Dynamic Modelling and Control of Dual Active Bridge Bi-directional DC-DC Converters for Smart Grid Applications

A thesis submitted in accordance with the regulations of the Royal Melbourne Institute of Technology University in fulfillment of the requirements for the degree of Doctor of Philosophy.

by

Dinesh Sekhar Segaran

BEng(Hons), Monash University, 2006

Supervisor: Prof. Grahame Holmes
Associate Supervisor: Dr. Brendan McGrath

School of Electrical and Computer Engineering (SECE), RMIT University.

February 7, 2013

© Copyright

by

Dinesh Sekhar Segaran

2013

Dynamic Modelling and Control of Dual Active Bridge Bi-directional DC-DC Converters for Smart Grid Applications

Thesis Acceptance

This student's Thesis, entitled **Dynamic Modelling and Control of Dual Active Bridge Bi-directional DC-DC Converters for Smart Grid Applications** has been examined by the undersigned committee of examiners and has received full approval for acceptance for fulfillment of the requirements for the degree of Doctor of Philosophy.

APPROVAL: ______________________________ Chief Examiner

____

____

____

____

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Dinesh Sekhar Segaran
February 7, 2013

Abstract

Since the modern Smart Grid includes highly dynamic energy sources such as wind turbines and solar cells, energy storage is required to sustain the grid in the face of fluctuations in power generation. Possible energy storage elements that have been proposed include Plug-in Hybrid Electric Vehicles (P-HEVs) and battery banks, with power electronic converters employed to link the Direct Current (DC) energy storage elements to the Alternating Current (AC) Smart Grid. These systems all demand bi-directional DC-DC energy transfer capability as well as galvanic isolation as part of their core functionality. At power levels greater than a kilowatt, these complex power flow requirements are typically met with a Dual Active Bridge (DAB) Bi-directional DC-DC converter.

The DAB converter is made up of two single-phase H-bridge converters, connected back-to-back across a high-frequency AC link that is made up of an inductor and an isolation/scaling transformer. Each bridge is modulated using a phase-shifted square wave (PSSW) modulation scheme, where the phase difference between the bridge output voltage waveforms governs the magnitude and direction of power flow. This converter also relies upon a capacitor to provide DC output voltage stabilisation as well as ride-through during transient events (e.g. changes in the desired output voltage or load condition). To guarantee steady state stability and provide a fast transient response, fast and accurate regulation of these converters is essential towards maximising overall grid performance. This makes the DAB converter a more attractive solution at lower power levels and significantly boosts their viability at higher power levels. This thesis therefore aims to maximise closed loop regulator performance for these converters.

To investigate the limits of controller performance, a highly accurate dynamic converter model is required. Previous modelling techniques applied to the DAB converter are complex, computationally intensive and do not easily account for 2nd order effects such as deadtime, which significantly affect the dynamic response of the converter. This thesis presents a novel *harmonic modelling* technique that results in a simple yet accurate and flexible converter dynamic model. The basic premise of

the harmonic model is that the converter modulation functions drive the converter dynamics. Fourier analysis is used to decompose the modulation functions into their harmonic components, so the converter response to each significant harmonic can be determined. These responses are then summed together to give the full dynamic model. It is also identified in this work that deadtime changes the converter operating point, and that its effect is dependent on the AC inductor current. A series of closed form expressions that define the inductor current were developed and used to predict the effect of deadtime across all operating conditions. This prediction was used to extend the harmonic model, achieving a first order, two-input, small-signal state space model that was verified in simulation and then matched to an experimental DAB converter.

The new harmonic model was then used to investigate the performance limits of a closed loop regulator for the DAB converter. Since the aim of the regulator is DC voltage regulation, a Proportional + Integral (PI) control structure was chosen and implemented using a digital microprocessor. This thesis presents several enhancements to maximise the performance of this controller. First, maximum controller gains are calculated by precisely accounting for the limiting effects of the digital controller implementation (transport delay). Second, the harmonic model identifies that the forward path gain of the converter varies significantly with operating point, so an adaptive gain calculation algorithm was implemented to match the changes in plant characteristics, ensuring consistently high performance across the operating range. Third, the model also identifies that the load current acts as a disturbance input that significantly compromises performance, so a feed-forward disturbance rejection algorithm was implemented to minimise this effect. Finally, an AC load condition was also investigated to guarantee feasibility in a Smart Grid context. The excellent performance achieved by this new DAB voltage regulator minimises the capacitance needed to maintain the DAB output voltage in both steady-state and transient conditions. This offers the potential to eliminate the traditional electrolytic capacitor used in these applications, with associated size, cost and lifetime benefits.

All design, modelling and control ideas presented in this thesis were extensively verified both in simulation as well as on a 1 kW prototype DAB converter.

Acknowledgements

First and foremost I would like to thank my supervisors, Professor Grahame Holmes and Dr. Brendan McGrath. The Power Electronics Group is a fantastic working environment, and I hope it always will be a centre for excellence. I thank you both for your guidance and knowledge, but most of all for always having faith in me, and for always being the champions for my rights, especially when no one else would.

Among all those I would like to thank, I would like to make special mention of Dr. Peter Freere. Thank you, Peter, for showing us all that engineering, education and understanding all are truly right at our fingertips. Also, to the brilliant and ever helpful people at Creative Power Technologies – Pat, Mac, Mike – thank you for all the support – engineering and otherwise. To the technical and administrative staff of both Monash University and RMIT, thank you. Your tireless efforts make everything I have done possible. In particular, I would like to thank Ivan Kiss for his assistance, technical knowledge and friendship.

I would also like to thank all my contemporary postgraduate associates. Your friendship and understanding of the PhD. burden has helped me shoulder it better. In particular I would like to thank Wang Kong, for all his help and friendship over the years. Sorrell Grogan, I thank you for the gift of laughter and innumerable silly (yet disturbing) jokes. I would also like to extend my thanks to Richard Watson, Reza Davoodnezhad, Carlos Teixeira, Zaki Mohzani and Stewart Parker.

Lastly, but most importantly, I would like to thank my parents, Dr. Segaran Muthu and Dr. Vineetha Das, I add yet another Doctor to the family. ***Thank you*** is simply not enough to acknowledge your love and support over the years. I am the man I am today because of you both, and you are ever my inspirations. I dedicate this thesis to you.

Table of Contents

Abstract . . . . **vi**

Acknowledgements . . . . **viii**

Table of Contents . . . . **xiv**

List Of Figures . . . . **xx**

List of Tables . . . . **xxii**

Glossary Of Terms . . . . **xxiii**

List of Symbols . . . . **xxv**

1 Introduction **1**

1.1 Background 1

1.2 Objectives 3

1.3 Thesis Structure 3

1.4 Identification of Original Contributions 5

1.5 List of publications 6

2 Literature Review **8**

2.1 Topology selection 9

2.1.1 Generic Structure 9

2.1.2 Flyback Converters 10

2.1.3 Current fed Push-pull Converters 11
2.1.4 Bridge Converters 13
2.1.5 Summary – Topology 18
2.2 Modulation 18
2.2.1 Pulse Width Modulation 18
2.2.2 Block Modulation 19
2.2.3 Soft-switching 21
2.2.4 Summary – Modulation 25
2.3 Dynamic Modelling 26
2.3.1 State Averaged Models 27
2.3.2 Fundamental Averaged Models 31
2.3.3 Non-ideal effects: Deadtime 33
2.3.4 Summary – Modelling 34
2.4 Closed loop Control 35
2.4.1 Non-linear Control 36
2.4.2 Linear Control 38
2.4.3 Summary – Control 43
2.5 Conclusion 44

3 Converter Modelling 45

3.1 DAB Converter Principles of Operation 46
3.2 DAB Dynamic Equations 49
3.3 Deriving the switching functions 51
3.4 The Choice of N 53
3.5 Harmonic Model Derivation 56
3.6 Deadtime Compensation 67
3.6.1 The Deadtime Effect 67
3.6.2 Converter Behaviour During Deadtime 68

3.6.3 Modelling the Deadtime effect 71
3.6.4 Analytical calculation of the phase shift error effect 74
3.7 Final Model Derivation & Validation 77
3.8 Summary 77

4 Closed Loop Control 79
4.1 Choice of Feedback Controller 80
4.2 The digital modulator/PI controller & its performance limitations . . 83
4.2.1 The Digital Modulator 83
4.2.2 The Digital PI Controller 84
4.3 Delays in the Digital Implementation 85
4.3.1 The Effect of Transport Delay 86
4.4 Optimising PI controller gains 87
4.5 Load Step Performance 92
4.5.1 Exploring the load transient 93
4.5.2 Disturbance Rejection 95
4.5.3 Improvement in Load Transient Performance 96
4.6 Summary 98

5 System Performance with an AC Load 100
5.1 Challenges of Smart Grid Converter Design 101
5.2 Converter Principles of Operation 102
5.2.1 Single-phase Voltage Source Inverter (VSI) 102
5.2.2 DAB Bi-directional DC-DC Converter 105
5.3 Closed loop controller design 105
5.3.1 Choice of controller architecture 106
5.3.2 VSI current regulator 107
5.3.3 DAB voltage regulator 110
5.4 Results 114
5.5 Summary 115

6 Description of Simulated & Experimental Systems 117

6.1 Simulated Systems 118

6.1.1 Overview 118

6.1.2 Power Stage 118

6.1.3 Modulators 120

6.1.4 Controllers 120

6.2 Experimental Prototype 124

6.2.1 Overview 124

6.2.2 Power Stage 124

6.2.3 Controller Hardware 132

6.2.4 Inter-GIIB Communication 135

6.3 Summary 138

7 Simulation & Experimental Results 139

7.1 Overview 140

7.2 Steady-state Operating Waveforms 140

7.3 Open Loop Transients 144

7.4 Closed Loop Transients 146

7.4.1 Voltage Reference Step 147

7.4.2 Load Change 149

7.4.3 AC Load 151

7.5 Summary 154

8 Conclusions 155

8.1 Contributions 156

8.1.1 Harmonic Model 156

8.1.2 Deadtime Modelling 156

8.1.3 Maximised controller gains 156

8.1.4 Adaptive controller gains 157

8.1.5 Improved Load Transient Response . . . 157
8.1.6 AC Load Condition . . . 157
8.2 Future Work . . . 158
8.2.1 Multiport Converters . . . 158
8.2.2 Magnetics Design . . . 158
8.2.3 Extending the Harmonic Model . . . 158
8.2.4 Controller Performance . . . 159
8.3 Closure . . . 159

Appendix A Simulation & Experimental Code . . . 160
A.1 Simulation Code . . . 160
A.2 Experimental Code . . . 175
A.2.1 CPLD Code – Dual Active Bridge . . . 175
A.2.2 DSP Code – Dual Active Bridge . . . 188
A.2.3 DSP Code – Voltage Source Inverter . . . 224

References . . . 262

List of Figures

2.1 The Generic Bi-directional DC-DC Converter Topology 9
2.2 A simple Flyback Converter 10
2.3 Flyback Converter Operating Waveforms 10
2.4 The Actively Clamped Bi-directional Flyback Converter [1] 11
2.5 A Current fed push-pull converter 12
2.6 CFPP Operating Waveforms 12
2.7 Bi-directional CFPP 13
2.8 Phase Leg Topology & Operating Waveforms 14
2.9 Bridge Converter Topologies 15
2.10 Bi-directional half-bridge 15
2.11 Dual Active Bridge Converters 16
2.12 H-bridge and Modulator 19
2.13 Pulse Width Modulation 20
2.14 Block Modulation 21
2.15 Switching Loss 22
2.16 Ideal Soft-switching Waveforms 22
2.17 Parallel Capacitance 23
2.18 Active Clamp 24
2.19 Series Resonance 25
2.20 Buck Converter 28
2.21 The DAB Converter & Operating Waveforms 31

2.22 DAB Equivalent Circuits 32

2.23 A Basic Feedback Controller 35

2.24 Multiport 38

2.25 DAB bi-directional DC-DC Converter 39

2.26 Single-loop Feedback Controller 40

2.27 Dual-loop Feedback Controller 41

2.28 Parallel-loop Feedback Controller [2] 41

2.29 PI Controller with Feed-forward 43

3.1 The DAB Converter 46

3.2 Phase Leg Structure 47

3.3 Phase Leg Equivalent Circuits & Truth Table 47

3.4 H-bridge Converter & Truth table 48

3.5 H-bridge Operating Waveforms 48

3.6 DAB Operating Waveforms 49

3.7 KVL of the DAB Converter 51

3.8 Square Wave Harmonics 53

3.9 DAB Converter Equivalent Circuits 54

3.10 Harmonic Model Verification: Inductor Current 59

3.11 Harmonic Model Verification: Capacitor Current ($N = 3$) 60

3.12 Harmonic Model Verification: Steady-state Output Voltage ($N = 3$) . 61

3.13 Harmonic Model Verification: AC Inductor Current (Transient step) ($N = 3$) 62

3.14 Harmonic Model Verification: Output Voltage ($N = 3$) 63

3.15 Harmonic Model Verification: Output Voltage (DC Terms Only) ($N = 3$) 64

3.16 Variation in B_δ 65

3.17 Harmonic Model Verification: Output Voltage (Linearised Model) ($N = 3$) 66

3.18 Operating point dependence of the deadtime effect 68

3.19 Single Phase Leg with an Inductive Load 69

3.20 Deadtime Effect in a Phase Leg 70

3.21 Deadtime influence - HV bridge *lags* the LV bridge 72

3.22 Deadtime influence - HV bridge *leads* the LV bridge 73

3.23 Deadtime influence in the DAB converter 77

3.24 Block Diagram of Final Dynamic Model. 77

3.25 Validating the Final Model ($N = 3$) 78

4.1 Basic closed loop block diagram of the DAB converter. 80

4.2 Ideal Bode plot 82

4.3 Transient Responses of Ideal & Digital implementations 83

4.4 Digital PSSW Modulator 84

4.5 Sample & Hold 85

4.6 Controller Calculation & Update 86

4.7 Closed loop block diagram - Including Transport Delay. 86

4.8 Forward Path Bode Plot - Including Transport Delay. 87

4.9 Linearised Transient Responses. 87

4.10 B_δ term variation with operating phase shift 90

4.11 Closed Loop Block diagram of the DAB converter with an Adaptive PI controller 91

4.12 Closed loop Step Response Comparison 91

4.13 Comparison of Load & Reference Transient Responses 92

4.14 DAB Block Diagram (with Disturbance) 94

4.15 Pole Zero map of Closed Loop Transfer Functions 94

4.16 DAB Closed Loop Block Diagram with Feed-forward 95

4.17 Final Closed Loop Block Diagram of the DAB Converter 96

4.18 Load Step Response - Without Feed-forward 97

4.19 Load Step Response - With Feed-forward 99

5.1 AC-DC converter topology 101
5.2 Single-phase VSI 103
5.3 Fundamental VSI model 103
5.4 Fundamental VSI Power Flow 105
5.5 Proposed Closed Loop DC-AC Converter Architecture 107
5.6 Structure of the Current Regulated VSI 108
5.7 Closed-loop block diagram of the Current Regulated VSI 108
5.8 Forward path Bode plot of the Current-regulated VSI 109
5.9 Step response of the current regulated VSI [15A $\rightarrow$ 20A step] 109
5.10 Closed-loop block diagram of the Voltage Regulated DAB 110
5.11 Forward path Bode plot of the Voltage-regulated DAB 111
5.12 Step response of the voltage regulated DAB [195V $\rightarrow$ 200V step] 111
5.13 AC Load - No Feed Forward 112
5.14 Harmonic Spectrum - No Feed Forward 112
5.15 DAB DC link current 113
5.16 AC Load - Feed Forward 114
5.17 Harmonic Spectrum - Feed Forward 114
5.18 Converter Transient Waveforms 115

6.1 PSIM Power Stage - DAB Converter 119
6.2 PSIM Power Stage - DAB Load 120
6.3 PSIM - Modulators 121
6.4 Modulator Features 121
6.5 PSIM Simulation - DLL Block 122
6.6 Experimental Setup 124
6.7 Laboratory Setup 125
6.8 MagnaPower DC Supply 126

6.9 Experimental DAB Converter 127

6.10 Experimental High Frequency AC Inductor 128

6.11 Experimental High Frequency Transformer 128

6.12 Experimental Load Circuit Configuration 129

6.13 Experimental Load Elements 129

6.14 6 phase leg Circuit Diagram 130

6.15 PCB DC Bus Structure 130

6.16 Experimental 6 phase leg IGBT platform. 131

6.17 DA2810 DSP Controller Board 133

6.18 Mini2810 Controller Board 134

6.19 GIIB Inverter Board 135

6.20 Linked GIIB Boards 136

6.21 Synchronisation 137

7.1 Experimental Setup 140

7.2 DAB Steady State Operating Waveforms 141

7.3 Deadtime Effect - HV bridge *Lagging* the LV bridge 142

7.4 Deadtime Effect - HV bridge *Leading* the LV bridge 143

7.5 Open Loop: Deadtime 144

7.6 Open Loop: No Deadtime 145

7.7 Fixed PI 147

7.8 Adaptive PI 148

7.9 Reducing Load 149

7.10 Increasing Load 150

7.11 Steady State AC Load 152

7.12 AC Step Change (4A $\leftrightharpoons$ 6A) 153

7.13 AC Step Change (6A $\leftrightharpoons$ 4A) 154

List of Tables

2.1 Converter Topology Comparison 18
2.2 State Averaged Models 30
3.1 DAB Converter Parameters 46
3.2 Switched DC current (i_{DC}) based on output bridge switching state . 50
3.3 Choice of N 56
3.4 Piecewise Linear Inductor Current Solutions 75
3.5 Phase Shift Error Effect. 76
4.1 DAB Converter PI Controller Parameters 90
5.1 DC-AC Converter Parameters 107
5.2 VSI Current Regulator Parameters 109
5.3 DAB Voltage Regulator Controller Parameters 110
6.1 DAB Voltage Regulator Controller Parameters 123
6.2 DC-AC Experimental Converter Parameters 125
7.1 DC-AC Experimental Converter Parameters 140
7.2 DAB Voltage Regulator Controller Parameters 146
7.3 H-bridge Current Regulator Parameters 151

Glossary Of Terms

AC	Alternating Current
ADC	Analog-to-Digital Converter
CFPP	Current Fed Push-pull
DAB	Dual Active Bridge
DAC	Digital-to-Analog Converter
DC	Direct Current
DHB	Dual Half Bridge
DSP	Digital Signal Processor
FC	Fuel Cell
HV	High Voltage
IGBT	Insulated Gate Bipolar Transistor
JTAG	Controller board programming device
KCL	Kirchoff's Current Law
KVL	Kirchoff's Voltage Law
LF	Low Frequency
LV	Low voltage
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
MISO	Multi Input Single Output
PCB	Printed Circuit Board
P-HEV	Plug-in Hybrid Electric Vehicle
PI	Proportional + Integral
PLL	Phase Lock Loop
PSIM	PowerSim Switched Simulation package
PSSW	Phase-Shifted Square Waves
PWM	Pulse Width Modulation
Q-point	Quiescent Point
R-L	Resistive-Inductive
RMS	Root Mean Square

RS-232	Serial Communication Protocol
SC	Supercapacitor
SMPS	Switchmode Power Supplies
SPI	Serial Peripheral Interface
TAB	Triple Active Bridge
THD	Total Harmonic Distortion
TTL	Truth Table Logic
UC	Ultracapacitor
UPS	Uninterruptible Power Supply
VSI	Voltage Source Inverter
ZCS	Zero Current Switching
ZIR	Zero Impulse Response
ZOH	Zero Order Hold
ZSR	Zero State Response
ZVS	Zero Voltage Switching

List of Symbols

α	Relative phase angle
δ	Phase shift
δ_0	Phase shift Q-point
δ_c	Commanded phase shift
δ_{db}	Phase shift error caused by deadtime
δ_{DT}	Deadtime period in radians
δ_e	Effective applied phase shift
δ_{FF}	Feed-forward command
ϕ_m	Phase margin
$\varphi_z [n]$	Impedance angle at n^{th} harmonic
ω	Frequency expressed in rad/s
ω_c	Controller bandwidth (in rad/s)
ω_s	Switching frequency (in rad/s)
a_n, b_n	Fourier Series harmonic coefficients
A, B_δ, B_I	State space coefficients
D	Duty Cycle
$G(s)$	Laplace domain representation of open loop plant
$H(s)$	Laplace domain representation of regulator
i_{load}	Load current
i_{load_0}	Load current Q-point
K_p	Proportional Gain
m	Modulation Depth
n	Harmonic number
N	Number of significant harmonics considered
$\frac{N_p}{N_s}$	Transformer turns ratio
S_k	Phase leg switch state
$S_k (t)$	Time-domain representation of S_k

T_d	Delay time
T_p	Plant time constant ($\frac{-1}{A}$)
T_r	Integrator reset time
T_s	Switching period
V_{out}	Output Voltage
V_{out_0}	Output Voltage Q-point
V_{ref}	Voltage reference command
$Z[n]$	Impedance at n^{th} harmonic
$\|Z[n]\|$	Impedance magnitude at n^{th} harmonic

Chapter 1

Introduction

1.1 Background

The Smart Grid is the emerging paradigm in energy generation and distribution, underpinning a concerted worldwide effort to improve and modernise electricity supply networks. A major feature of this new electrical network is the move to supply our energy demands with clean, renewable energy sources such as solar panels and wind turbines, rather than fossil fuel based generation systems [3--5]. In electrical terms, this represents a fundamental change in energy generation, moving away from non-volatile sources (e.g. fossil fuel fired power stations) towards volatile, non-schedulable sources (e.g. solar panels, whose output can be extremely variable). To sustain the grid in the face of these fluctuations in energy generation, the Smart Grid must include non-volatile energy storage as part of its core structure, to provide grid support and 'ride-through' capability during times of reduced primary energy production. Possible energy storage appliances that have been proposed for this function include Plug-in Hybrid Electric Vehicles (P-HEV) and battery banks [4--7].

