unsigned int IL_AVG1;
extern volatile unsigned int IL_AVG2;
extern volatile unsigned int IL_AVG3;
extern volatile long int IL_AVG;
extern volatile unsigned int HS_TEMP;

#endif

```

### JD\_PowerUnit.h

```

// header file for JD_PowerUnit.c
#ifndef JD_PowerUnit_H
#define JD_PowerUnit_H

    // define states
    #define RUN 0
    #define STANDBY 1
    #define RESET 2
    #define FAULT 3

    // define modes
    #define EV 0
    #define HBOOST 1
    #define HBUCK 2
    #define SHUTDOWN 3

    // function prototypes for application related functions
    void check_faults();
    long int control_manager(void);
    unsigned int fetch_state(void);
    long int run_handler(void);
    unsigned int reset_handler(void);
    unsigned int standby_handler(void);

#endif

```

### Sys\_fun.c

```

// ApECOR
// John Deere Bi-Directional Power Unit
// Michael Pepper
// Septempber 27, 2006

#include "DSP280x_Device.h" // DSP280x Headerfile Include File
#include "DSP280x_Examples.h" // DSP280x Examples Include File

```

```

#include "JD_PowerUnit.h"           // Contains application specific
information
#include "sys_fun.h"               // Contains converter specific
information
#include "settings.h"             // Settings and general defines

#include "DSP280x_CAN_GlobalVariableDefs.h" //MQ: DSP28 MailBoxes
Data Messages File

#ifndef FLASH
#include "DSP280x_PieVect.h"      // MQ
#endif

#ifdef FLASH
// Murad Qahwash: These are defined by the linker (see F2808.cmd)
extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsLoadEnd;
extern Uint16 RamfuncsRunStart;

// Functions that will be run from RAM need to be assigned to
// a different section. This section will then be mapped using
// the linker cmd file.
#pragma CODE_SECTION(ecan0inta_isr, "ramfuncs");
#pragma CODE_SECTION(ecanlinta_isr, "ramfuncs");
#pragma CODE_SECTION(ecan0intb_isr, "ramfuncs");
#pragma CODE_SECTION(ecanlintb_isr, "ramfuncs");
#pragma CODE_SECTION(epwml_timer_isr, "ramfuncs");
#pragma CODE_SECTION(adc_isr, "ramfuncs");

//End MQ
#endif

void DSP280x_CANA_TX();

void main(void)
{
    sys_init();

    // Wait for interrupts
    for(;;)
        ;
}

interrupt void ecan0inta_isr(void)
{
    // Insert ISR Code here
    int iMBox;
    iMBox = ECanaRegs.CANGIF0.bit.MIV0 ;

    // michael added
    comm_cnt = 0;

    //

    /* Begin Receiving */
    while(ECanaRegs.CANRMP.all != ((long)1<<iMBox)) {} //MQ: wait for
RMPi to be set. i = 16:31 "RX MBoxes"

```

```

        ECanaRegs.CANRMP.all |= iMBox;
        DSP280x_CANA_RX(iMBox);

    // To receive more interrupts from this PIE group, acknowledge this
    interrupt
        PieCtrlRegs.PIEACK.all = PIEACK_GROUP9;
    }

interrupt void ecanlinta_isr(void)
{
    // Insert ISR Code here
    int iMBox;
    iMBox = ECanaRegs.CANGIF1.bit.MIV1 ;

    // michael added
    comm_cnt = 0;

    /* Begin Receiving */
    while(ECanaRegs.CANRMP.all != ((long)1<<iMBox)) {} //MQ: wait for
    RMPi to be set. i = 16:31 "RX MBoxes"
        ECanaRegs.CANRMP.all |= iMBox;
        DSP280x_CANA_RX(iMBox);

    // To receive more interrupts from this PIE group, acknowledge this
    interrupt
        PieCtrlRegs.PIEACK.all = PIEACK_GROUP9;
    }

interrupt void ecan0intb_isr(void)
{
    // Insert ISR Code here
    int iMBox;
    iMBox = ECanbRegs.CANGIF0.bit.MIV0 ;

    /* Begin Receiving */
    while(ECanbRegs.CANRMP.all != ((long)1<<iMBox)) {} //MQ: wait for
    RMPi to be set. i = 16:31 "RX MBoxes"
        ECanbRegs.CANRMP.all |= iMBox;
        DSP280x_CANB_RX(iMBox);

    // To receive more interrupts from this PIE group, acknowledge this
    interrupt
        PieCtrlRegs.PIEACK.all = PIEACK_GROUP9;
    }

interrupt void ecanlintb_isr(void)
{
    // Insert ISR Code here
    int iMBox;
    iMBox = ECanbRegs.CANGIF1.bit.MIV1 ;

    /* Begin Receiving */
    while(ECanbRegs.CANRMP.all != ((long)1<<iMBox)) {} //MQ: wait for
    RMPi to be set. i = 16:31 "RX MBoxes"
        ECanbRegs.CANRMP.all |= iMBox;
        DSP280x_CANB_RX(iMBox);

```

```

    // To receive more interrupts from this PIE group, acknowledge this
interrupt
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP9;
}

interrupt void epwm1_timer_isr()
{
// Define local variables
    static signed int duty = 0;

// Trigger an ADC conversion
    call_ADC();

// Load previously calculated duty cycle
    EPwm1Regs.CMPA.half.CMPA = duty;
    EPwm2Regs.CMPA.half.CMPA = duty;
    EPwm3Regs.CMPA.half.CMPA = duty;

// Call control manager, returns new duty cycle values
    duty = control_manager();

// Duty cycle conditioning
    if(duty > DUTY_MAX)
        duty=DUTY_MAX;
    if(duty < DUTY_MIN)
        duty=DUTY_MIN;

// Blink the heart beat LED
    heartBeat_LED(0x7FF);