Connecting these energy storage devices to the Smart Grid is a challenging task, because most storage elements are electrically Direct Current (DC) in nature, while the Smart Grid uses Alternating Current (AC). To link these two very different forms of power, intermediate processing of the energy flow is required. This is achieved using *power electronic converters*, which are systems that use semiconductor switching devices to alter and manage the flow of electrical energy. They can therefore be used to convert this energy from one voltage level or frequency to another [8--13].

Power electronic conversion systems for such Smart Grid applications must meet two key design targets. First, safety regulations demand that they include galvanic isolation as part of their construction, almost invariably through a transformer.

Second, they must match the DC power flow required by energy storage devices to the fluctuating AC power flow of the Smart Grid. This task is quite challenging, as the power fluctuations in the Smart Grid are complex, ranging from the relatively consistent variation caused by the AC nature of the grid, to more severe transients caused by the volatility of Smart Grid energy sources. Managing this problem requires a converter that can achieve both a bi-directional power flow capability as well as high performance regulation. Bi-directional power flow is needed to allow charging of the energy storage elements during normal operation, as well as discharging when grid support is required. High performance regulation is required to enable effective and efficient management of this complex energy flow. These factors all combine to make design of the converter a complex task [3--5, 8, 9, 13, 14].

Modern solutions that achieve these targets use a two-stage power electronic converter. The first stage is a DC-AC inverter, which links the AC Smart Grid to an intermediate DC bus. The second stage employs a bi-directional DC-DC converter that couples the intermediate bus to the energy storage system while also providing galvanic isolation and voltage level translation (if necessary) [3, 4, 9].

DC-AC inverters have been the subject of significant research over the past two decades, exploring ways to improve the performance of these systems. As a result, there is a wealth of knowledge and algorithms available to optimise inverter design and performance. These range from advanced converter topologies (e.g. H-bridge inverters, flying capacitor multilevel inverters, etc.), to innovative modulation methods that produce high quality output waveforms, as well as enhanced closed loop regulation strategies that guarantee fast transient responses [8, 9].

However, the same is not true for bi-directional DC-DC converters. This area of research is not as mature, and several key research questions still remain unanswered. Of particular interest is the question of closed loop performance for these converters. When faced with a complex power flow profile (e.g. that of the Smart Grid), high performance regulation becomes a necessity, but the maximum achievable controller performance that can be achieved under these conditions has not been comprehensively identified, nor have the factors that underpin these limits been articulated.

This thesis addresses this issue. The central theme is to improve the performance of an isolated bi-directional DC-DC converter for a Smart Grid application by maximising its dynamic performance. This is achieved by developing a novel, high performance closed loop regulator. Towards this goal, a highly accurate converter dynamic model is derived, which is then used to construct the advanced closed loop

regulator. The factors that limit the performance of this regulator are also identified, ensuring maximised performance.

1.2 Objectives

The fundamental research objectives of this thesis are:

- To establish an accurate dynamic model of the bi-directional DC-DC converter. This model must include the non-linear effects of deadtime on output dynamics while still lending itself easily to closed loop controller design.
- To develop a closed loop control structure based on the previously derived dynamic model. This controller must give a fast response to transient events as well as provide good steady-state regulation.
- To determine the maximum achievable closed-loop performance. This involves identifying the factors that limit performance and designing an algorithm to optimise controller response based on these limits.
- To implement the proposed regulator on a suitable Smart Grid appliance, to verify the improvements achieved in terms of converter lifetime and reliability.

The following sections outline the overall thesis structure, as well as present a list of the significant contributions and a list of publications made during the course of the project.

1.3 Thesis Structure

This thesis is organised as follows:

Chapter 1 (this chapter) introduces the research context of this thesis, and pinpoints the fundamental research questions that this work addresses. It also provides an outline of the thesis structure (this section), and a list of publications made during the course of the research.

Chapter 2 presents a review of the current literature in the area of isolated bi-directional DC-DC converters, in terms of their topology, modulation, dynamic modelling and closed loop regulation. The first major finding of this chapter is that the dynamic models applied to this converter tend to either to be complex, or

have limited applicability. The next major finding is that most of these converters only deal with DC load conditions, not the AC load expected by the Smart Grid. Finally, although many closed loop controllers have been suggested, the controller performance limits have not yet been precisely articulated. It is concluded from this chapter that there is a need for a simpler, more flexible dynamic model that can be used to identify the limits of closed-loop controller performance, particularly in the context of an AC load.

Chapter 3 presents the derivation of the novel harmonic modelling technique, and applies it to the bi-directional DC-DC converter. The underlying principle of this technique is that converter dynamics can be expressed in terms of their switch states, which are time varying binary valued functions that represent the condition of the system switches. In order to solve the converter dynamic equations, these switching functions are broken down into a summation of significant harmonics using a Fourier Transform. The dynamic response of the converter to each significant harmonic is then determined, and summed together to give the full dynamic response.

The effect of deadtime on converter dynamics is also addressed in this chapter. It is identified that the flow of AC current during the deadtime period changes the effective converter operating point, changing the dynamic converter response. A piecewise linear closed form expression for this AC current is then developed, which allows the effect of deadtime at any operating condition to be determined analytically. The idealised harmonic model was extended to include this deadtime prediction, resulting in a simple yet accurate model of converter dynamics that successfully matches simulation predictions and reality.

Chapter 4 describes the development of a novel high performance closed loop regulator for the bi-directional DC-DC converter. Using the harmonic model derived in the previous chapter, an appropriate control structure and controller form are selected. Next, the effects of a digital controller implementation are identified as the primary factors that limit controller performance. This chapter then analytically quantifies these effects, resulting in a design procedure for a closed loop regulator that gives maximised transient performance across the entire converter operating range.

Chapter 5 extends the application of this closed loop regulator to an AC inverter load. This load inverter is necessary in order to link to the AC Smart Grid. The new closed loop regulator is applied to this system, and the benefits of the improved control architecture and high performance regulation are then described. This chapter also identifies that the major limitations of these systems is the large intermediate electrolytic capacitor, which has a limited lifetime. The

strong impact that high performance regulation can have on the required capacitance is demonstrated, potentially eliminating the need for these electrolytic capacitors, which has significant lifetime and cost benefits.

Chapter 6 provides a description of the simulated and experimental systems that have been developed to explore and verify the concepts presented in this thesis.

Chapter 7 presents salient experimental results from a prototype bi-directional DC-DC converter that was constructed in the laboratory to validate the proposed modelling and control schemes.

Chapter 8 concludes the thesis and suggests paths for future work in this area of research.

1.4 Identification of Original Contributions

This thesis presents several key contributions to the field of power electronic converters, which are listed in this section.

The first contribution is presented in Chapter 3, where the application of a generalised harmonic modelling strategy to the analysis of DAB bi-directional DC-DC converters is presented [15]. The development of Fourier series models for the converter switching functions is described, and the relationship between each significant harmonic and the overall dynamic response of the converter is identified. This model is then verified with detailed simulation results and matched to the experimental prototype in Chapter 7. The modelling methodology presented here is extremely powerful because it is not limited to DAB converters, but is general enough to be applied to any power electronic converter.

The second major contribution of this thesis is the analytical modelling of the effect deadtime has on bi-directional DC-DC converter dynamics, explored in Chapter 3. Although several authors have identified this effect, the compensation algorithms that have been suggested are heuristic in nature. This thesis presents a powerful analytic approach to modelling the effect deadtime has on this converter, by first identifying that during the deadtime interval, it is the flow of the AC inductor current that determines how converter dynamics are affected. A closed-form expression for this current is developed, which allows the effect of deadtime to be precisely determined. This is verified using detailed switched simulations, which are matched to the experimental prototype in Chapter 7.

The third major contribution of this thesis is the investigation into the limits of closed loop performance for this converter, and the subsequent development of

a high-performance closed loop voltage regulator, described in Chapter 4. It is shown that the sample and update delays caused by the digital implementation of the controller are the primary factors that limit controller performance. The nature of this delay is explored, and its effect analytically determined. This allows the maximum achievable controller gains to be calculated. The performance of this controller is verified in simulation as well as on the experimental prototype.

The fourth major contribution of this thesis is an optimised response to a load transient event. In Chapter 4, it is identified that the load current acts as a disturbance input to the closed-loop system, degrading transient performance. The precise effect of the load current is quantified, and a compensation algorithm derived and implemented, such that load transient performance too is optimised.

The fifth major contribution is presented in Chapter 5, and is the application of this new closed loop regulator to an AC load condition. It is identified that although the load power oscillates significantly, the new high performance voltage regulator is able to maintain the converter DC output voltage without the need for bulk capacitance. This potentially eliminates the electrolytic capacitor from these converters, with associated size, weight and lifetime benefits.

The majority of the ideas presented in this thesis have been published in IEEE conferences [16--19] and journal proceedings [20, 21]. These publications mark milestones in the research, and lend validity to the concepts presented by virtue of the peer review that is part of the publication process for these conference and journal proceedings.

1.5 List of publications

[1] D. Segaran, D. G. Holmes, and B. P. McGrath, ''Comparative analysis of single and three-phase dual active bridge bidirectional dc-dc converters,'' in Proc. Australasian Universities Power Engineering Conference (AUPEC), 2008, pp. 1--6.

[2] D. Segaran, D. Holmes, and B. McGrath, ''Comparative analysis of single and three-phase dual active bridge bidirectional dc-dc converters,'' Aust. J. Electr. Electron. Eng., vol. 6, no. 3, pp. 1--12, 2009.

[3] D. G. Holmes, B. P. McGrath, D. Segaran, and W. Y. Kong, ''Dynamic control of a 20kw interleaved boost converter for traction applications,'' in Proc. 43rd IEEE Industry Applications Society Annual Meeting (IAS), 2008, pp. 1--8.

[4] D. Segaran, B. P. McGrath, and D. G. Holmes, ''Adaptive dynamic control of a bi-directional dc-dc converter,'' in Proc. IEEE Energy Conversion Congress and Exposition (ECCE), 2010, pp. 1442--1449.

[5] D. Segaran, D. G. Holmes, and B. P. McGrath, ''Enhanced load step response for a bi-directional dc-dc converter,'' in Proc. IEEE Energy Conversion Congress and Exposition (ECCE), 2011, pp. 3649--3656.

[6] D. Segaran, D. G. Holmes, and B. P. McGrath, ''High-performance bi-directional ac-dc converters for PHEV with minimised dc bus capacitance,'' in Proc. 37th IEEE Annual Conference on Industrial Electronics (IECON), 2011, pp. 3620 -- 3625.

[7] D. Segaran, D. Holmes, and B. McGrath, ''Enhanced load step response for a bi-directional dc-dc converter,'' IEEE Trans. Power Electron., vol. 28, p. 371--379, 2013.

Chapter 2

Literature Review

The first step towards maximising the performance of an isolated bi-directional DC-DC converter is to review the limitations of existing systems. These converters have been the focus of significant research interest over the last three decades, which has resulted in an extensive body of literature. In order to properly manage the substantial number of publications and better review their contributions, this review groups the published material into four major areas:

- Topology Selection
- Converter Modulation
- Dynamic Modelling
- Closed-loop Control

Each of these research areas will be reviewed in turn, and the insight gained from each review section will then be applied to the next.

The first section (Section 2.1) begins by identifying the major converter topologies that have been used to achieve isolated bi-directional DC-DC conversion, summarising their principles of operation and contrasting their benefits and limitations. From this review, a suitable topology choice for a converter used in a Smart Grid application can be made. Section 2.2 then identifies the major modulation strategies that have been applied in this context and describes their fundamental operating principles, allowing an appropriate modulation strategy to be determined. Next Section 2.3 explores dynamic modelling, examining the techniques that have been presented in the literature to predict converter behavior by describing their underlying principles and identifying the benefits and drawbacks of each modelling approach. Finally, Section 2.4 summarises the closed loop regulation techniques that have been applied to these converter structures in the literature, and then analyses and compares their performance.

2.1 Topology selection

2.1.1 A Generic Structure for isolated Bi-directional DC-DC Converters

Almost all isolated bi-directional DC-DC converters reported in the literature follow the generic structure shown in Fig. 2.1, and are essentially made up of two switching converters connected via an intermediate AC link that includes an isolation/scaling transformer [22--30].

The primary side converter converts the incoming DC voltage to an AC waveform, which is applied to the intermediate transformer. The secondary converter then rectifies and filters this AC signal, creating a DC voltage that can be applied to a load. The symmetry of this structure allows the primary and secondary converters to swap roles without issue, allowing bi-directional power flow through the converter.

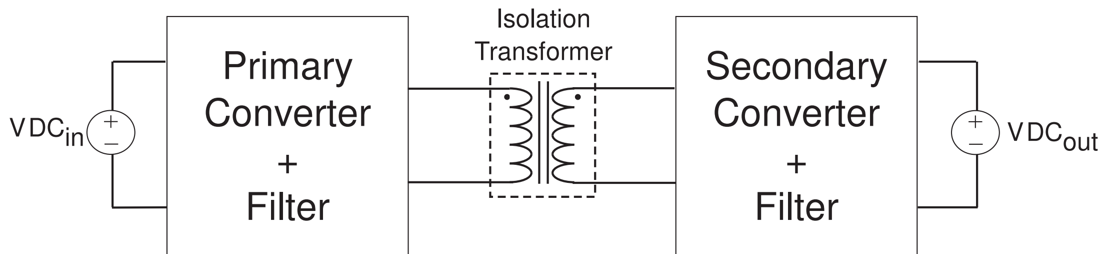


Figure 2.1: The Generic Bi-directional DC-DC Converter Topology

The literature has identified numerous topological alternatives for the primary and secondary converter. The major topologies proposed are:

- Flyback Converters
- Current Fed Push-pull Converters (CFPP)
- Bridge Converters (Half-bridge, Full-bridge, etc.)

Note: While the use of matrix converters for isolated bi-directional conversion has also been reported in the literature [31--39], these converters are mostly only used where an isolated AC-AC interface is required. Hence, they will not be explored any further in this thesis, which focuses on DC-DC conversion.

The choice of topology is substantially dependent on the converter ratings, i.e. upon the required voltage range and desired power level. This is summarised in refs. [26, 29, 30, 40--42], which identify suitable voltage and power ranges for each topology. This concept is discussed further in the following subsections, where the

operating principles of each topological alternative are described, together with their advantages and limitations.

2.1.2 Flyback Converters

Fig. 2.2 shows the basic circuit topology of a flyback converter. This is a well known isolated DC-DC converter structure, popular for its reduced component count since it does not require any output filter inductors, and its ability to simultaneously supply several different voltage levels simply by using a transformer with multiple output windings [10, 11].

The operation of this converter is explained with the aid of Fig. 2.3. When switch S_1 is turned *ON*, current flows through the primary side of the converter, charging the magnetising impedance of the isolating transformer. When S_1 is turned *OFF*, the current freewheels through the secondary winding of the transformer, supplying the load.

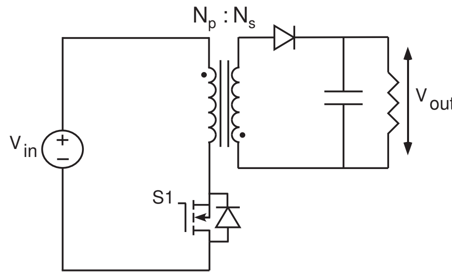


Figure 2.2: A simple Flyback Converter

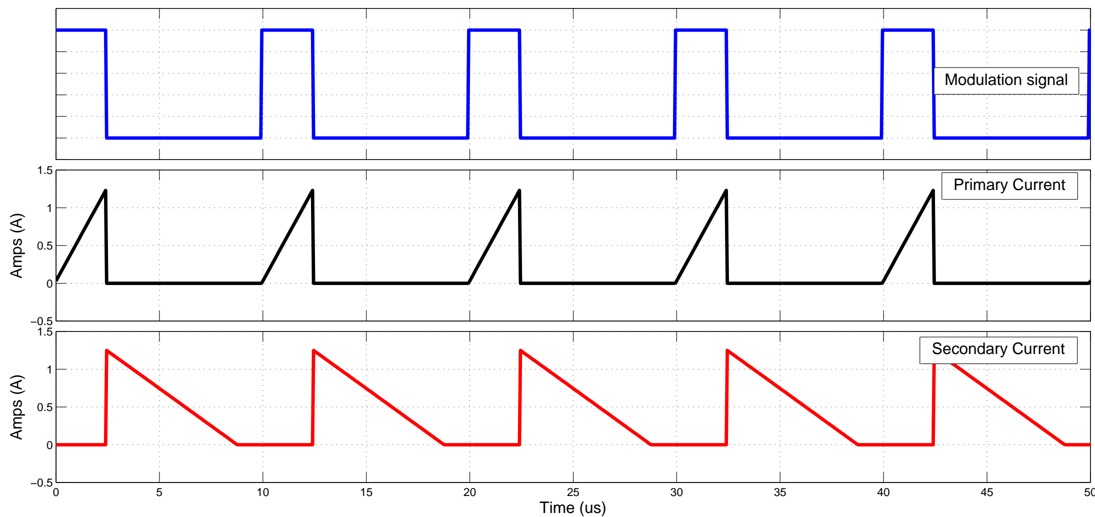


Figure 2.3: Flyback Converter Operating Waveforms

However, this structure is uni-directional in nature. To handle bi-directional power flow, [1, 41, 43, 44] suggests connecting two such converters back-to-back, as shown in Fig. 2.4.

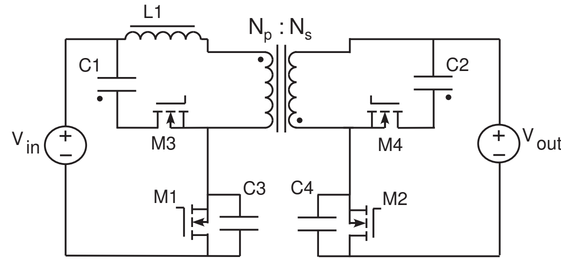


Figure 2.4: The Actively Clamped Bi-directional Flyback Converter [1]

This circuit is called an *actively clamped bi-directional flyback converter.* Bi-directional power flow is achieved by modulating either M_1 or M_2 depending on the desired power flow direction. The active clamp circuits (i.e. MOSFET-capacitor pairs M_3-C_1 & M_4-C_2) as well as the parallel capacitances of C_3 & C_4 are used to help the converter achieve soft-switching[1].

While flyback converters are a simple topology, they suffer from two key limitations. Firstly, the discontinuous current that usually flows in this converter causes relatively high peak currents to occur for a given power rating, illustrated in Fig. 2.3 [10, 41]. This reduces converter efficiency and increases switch ratings. Secondly, a large transformer magnetising inductance is required because all the converter energy is stored within it during converter operation. Lastly, in order for this converter to achieve efficient and effective energy transfer, a transformer with very low leakage inductance is required. This makes the transformer design very challenging, especially as power levels rise [10, 41].

As such, flyback based isolated bi-directional DC-DC converters are only attractive at low voltage and power levels (<100 V, <500 W) [1, 41, 43, 44].

2.1.3 Current fed Push-pull Converters

The second major topology that has been proposed for isolated bi-directional DC-DC converters is the current fed push-pull (CFPP) converter, whose topology is illustrated in Fig. 2.5. This is a popular switch mode power supply topology due to its simplicity and good power-to-weight ratio [11, 41, 45, 46].

The operation of this converter is illustrated in Fig. 2.6. During the overlap period both switches S_1 and S_2 are turned on, so the current builds up in the inductor L_1. When only switch S_1 is on, a net positive voltage appears on the transformer secondary (V_{sec}). Conversely, a net negative voltage appears when only S_2 is on. The resulting AC waveform is rectified to generate an isolated DC output voltage.

[1] Soft-switching concepts are discussed in Section 2.2.

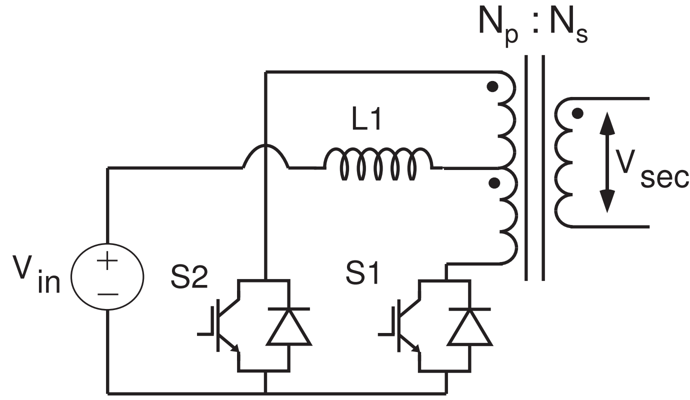


Figure 2.5: A Current fed push-pull converter

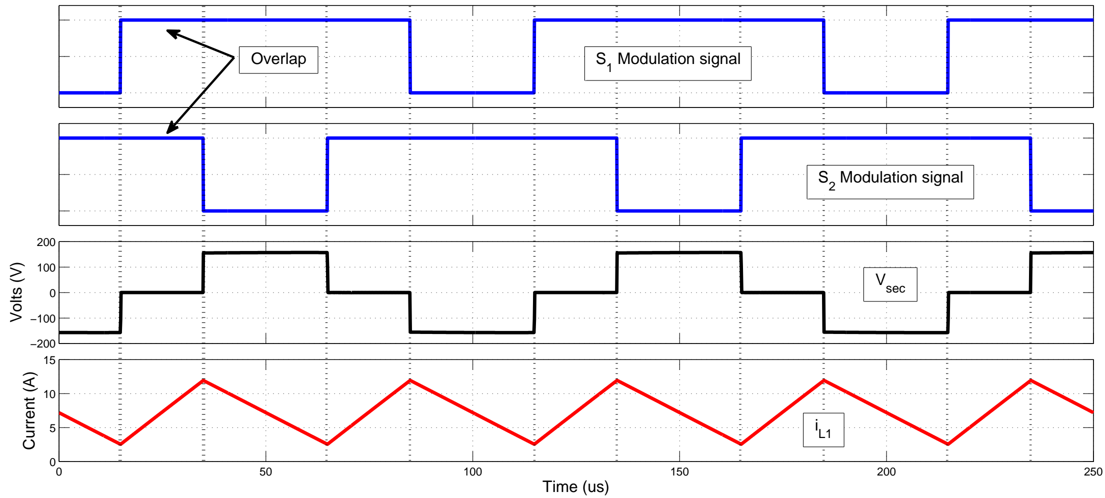


Figure 2.6: CFPP Operating Waveforms

A major advantage of this topology over its voltage-fed counterpart (which does not include a DC inductor in its construction) is that it avoids staircase saturation of the transformer, which is a major failure mode of voltage-fed push-pull converters. This effect occurs when circuit non-idealities cause an imbalance in the modulation signal [10, 11]. This means that the voltage applied to the transformer has a DC component, which grows with every switching cycle. The current that is generated due to this DC voltage component eventually saturates the transformer core, resulting in an over voltage event that often causes converter failure [10,11]. CFPP converters avoid this hazard by including the inductor L_1 in its construction, which allows the input current to be regulated. Any DC component in this current waveform is then eliminated using closed-loop control, thus avoiding staircase saturation.

To achieve bi-directional power flow, a secondary converter is coupled to the transformer secondary. This secondary converter does not necessarily have to be another push-pull converter. For example, Fig. 2.7 shows how a half-bridge converter[2]

[2] Half-bridge converter operation is discussed in the following subsection.

is linked to the transformer secondary [41, 47]. Converters that employ different topologies within their structure are known as a *hybrid converters*. These converters are commonly utilised in the literature when primary and secondary voltages differ greatly, as is the case in [47], where a 48 V battery bank is linked to a 350 V output. Combining two converter topologies in this way is advantageous because each converter topology is used to its best advantage. It is important to note that an intermediate filter/impedance between the two converters is essential to provide voltage limiting (in a current source system) or current limiting (in a voltage source system). This is reflected in the hybrid converter of (Fig. 2.7), as the DC filter inductor L_1 limits the current that flows between the two converters during operation.

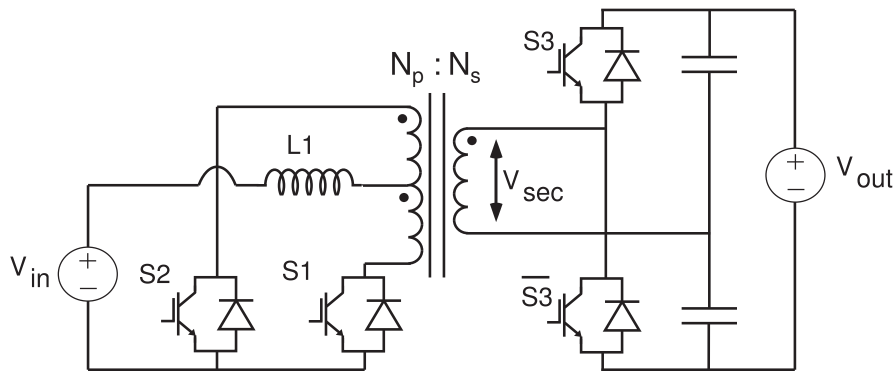


Figure 2.7: An isolated bi-directional DC-DC converter using a CFPP [41, 47]

The literature has proposed several applications for a CFPP converter as part of an isolated bi-directional converter, i.e. Power Factor Correction (PFC) systems [48], inverter/battery chargers [49], Fuel Cell systems that need to be linked to batteries [26] or supercapacitors [41], UPS systems with battery storage [28, 47], and Hybrid Electric Vehicles (HEVs) [27, 50--52].

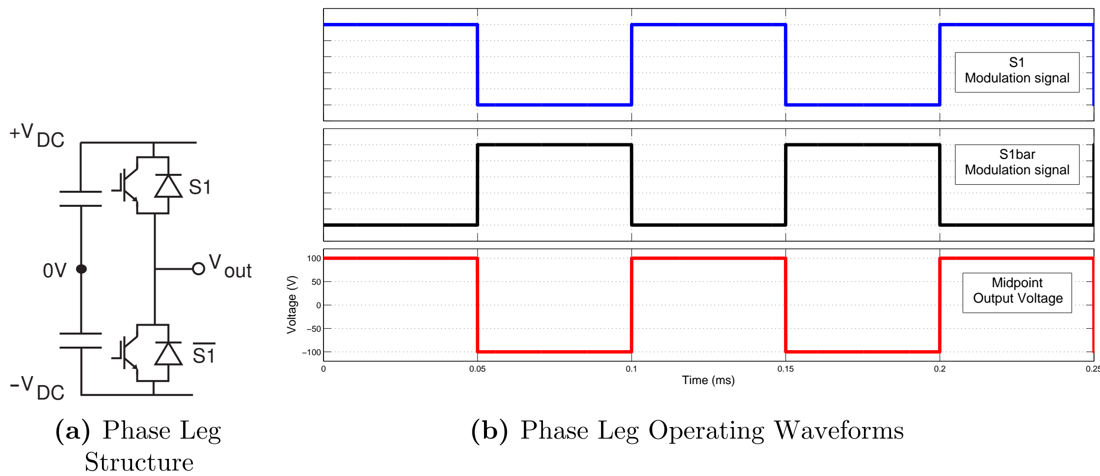
However, this topology has one significant drawback. During each switching cycle, the inactive switch must block double the input DC voltage ($2V_{in}$) [10, 11]. As a result, the switches for CFPP converters require a high blocking voltage rating, making them more expensive. This usually limits the applicability of CFPP converters to lower voltage and power level applications, i.e. below 400 V and 2 kW [27, 41, 47, 50, 51].

2.1.4 Bridge Converters

The bridge converter is the most common power electronic converter structure used for isolated bi-directional DC-DC converters because of its versatility and high power density [53, 54].

All bridge converters are made up of *phase legs*, which are two switches, series-connected across a DC link, as shown in Fig. 2.8a. The phase leg operates by turning the switch pair on (and off) in complementary fashion, as shown in Fig. 2.8b. This oppositional switching causes the voltage at the phase leg output (V_{out}) to switch between the upper and lower voltage rails ($+V_{DC}$ and V_{DC}) [10, 12].

Since switching devices have non-zero and potentially asymmetric turn-on and turn-off times, a blanking time is inserted between the two gate signals to ensure that the two switches in a phase leg are never conducting simultaneously, as this causes a destructive short-circuit condition known as *shoot-through*. This blanking time is known as *deadtime*, and is common to all voltage-fed bridge converter structures [10--12].



(a) Phase Leg Structure

(b) Phase Leg Operating Waveforms

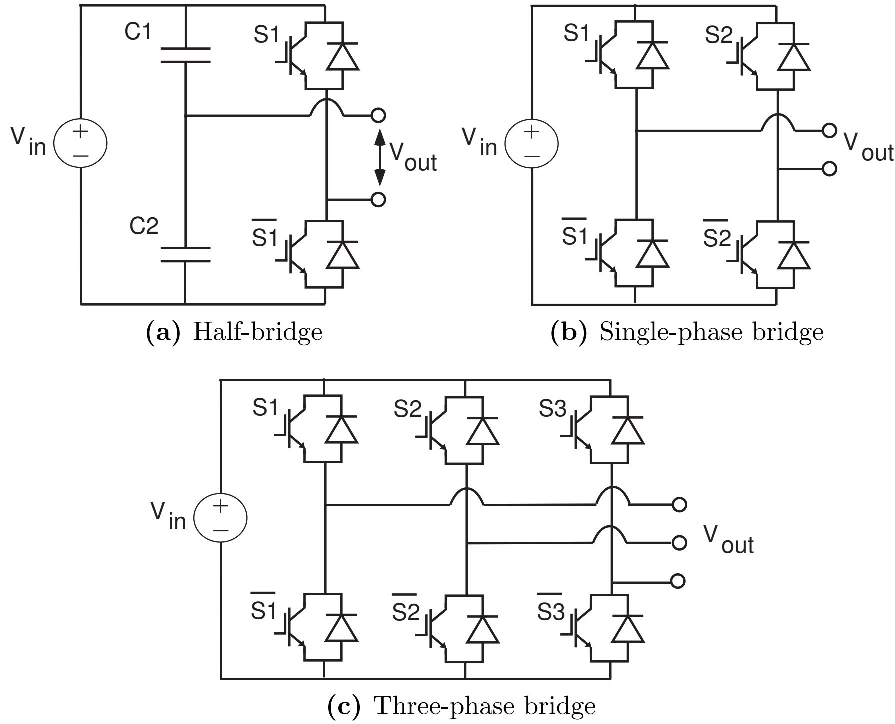
Figure 2.8: Phase Leg Topology & Operating Waveforms

Phase legs can be combined to form the three most common bridge structures, i.e. the half-bridge, single-phase and three-phase bridge topologies, as shown in Fig. 2.9 [10, 11].

Half-bridge converters

Half-bridge converters (Fig. 2.9a) consist of a single phase leg in parallel with a split capacitor bus. They are a popular topology choice for isolated bi-directional DC-DC converters [40, 41, 46], used in UPS systems [30, 47], Fuel Cell converters (often for Electric Vehicle applications) [2, 25--27, 30, 55--58] and even photovoltaic arrays [59].

Fig. 2.10 shows a half-bridge based topology that achieves bi-directional power flow – the Dual Half Bridge converter. Each bridge of this converter is modulated



(a) Half-bridge **(b)** Single-phase bridge

(c) Three-phase bridge

Figure 2.9: Bridge Converter Topologies

to produce an AC waveform across the intermediate link[3], while the inductive filter L limits the current between the two bridges.

Although half-bridge converters offer a reduced switch count advantage compared to their single and three-phase bridge counterparts, their primary drawback is the size and cost of the DC capacitors required (C_1 – C_4 in Fig. 2.10). These capacitors must also sustain large ripple currents, as the full AC current (i_L) must flow through them during operation [10--12]. As power and voltage levels rise, these capacitors become prohibitively bulky and expensive [10, 11].

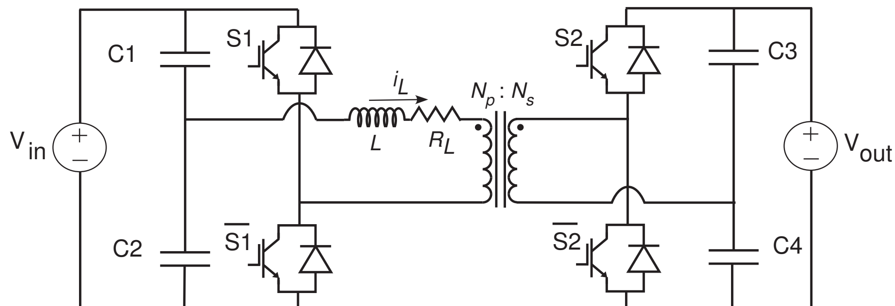


Figure 2.10: An isolated bi-directional DC-DC converter using a half-bridge [58].

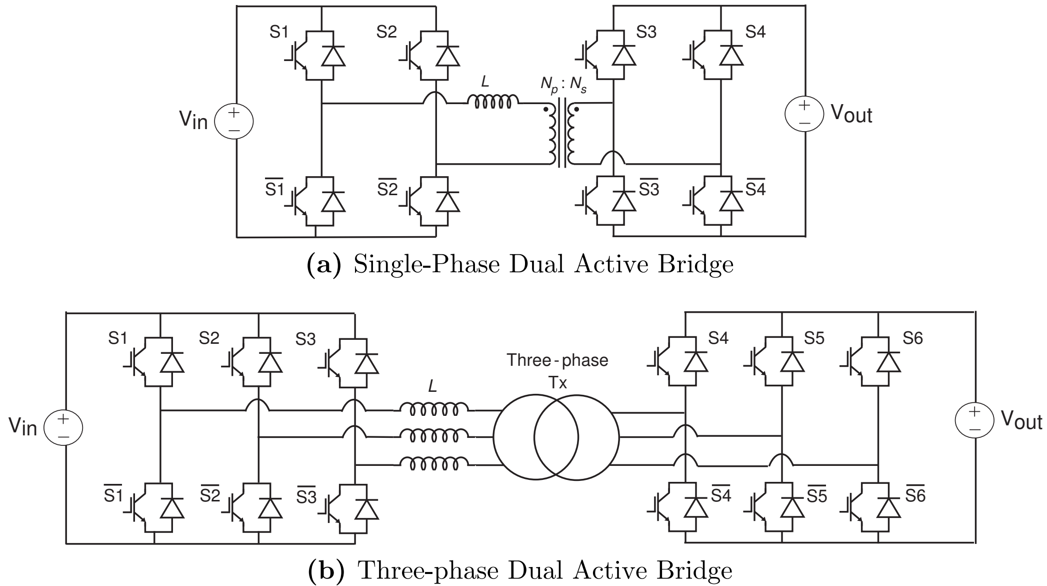
Consequently, the ratings of half-bridge topologies are limited to below 400 V and 2 kW [10, 11, 53].

[3] Converter modulation will be addressed in the following review section.

Full-bridge converters

Full-bridge converters, such as single-phase 'H-bridge' converters and three-phase bridge converters, are a very popular alternative for construction of isolated bi-directional DC-DC converters. These structures are made up of two and three phase legs respectively, as shown in Figs. 2.11a & 2.11b, and are known as Dual Active Bridge (DAB) converters [53].

DAB converters have a relatively high switch count (8 devices & 12 devices for the single and three-phase bridges respectively) compared to the half-bridge converter presented earlier, but they do not suffer from high capacitor ripple currents. This is because while the AC inductor current (i_L) flows through the DC capacitors in a half-bridge converter, in a full-bridge converter, it flows through the active switches (or their anti-parallel diodes) instead.



(a) Single-Phase Dual Active Bridge

(b) Three-phase Dual Active Bridge

Figure 2.11: Dual Active Bridge Bi-directional DC-DC Converter Topologies [53]

In order to choose between these two alternative full-bridge structures, the benefits and drawbacks of each topology must be evaluated. Fundamental AC circuit theory has been used to compare the single-phase and the three-phase topology alternatives, and predicts significant advantages in favour of the three-phase bridge, such as:

- **Reduced Current Stress**
  The current in the three-phase converter is shared between more phase legs than for the single-phase converter, reducing the current stress on the devices [20].

- **Constant Power**
  During operation of a single-phase DAB converter, the total power flow through

the converter is AC. This requires a large DC link capacitor to absorb the oscillations in the energy flow. However, in a three-phase DAB converter, the total energy flow is DC. This is because the 120° phase offset that exists between each phase leg cancels the AC component of the total energy flow, leaving a constant flow of power [60, 61].

The DC link capacitance of a DAB converter depends on the flow of power through it. To maintain a constant DC bus, this capacitance must be large enough to absorb any oscillations in total power flow. The constant power flow seen by the three-phase converter should therefore lead to a smaller DC link capacitance, with potential size and cost benefits.

- **Flux cancellation**
  When three-phase current flows into a transformer, the 120° offset between the phase currents generates flux that is also offset by 120°. Assuming a balanced system, the summation of these fluxes is zero, so the required transformer core material should be reduced [61, 62].

These issues have been examined and evaluated in detail in publications such as [20, 53, 63]. However, the conclusion drawn from these papers is that the theoretical benefits of a three-phase structure presented above do not translate for practical converters. Firstly, while the lower peak current seen in three-phase converters reduces device current stress, any loss benefit is negated by the higher switch count [20, 63]. Secondly, any potential size reduction benefits for the three-phase transformer are almost completely negated for thermal reasons, since the smaller core does not provide enough surface area to dissipate the heat generated by the magnetic/ohmic losses [19, 63].

Bridge converters in general are very flexible in their application, and are the most popular topology choice for isolated bi-directional DC-DC converters. They are used at voltage levels up to 1 kV and quite high power levels, such as Pavlovsky et al. [64] who constructed a 50 kW DAB converter system. These systems are so popular that they have appeared in over 100 research papers, focusing on a variety of different aspects of converter operation. For example, papers such as [54, 64--66] explore converter construction to attain high power density. Others, such as [30, 52, 53, 67] investigate soft-switching techniques for maximising converter efficiency[4], while others, e.g. [68--70] look to improve closed-loop converter performance, just to name a few. The particular contributions of the most significant of these papers will be discussed in later sections of this literature review.

[4] Soft-switching is examined in greater detail in Section 2.2.

2.1.5 Summary – Topology

Refs. [26, 29, 30, 40--42] conclude that the choice of converter topology for isolated bi-directional DC-DC converters is primarily based on the required converter ratings.

Table 2.1 summarises the reviewed converter structures and the appropriate limits of each topology that has been presented in this review. At low voltage and power levels, flyback converters are popular, but as ratings increase beyond 100 V and 1 kW, current fed push-pull converters and half-bridges become more appropriate. As voltage and power levels rise still further (400 V, 2 kW and above), full-bridge converters become the topology of choice.

	Flyback Converter	**Current-fed Push-pull Converter (CFPP)**	**Bridge Converters**	
			Half Bridge	**Full Bridge**
Voltage Rating	Low (<100V)	Low (100-400V)	Low (<400V)	High (>400V)
Power Rating	Low (≈500W)	Medium (>2 kW)	Medium (>2 kW)	High (>2kW)

Table 2.1: Converter Topology Comparison

This review suggests that a single-phase full-bridge topology is the most appropriate for a higher power Smart Grid application, so the other alternative topologies will not be considered any further in this thesis.