// Reset the pwm timer interrupt
    epwm_replenish();

    return;
}

void call_ADC()
{
// Start ADC and average results
    AdcRegs.ADCTRL2.bit.SOC_SEQ1 = 1;
    while(AdcRegs.ADCTRL2.bit.SOC_SEQ1)
    {}    // wait for SOC

    IL_AVG1 = ((long)IL_AVG1_R1+IL_AVG1_R2+IL_AVG1_R3+IL_AVG1_R4)>>2;
    IL_AVG2 = ((long)IL_AVG2_R1+IL_AVG2_R2+IL_AVG2_R3+IL_AVG2_R4)>>2;
    IL_AVG3 = ((long)IL_AVG3_R1+IL_AVG3_R2+IL_AVG3_R3+IL_AVG3_R4)>>2;
    HS_VBUS = ((long)HS_VBUS_R1+HS_VBUS_R2)>>1;
    HS_TEMP = ((long)HS_TEMP_R1+HS_TEMP_R2)>>1;

    IL_AVG = ((long)IL_AVG1+IL_AVG2+IL_AVG3);

//    adc_replenish();
}

interrupt void adc_isr()
{

```



```

    adc_replenish();

    return;
}

signed long OVR_Z(signed long vo_ref, long u_limit, int reset)
{
    // Define local variables
    static long vo_meas=0;
    static long diff=0,output=0,comp_diff=0,u1=0,u2=0;

    if(reset)
    {
        output=0;
        u1 = 0;
        u2 = 0;
    }
    else
    {
        // Subtract any offset
        vo_meas=(long)HS_VBUS-(long)HS_VBUS_OFFSET;           //
r16: 0.0 -> 1.0

        // difference between ref and feedback
        diff=(long)vo_ref-(long)vo_meas;                       //
r15: -1.0 -> 1.0

        // compensated difference
        comp_diff= ( (long)diff<<10 )                          //
r23: -3.0 -> 3.0
                -( (long)u1<<11 )
                +( (long)u1 )
                +( (long)u2<<10 );

        u2=u1;
        u1=diff;

        // gain and integrator
        output+=( (comp_diff>>10)<<VGAIN );                    //
r15:

        // limitation (OVR limit to ZERO and variable upper limit)
        if(output<0)
            output=0;
        if(output>INT_SAT_UPPER)
            output=INT_SAT_UPPER;
    }

    return(output);
    // output r15:
}

signed long OVR_N(signed long vo_ref, int reset)
{
    // Define local variables
    static long vo_meas=0;
    static long diff=0,output=0,comp_diff=0,u1=0,u2=0;

```

```

    if(reset)
    {
        output=0;
        u1 = 0;
        u2 = 0;
    }
    else
    {
        // Subtract any offset
        vo_meas=(long)HS_VBUS-(long)HS_VBUS_OFFSET;           //
r16:  0.0 -> 1.0

        // difference between ref and feedback
        diff=(long)vo_ref-(long)vo_meas;                       //
r15: -1.0 -> 1.0

        // compensated difference
        comp_diff=    ( (long)diff<<10    )                   //
r23: -3.0 -> 3.0
                                -( (long)u1<<11    )
                                +( (long)u1          )
                                +( (long)u2<<10    );

        u2=u1;
        u1=diff;

        // gain and integrator
        output+=( (comp_diff>>10)<<VGAIN );                   //
r15:

        // limitation (OVR can be negative)
        if(output<-629146)
            output=-629146;
        if(output>INT_SAT_UPPER)
            output=INT_SAT_UPPER;
    }

    return(output);
    // output r15:
}

signed long IVR(signed long vo_ref, int reset)
{
    // Define local variables
    static long vo_meas=0;
    static long diff=0,output=0;

    if(reset)
    {
        output=0;
    }
    else
    {
        // difference between ref and feedback
        vo_meas=(long)rx.LSVBUS;
        // r16: 0.0 -> 1.0

```

```

        diff=(long)vo_ref-(long)vo_meas;//<<4);
// r20: -1.0 -> 1.0

        // gain and integrator
        output+=(diff>>VGAIN);
// r20

        // limitation (OVR can be negative)
        if(output<0)
            output=0;
        if(output>INT_SAT_UPPER)
            output=INT_SAT_UPPER;
// r20
    }

    return(output);
// output r20
}

signed long ICR(signed long io_ref, int reset)
{
    // Define local variables
    static long io_meas=0;
    static long diff=0,output=0,comp_diff=0,u1=0,u2=0;

    if(reset)
    {
        output=0;
        u1 = 0;
        u2 = 0;
    }
    else
    {
        // take the absolute value of io_ref
        if(io_ref<0)
        {
            io_ref=-io_ref;
        }

        io_meas = (long)IL_AVG - (long)CUR_OFFSET;

        if(io_meas<0)
        {
            io_meas=-io_meas;
        }

        // difference between ref and feedback
        diff=(long)io_ref-(long)io_meas; // r15: -1.0 ->
1.0

        // compensated difference
        comp_diff= ( (long)diff<<6 ) // r21: -3.0 ->
3.0
                -( (long)u1<<7 )
                +( (long)u1 )
                +( (long)u2<<6 );
    }
}

```

```

        u2=u1;
        u1=diff;

        // gain and limited integrator
        output+=((comp_diff>>6)<<IGAIN);           // r15:

        // limitation (ICR cannot be negitave)
        if(output<0)
            output=0;
        if(output>INT_SAT_UPPER)
            output=INT_SAT_UPPER;
    }

    return (output);                               // r15:
}

void load_can_data()
{
    // Load recived CAN data into global variables
    rx.cmd.all = ECana_MBoxes_Data.MBOX26.MDL.byte.BYTE0;
    rx.HSVBUS = ECana_MBoxes_Data.MBOX26.MDH.word.HI_WORD;
    rx.LSVBUS = ECana_MBoxes_Data.MBOX26.MDL.word.LOW_WORD;
    rx.AVGCURBUCK = (32000-
ECana_MBoxes_Data.MBOX27.MDL.word.LOW_WORD);
    rx.AVGCURBOOST = (ECana_MBoxes_Data.MBOX27.MDH.word.HI_WORD-
32000);