2.2 Modulation

In this section, the modulation strategies that have been applied in the literature to full-bridge isolated bi-directional DC-DC converters are presented and their key features described. The two key modulation strategies that have been applied to these converters are Pulse Width Modulation (PWM) & block modulation [10, 12, 53]. This section describes both these strategies in terms of the H-bridge converter of Fig. 2.12, then evaluates their benefits and drawbacks.

2.2.1 Pulse Width Modulation

PWM is one of the most popular bridge converter modulation schemes. Many different types of PWM schemes have been proposed in modulation literature,

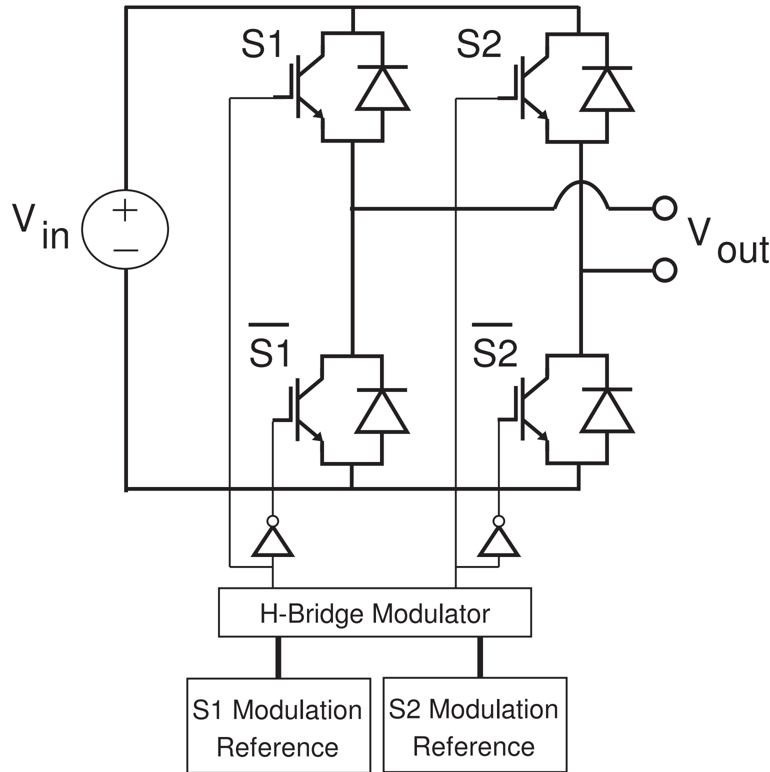


Figure 2.12: H-bridge and Modulator

ranging from Naturally Sampled & Regular Sampled PWM through to Discontinuous Modulation schemes, to Space Vector modulation strategies [12]. All these modulation strategies share a common operating principle, i.e. a high frequency switching pulse train whose widths vary more slowly to give a Low Frequency average (fundamental) output AC waveform [12].

This is illustrated in Fig. 2.13, which shows the operation of a Naturally Sampled sine-triangle PWM modulator. A high frequency triangular carrier signal is compared to a lower frequency modulation reference to give a PWM switching pattern.

This strategy is popular in power electronics because the output AC waveform has very low levels of distortion. This is because PWM ensures that the bulk of the waveform energy is transferred at the frequency of the fundamental harmonic. However, this relatively low frequency of energy transfer leads to bulky magnetic components [10, 62].

2.2.2 Block Modulation

Block modulation is made up of a train of high frequency pulses of constant width. The two main types of block modulation are two-level and three-level modulation, illustrated in Fig. 2.14. These modulation patterns are generated by modulating each phase leg of a bridge converter with square waves, so the difference between

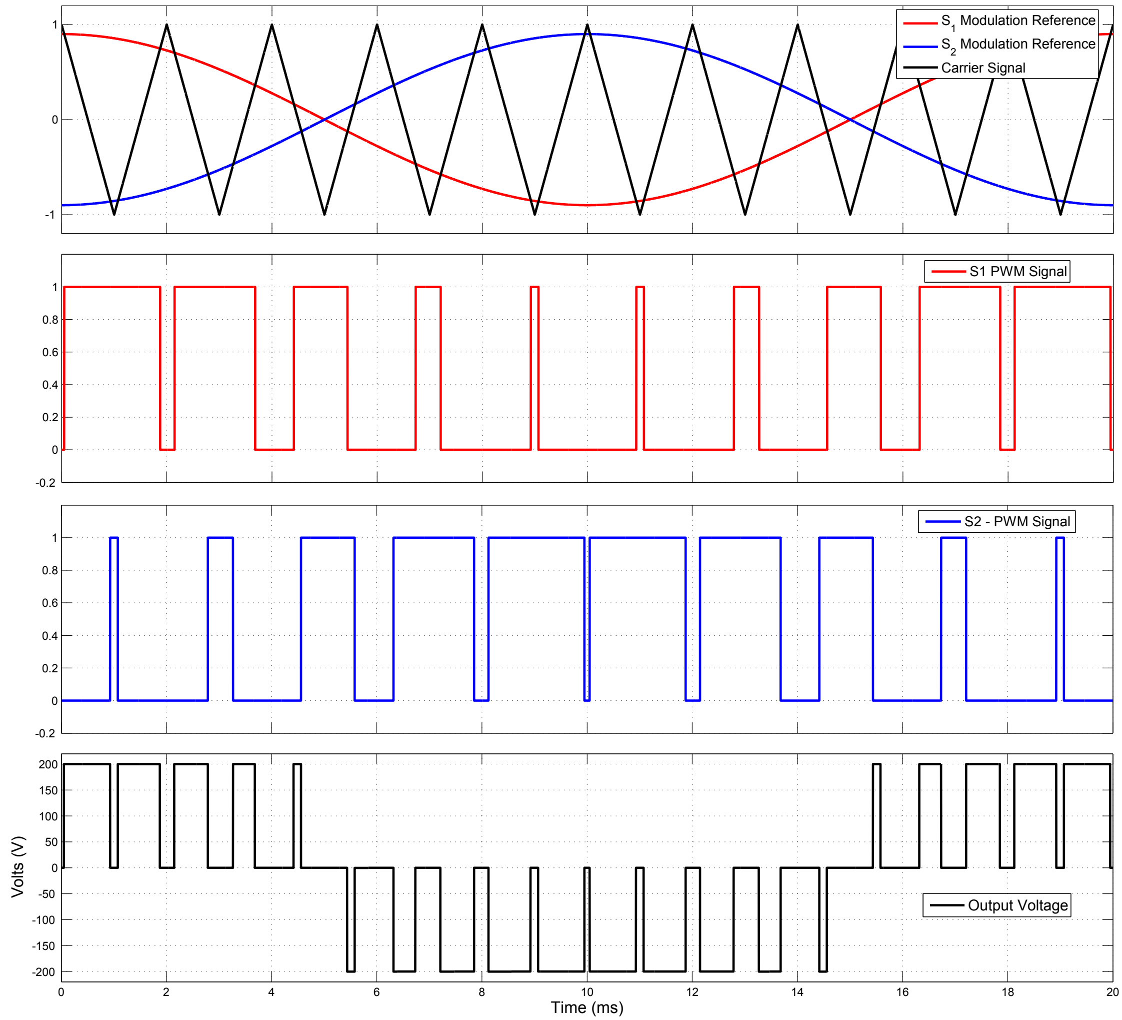
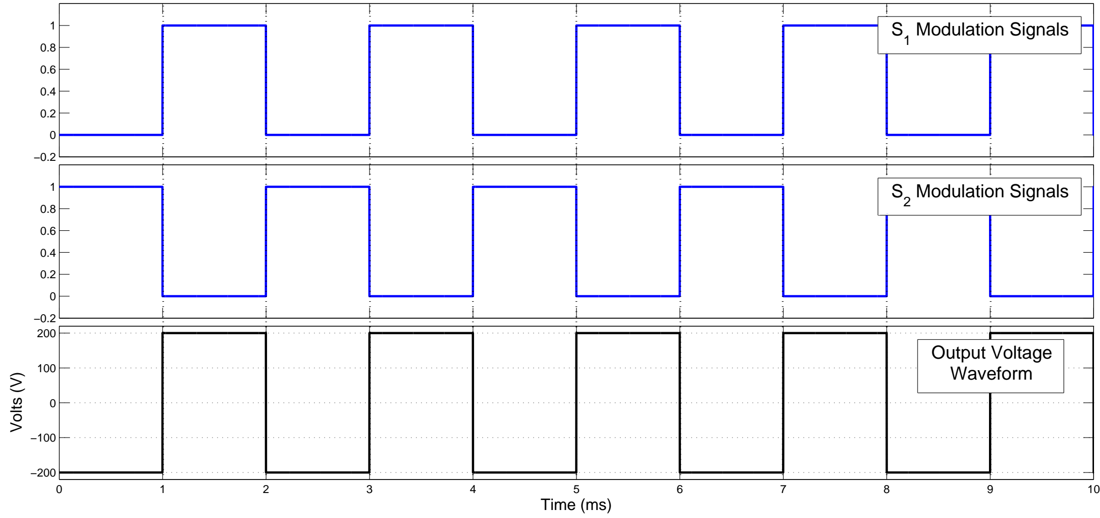


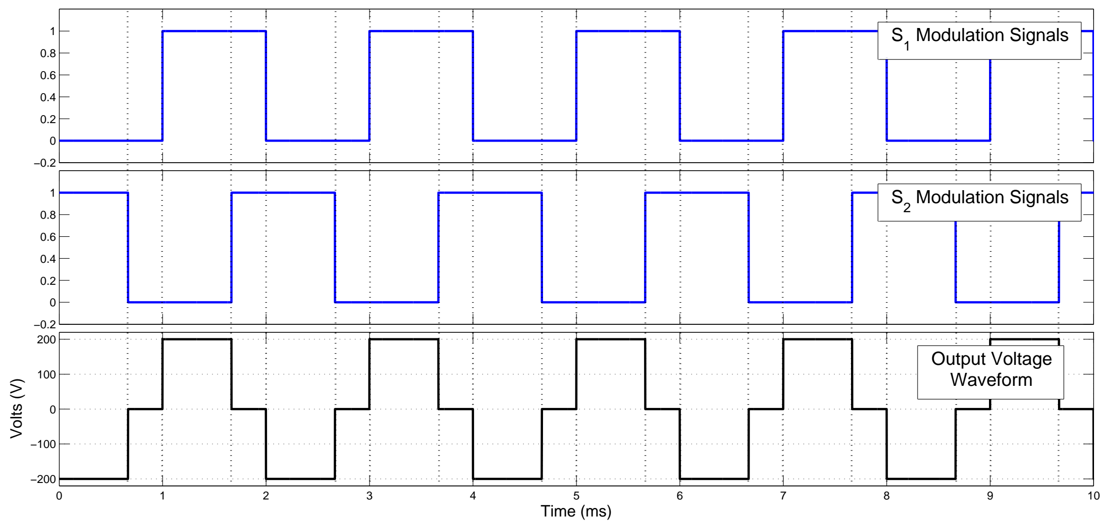
Figure 2.13: Pulse Width Modulation

the two waveforms appears on the bridge output terminals. From the waveforms of Fig. 2.14, it can be seen that the only difference between the two schemes is that two-level modulation has a constant duty ratio (50%) while three-level modulation has a variable duty ratio [12]. This modulation scheme is also known as Phase Shifted Square Waves (PSSW).

Unlike PWM, block modulation does not have a low frequency average output. Instead, the waveform energy is transferred at higher frequencies, i.e. at the switching frequency and its higher order harmonics [12]. This has the potential for smaller, lighter magnetic components (e.g. inductors, transformers), as identified in [39, 41, 43, 62--66, 71--76]. It also can give a faster dynamic response, as suggested by [12, 18, 20, 77], because the flow of energy can be changed and varied more quickly.



(a) Two-level Modulation



(b) Three-level Modulation

Figure 2.14: Block Modulation

2.2.3 Soft-switching

Switching loss is the loss of energy incurred each time a switching device turns on or turns off, illustrated in Fig. 2.15 [10]. This figure shows that device turn-on and turn-off events do not occur instantaneously, so if the voltage across the switching device and the current flowing through it is non-zero during this interval, there is a short period of elevated loss. This loss scales up with switching frequency (since more transitions occur) and power level (since more energy is lost per switching event), and is one of the major loss mechanisms in power electronic converters [10, 11, 61].

Soft-switching aims to minimise this loss by ensuring that switching events only occur when the voltage across the device or the current through it is zero. One of

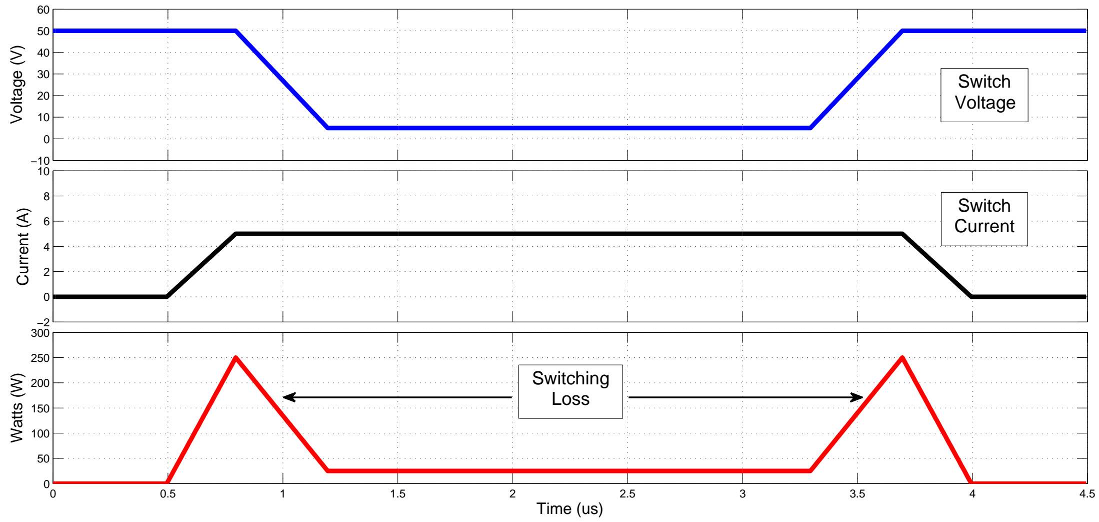
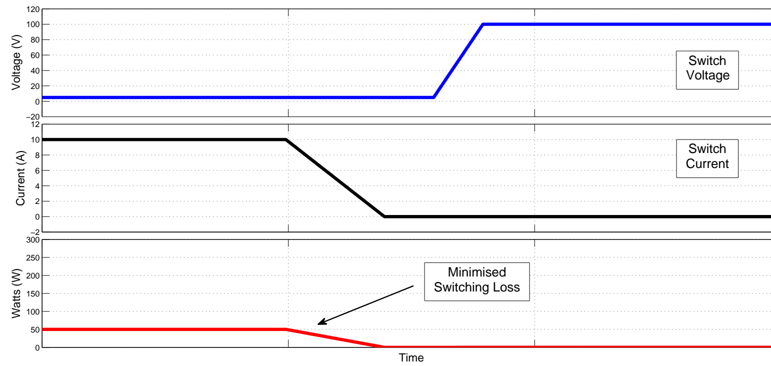
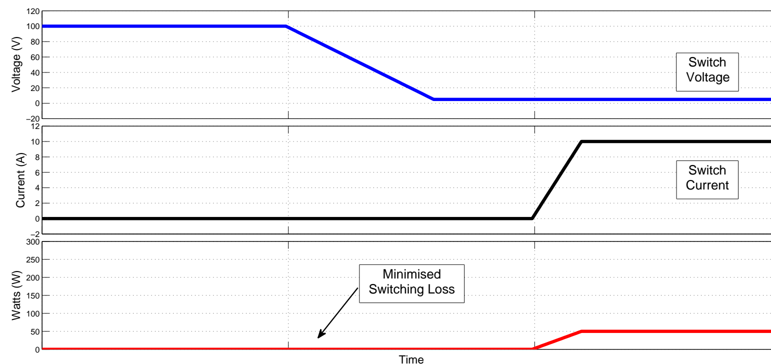


Figure 2.15: Switching Loss

the earliest views of this idea was presented by Divan et al. [78] in the 1980s to help minimise the switching loss in power electronic converters.



(a) A ZVS Transition



(b) A ZCS Transition

Figure 2.16: Ideal Soft-switching Waveforms

The two major soft-switching modes are known as Zero Voltage Switching (ZVS) and Zero Current Switching (ZCS) [78]. This is illustrated in Fig. 2.16 – ZVS is achieved in Fig. 2.16a because the voltage across the switch is held low as it turns

off (i.e. its current drops to zero). In similar fashion, ZCS is achieved in Fig. 2.16b because the current through the switch is held at zero until the switch turns on (i.e. the voltage across it collapses).

ZVS & ZCS are achieved by adding auxiliary resonant components (e.g. capacitors and/or inductors) to the converter structure. Exciting the resonance between these components creates an oscillatory voltage/current waveform, and soft-switching is achieved by then adjusting the primary device switching instants to coincide with the zero-crossings of this oscillation.

Research into soft-switching strategies has been a major research focus for isolated bi-directional DC-DC converters. From this work, three techniques stand out as the most commonly used approaches, i.e.:

- **Parallel Device Capacitance**

  The circuit diagram of this technique is shown in Fig. 2.17, and involves augmenting the parasitic capacitance of the switching devices with another parallel capacitor (C). ZVS is thus achieved at turn-off because the capacitor holds the voltage across the device low as the device turns-off.

  To also achieve ZVS at turn-on, the switching event is timed to occur at the zero-crossings of the resonance between the capacitance and the transformer leakage inductance.

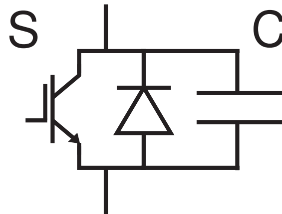


Figure 2.17: Parallel Capacitance

  In DAB isolated bi-directional DC-DC converters, non-ideal features such as the parasitic capacitance of the switching devices and the leakage inductance of the AC transformer can help achieve natural soft-switching at some operating conditions [53, 57, 79--81]. However, this effect is strongly dependent on the current in the leakage inductance, so the soft-switching range is often limited. The literature in this area has primarily explored extending this range by augmenting the natural device capacitance with external capacitors.

  The primary limitation of this technique is that ZVS at turn-on is dependent upon the energy in the transformer leakage inductance, which means it is load dependent. It is therefore difficult to ensure soft-switching across the entire load range [53, 82--85]. However, the simplicity of this technique makes it very

popular, and features in numerous publications, achieving soft-switching for a variety of applications, such as Electric Vehicles, UPS systems and Fuel Cell converters [23, 37, 53, 71, 85--91].

- **Active Clamp Circuits**
  A basic active clamp circuit is shown in Fig. 2.18, and consists of a DC inductor (L) in series with the source, and an auxiliary capacitor (C) with a series switch (S_a).

  The system operates by modulating switch S_a to excite the resonance between the series inductor L and the parallel capacitor C. This results in an oscillatory waveform on the DC link (V_{res}). The full-bridge is then switched such that the turn-on and turn-off events occur at the zero-crossings of this resonant voltage waveform, ensuring ZVS.

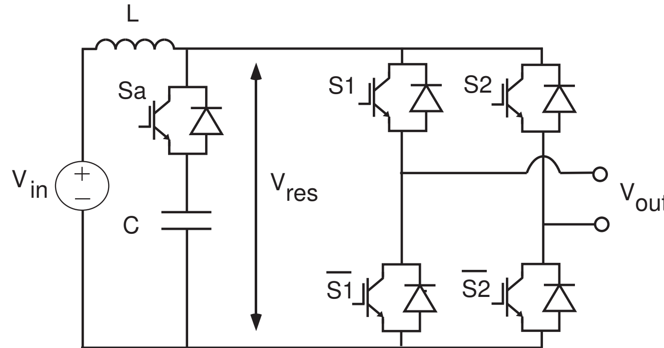


Figure 2.18: A bridge converter with an active clamp [26]

This soft-switching technique is not as common as the parallel capacitance technique because it needs additional resonant components as well as an active switch, leading to increased cost and more complex control requirements. However, in the literature, this technique has still been successfully applied in many publications, which explore soft-switching in the context of Electric Vehicles as well as Fuel Cell systems [1, 22, 23, 26, 28, 30, 44, 89, 92, 93].