    // Load transmitted CAN data into global variables
    tx.HSVBUS = (signed long)(HS_VBUS-HS_VBUS_OFFSET)/3;
    tx.other.HSTEMP = ((signed long)(HS_TEMP-HS_TEMP_OFFSET)/129);
    tx.AVGCURPH1 = (signed long)(((signed)(IL_AVG1-
CUR_OFFSET1)/60)+32000);
    tx.AVGCURPH2 = (signed long)(((signed)(IL_AVG2-
CUR_OFFSET2)/60)+32000);
    tx.AVGCURPH3 = (signed long)(((signed)(IL_AVG3-
CUR_OFFSET3)/60)+32000);
    tx.AVGCURLOW = (signed long)(((long)(IL_AVG-
CUR_OFFSET)/60)+32000);
    //tx.other.STATUS.bit.MODE = rx.cmd.bit.mode;           // Update
these in different location in code
    //tx.other.STATUS.bit.STATE = rx.cmd.bit.state;       // Update
these in different location in code

    // validate date
    // if(rx.HSVBUS > HSVBUS_MAX)
    //     fault.bit.crit_fault_1 = 1;           // invalid data
    // if(rx.HSVBUS < HSVBUS_MIN)
    //     fault.bit.crit_fault_1 = 1;           // invalid data

    return;
}

void trans_can_data()
{
    // define local variables
    unsigned char Message[8];

```

```

    Message[0]= tx.other.STATUS.all;
    Message[1]= tx.other.HSTEMP;
    Message[2]= tx.HSVBUS;
    Message[3]= tx.HSVBUS>>8;
    Message[4]= tx.FAULTS;
    Message[5]= tx.FAULTS>>8;
    Message[6]= 0;
    Message[7]= 0;

    DSP280x_CANA_TX(Message,10);

    Message[0]= tx.AVGCURLLOW;
    Message[1]= tx.AVGCURLLOW>>8;
    Message[2]= tx.AVGCURPH1;
    Message[3]= tx.AVGCURPH1>>8;
    Message[4]= tx.AVGCURPH2;
    Message[5]= tx.AVGCURPH2>>8;
    Message[6]= tx.AVGCURPH3;
    Message[7]= tx.AVGCURPH3>>8;

    DSP280x_CANA_TX(Message,11);

    return;
}

unsigned int current_duty()
{
    // return the current duty cycle
    return(EPwm1Regs.CMPA.half.CMPA);
}

void heartBeat_LED(int speed)
{
    static unsigned int cnt = 0;
    if( !(++cnt & speed) )
        GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;           // toggle
heartBeat

    return;
}

unsigned polarity(unsigned int value)
{
    static unsigned current=0;

    switch(value)
    {
        case HBOOST:           // Boost
        {
            EPwm1Regs.AQCTLA.bit.CAU = AQ_CLEAR;           //
Active low for lower switch
            EPwm1Regs.AQCTLA.bit.PRD = AQ_SET;
            EPwm1Regs.AQCTLB.bit.CAU = AQ_CLEAR;           //
Disable upper switch
            EPwm1Regs.AQCTLB.bit.PRD = AQ_CLEAR;

```

```

        EPwm2Regs.AQCTLA.bit.CAU = AQ_CLEAR;           //
Active low for lower switch
        EPwm2Regs.AQCTLA.bit.PRD = AQ_SET;
        EPwm2Regs.AQCTLB.bit.CAU = AQ_CLEAR;           //
Disable upper switch
        EPwm2Regs.AQCTLB.bit.PRD = AQ_CLEAR;

        EPwm3Regs.AQCTLA.bit.CAU = AQ_CLEAR;           //
Active low for lower switch
        EPwm3Regs.AQCTLA.bit.PRD = AQ_SET;
        EPwm3Regs.AQCTLB.bit.CAU = AQ_CLEAR;           //
Disable upper switch
        EPwm3Regs.AQCTLB.bit.PRD = AQ_CLEAR;

        current=value;
// update current state
        return current;
    }
    case HBUCK:      // Buck
    {
        EPwm1Regs.AQCTLA.bit.CAU = AQ_CLEAR;           //
Active low for lower switch
        EPwm1Regs.AQCTLA.bit.PRD = AQ_CLEAR;
        EPwm1Regs.AQCTLB.bit.CAU = AQ_CLEAR;           //
Disable upper switch
        EPwm1Regs.AQCTLB.bit.PRD = AQ_SET;

        EPwm2Regs.AQCTLA.bit.CAU = AQ_CLEAR;           //
Active low for lower switch
        EPwm2Regs.AQCTLA.bit.PRD = AQ_CLEAR;
        EPwm2Regs.AQCTLB.bit.CAU = AQ_CLEAR;           //
Disable upper switch
        EPwm2Regs.AQCTLB.bit.PRD = AQ_SET;

        EPwm3Regs.AQCTLA.bit.CAU = AQ_CLEAR;           //
Active low for lower switch
        EPwm3Regs.AQCTLA.bit.PRD = AQ_CLEAR;
        EPwm3Regs.AQCTLB.bit.CAU = AQ_CLEAR;           //
Disable upper switch
        EPwm3Regs.AQCTLB.bit.PRD = AQ_SET;

        current=value;
// update current state
        return current;
    }
    case '?':
    {
        return current;
    }
    default:      // ShutDown
    {
        EPwm1Regs.AQCTLA.bit.CAU = AQ_CLEAR;           //
Active low for lower switch
        EPwm1Regs.AQCTLA.bit.PRD = AQ_CLEAR;
        EPwm1Regs.AQCTLB.bit.CAU = AQ_CLEAR;           //
Disable upper switch
        EPwm1Regs.AQCTLB.bit.PRD = AQ_CLEAR;

```

```

        EPwm2Regs.AQCTLA.bit.CAU = AQ_CLEAR;           //
Active low for lower switch
        EPwm2Regs.AQCTLA.bit.PRD = AQ_CLEAR;
        EPwm2Regs.AQCTLB.bit.CAU = AQ_CLEAR;           //
Disable upper switch
        EPwm2Regs.AQCTLB.bit.PRD = AQ_CLEAR;

        EPwm3Regs.AQCTLA.bit.CAU = AQ_CLEAR;           //
Active low for lower switch
        EPwm3Regs.AQCTLA.bit.PRD = AQ_CLEAR;
        EPwm3Regs.AQCTLB.bit.CAU = AQ_CLEAR;           //
Disable upper switch
        EPwm3Regs.AQCTLB.bit.PRD = AQ_CLEAR;

        current=value;
        // update current state
        return current;
    }
}

void InitEPwm1Example()
{
    EPwm1Regs.TBPRD = TPERIOD;                           //
Period
    EPwm1Regs.TBPHS.half.TBPHS = 0;                       // Set
Phase register to zero
    EPwm1Regs.TBCTR = 0x0000;                             // Clear
counter
    EPwm1Regs.CMPA.half.CMPA = 0;                         // set duty
cycle
    EPwm1Regs.CMPB = 25;

    EPwm1Regs.TBCTL.bit.FREE_SOFT = 0x10;                 // Free
running mode
    EPwm1Regs.TBCTL.bit.CTRMODE = 0;                       // Up count
    EPwm1Regs.TBCTL.bit.PHSEN = TB_DISABLE;               // Master module
    EPwm1Regs.TBCTL.bit.PRDLN = TB_SHADOW;                // Shadow
the period reg
    EPwm1Regs.TBCTL.bit.SYNCSEL = TB_CTR_ZERO;           // Sync down-
stream module

    EPwm1Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;           // shadow
mode
    EPwm1Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;           // shadow
mode
    EPwm1Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;         // load on
CTR=Zero
    EPwm1Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;         // load on
CTR=Zero

    EPwm1Regs.AQCTLA.bit.CAU = AQ_CLEAR;                 // init
both PWMs forced low
    EPwm1Regs.AQCTLA.bit.PRD = AQ_CLEAR;
    EPwm1Regs.AQCTLB.bit.CAU = AQ_CLEAR;

```

```

    EPwm1Regs.AQCTLB.bit.PRD = AQ_CLEAR;

    EPwm1Regs.DBCTL.bit.OUT_MODE = 00;           // Disable
Dead-band module

    // Set up interupt
    EPwm1Regs.ETSEL.bit.INTSEL = ET_CTR_ZERO;    // Select INT on
Zero event
    EPwm1Regs.ETSEL.bit.INTEN = 1;             // Enable
INT
    EPwm1Regs.ETPS.bit.INTPRD = ET_1ST;        // Generate INT
on 1st event

    return;
}

void InitEPwm2Example()
{
    EPwm2Regs.TBPRD = TPERIOD;                 //
Period =
    EPwm2Regs.TBPHS.half.TBPHS = TPERIOD/3;    // Phase =
120d
    EPwm2Regs.TBCTR = 0x0000;                 // Clear counter
    EPwm2Regs.CMPA.half.CMPA = 0;             // set
initial duty cycle

    EPwm2Regs.TBCTL.bit.FREE_SOFT = 0x10;      // Free
running mode
    EPwm2Regs.TBCTL.bit.CTRMODE = 0;          // Up Count
    EPwm2Regs.TBCTL.bit.PHSEN = TB_ENABLE;     // Slave
module

    EPwm2Regs.TBCTL.bit.PRDL = TB_SHADOW;      // shadow
period reg
    EPwm2Regs.TBCTL.bit.SYNCSEL = TB_SYNC_IN; // sync
flow-through

    EPwm2Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW; // shadow
mode
    EPwm2Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW; // shadow
mode
    EPwm2Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO; // load on
CTR=Zero
    EPwm2Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO; // load on
CTR=Zero

    EPwm2Regs.AQCTLA.bit.CAU = AQ_CLEAR;       // init
both PWMs forced low
    EPwm2Regs.AQCTLA.bit.PRD = AQ_CLEAR;
    EPwm2Regs.AQCTLB.bit.CAU = AQ_CLEAR;
    EPwm2Regs.AQCTLB.bit.PRD = AQ_CLEAR;

    EPwm2Regs.DBCTL.bit.OUT_MODE = 0;         // disable
Dead-band module

```



```

        return;
    }

void InitEPwm3Example()
{
    EPwm3Regs.TBPRD = TPERIOD; //
    Period =
    EPwm3Regs.TBPHS.half.TBPHS = 2*TPERIOD/3; // Phase = 240d
    EPwm3Regs.TBCTR = 0x0000; // Clear counter
    EPwm3Regs.CMPA.half.CMPA = 0; //
    set duty cycle

    EPwm3Regs.TBCTL.bit.FREE_SOFT = 0x10; // Free
    running mode
    EPwm3Regs.TBCTL.bit.CTRMODE = 0; // Up count
    EPwm3Regs.TBCTL.bit.PHSEN = TB_ENABLE; // Slave
    module

    EPwm3Regs.TBCTL.bit.PRDL = TB_SHADOW; // shadow
    the period reg
    EPwm3Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_IN; // sync
    flow-through

    EPwm3Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW; // shadow
    mode
    EPwm3Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW; // shadow
    mode
    EPwm3Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO; // load on
    CTR=Zero
    EPwm3Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO; // load on
    CTR=Zero

    EPwm3Regs.AQCTLA.bit.CAU = AQ_CLEAR; // init
    both PWMs forced low
    EPwm3Regs.AQCTLA.bit.PRDL = AQ_CLEAR;
    EPwm3Regs.AQCTLB.bit.CAU = AQ_CLEAR;
    EPwm3Regs.AQCTLB.bit.PRDL = AQ_CLEAR;