- **Series-resonance**
  The series resonance soft-switching technique is illustrated in Fig. 2.19, and uses an AC capacitor (C_{series}) in series with the transformer leakage inductance (L_{leak}). The switching processes of the bridge converter excites this resonance, resulting in an oscillatory output waveform (V_{out}). Soft-switching (ZVS) is achieved by ensuring that the bridge switching transitions take place at the zero-crossings of the resonant voltage waveform.

  The main drawback of this method is that C_{series} must withstand the rated voltage and current of the converter. As converter ratings rise, the size and cost of this capacitor becomes prohibitively large.

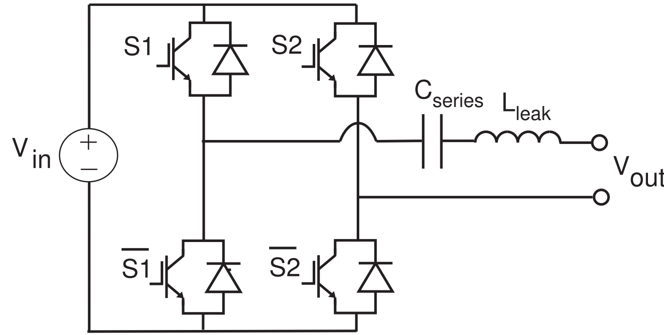


Figure 2.19: A series resonant bridge converter

> The literature in this area focuses on several different aspects of this converter, e.g. dynamic modelling and control[5] [23, 90, 94, 95], magnetics design [76], maximising efficiency and power density [66, 75, 77, 96, 97], as well as their applications, such as electric vehicles [77,97] and telecoms applications [66,75].

Although the potential advantages of soft-switching are compelling, it has some significant drawbacks. [53,82,98] have shown that it is very difficult for a converter to maintain soft-switching at all load conditions. For instance, H-bridge converters are unable to maintain soft-switching at lower load load conditions as there is insufficient load current to charge the ZVS capacitors. This causes the switch transitions to revert to being hard-switched in nature, limiting the available operating range for the converter and making it less flexible in application. Attempts to improve and extend this range, either with resonant components or with auxiliary circuitry, tend to increase converter cost, size and complexity, reducing its feasibility at higher power and voltage levels [53, 82].

Furthermore, the fundamental operating principles of hard-switched and soft-switched converters are essentially identical in nature, as concluded by de Doncker et al. [53, 80]. This is because although the switch transitions in soft-switched converters are resonant and require a finite time to complete, first-order analysis can assume that they occur almost instantaneously, considerably simplifying converter analysis.

2.2.4 Summary – Modulation

This section has reviewed the major modulation techniques that have been applied to isolated bi-directional DC-DC converters, so the selection of an appropriate modulation strategy for a Smart Grid application can be addressed.

[5] The closed-loop regulation and the associated dynamic models of these converters will be addressed later on in this chapter.

From this review, Phase-Shifted Square Wave block modulation is the more attractive strategy for higher power converters because it requires smaller magnetic components, and can also achieve a fast dynamic response. Of the PSSW strategies presented, two-level modulation seems more attractive for Smart Grid applications because it achieves the maximum power transfer for a given operating condition [53].

This review also suggests that hard-switching can be more attractive than soft-switching for Smart Grid applications since it is cheaper to implement and yet can still achieve comparable levels of converter performance, which is the primary focus of this thesis [53].

2.3 Dynamic Modelling

Having reviewed the different topologies that have been applied to isolated bi-directional DC-DC converters as well as their modulation techniques, this literature review now shifts focus to the dynamic modelling and control of these converters.

An accurate dynamic model is essential for the design of a high performance closed loop controller [99, 100]. Without such a model, regulator design is essentially a heuristic process and maximised performance is not guaranteed.

A dynamic plant model is a series of mathematical equations that describe the relationship between the output conditions of a system based on input stimuli [99, 100]. These models are time-based in nature, as they need to manage the time-varying nature of the system inputs and outputs. Therefore, differential or difference equations[6] are usually employed in this context because they lend themselves easily to time-domain analysis [99, 100].

The models that have been presented in the literature to predict the dynamic behaviour of isolated bi-directional DC-DC converters can be grouped into two families, i.e. models based on state averaging techniques, and models based on the fundamental power flow. This section describes the underlying principles of these dynamic models, and evaluates the benefits and drawbacks of each one.

[6] Differential equations and difference equations are used for continuous-time and discrete time systems, respectively.

2.3.1 State Averaged Models

State averaging is a popular modelling technique that is very powerful when applied to power electronic converters, so a wide body of literature exists in this area [10, 101, 102].

The underlying principles of state averaged modelling are outlined here, using a simple Buck DC-DC converter as an example [103]. Fig. 2.20 shows the circuit diagram of the Buck converter as well as its operating waveforms. Assuming continuous conduction of the inductor L, there are two clear modes of operation, i.e. when switch S_1 is turned on, and when switch S_1 is turned off. When switch S_1 is turned on, the inductor L is charged by the input voltage source (V_{in}), so the current ramps up. Conversely, when switch S_1 is turned off, the inductor current ramps down, freewheeling through diode D_1.

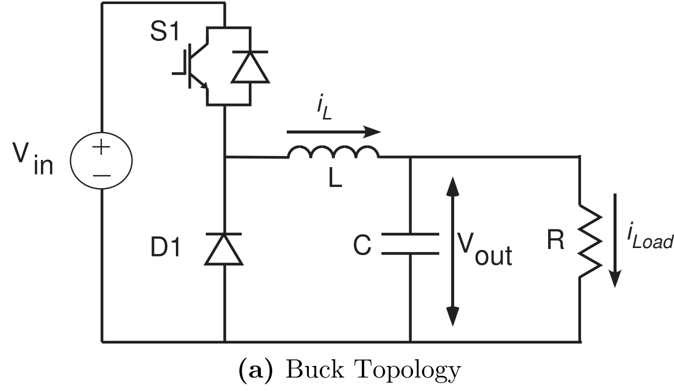
The transition between these two modes is assumed to be instantaneous because although device turn-on and turn-off times are non-zero, they are generally designed to be only a small fraction of the total switching cycle [10, 11]. The system can therefore be described as switching between two modes of operation during the course of a single switching cycle.

To model the Buck converter, the inputs, outputs and internal state variables of the system in each operating mode must first be identified. The system inputs are the input voltage (V_{in}) and the load current[7] (i_{load}), while its output is the capacitor voltage V_{out}.

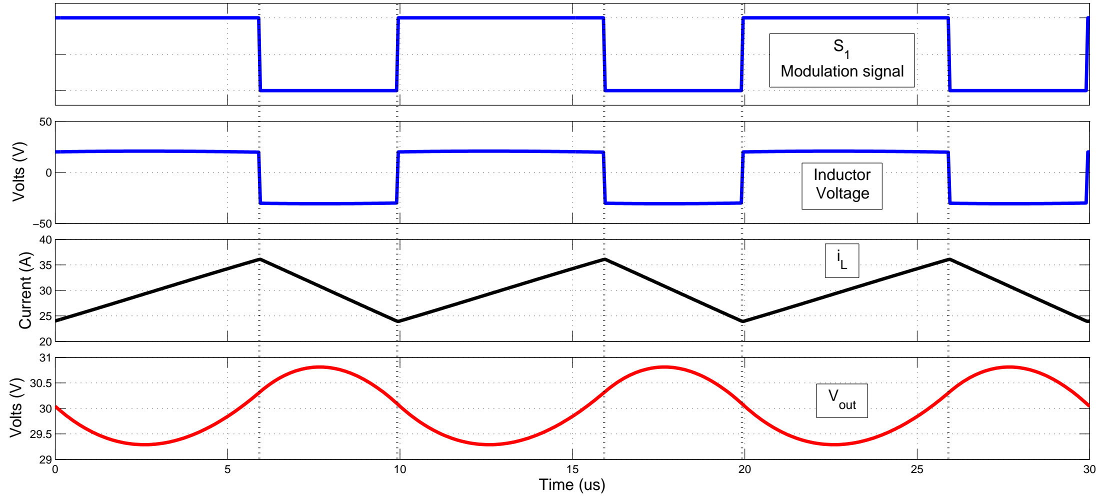
The state variables are usually chosen to represent energy storage elements (e.g. inductor currents, capacitor voltages etc.), as their values are continuous functions that cannot change instantaneously. This is because state averaging cannot easily model discontinuous states. In general this means that each energy storage element contributes one state variable to the system. Therefore the Buck converter is a second-order system, and its state variables are the output capacitor voltage (V_{out}) and the inductor current (i_L).

Note: There are cases where a state can be omitted from the overall model, but only if it does not contribute significantly to the dynamic response [101, 103]. This usually happens if the dynamics of one state are significantly faster than the others. The slower state dynamics dominate, so the fast state can be omitted from the model.

[7] The load current input is included to model the effect of load variation.



(a) Buck Topology



(b) Buck Operating Waveforms

Figure 2.20: Buck Converter & Idealised Operating Waveforms

The evolution of these state variables during each mode of operation is then described by a series of equations, represented in state space form as:

$$\left.\begin{aligned} \dot{x}(t) &= A_n x(t) + B_n u(t) \\ y(t) &= C_n x(t) \end{aligned}\right\} \quad n = \text{ON or OFF} \tag{2.1}$$

where $x(t) = \begin{bmatrix} i_L(t) \\ V_{out}(t) \end{bmatrix}$, $u(t) = \begin{bmatrix} V_{in}(t) \\ I_{load}(t) \end{bmatrix}$, $y(t) = V_{out}(t)$

These piecewise linear equations describe the static behaviour of both states of the Buck converter. This type of model is often used to perform a loss analysis [104, 105]. In order to derive dynamics from these models however, each state must be averaged with respect to their duration over the entire switching period (i.e. the switch duty cycle, D), viz.:

$$\begin{aligned} \dot{x}(t) &= A'x(t) + B'u(t) \\ y(t) &= C'x(t) \end{aligned} \tag{2.2}$$

where
$$\begin{aligned} A' &= D(t)A_{ON} + \{1 - D(t)\}A_{OFF}, \\ B' &= D(t)B_{ON} + \{1 - D(t)\}B_{OFF}, \\ C' &= D(t)C_{ON} + \{1 - D(t)\}C_{OFF} \end{aligned}$$

The duty cycle D is an input to this combined, averaged system, so a new input matrix $u'(t)$ is defined as:

$$u'(t) = \begin{bmatrix} u(t) \\ D(t) \end{bmatrix} \tag{2.3}$$

Standard linearisation techniques are then applied to the non-linear state space averaged converter model (see eq. 2.2) by selecting an operating point and deriving a linearised model of the system about this point [99], i.e.:

$$\begin{aligned} x(t) &= x_0 + \hat{x} \\ u'(t) &= u'_0 + \hat{u}' \\ y(t) &= y_0 + \hat{y} \end{aligned} \tag{2.4}$$

The partial derivatives of each variable are taken and summed together to give the final linearised small-signal state averaged model:

$$\begin{aligned} \dot{\hat{x}} &= A'\hat{x}(t) + B'\hat{u}'(t) \\ \hat{y} &= C'\hat{x}(t) \end{aligned} \tag{2.5}$$

Numerous publications have applied these state averaging concepts to model the dynamics of isolated bi-directional DC-DC converters. The key features of the resulting models are summarised in Table 2.2.

The primary difference between these models is that various publications present different sets of state variables to model, without any clear justification for this decision. It is thus not uncommon to see several alternative models derived for the same converter structure that differ significantly in terms of model order as well as choice of system state. For example, the dual half bridge converter is modelled as a 4th order system by Liping et al. [23, 106] as well as Hui et al. [87]. However Liping's work includes the dynamics of the converter current while Hui's does not, and no justification for this difference is presented. A similar problem can be seen

Author	Topology	Model Order	State Variables
Gang et al. [44]	Actively Clamped Flyback	5th	• Input & Output Current • Clamping Capacitor Voltage • Transformer Magnetising Current
Swingler et al. [49]	Dual push-pull	2nd	• Input Current • Output Voltage
Gang et al. [46]	Hybrid converter (Half-bridge linked to a series-resonant CFPP)	6th	• Input & Output Currents • Input & Output Voltages • Inductor Current
Liping et al. [23, 106]	Dual half-bridge	4th	• Input & Output Voltages • Inductor Current • Output Current
Li et al. [87]	Dual half-bridge	4th	• Input & Output Voltages
Bai et al. [107]	Dual active bridge	3rd	• Input Voltage • Output Voltage • Inductor Current
Krismer et al. [69]	Dual active bridge	5th	• Input & Output Voltage • Input & Output Current • Inductor Current
Demetriades et al. [70]	Dual active bridge	2nd	• Inductor Current • Output Voltage
Alonso et al. [67]	Dual active bridge	3rd	• Inductor Current • Input & Output Currents
De Doncker et al. [53]	Dual active bridge	1st	• Output Current

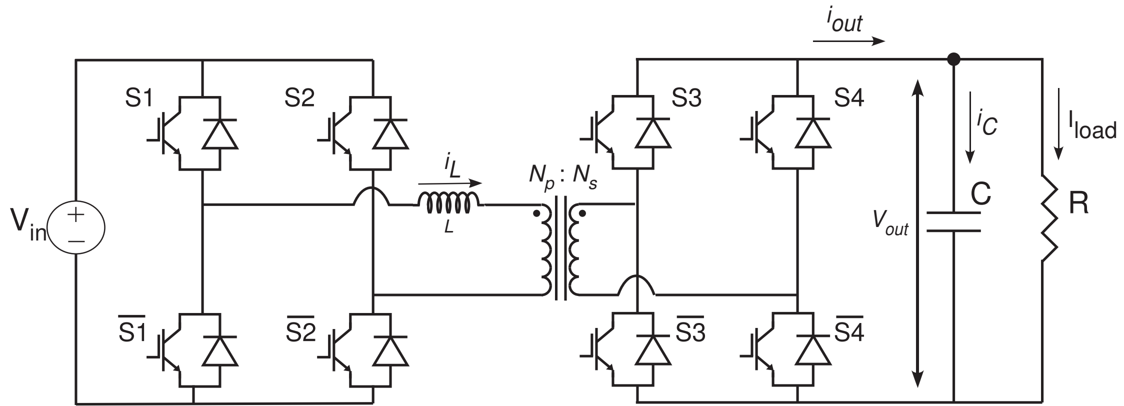
Table 2.2: State Averaged Models

for DAB converters, since de Doncker et al. [53] models them as a 1st order system, while much more complex models have been proposed by Demetriades et al. [70] (2nd order), Alonso et al. [67] (3rd order) and Krismer et al. [69] (5th order).

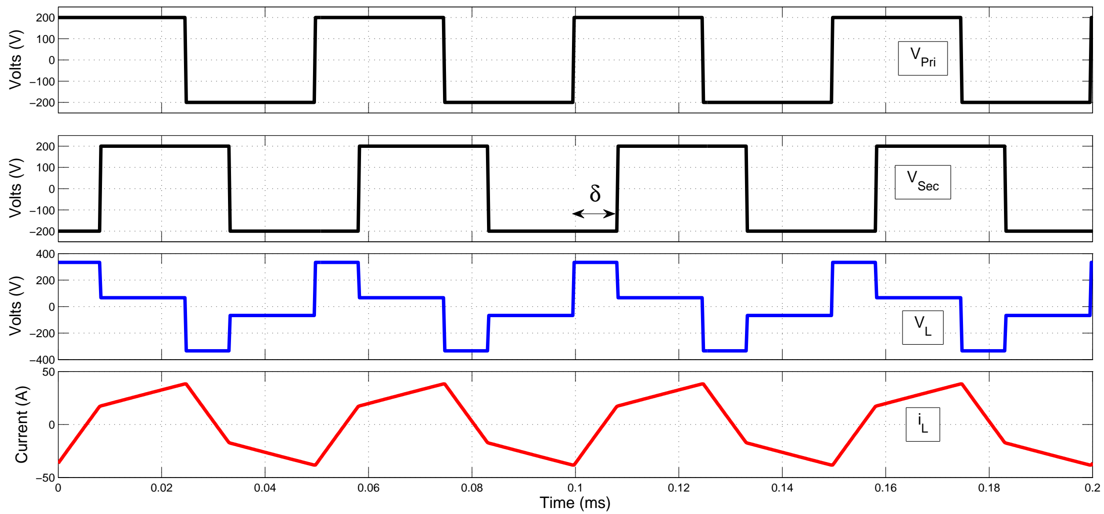
As a result of these variations in plant models proposed in the literature and the lack of comparison between them, choice of system state in a particular context is often unclear, and hence model development in this research field can be difficult and uncertain.

2.3.2 Fundamental Averaged Models

The second family of dynamic models applied to isolated, bi-directional DC-DC converters use dynamic equations based on the converter fundamental power expressions. The underlying principles of this method are outlined here, using a block modulated single-phase DAB converter as an example.



(a) DAB Converter Structure

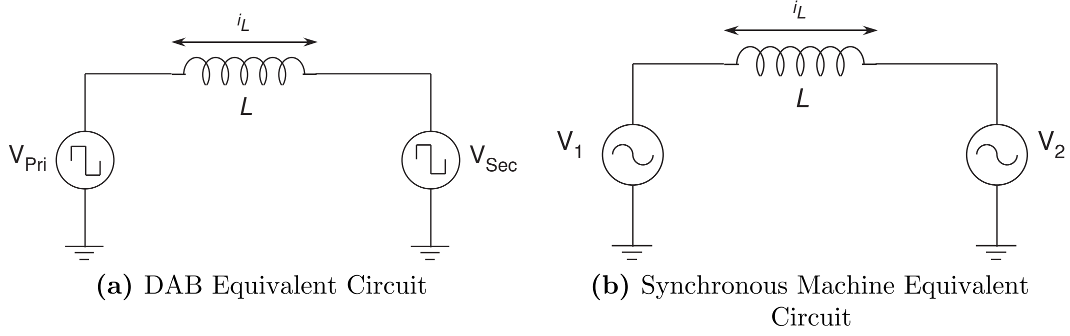


(b) DAB Operating Waveforms

Figure 2.21: The DAB Converter & Operating Waveforms

The DAB converter topology and its two-level modulation scheme have been presented in Sections 2.1 & 2.2 respectively. It is redrawn for clarity here in Fig. 2.21. The first step to model this converter is to represent it with the equivalent circuit shown in Fig. 2.22a, where each bridge is replaced by square-wave voltage sources V_{Pri} and V_{Sec}, and the AC link and its associated impedance are represented by an inductance L.

This structure is similar to that of two synchronous machines connected by an inductive transmission line, shown in Fig. 2.22b. V_1 and V_2 are the RMS machine



(a) DAB Equivalent Circuit

(b) Synchronous Machine Equivalent Circuit

Figure 2.22: Equivalent Circuits for the DAB Bi-directional DC-DC Converter

output voltages, and L is the inductance of the transmission line between them. Although the DAB converter uses square-wave voltages rather than sinusoidal waveforms, fundamental power flow analysis proposes that the fundamental harmonic of these square-waves dominates, so the other (higher order) harmonics that make up the square wave can be ignored [74]. This allows the average power flow of this system to be expressed using AC phasor theory as:

$$P_{AC} = \frac{V_1 V_2 \sin \delta}{\omega L} \tag{2.6}$$

where δ is the phase shift between the two sinusoidal voltage signals.

To derive dynamic equations for this static power transfer model, the DAB converter is assumed to be lossless. Therefore the DC output power of the converter (P_{out}) is equal to the average AC power transfer defined in eq. 2.6. If the system is also assumed to be operating in steady state, the time domain expressions for the DC output voltages can be expressed in terms of the static AC RMS average quantities:

$$V_1 = \frac{V_{1_{pk}}}{\sqrt{2}} sin(\omega t) \quad \therefore \quad V_1\left(t\right) = \frac{V_{1_{pk}}\left(t\right)}{\sqrt{2}} sin(\omega t) \tag{2.7a}$$

$$V_2 = \frac{V_{2_{pk}}}{\sqrt{2}} sin(\omega t) \quad \therefore \quad V_2\left(t\right) = \frac{V_{2_{pk}}\left(t\right)}{\sqrt{2}} sin(\omega t) \tag{2.7b}$$

where $V_{1_{pk}}$ & $V_{2_{pk}}$ are the peak machine voltages and ω is the fundamental frequency.