    EPwm3Regs.DBCTL.bit.OUT_MODE = 0; // disable
    Dead-band module
    // fix this to give comp. signals

    return;
}

void InitEPwm4Example()
{
    EPwm4Regs.TBPRD = ADC_PERIOD; //
    Period =
    EPwm4Regs.TBPHS.half.TBPHS = 0; //
    Phase = 0
    EPwm4Regs.TBCTR = 0x0000; // Clear counter
    EPwm4Regs.CMPA.half.CMPA = ADC_PERIOD>>1; // set duty cycle
}

```

```

        EPwm4Regs.TBCTL.bit.FREE_SOFT = 0x10;           // Free
running mode
        EPwm4Regs.TBCTL.bit.CTRMODE = 0;              // Up count
        EPwm4Regs.TBCTL.bit.PHSEN = TB_DISABLE;      // Slave module

        EPwm4Regs.TBCTL.bit.PRDLN = TB_SHADOW;       // shadow
the period reg
        EPwm4Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_IN;   // sync
flow-through

        EPwm4Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;  // shadow
mode
        EPwm4Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;  // shadow
mode
        EPwm4Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO; // load on
CTR=Zero
        EPwm4Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO; // load on
CTR=Zero

        EPwm4Regs.AQCTLA.bit.CAU = AQ_CLEAR;        // init
both PWMs forced low
        EPwm4Regs.AQCTLA.bit.PRD = AQ_CLEAR;
        EPwm4Regs.AQCTLB.bit.CAU = AQ_CLEAR;
        EPwm4Regs.AQCTLB.bit.PRD = AQ_CLEAR;

        EPwm4Regs.DBCTL.bit.OUT_MODE = 0;           // disable
Dead-band module

        // Start the conversion for the ADC
//      EPwm4Regs.ETSEL.bit.SOCAEN = 1;              //
enable triggering
//      EPwm4Regs.ETSEL.bit.SOCASEL = 6;            // trigger
ADC on period
//      EPwm4Regs.ETPS.bit.SOCAPRD = 1;            //
trigger on first event

        return;
}

void sys_init()
{
// Step 1. Initialize System Control:
// PLL, WatchDog, enable Peripheral Clocks
// This example function is found in the DSP280x_SysCtrl.c file.
    InitSysCtrl();

// For this example, set HSPCLK to SYSCLKOUT / 8 (12.5Mhz assuming
100Mhz SYSCLKOUT)
    EALLOW;
    SysCtrlRegs.HISPCP.all = 0x4; // HSPCLK = SYSCLKOUT/8
    EDIS;

// Step 2. Initialize GPIO:
// This example function is found in the DSP280x_Gpio.c file and
// illustrates how to set the GPIO to it's default state.

```

```

// InitGpio(); // Skipped for this example

// For this case just init GPIO pins for ePWM1, ePWM2, ePWM3
// These functions are in the DSP280x_EPwm.c file
InitEPwm1Gpio();
InitEPwm2Gpio();
InitEPwm3Gpio();
InitEPwm4Gpio();

// Just initialize eCAN pins for this example
// This function is in DSP280x_ECan.c
InitECanGpio();

EALLOW;
// Set up the heartBeat_LED
GpioCtrlRegs.GPBMUX1.bit.GPIO34 = 0; // select GPIO function
GpioCtrlRegs.GPBDIR.bit.GPIO34 = 1; // Output

// Set up spare GPIO
GpioCtrlRegs.GPAMUX1.bit.GPIO8 = 0; // select GPIO function
GpioCtrlRegs.GPADIR.bit.GPIO8 = 1; // Output
EDIS;

// Step 3. Clear all interrupts and initialize PIE vector table:
// Disable CPU interrupts
DINT;

// Initialize the PIE control registers to their default state.
// The default state is all PIE interrupts disabled and flags
// are cleared.
// This function is found in the DSP280x_PieCtrl.c file.
InitPieCtrl();

// Disable CPU interrupts and clear all CPU interrupt flags:
IER = 0x0000;
IFR = 0x0000;

// Initialize the PIE vector table with pointers to the shell Interrupt
// Service Routines (ISR).
// This will populate the entire table, even if the interrupt
// is not used in this example. This is useful for debug purposes.
// The shell ISR routines are found in DSP280x_DefaultIsr.c.
// This function is found in DSP280x_PieVect.c.
InitPieVectTable();

// Interrupts that are used in this example are found in
DSP280x_DefaultIsr.c.

DSP280x_InterruptsConfig();
//EnableInterrupts();

EALLOW;
PieVectTable.ADCINT = &adc_isr; // isr for avg
samples
PieVectTable.EPWM1_INT = &epwm1_timer_isr; // isr for main section
of code
PieVectTable.ECAN0INTA = &ecan0inta_isr;

```

```

PieVectTable.ECAN1INTA = &ecanlinta_isr;
PieVectTable.ECAN0INTB = &ecan0intb_isr;
PieVectTable.ECAN1INTB = &ecanlintb_isr;
EDIS;

#ifdef FLASH
    // Murad Qahwash. User specific code, enable interrupts:

    // Copy time critical code and Flash setup code to RAM
    // This includes the following ISR functions: epwm1_timer_isr(),
    epwm2_timer_isr()
    // epwm3_timer_isr and and InitFlash();
    // The RamfuncsLoadStart, RamfuncsLoadEnd, and RamfuncsRunStart
    // symbols are created by the linker. Refer to the F2808.cmd file.
    MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);

    // Call Flash Initialization to setup flash waitstates
    // This function must reside in RAM
    InitFlash();
#endif

// Step 4. Initialize all the Device Peripherals:
// This function is found in DSP280x_InitPeripherals.c
// InitPeripherals(); // Not required for this example
InitAdc();

EALLOW;
SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;
EDIS;

InitEPwm1Example();
InitEPwm2Example();
InitEPwm3Example();
InitEPwm4Example();

EALLOW;
SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;
EDIS;

// In this case just initialize eCAN-A and eCAN-B
// This function is in DSP280x_ECan.c
InitECan();

// Step 5. User specific code, enable interrupts

// Enable ADCINT in PIE
PieCtrlRegs.PIEIER1.bit.INTx6 = 1; // ADC
PieCtrlRegs.PIEIER3.bit.INTx1 = 1; // PWM

IER |= M_INT1; // Enable CPU Interrupt Level 1
IER |= M_INT3; // Enable CPU INT3 which is connected to EPWM1-6 INT:
IER |= M_INT9; // Enable CPU Interrupt Level 9
EINT; // Enable Global interrupt INTM
ERTM; // Enable Global realtime interrupt DBGM

// configure the CAN
DSP280x_ECanConfig(); //MQ

```

```

// Configure ADC
AdcRegs.ADCMAXCONV.bit.MAX_CONV1 = 7;
AdcRegs.ADCMAXCONV.bit.MAX_CONV2 = 7;

AdcRegs.ADCCHSELSEQ1.bit.CONV00 = HS_VBUS_PIN; // Setup 1st SEQ1
conv.
AdcRegs.ADCCHSELSEQ1.bit.CONV01 = HS_TEMP_PIN; // Setup 2nd SEQ1
conv.
AdcRegs.ADCCHSELSEQ1.bit.CONV02 = HS_VBUS_PIN; // Setup 3rd SEQ1
conv.
AdcRegs.ADCCHSELSEQ1.bit.CONV03 = HS_TEMP_PIN; // Setup 4th SEQ1
conv.

AdcRegs.ADCCHSELSEQ2.bit.CONV04 = IL_AVG1_PIN;
AdcRegs.ADCCHSELSEQ2.bit.CONV05 = IL_AVG2_PIN;
AdcRegs.ADCCHSELSEQ2.bit.CONV06 = IL_AVG3_PIN;
AdcRegs.ADCCHSELSEQ2.bit.CONV07 = IL_AVG1_PIN;

AdcRegs.ADCCHSELSEQ3.bit.CONV08 = IL_AVG2_PIN;
AdcRegs.ADCCHSELSEQ3.bit.CONV09 = IL_AVG3_PIN;
AdcRegs.ADCCHSELSEQ3.bit.CONV10 = IL_AVG1_PIN;
AdcRegs.ADCCHSELSEQ3.bit.CONV11 = IL_AVG2_PIN;

AdcRegs.ADCCHSELSEQ4.bit.CONV12 = IL_AVG3_PIN;
AdcRegs.ADCCHSELSEQ4.bit.CONV13 = IL_AVG1_PIN;
AdcRegs.ADCCHSELSEQ4.bit.CONV14 = IL_AVG2_PIN;
AdcRegs.ADCCHSELSEQ4.bit.CONV15 = IL_AVG3_PIN;

AdcRegs.ADCTRL1.bit.CONT_RUN = 0; // Set ADC
to start/stop mode
AdcRegs.ADCTRL1.bit.SEQ_CASC = 1; // Cascade
Seq1 and 2
//AdcRegs.ADCTRL2.bit.EPWM_SOCA_SEQ1 = 1; // Enable SOCA
from ePWM to start SEQ1
//AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = 1; // Enable SEQ1
interrupt (every EOS)

// Initilize with a fault to force start up in standby
rx.cmd.bit.state=0;
ECana_MBoxes_Data.MBOX26.MDL.byte.BYTE0=0;
fault.bit.START_UP=1;

}

void adc_replenish()
{
// Reinitialize for next ADC sequence
AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1; // Reset SEQ1
AdcRegs.ADCTRL2.bit.RST_SEQ2 = 1; // Reset SEQ2
AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1; // Clear INT SEQ1 bit
// PieCtrlRegs.PIEACK.all = PIEACK_GROUP1; // Acknowledge
interrupt to PIE
}

void epwm_replenish()
{

```

```

// Clear INT flag for this timer
    EPwmlRegs.ETCLR.bit.INT = 1;

// Acknowledge this interrupt to receive more interrupts from group 3
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
}
//=====
// No more.
//=====
=====

```

### JD\_PowerUnit.c

```

// code for digital control of John Deere's power unit
// ApECOR
// Created August 25, 2006
//-----

#include "DSP280x_Device.h"    // DSP280x Headerfile Include File
#include "DSP280x_Examples.h"  // DSP280x Examples Include File
#include "DSP280x_CAN_GlobalVariableDefs.h" //MQ: DSP28 MailBoxes
Data Messages File

#include "settings.h"
#include "JD_PowerUnit.h"
#include "sys_fun.h"

// First fuction to be called
// All control starts from here
// Function returns a refrence
// for Duty in R20 format
long int control_manager()
{
// Define local variables
    static signed duty = 0;
    static signed soft = 0;
    static unsigned tx_cnt = 0;

// Transmit data to CAN
    if(++tx_cnt == 0xFFFF)    // 2000 = 100ms at 20kHz
    {
        GpioDataRegs.GPATOGGLE.bit.GPIO8 = 1;
        trans_can_data();
        tx_cnt=0;
    }

// Load values recived from CAN
    load_can_data();

// Check for fault conditions
    check_faults();

//
    switch (fetch_state())

```

```

{
    case RUN:                //run
    {
        //    duty = (( (long)TPERIOD*run_handler() )>>20);
        duty = run_handler()>>8;

        //return(duty);

        ++soft;                // increment the soft start

        if(soft<0)            // limit the soft start
            soft=0;
        if(soft>(DUTY_MAX<<3))
            soft=(DUTY_MAX<<3);

        return(min(duty,soft>>3));
    }
    case RESET:              //reset
    {

        duty = current_duty();

        --soft;                // increment the soft start

        if(soft<0)            // limit the soft start
        {
            soft=0;
            reset_handler();
        }
        if(soft>(DUTY_MAX<<3))
            soft=(DUTY_MAX<<3);

        return(min(duty,soft>>3));
    }
    default:                  //standby
    {

        duty = current_duty();