The time domain representation of the average DAB output power can now be defined as:

$$P_{DC}\left(t\right) = \frac{V_1\left(t\right) V_2\left(t\right) \sin \delta\left(t\right)}{\omega L} \tag{2.8}$$

Since the converter is assumed lossless, the output voltage V_{out} can be assumed equal to $V_{2_{pk}}$, so an expression for the DAB output current i_{out} is given as:

$$
\begin{aligned}
P_{DC}(t) &= V_{out}(t)\, i_{out}(t) \\
&= \frac{V_1(t)\, V_2(t) \sin\delta(t)}{\omega L} \\
\therefore i_{out}(t) &= \frac{\sqrt{2}\, V_1(t) \sin\delta(t)}{\omega L}
\end{aligned}
\tag{2.9}
$$

The DAB output voltage dynamic equation is given by basic circuit theory as [60, 61, 102]:

$$
\frac{dV_{out}(t)}{dt} = \frac{i_C(t)}{C} \tag{2.10}
$$

From Fig. 2.21, Kirchhoff's Current Law gives $i_{out}(t) = i_C(t) + i_{load}(t)$, so the final output voltage expression is [60, 61, 102]:

$$
\frac{dV_{out}(t)}{dt} = \frac{1}{C}\left(i_{out}(t) - i_{load}(t)\right) \tag{2.11}
$$

This expression is non-linear, so researchers such as Cardozo et al. [108] use this full non-linear form to develop a non-linear controller. However, most publications first simplify this model by linearising the current expression about an operating point, and only then forming the output voltage equation. This gives a linear model that is then used for closed loop control purposes [38, 74, 94, 109]. The design of closed loop controllers based on these models will be addressed in Section 2.4.

2.3.3 Non-ideal effects: Deadtime

It is well known in power electronics that the behaviour of an idealised system can significantly differ from a practical implementation because of the non-idealities that exist in reality. Examples of such non-ideal effects include parasitic impedances, device voltage drops and source impedances [10, 11, 102].

In the case of isolated bi-directional DC-DC converters, the literature has mostly identified *deadtime* as the primary second order effect [68, 110, 111]. The principles of deadtime were presented in Section 2.1, which define it as the blanking time included between the gate signals of a phase leg to prevent catastrophic shoot-through.

During the deadtime interval, the midpoint output voltage is defined by the flow of current through the converter, rather than a switch state. Since the switches

have been turned off, this current is forced to conduct through the anti-parallel diodes of the active devices instead. This forces the phase leg output voltage to either the positive or the negative bus, depending on current direction. This causes a discrepancy between the commanded output voltage and the actual voltage seen at the phase leg midpoint.

Publications such as Akagi et al. [110], Bai et al. [111] and Xie et al. [68] have shown that this discrepancy in phase leg output voltage changes the converter operating point and even affects its dynamic response.

Several alternative methods have been proposed in the literature to include the effect of deadtime when modelling converter dynamics. The simplest method proposes measuring the error in operating point caused by deadtime and updating the system model accordingly [110]. However, this approach is converter-specific, and load dependent, so that while simple, it is substantially limited and unattractive. State averaged models can analytically include the effect of deadtime by modelling it as an additional mode of operation and then adjusting the behaviour accordingly. However, this comes at the cost of significantly increased model complexity [69, 101, 103]. Fundamental averaged models also can account for the deadtime effect by separately modelling the operating point distortion caused, and adjusting the operating point of the model accordingly, resulting in a more accurate dynamic model [111].

2.3.4 Summary – Modelling

This section has presented a review of the literature in the area of dynamic modelling for isolated bi-directional DC-DC converters. The design of these models can be challenging since they must not only model the switched behavior of these converters, but must also accommodate the effect of deadtime, which is known to affect converter dynamics. Two alternative modelling techniques emerge from the literature, i.e. state averaged modelling and fundamental averaged modelling. Each has their strengths and drawbacks.

State averaged modelling is very powerful, generating very accurate dynamic models that can also include the effect of deadtime on the converter. However, this technique often results in high order models, and can be very complex, especially when the effect of deadtime is included. This is undesirable as such complex models often require equally complex regulators, which are hard to implement.

Fundamental averaged modelling is a much more elegant technique than state averaging, and gives a simple dynamic model that easily includes the effect of deadtime. However, it can lack accuracy for two reasons – firstly, it assumes that the

fundamental component is sufficient to model converter dynamics. Unfortunately, in the case of a block modulated converter, significant energy is contained in the higher order harmonics, limiting the validity of this assumption. Secondly, although the effect of deadtime can be included in the dynamic model, the analytic deadtime models currently presented in the literature are very complex in nature [68, 111]. Including the deadtime effect therefore results in a final dynamic model that is complex, yet still lacks accuracy due to the assumption of fundamental component power flow equivalence.

2.4 Closed loop Control

Power electronic converters need closed loop regulation to ensure that the correct output is maintained irrespective of operating conditions, as well as to guarantee stability and fast recovery in the face of transient events. There are two basic types of transient that affect this class of system, i.e. changes in load condition and variations in reference command. A good controller should achieve a similar level of performance for both events.

Closed loop controllers work on the principle of *feedback*, illustrated in Fig. 2.23. These feedback controller structures are made up of a plant G that needs to be controlled, and a controller H. The plant output y is compared to its target reference value r, and the difference between them (e) is fed into the controller. The controller then adjusts the control signal u, such that the plant output achieves its target value [99].

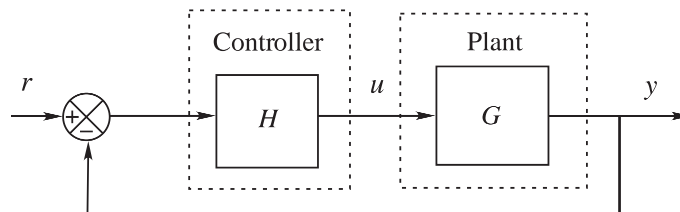


Figure 2.23: A Basic Feedback Controller

> **Note:** Open loop regulation is proposed by Akagi et al. in [110], where a pre-calculated lookup table generates the control signal for a DAB converter based on the desired output power. Although simple, open loop control cannot guarantee transient performance, so it is not considered any further in this review.

While a large number of closed loop regulation strategies have been identified in this literature survey, they essentially fall into two major categories, i.e. linear and

non-linear controllers. This section reviews the basic operational concepts of each closed loop control technique presented, and identifies the benefits and drawbacks of each.

2.4.1 Non-linear Control

Numerous non-linear control techniques exist in the literature, including passivity based control, sliding mode control and model predictive control [112]. However, in the literature surrounding isolated bi-directional DC-DC converters, the main non-linear control techniques that have been applied are Feedback Linearisation and Flatness based control.

Feedback Linearisation

Feedback linearisation has been successfully applied to isolated bi-directional DC-DC converters in [108,113]. This technique uses an accurate non-linear converter model to mathematically identify the system non-linearities before employing feedback to cancel out their effect [112]. This leaves an equivalent linear system that can be regulated using classical linear control techniques [99]. A detailed description of this design process is presented here to better understand the underpinning principles of this control strategy.

In Cardozo et al. [108], a feedback linearised non-linear regulator was used to regulate the DC output voltage of a DAB bi-directional DC-DC converter (see Fig. 2.21). This controller was realised by deriving an expression for the average converter output current using a state averaging method [53, 108]:

$$i_{out}(t) = \frac{V_{in}(t)}{2Lf_{sw}}\alpha(t)(1-\alpha(t)) \tag{2.12}$$

where f_{sw} is the switching frequency, and the phase shift between the bridges is represented by $\alpha = \frac{\delta(t)}{2\pi}$.

The converter dynamic model was then derived from this static equation using the same method outlined in Section 2.3 to derive fundamental averaged dynamic models. This results in Eq. 2.13, which is a non-linear differential equation that describes the converter output voltage dynamics.

$$\begin{aligned}\frac{dV_{out}(t)}{dt} = \frac{1}{C}i_C &= \frac{1}{C}\left(-i_{load}(t) + i_{out}(t)\right) \\ &= -\frac{1}{RC}V_{out}(t) + \frac{1}{C}\left(\frac{V_{in}(t)}{2Lf_{sw}}\alpha(t)\left(1-\alpha(t)\right)\right)\end{aligned} \tag{2.13}$$

To apply feedback linearisation, an auxiliary system input is defined as:

$$V_{aux}(t) = V_{in}(t)\,\alpha(t)\left(1-\alpha(t)\right) \tag{2.14}$$

Reforming the output voltage expression in terms of the auxiliary input V_{aux} eliminates the non-linearity of eq. 2.13, resulting in the following simplified expression [108]:

$$\frac{dV_{out}(t)}{dt} = -\frac{1}{RC}V_{out}(t) + \frac{1}{2CLf_{sw}}V_{aux}(t) \tag{2.15}$$

Since this expression is linear, classical linear control theory can now be applied to design a controller for this system. A simple Proportional + Integral (PI) controller was chosen in [108] because it achieves zero steady-state error. Controller gains were then calculated to achieve the desired bandwidth and level of damping.

This non-linear control design process is relatively simple, but its primary weakness is that the achieved performance depends heavily upon the ability of the controller to cancel out the system non-linearities. Any errors in the system model (due to non-idealities such as device voltage drops, losses, deadtime etc.) will degrade this cancellation, compromising performance.

Flatness Based Control

Fliess et al. [114] defines a system as 'flat' if the number of inputs and outputs are equal, and if all states and inputs can be determined from these outputs without integration. Nieuwstadt et al. elaborates on this concept in [115], where it is assumed that all states and control variables of such flat systems are known in both steady-state and transient. This accurate knowledge of system behaviour allows a control signal to be generated that will give precisely the desired output response.

In the scope of isolated bi-directional DC-DC converters, flatness based control was proposed by Phattanasak et al. in [116, 117]. The application context of this paper was a Triple Active Bridge (TAB) converter, where a Fuel Cell as well as a Supercapacitor were used to supply a variable load (see Fig. 2.24). The paper first proves that this system is flat, before designing a controller with two design targets,

i.e. minimal transient voltage overshoot/undershoot, and slew rate limited Fuel Cell current. This ensured good output voltage regulation while also minimising stress on the Fuel Cell. The flatness technique was then used to determine a set of phase shift trajectories that achieved these goals.

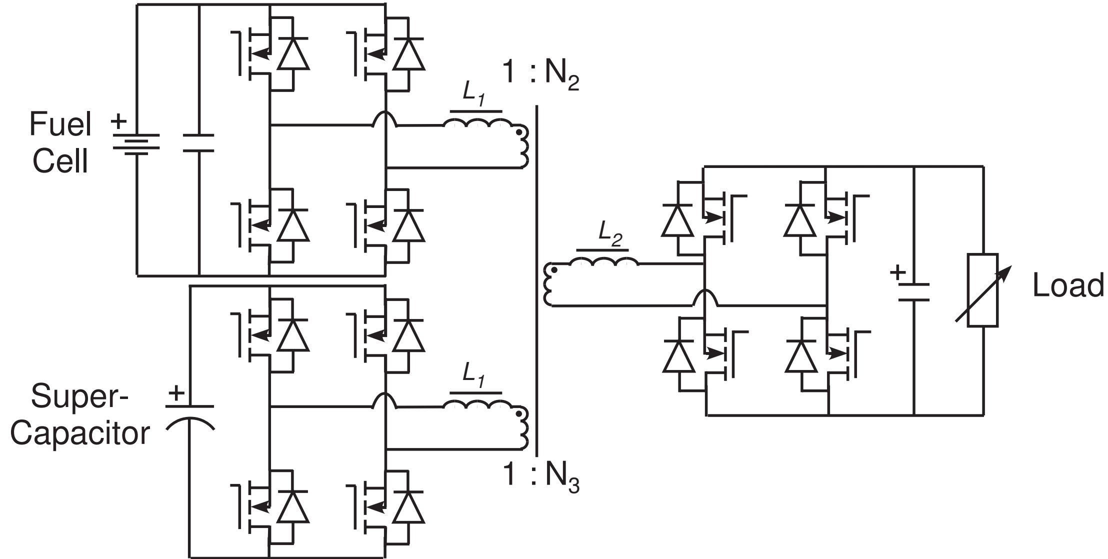


Figure 2.24: Triple Active Bridge Converter [116]

Although Flatness based control can achieve a very high level of performance, its complexity makes implementation of such a controller difficult and expensive in terms of both hardware and software.

2.4.2 Linear Control

Linear controllers are the most popular type of closed loop strategy proposed in the literature to regulate isolated bi-directional DC-DC converters. A large number of linear controllers exist in the literature (Proportional + Integral controllers, pole placement controllers, etc.). This section outlines their key design features and evaluates their performance.

This review is simplified by the fact that all linear controller designs essentially follow the same sequential process, i.e.:

1. Regulator target variable selection
2. Loop design
3. Regulator design

The step-by-step nature of this process is utilised in this review by presenting each alternative controller solution in the context of this process.

Regulator Target Variable Selection

The first stage of linear regulator design is to select the converter state variable(s) to be controlled. The typical states that have been selected for regulating isolated bi-directional DC-DC converters are reviewed in this section. For clarity, Fig. 2.25 identifies these control states on a DAB isolated bi-directional DC-DC converter.

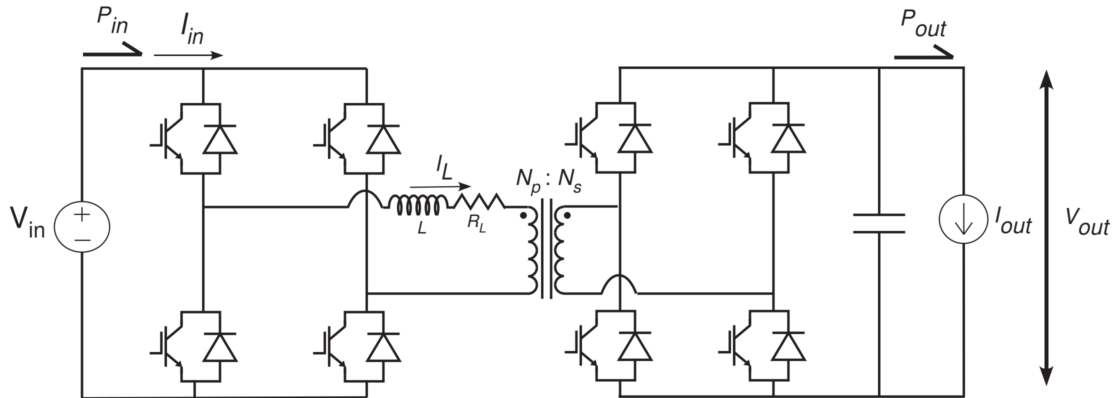


Figure 2.25: DAB bi-directional DC-DC Converter

- **Input Power** (P_{in})
  Tao et al. [2,81] proposes a Triple Active Bridge converter powered by a Fuel Cell. Input power regulation is then used to minimise the dynamic stress on the Fuel Cell, as these devices are unable to change their power output quickly.

- **Input Current** (I_{in})
  Haihua et al. [113] presented the use of several parallel connected DAB converters sourced from the same ultracapacitor. Input current control is used to ensure sharing between the converters.

- **Output Current** (I_{out})
  Output current control is proposed in Kunrong et al. [118] to allow a DAB converter to safely and effectively charge a battery load.

- **AC Inductor Current** (I_L)
  Demetriades et al. [70] and Lei et al. [119] propose controlling the intermediate AC inductor current in a DAB converter to provide inherent current limiting as part of a dual loop controller.

- **Output Voltage** (V_{out})
  Output voltage control is very effective in managing the most popular load scenarios for isolated bi-directional DC-DC converters, which are resistive loads and AC inverter loads [65,69,74,120]. It is therefore the most popular control variable used in the literature.

The choice of regulator target state is primarily dependent upon application requirements. These include primary design objectives, such as using output current control for a battery charger application to ensure safe operation and extend battery life [118], and secondary design objectives, such as input current regulation to guarantee current sharing in parallel connected converters [113].

Loop Design

The second stage of linear controller design is to select an appropriate feedback control loop structure. Three main approaches dominate the literature:

- **Single-loop structures**
  The single loop feedback controller is the simplest control loop, containing a single controller ($H(s)$) regulating a single plant output, as illustrated in Fig. 2.26. This system is commonly employed in Single Input Single Output (SISO) systems, as there is only one output variable that requires regulation [99, 100].

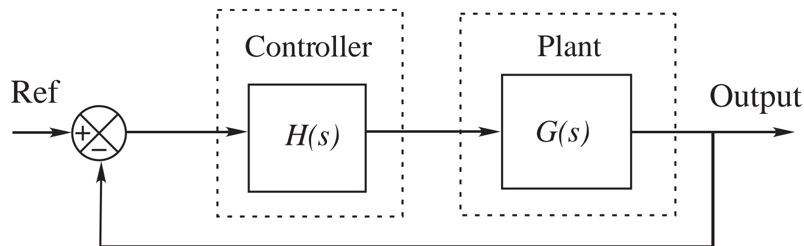


Figure 2.26: Single-loop Feedback Controller

  The simplicity of this loop structure makes it very attractive, as it can achieve a fast transient response with low implementation costs (e.g. due to minimal sensor requirements, reduced processing, etc.) [99]. This loop structure has been used in a variety of publications, such as Kheraluwala et al. [65], Akagi et al. [110], Watson et al. [121], where single loop feedback controllers are used to regulate the converter output voltage.

- **Nested loop structures**
  Nested loop structures are made up of several concentric control loops, and are usually employed when several control targets need to be simultaneously met (e.g. voltage regulation as well as current limiting). Fig. 2.27 shows the most common nested loop structure employed in the literature – the dual loop controller, made up of an outer controller that generates a reference for the inner controller.

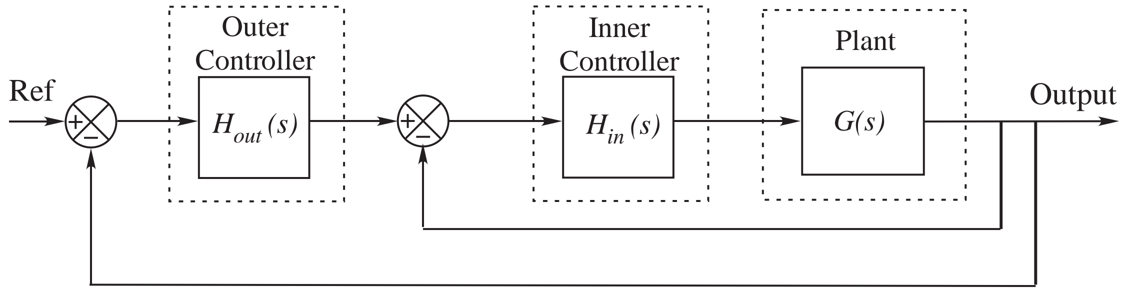


Figure 2.27: Dual-loop Feedback Controller

In the field of isolated bi-directional DC-DC converter regulation, the inner loop usually controls current, while the outer loop typically controls voltage [70]. This structure can therefore achieve output voltage regulation while also providing inherent current limiting. However, a limitation of this loop structure is that interaction between the two loops must be minimised, as this can cause instability. This is usually achieved by designing the outer loop to react several times slower than the inner loop, but this slows down the overall transient response [99, 100, 122].

- **Parallel loop structures**

  Parallel loop controllers consist of several closed loop controllers operating together to regulate a single system, as shown in Fig. 2.28. These structures are often used in Multi Input Multi Output (MIMO) systems [99], which makes them popular for multiport converter applications, such as Tao et al. [2, 74, 81], who present a Triple Active Bridge converter (see Fig. 2.24) that uses a parallel loop controller to regulate the load output voltage while also maintaining the supercapacitor state of charge.