        --soft;                // increment the soft start

        if(soft<0)            // limit the soft start
        {
            soft=0;
            standby_handler();
        }
        if(soft>(DUTY_MAX<<3))
            soft=(DUTY_MAX<<3);

        return(min(duty,soft>>3));
    }
}

}

unsigned int fetch_state()
{

```

```

// Define local variables
unsigned int state=rx.cmd.bit.state;          //init at received value

        tx.other.STATUS.bit.STATE=state;

// If fault is present update mode to
// standby and trasmitte a FAULT
        if(fault.all)
        {
                state=STANDBY;
                tx.other.STATUS.bit.STATE=FAULT; // tx a fault
        }

// If a RESET command is sent ignore the
// fault and set state to reset
        if(rx.cmd.bit.state == RESET)
        {
                state=RESET;
                tx.other.STATUS.bit.STATE=RESET;
                // might tx data later
        }

        return(state);
}

long int run_handler()
{
// define local variables
int reset = 0;

// Select mode recived from the CAN
        switch(rx.cmd.bit.mode)
        {
                case EV: // EV
                {
                        // define local variables
                        static long io_ref=0,vo_ref=0;
                        static long io_out=0,vo_out=0;

                        if(tx.other.STATUS.bit.MODE == EV)
                                reset=0;
// run in normal mode
                        else
                        {
                                reset=1;
// reset the controller
                                tx.other.STATUS.bit.MODE = EV;
                        }

                        // set voltage regulation reference and run OVR_N
(negative output enabled)
                                vo_ref=(long) rx.HSVBUS*3;

                                vo_out=OVR_N(min(VO_REF_LIM,vo_ref), reset);

                                // update polarity based on current command
                                if(vo_out>0)

```



```

        {
            // set current regulation reference
            io_ref=(long)rx.AVGCURBOOST*60;           //
check this scaling
            polarity(HBOOST);
        }
        else
        {
            // set current regulation reference
            io_ref=(long)rx.AVGCURBUCK*60;           //
check this scaling
            polarity(HBUCK);
        }

        io_out=ICR(min(IO_REF_LIM,io_ref),reset);

        if(vo_out<0)
            vo_out=-vo_out;

        return(min(io_out,vo_out));
    }
    case HBOOST: // HBOOST
    {
        // define local variables
        static long io_ref=0,vo_ref=0;
        static long io_out=0,vo_out = 0;

        if(tx.other.STATUS.bit.MODE == HBOOST)
        {
            reset=0;                                // run in normal
mode
            polarity(HBOOST);                        // set polarity to
HBOOST
        }
        else
        {
            reset=1;
            // reset the controller
            tx.other.STATUS.bit.MODE = HBOOST;
            polarity(SHUTDOWN);                      // set polarity
to SHUTDOWN
        }

        // set current command
        io_ref=(long)rx.AVGCURBOOST*60;           //
check this scaling

        io_out=ICR(min(IO_REF_LIM,io_ref),reset);

        // set over voltage regulation
        //vo_ref = freq_resp((long)rx.HSVBUS*3);
        vo_ref=(long)rx.HSVBUS*3;

        vo_out=OVR_Z(min(VO_REF_LIM,vo_ref),io_out,reset);

        return(min(io_out,vo_out));
    }
}

```

```

        case HBUCK: // HBUCK
        {
            // define local variables
            static long io_ref=0,vo_ref=0;
            static long io_out=0,vo_out=0;

            // reset controllers if first time entering HBUCK
mode
            if(tx.other.STATUS.bit.MODE == HBUCK)
            {
mode
                reset=0; // run in normal
mode
                polarity(HBUCK); // set polarity to
HBUCK
            }
            else
            {
                reset=1;
                // reset the controller
                tx.other.STATUS.bit.MODE=HBUCK;
                polarity(SHUTDOWN); // set polarity
to SHUTDOWN
            }

            // set current regulation reference
            io_ref=(long)rx.AVGCURBUCK*60; //
check this scaling

            if(io_ref)
                io_out=ICR(min(IO_REF_LIM,io_ref),reset);
            else
                io_out=ICR(min(IO_REF_LIM,io_ref),reset);

            // set voltage regulation reference
            vo_ref=280*20; // check
this scaling
            vo_out=IVR(min(VB_REF_LIM,vo_ref),reset);

            return(min(io_out,vo_out));
        }
        default: // invalid state
        {
            fault.bit.crit_fault_0 = 1;
            // report invalid state error
            return(current_duty());
            // do not change duty cycle
        }
    }
}

unsigned int reset_handler()
{
    fault.all = 0x0000;
    polarity(SHUTDOWN);
    return(0);
}

```

```

unsigned int standby_handler()
{
    polarity(SHUTDOWN);
    return(0);
}

void check_faults()
{
    // define local variables
    long curr_meas=IL_AVG;

    curr_meas=(long)IL_AVG-(long)CUR_OFFSET;

    // if(curr_meas<0)
    //     curr_meas=-curr_meas;

    // check all ADC for fault conditions
    // critical faults
    if(polarity('?')==HBOOST)
    {
        if((-curr_meas) > IL_BOOST_AVG_MAX)
        {
            //polarity(SHUTDOWN);
            fault.bit.IL_OC=1;
        }
    }
    else
    {
        if(curr_meas > IL_BUCK_AVG_MAX)
        {
            // polarity(SHUTDOWN);
            fault.bit.IL_OC=1;
        }
    }

    if(HS_TEMP > HS_TEMP_MAX)
    {
        // polarity(SHUTDOWN);
        fault.bit.TEMP=1;
    }