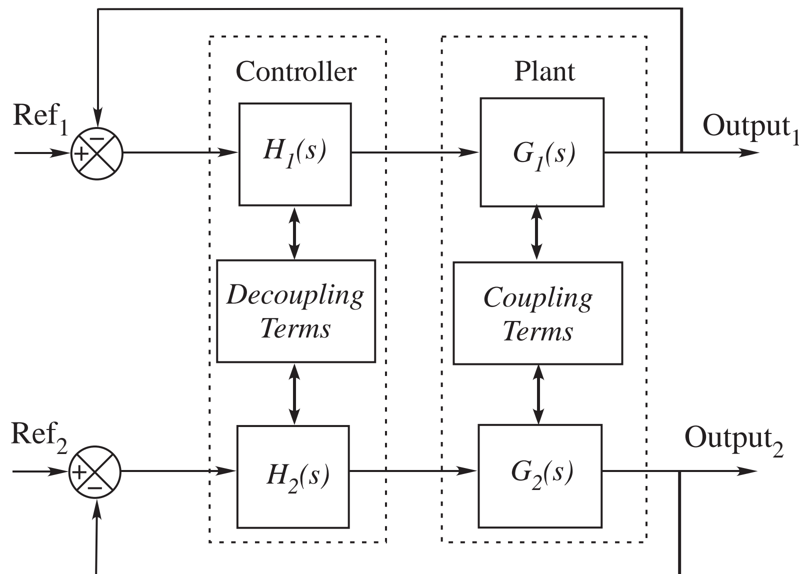


Figure 2.28: Parallel-loop Feedback Controller [2]

An additional advantage of parallel loop controllers is presented in Zhao et al. [109]. MIMO plants are often made up of a coupled network of several interacting systems, complicating the control process. Parallel loop controllers can solve this problem by incorporating decoupling networks within their structure, as shown in Fig. 2.28. These networks decompose the complex MIMO system into a series of independent SISO systems, considerably simplifying controller design.

Regulator Design

The most popular regulator forms identified in the literature to manage isolated bi-directional DC-DC converters are PI controllers and pole placement controllers.

Pole placement controllers are developed by first defining the desired level of closed loop performance in terms of criteria such as bandwidth, steady-state error and overshoot. Next, closed loop pole-zero locations that can achieve this criteria are then identified [99]. The open loop transfer function of the plant is then analysed (using Bode or Root Locus techniques), and a controller transfer function that moves the closed loop system pole locations to their desired locations is derived [49, 87, 109]. However, the main disadvantage of this technique is that the resulting controller transfer function is often complex and hard to implement.

Proportional + Integral (PI) controllers are by far the most common regulator structure, made up of a proportional gain term (K_p) that determines the speed of controller response, and an integral term (T_r) that eliminates steady-state error. The typical transfer function for this controller is:

$$H_{PI}(s) = K_p \left(1 + \frac{1}{sT_r}\right) \tag{2.16}$$

However, the bulk of the literature presents controller gain selection processes (K_p, T_r) that are heuristic in nature. This does not guarantee maximised performance. Only Krismer et al. [69] seeks to maximise controller performance by identifying that the primary performance limitation are the delays caused by the digital controller implementation. By accounting for these delays, this publication calculates controller gains that can achieve high performance. However, the resulting controller is only tested with a reference command transient, so its response to a change in load condition is unclear.

To further improve the performance of a PI controller, some publications have suggested the use of feed-forward terms in the controller structure. Fig. 2.29

illustrates this technique, where the feed-forward term augments the controller output with an estimate of the desired control signal. Hence the PI controller only needs to manage residual errors in this estimate (possibly caused by system non-idealities), and therefore has the potential to achieve a very rapid controller response [99, 100]. This technique has been employed by Bai et al. [107], who estimated the desired feedforward signal by assuming a constant load. However, this assumption is not adequate in general, especially since many converter applications face highly variable loads.

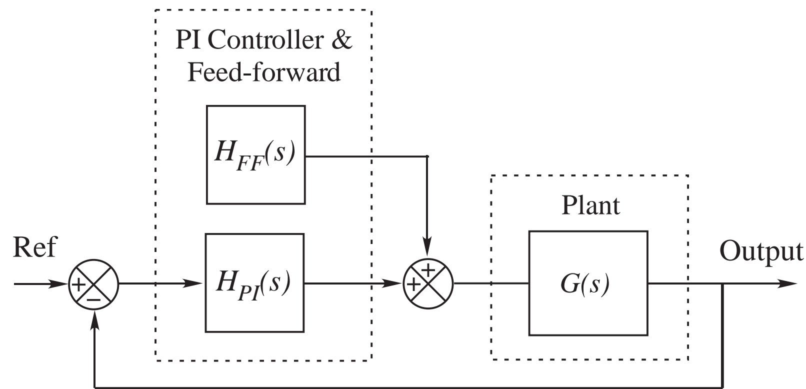


Figure 2.29: PI Controller with Feed-forward

2.4.3 Summary – Control

The literature has proposed many different types of regulators to control isolated bi-directional DC-DC converters, which include both linear and non-linear forms of control.

Non-linear controllers can give very high performance, but suffer from two main drawbacks. Their complexity makes them hard to implement, and they are very sensitive to variations in converter parameters, leading to reduced robustness.

Linear controllers are substantially simpler and easier to implement, but the linear control techniques presented in the literature suffer from two drawbacks. Firstly, their linear nature means that they are designed for a particular operating point, and therefore do not guarantee consistent performance across the entire operating range. Secondly, although some strategies for maximising controller gains have been presented, it is not clear in the literature whether these controllers are sufficient to give a good transient response for both reference and load transient events.

2.5 Conclusion

This chapter has provided an overview of the major literature in the area of isolated bi-directional DC-DC converters. It has outlined the converter topologies and their modulation strategies that have been used to achieve bi-directional power flow in this context, and also reviewed the major dynamic modelling and closed loop control techniques that have been applied to these converters.

From this review, an appropriate converter topology for Smart Grid applications can be identified. Operating in the Smart Grid environment usually requires voltages above 200 V, and power ratings in the kilowatt range. Based on the literature presented in this section, these ratings suggest that the *full-bridge* is the most appropriate choice for both primary and secondary converters in such a system. A single-phase topology (see Fig. 2.11a) is more attractive than its three-phase counterpart because it offers a reduced switch count without particularly compromising efficiency or dynamic performance [53, 63]. This review also suggests that the most attractive modulation strategy for a higher power DAB converter is a hard-switched two-level block modulation approach. This is because this modulation method achieves high frequency power transfer, which leads to minimised magnetic components and a fast dynamic response.

This review has also identified limitations in the area of converter modelling and control. The dynamic models presented in the literature tend to trade off simplicity for accuracy, and do not adequately accommodate the effect of deadtime on converter dynamics. This presents an opportunity to develop a simpler yet accurate dynamic model that also includes deadtime and its effects.

Several types of closed loop controllers have been proposed in the literature, but they do not guarantee maximised performance across the entire converter operating range, and have not been proven to achieve similar performance for changes in both load as well as reference command. Hence there is scope to develop an improved closed loop regulator that achieves maximised performance for both load and reference transients across the entire operating range.

This thesis now presents new converter modelling and control concepts to fill these gaps in current knowledge, allowing converter performance to be maximised.

Chapter 3

Converter Modelling

The literature analysis presented in the previous chapter has suggested that the most appropriate isolated bi-directional DC-DC converter for a Smart Grid application is a single-phase Dual Active Bridge (DAB) converter that employs two-level block modulation.

Previous attempts to model the dynamics of this converter have achieved only limited success as they tend to trade off simplicity for accuracy, and often do not properly account for 2nd order non-idealities such as deadtime that are known to affect dynamics. This thesis now proposes a new dynamic modelling strategy known as *harmonic modelling* to predict the dynamics of the DAB converter. This novel modelling technique aims to create a simple yet accurate dynamic converter model that also easily accounts for the effect of deadtime.

This chapter is structured as follows. First, the basic operating principles of the DAB converter are presented in terms of time domain switching functions. Next, dynamic equations for this converter are derived in terms of these switching functions. To apply the harmonic analysis to these dynamic equations, a summation of harmonics that represents the converter modulating signals is derived using a Fourier Transform. The resulting Fourier Series summations are substituted into the converter dynamic equations, to create a highly accurate model of the converter dynamics. This non-linear form is then linearised to give a small-signal model of the DAB converter. Next, deadtime is identified to affect converter dynamics by changing the effective system operating point. A set of analytical expressions that predict this change in operating point are derived and the resulting prediction included in the harmonic model. Finally, the model is validated by comparing its predicted response to that of a detailed switched simulation of a DAB converter.

To better illustrate the ideas presented, this chapter also includes selected simulation results, whose salient circuit parameters are listed in Table 3.1. The ratings of this converter were chosen to be representative of the voltage and power levels required for a household P-HEV battery charger. These simulation results will provide visual aids help validate the theories and mathematical derivations described.

Circuit Parameter		Value
DC Input Voltage	(V_{in})	200 V
DC Output Voltage (Nominal)	(V_{out})	200 V
DC Capacitance	(C)	20 μF
AC Inductance	(L)	50 μH
AC Resistance	(R_L)	0.1 Ω
Transformer Turns Ratio	$(N_{Pri} : N_{Sec})$	10 : 15
Switching Frequency	(f_s)	20 kHz
Deadtime	(t_{DT})	1.5 μs
Nominal Output Power	(P_{out})	3 kW

Table 3.1: DAB Converter Parameters

3.1 DAB Converter Principles of Operation

The structure of the DAB converter is shown again in Fig. 3.1. It is made up of two single-phase H-bridges, connected back-to-back across an AC link. This AC link comprises an isolating/scaling transformer and an intermediate inductor L.

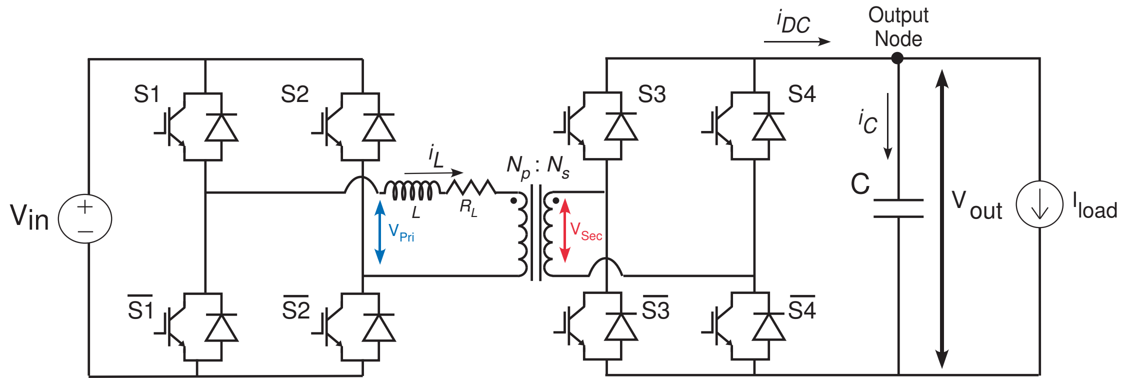


Figure 3.1: The DAB Converter

To analyse the operation of the Dual Active Bridge converter, it is useful to begin with the behavior of a single phase leg, shown in Fig. 3.2. Each switch of the leg is turned on and off in a complementary fashion, causing the the voltage at the phase leg output (V_{out}) to switch between the upper and lower voltage rails $(+V_{DC}$

and $-V_{DC}$) [10, 12]. The equivalent circuits of Fig. 3.3a & Fig. 3.3b illustrate this oppositional switching method, which is summarised in Truth Table 3.3c.

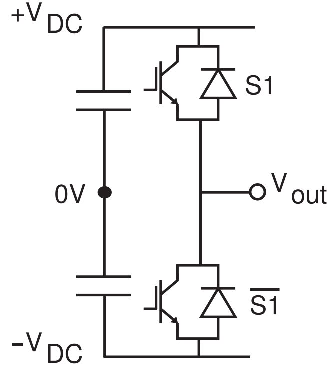
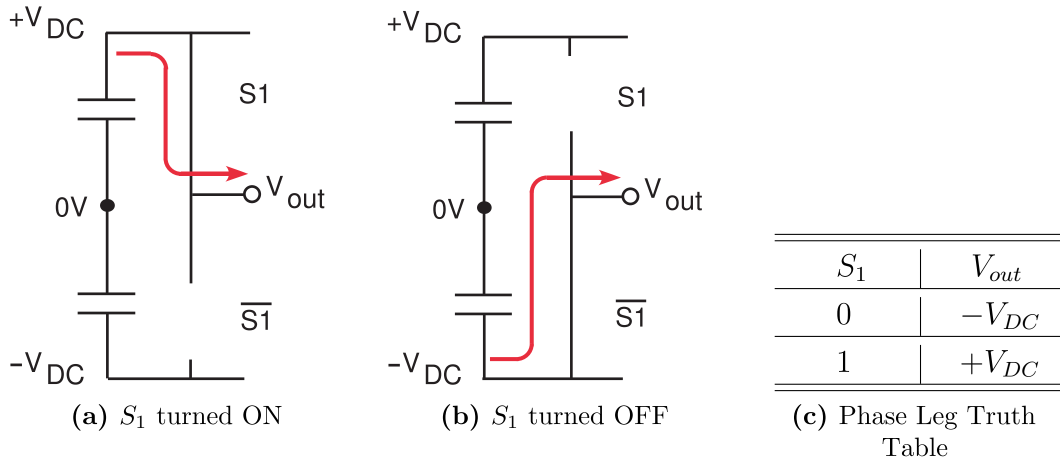


Figure 3.2: Phase Leg Structure



(a) S_1 turned ON (b) S_1 turned OFF

S_1	V_{out}
0	$-V_{DC}$
1	$+V_{DC}$

(c) Phase Leg Truth Table

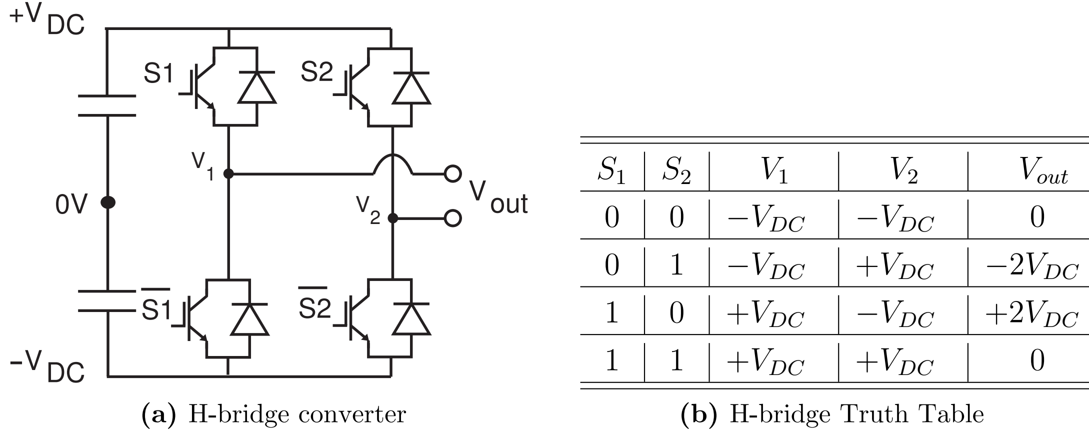
Figure 3.3: Phase Leg Equivalent Circuits & Truth Table

This analysis readily extends to describe the operation of a H-bridge converter, as it is made up of two phase legs, shown in Fig. 3.4a. The bridge output voltage, V_{out}, is given by the voltage difference between the midpoints of each phase leg (V_1 & V_2). The H-bridge has four possible states of operation, depending on the condition of its switches ($S_1, \bar{S}_1, S_2, \bar{S}_2$). The truth table of Table 3.4b describes these four states, and shows that they produce three possible output voltage levels – positive ($2V_{DC}$), negative ($-2V_{DC}$) and zero. This table can therefore be summarised by the following static equation:

$$V_{out} = 2V_{DC}\left\{S_1 - S_2\right\} \tag{3.1}$$

where S_1 & S_2 are logic variables that define the switched state of each phase leg.

Having described the switching states of the H-bridge, the concept of converter modulation can now be presented. Modulation can be defined as the process of



(a) H-bridge converter

S_1	S_2	V_1	V_2	V_{out}
0	0	$-V_{DC}$	$-V_{DC}$	0
0	1	$-V_{DC}$	$+V_{DC}$	$-2V_{DC}$
1	0	$+V_{DC}$	$-V_{DC}$	$+2V_{DC}$
1	1	$+V_{DC}$	$+V_{DC}$	0

(b) H-bridge Truth Table

Figure 3.4: H-bridge Converter & Truth table

changing switch states as a function of time to achieve the output condition (e.g. desired average output voltage, etc.) [12]. The modulation command for each converter phase leg is therefore a time varying, binary valued signal. The modulation commands employed by the proposed two-level block modulation strategy are a pair of a 50% duty cycle square wave signals that are offset 180° from each other, as shown in Fig. 3.5.

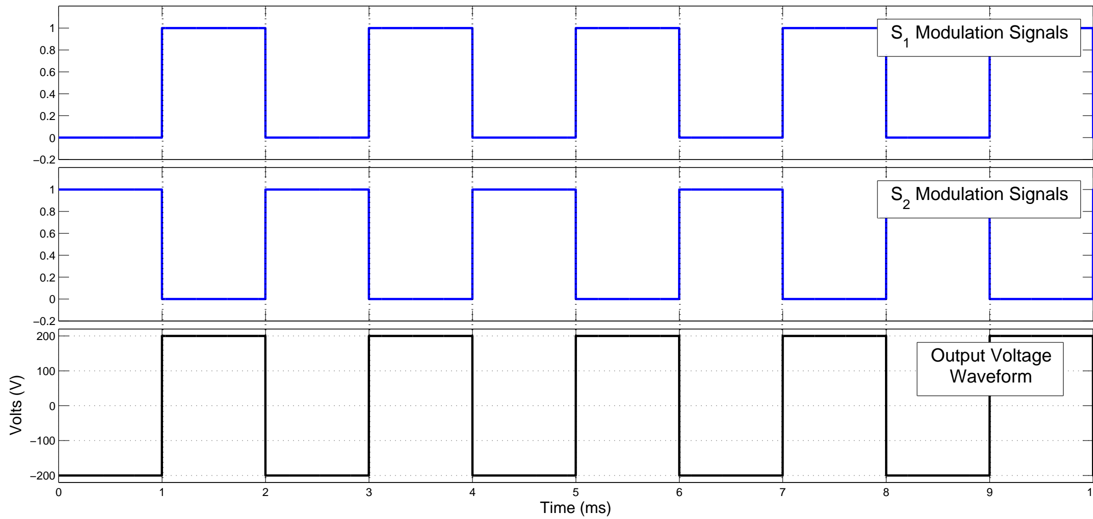


Figure 3.5: H-bridge Operating Waveforms

To model the time varying nature of the bridge output voltage shown in Fig. 3.5, the static switch state expression of eq. 3.1 is clearly insufficient. To resolve this issue, a time domain expression for the switching states of each phase leg is defined as:

$$S_k(t) \in \{0,1\}\ , \text{ where } k = 1,2,\ldots \tag{3.2}$$

This switching function allows the time domain representation of the H-bridge output voltage ($V_{out}(t)$) to be developed as:

$$V_{out}(t) = 2V_{DC}\{S_1(t) - S_2(t)\} \tag{3.3}$$

The modulation principles of a H-bridge can now be extended to describe the operation of the DAB converter in Fig. 3.1. Each bridge of the converter is modulated using a two-level block Phase Shifted Square Wave (PSSW) strategy, as illustrated in Fig. 3.6. The resulting bridge output voltage waveforms ($V_{Pri}(t)$ & $V_{Sec}(t)$) can be described in terms of the converter switching functions as:

$$V_{Pri}(t) = V_{in}(t)\{S_1(t) - S_2(t)\} \tag{3.4a}$$

$$V_{Sec}(t) = V_{out}(t)\{S_3(t) - S_4(t)\} \tag{3.4b}$$

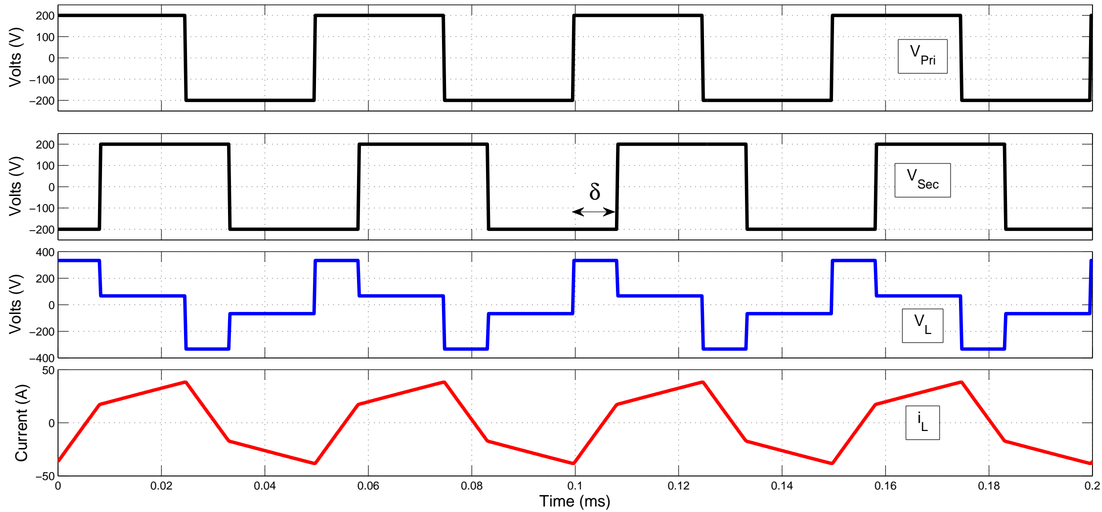


Figure 3.6: DAB Operating Waveforms

The two bridge output voltages $V_{Pri}(t)$ & $V_{Sec}(t)$ are offset from each other by a phase difference δ. This causes a non-zero net voltage V_L to appear across the AC link inductor, which in turn causes the current i_L to flow.