    // non critical faults
    if(HS_VBUS > HS_VBUS_MAX)
    {
        polarity(SHUTDOWN);
        fault.bit.HS_OV=1;
    }

    if(rx.LSVBUS > LS_VBUS_MAX)
    {
        fault.bit.LS_OV=1;
    }

    // check communication
    if(++comm_cnt>5000) // 5000 = 250ms at 20kHz

```

```
    {
        comm_cnt--;
        fault.bit.COMM_ERR = 1;
    }
    //else
    //    fault.bit.COMM_ERR = 0;

    return;
}

// end of file
```

## APPENDIX C: EQUATIONS

$$\begin{aligned}
V_{bus}(s) &= \frac{V_{batt}^2 T_s D^2(s)}{s2CL(V_{bus}(s) - V_{batt})} - \frac{V_{bus}(s)}{sR_{Load}C} \\
V_{bus}(s) &= V_{bus} + \tilde{v}_{bus} \\
D(s) &= D + \tilde{d} \\
V_{bus} + \tilde{v}_{bus} &= \frac{V_{batt}^2 T_s (D + \tilde{d})^2}{s2CL(V_{bus} + \tilde{v}_{bus} - V_{batt})} - \frac{V_{bus} + \tilde{v}_{bus}}{sR_{Load}C} \\
(V_{bus} + \tilde{v}_{bus})(V_{bus} + \tilde{v}_{bus} - V_{batt}) \left(1 + \frac{1}{sR_{Load}C}\right) &= \frac{V_{batt}^2 T_s (D + \tilde{d})^2}{s2CL} \\
(V_{bus} + \tilde{v}_{bus})(V_{bus} + \tilde{v}_{bus} - V_{batt}) \left(s2CL + \frac{2L}{R_{Load}}\right) &= V_{batt}^2 T_s (D^2 + 2D\tilde{d} + \tilde{d}^2) \\
(V_{bus}^2 + 2V_{bus}\tilde{v}_{bus} - V_{bus}V_{batt} - V_{batt}\tilde{v}_{bus} + \tilde{v}_{bus}^2) \left(s2CL + \frac{2L}{R_{Load}}\right) &= V_{batt}^2 T_s (D^2 + 2D\tilde{d} + \tilde{d}^2)
\end{aligned}$$

Cancel pure DC and higher order terms

$$\begin{aligned}
(2V_{bus}\tilde{v}_{bus} - V_{batt}\tilde{v}_{bus}) \left(s2CL + \frac{2L}{R_{Load}}\right) &= V_{batt}^2 T_s 2D\tilde{d} \\
\tilde{v}_{bus}(2V_{bus} - V_{batt}) &= \frac{V_{batt}^2 T_s 2D\tilde{d}}{\left(s2CL + \frac{2L}{R_{Load}}\right)} \\
\tilde{v}_{bus}(2V_{bus} - V_{batt}) &= \frac{V_{batt}^2 T_s D\tilde{d}R_{Load}}{(sR_{Load}CL + L)} \\
\frac{\tilde{v}_{bus}}{\tilde{d}} &= \frac{\left(\frac{V_{batt}^2 T_s DR_{Load}}{2V_{bus} - V_{batt}}\right)}{sR_{Load}CL + L}
\end{aligned}$$

**Equation 8**

## LIST OF REFERENCES

- [1] “Comparing DC-DC converters for power management in hybrid electric vehicles” Schupbach, R.M.; Balda, J.C.; Electric Machines and Drives Conference, IEMDC'03. IEEE International Volume 3, June 2003 Pp1369–1374.
- [2] “Interleaving optimization in synchronous rectified DC/DC converters” Gerber, M.; Ferreira, J.A.; Hofsjager, I.W.; Seliger, N.; Power Electronics Specialists Conference, 2004. IEEE 35th Annual Volume 6, 20-25 June 2004 Page(s):4655 - 4661 Vol.6
- [3] “Digital Controller Design for a Practicing Power Electronics Engineer” Al-Atrash, H.; Batarseh, I.; Applied Power Electronics Conference, - Twenty Second Annual IEEE Feb. 2007 Page(s):34 – 41
- [4] “Automotive DC-DC bidirectional converter made with many interleaved buck stages,” Garcia, O., Zumel, P., de Castro, A., Cobos, A., Power Electronics, IEEE Transactions on Volume 21, Issue 3, May 2006 Page(s):578 – 586
- [5] “Bidirectional buck-boost converter with variable output voltage” Krishnamachari, B.; Czarkowski, D.; Circuits and Systems, 1998. ISCAS '98. Proceedings of the 1998 IEEE International Symposium on Volume 6, 31 May-3 June 1998 Page(s):446 - 449 vol.6
- [6] “High Efficiency Energy Storage System Design for Hybrid Electric Vehicle with Motor Drive Integration” Shuai Lu; Corzine, K.A.; Ferdowsi, M.; Industry Applications Conference, 2006. Conference Record of the 2006 IEEE Volume 5, 8-12 Oct. 2006 Page(s):2560 – 2567
- [7] “Battery usage and thermal performance of the Toyota Prius and Honda Insight during chassis dynamometer testing” Kelly, K.J.; Mihalic, M.; Zolot, M.; Battery Conference on Applications and Advances, 2002. The Seventeenth Annual 15-18 Jan. 2002 Page(s):247 – 252
- [8] “Topological overview of hybrid electric and fuel cell vehicular power system architectures and configurations” Emadi, A.; Rajashekara, K.; Williamson, S.S.; Lukic, S.M.; Vehicular Technology, IEEE Transactions on Volume 54, Issue 3, May 2005 Page(s):763 – 770
- [9] “A multiphase dc/dc converter for automotive dual-voltage power systems” A. Consoli, M. Cacciato, G. Scarcella, A. Testa, IEEE Industry Applications Magazine, Nov|Dec 2004
- [10] “Modern Electric, Hybrid Electric, and Fuel Cell Vehicles” M. Ehsani, Y. Gao, S. E. Gay, A. Emadi, CRC Press LLC, 2005

<sup>1</sup> Image reprinted under the Creative Commons license agreement  
<http://creativecommons.org/licenses/by-sa/2.5/>