3.2 DAB Dynamic Equations

In this section, the DAB converter dynamic equations are derived in terms of their switching functions.

The dynamics of the output capacitor voltage ($V_{out}(t)$) are of primary interest, and are defined by basic circuit theory as:

$$\frac{dV_{out}(t)}{dt} = \frac{i_C(t)}{C} \tag{3.5}$$

where $i_C(t)$ is the capacitor current[1].

To determine $i_C(t)$, Kirchhoff's Current Law (KCL) is applied to the output node of the DAB converter, which gives:

$$i_C(t) = i_{DC}(t) - i_{load}(t) \tag{3.6}$$

where $i_{load}(t)$ is the load current and $i_{DC}(t)$ is the current injected by the secondary bridge.

Determining i_{load} is relatively simple, as it is a measurable quantity. However, defining the secondary bridge current is more complex, since the flow of i_{DC} is dependent upon the state of the secondary bridge switches S_3 and S_4 as well as the intermediate AC inductor current, i_L. Specifically, the inductor current i_L can only flow through the secondary bridge to the output (i_{DC}) when switches S_3 & S_4 are in their complementary position, as summarised by the truth table in Table 3.2.

S_3	S_4	i_{DC}
0	0	0
0	1	$-i_L$
1	0	i_L
1	1	0

Table 3.2: Switched DC current (i_{DC}) based on output bridge switching state

A time domain expression for i_{DC} in terms of the switching states expressed in Table 3.2 can now be established as:

$$i_{DC}(t) = i_L(t)\{S_3(t) - S_4(t)\} \tag{3.7}$$

The next stage of converter modelling is to derive a time domain expression for the inductor current, $i_L(t)$. To find this current, a Kirchhoff Voltage Loop (KVL) summation is taken around the DAB converter, illustrated in Fig. 3.7. This results in the following KVL loop expression:

[1] An ideal capacitor is assumed.

$$V_{Pri}(t) - \frac{N_p}{N_s} V_{Sec}(t) - R_L i_L(t) - L\frac{di_L}{dt}(t) = 0 \tag{3.8}$$

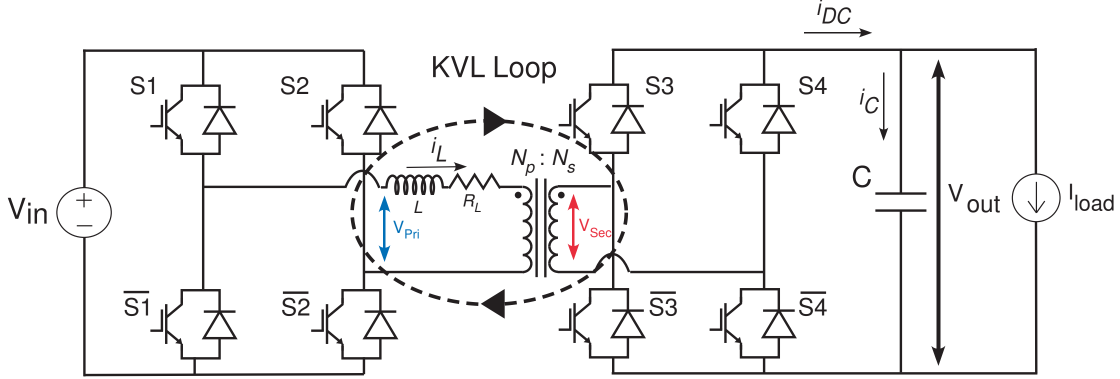


Figure 3.7: KVL of the DAB Converter

The time domain representations of the bridge output voltages presented in eq. 3.4 can now be substituted into eq. 3.8 and rearranged to give:

$$\begin{aligned} R_L i_L(t) + L\frac{di_L}{dt}(t) &= V_{in}\{S_1(t) - S_2(t)\} \\ &\quad - \frac{N_p}{N_s} V_{out}(t)\{S_3(t) - S_4(t)\} \end{aligned} \tag{3.9}$$

Solving these dynamic equations is non-intuitive problem since the continuous time converter state variables are driven by the binary valued switching functions. Such systems are defined as *mixed-mode dynamic systems* for they include both discrete and continuous time functions within their structure [100]. In the following section, a new method of describing the switching functions is presented that makes this dynamic model more tractable.

3.3 Deriving the switching functions

The dependence of DAB dynamics on the non-linear switching functions makes them difficult to solve analytically. To overcome this problem, this thesis presents a new approach for representing the non-linear binary valued switching functions. This *harmonic modelling* approach proposes decomposing the switching function into its Fourier Series components using a Fourier Transform [15, 123]. This gives a continuous time summation of harmonics that can be used to solve the converter dynamic equations.

Fourier theory states that any real-valued signal can be represented by the infinite series of sinusoids of [124]:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \{a_n cos(nx) + b_n sin(nx)\} \tag{3.10}$$

where the harmonic coefficients a_n & b_n are defined as:

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) cos(nx)\, \mathrm{d}x \,,\; n \geq 0 \tag{3.11a}$$

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) sin(nx)\, \mathrm{d}x \,,\; n \geq 1 \tag{3.11b}$$

Section 3.1 described how each phase leg of the DAB converter is modulated using a 50% square wave switching pattern. A Fourier transform is applied to this square wave, which gives the well known summation of [124]:

$$S_k(t) = \frac{1}{2} + \frac{2}{\pi} \sum_{n=0}^{\infty} \frac{\sin\left([2n+1]\{\omega_s t - \alpha_k\}\right)}{[2n+1]} \,,\; k = 1, 2, 3 \ldots \tag{3.12}$$

where ω_s is the switching frequency of the square wave (in rad/s) and α_k is the phase delay of the square wave relative to an arbitrary reference phasor.

For simplicity, this infinite Fourier Series is truncated to only the first N significant harmonics, which restates eq. 3.12 as:

$$S_k(t) = \frac{1}{2} + \frac{2}{\pi} \sum_{n=0}^{N} \frac{\sin\left([2n+1]\{\omega_s t - \alpha_k\}\right)}{[2n+1]} \,,\; N \geq 0 \,,\; k = 1, 2, 3 \ldots \tag{3.13}$$

Fig. 3.8 illustrates the resulting harmonic summation by comparing it to an ideal square wave. As suggested by Fourier theory, the inclusion of a more significant harmonics in the summation gives a better match to the ideal square wave.

The DAB converter has four sets of switches, so four switching functions need to be expressed based on eq. 3.13. This formulation makes the following assumptions:

- S_1 is chosen as the reference phasor, i.e. $\alpha_1 = 0$.
- Two-level PSSW modulation is employed, so the phase shift between the phase leg pairs of each bridge ($\{S_1 - S_2\}$, $\{S_3 - S_4\}$) is always π, and the phase shift between the primary and secondary bridges is defined as δ.

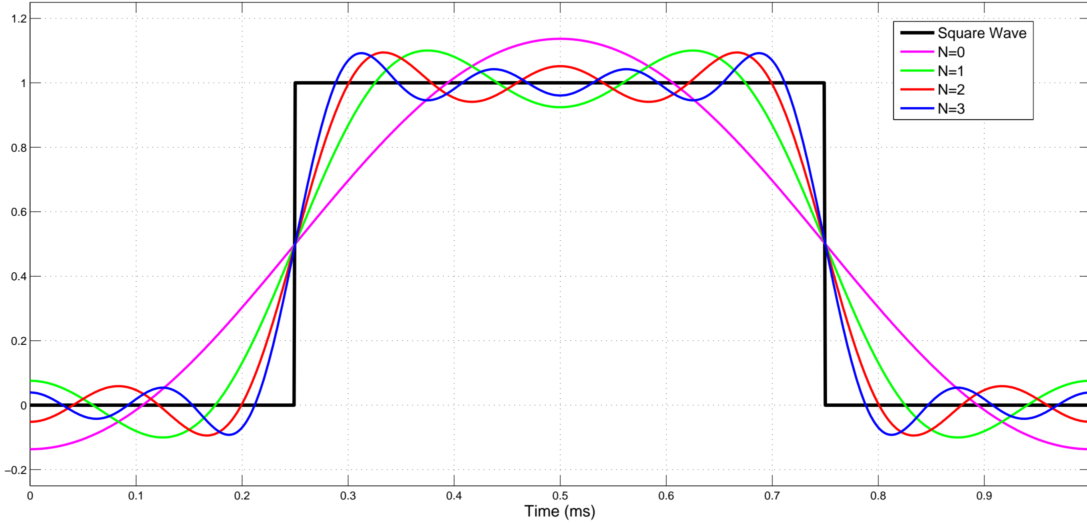


Figure 3.8: Square Wave Harmonics

This gives the following switching functions:

$$S_1(t) = \frac{1}{2} + \frac{2}{\pi}\sum_{n=0}^{N}\frac{\sin\left([2n+1]\{\omega_s t\}\right)}{[2n+1]} \tag{3.14a}$$

$$S_2(t) = \frac{1}{2} + \frac{2}{\pi}\sum_{n=0}^{N}\frac{\sin\left([2n+1]\{\omega_s t-\pi\}\right)}{[2n+1]} \tag{3.14b}$$

$$S_3(t) = \frac{1}{2} + \frac{2}{\pi}\sum_{n=0}^{N}\frac{\sin\left([2n+1]\{\omega_s t-\delta\}\right)}{[2n+1]} \tag{3.14c}$$

$$S_4(t) = \frac{1}{2} + \frac{2}{\pi}\sum_{n=0}^{N}\frac{\sin\left([2n+1]\{\omega_s t-\delta-\pi\}\right)}{[2n+1]} \tag{3.14d}$$

These continuous time harmonic representations of the binary valued switching functions can now be used to solve the DAB converter dynamics equations.

3.4 The Choice of N

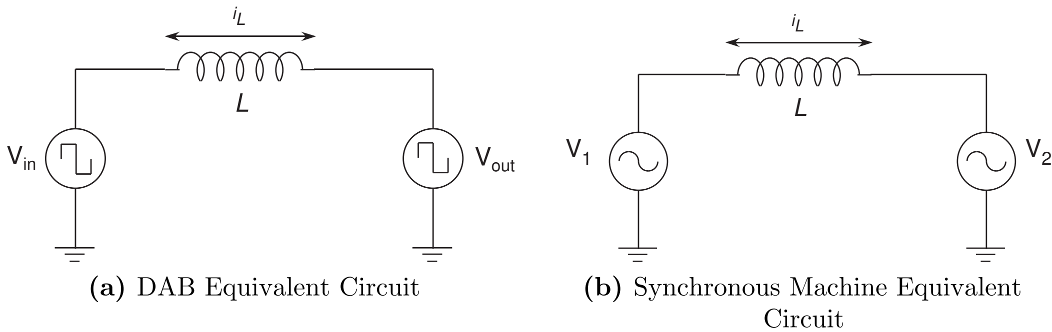
The Fourier representations of the switching functions presented are a truncated summation of harmonics. In these expressions, the value of N determines the number of significant harmonics that are included in the Fourier representation of the switching function. The choice of N is important because if too few harmonics are considered, the model lacks accuracy, but if too many are included, the model becomes too complex.

This thesis proposes choosing N based on a power transfer approach. The steady state power that flows between the two bridges of the DAB converter depends on the phase shift between the two bridge output voltages. Since the bridge output voltages can be represented by a summation of harmonics, it is therefore possible to represent the power that flows between the two bridges in a similar fashion. This harmonic power summation can then be compared to the analytical power flow expression derived by de Doncker et al. [53] (eq. 3.15, also presented in [69, 88]). The number of significant harmonics is then chosen such that a good match is obtained between these two predictions.

$$P = \frac{N_p}{N_s} \frac{V_{in} V_{out}}{\omega L} \frac{\delta(\pi - |\delta|)}{\pi} \tag{3.15}$$

where δ is the phase shift between the two bridges, and the impedance between them is represented by an inductance L.

The harmonic power flow model is derived by first representing the PSSW-modulated DAB converter of Fig. 3.1 as a pair of square-wave voltage sources connected across an impedance L, as shown in Fig. 3.9a. This equivalent circuit was then further simplified into its fundamental power flow component, illustrated in Fig. 3.9b. In this figure, V_1 & V_2 represent the RMS values of the two sinusoidal voltage sources.



(a) DAB Equivalent Circuit

(b) Synchronous Machine Equivalent Circuit

Figure 3.9: DAB Converter Equivalent Circuits

AC phasor theory gives the steady state expression for the power transfer in the fundamental equivalent circuit as:

$$P_{fund} = \frac{V_1 V_2 \sin \delta}{\omega L} \tag{3.16}$$

This equation shows that the real power in the DAB converter is primarily determined by the phase shift (δ) between the two voltage sources.

However, the DAB converter uses square-wave voltage signals rather than sinusoidal waveforms. To determine the power transferred by the higher order harmonics, the primary and secondary bridge voltages ($V_{Pri}(t)$ & $V_{Sec}(t)$) must first be expressed in harmonic terms. This is achieved by substituting the harmonic representation of the switching functions (eq. 3.14) into the bridge voltage equations (eq. 3.4), yielding:

$$V_{Pri}(t) = V_{in}\frac{4}{\pi}\sum_{n=0}^{N}\frac{1}{[2n+1]}\sin\left([2n+1]\,\omega_s t\right) \tag{3.17a}$$

$$V_{Sec}(t) = V_{out}\frac{4}{\pi}\sum_{n=0}^{N}\frac{1}{[2n+1]}\sin\left([2n+1]\,\omega_s t - \delta\right) \tag{3.17b}$$

The RMS magnitude of each harmonic is then extracted from these voltage expressions, resulting in:

$$V_{Pri_{RMS}} = \frac{V_{in}}{\sqrt{2}}\frac{4}{\pi}\frac{1}{[2n+1]} \tag{3.18a}$$

$$V_{out_{RMS}} = \frac{V_{out}}{\sqrt{2}}\frac{4}{\pi}\frac{1}{[2n+1]} \tag{3.18b}$$

The total power transferred between the two bridges can now be described in harmonic form by substituting each RMS voltage term of eq. 3.18b into the power flow expression of eq. 3.16 to determine the power transferred by each harmonic, and summing their contributions. This gives the final harmonic power summation formula of:

$$P_{Sec} = \frac{8}{\pi^2}V_{in}V_{out}\frac{N_p}{N_s}\sum_{n=0}^{N}\left\{\frac{1}{[2n+1]^3}\frac{\sin\left([2n+1]\,\delta\right)}{\omega_s L}\right\} \tag{3.19}$$

The power flow predicted by the harmonic summation is then compared to the solution of the analytic expression (eq. 3.15). N is determined by including additional harmonics to the harmonic power summation until the difference between the two solutions is negligible.

For the simulated circuit parameters listed in Table 3.1, the differences between the harmonic power summation and the analytic expression are listed in Table 3.3. When $N = 3$, the difference is less than 0.1%, which is deemed negligible. Hence $N = 3$ has been used for the analysis presented in this thesis.

Significant Harmonics (N)	**Difference**
0 (Fundamental)	3.131 %
1	-0.573 %
2	0.178 %
3	-0.070 %
4	0.031 %
5	-0.014 %
6	0.006 %

Table 3.3: Accuracy of the harmonic model compared to the switched model as the number of significant harmonics (N) are increased.

3.5 Harmonic Model Derivation

In this chapter, the DAB dynamic equations have been developed in terms of their switching functions and the switching functions themselves have been expressed as a summation of their Fourier Series components. This section solves these dynamic expressions, presenting the derivation of the harmonic model in its entirety. At key points during the derivation process, selected simulation results are included for visual aid and validation purposes (see Table 3.1 for simulation parameters).

The harmonic model is derived by first substituting the harmonic representation of switching functions into the converter dynamic expressions. This results in a set of equations that describe the contribution of each significant harmonic to the overall converter dynamic response. Summing the contributions from each harmonic forms the full non-linear dynamic converter model. For the purposes of closed-loop regulator design, the key dynamics are then extracted from this model, and are finally linearised to generate the final harmonic model.

The first step in developing the harmonic model is to determine the dynamics of the AC inductor current. These dynamics are described by the KVL expression of eq. 3.9. This equation is solved by substituting the switching functions of eq. 3.14 into the expression, giving:

$$\begin{aligned} R_L i_L(t) + L\frac{di_L}{dt}(t) &= V_{Pri}(t) - \frac{N_p}{N_s}V_{Sec}(t) \\ &= \left\{ \begin{aligned} & V_{in}\left\{\frac{4}{\pi}\sum_{n=0}^{N}\frac{\sin\left([2n+1]\left\{\omega_s t\right\}\right)}{[2n+1]}\right\} \\ & - V_{out}(t)\left\{\frac{4}{\pi}\sum_{n=0}^{N}\frac{\sin\left([2n+1]\left\{\omega_s t-\delta\right\}\right)}{[2n+1]}\right\} \end{aligned} \right\} \end{aligned} \tag{3.20}$$

However, using eq. 3.20 to extract an expression for the inductor current is complicated by the derivative term. Steady-state AC phasor theory presents a possible solution to this difficulty – it states that if an AC system is operating in *cyclic steady-state*, derivative terms ($\frac{d}{dt}$) can be represented instead by steady-state $j\omega$ terms. This assumption is valid for the DAB converter because the switching behaviour of the converter is essentially constant from one switching cycle to the next. Each harmonic component of the KVL expression in eq. 3.20 can therefore be solved independently and represented in the phasor domain as:

$$R_L i_L(t) + L\frac{di_L}{dt}(t) = \{R_L + j[2n+1]\omega_s L\} I_{L[2n+1]} \tag{3.21a}$$

$$V_{Pri[2n+1]} - \frac{N_p}{N_s} V_{Sec[2n+1]} = \frac{4}{\pi}\frac{1}{[2n+1]}\left\{V_{in}\angle 0 - \frac{N_p}{N_s}V_{out}\angle - [2n+1]\delta\right\} \tag{3.21b}$$

Eq. 3.21 can be rearranged to give an expression for each harmonic component of the inductor current:

$$\begin{aligned} \{R_L + j[2n+1]\omega_s L\} I_{L[2n+1]} &= \frac{4}{\pi}\frac{1}{[2n+1]}\left\{V_{in}\angle 0 - \frac{N_p}{N_s}V_{out}\angle - [2n+1]\delta\right\} \\ \therefore I_{L[2n+1]} &= \frac{\frac{4}{\pi}\frac{1}{[2n+1]}\left\{V_{in}\angle 0 - \frac{N_p}{N_s}V_{out}\angle - [2n+1]\delta\right\}}{R_L + j[2n+1]\omega_s L} \end{aligned} \tag{3.22}$$

Finally, to establish a time-domain model of the inductor current, this phasor domain expression needs to be converted back to the time domain. Summing the responses of each significant harmonic therefore gives the steady-state time domain expression for the AC inductor current of:

$$i_L(t) = \frac{4}{\pi}\sum_{n=0}^{N}\frac{1}{[2n+1]}\left\{\begin{array}{l} \frac{V_{in}}{|Z[n]|}\sin([2n+1]\omega_s t - \varphi_z[n]) - \\ \frac{V_{out}(t)}{|Z[n]|}\frac{N_p}{N_s}\sin([2n+1](\omega_s t - \delta) - \varphi_z[n]) \end{array}\right\} \tag{3.23}$$

where $|Z[n]| = \sqrt{{R_L}^2 + ([2n+1]\omega_s L)^2}$ and $\varphi_z[n] = tan^{-1}\left(\frac{[2n+1]\omega_s L}{R_L}\right)$, i.e. the magnitude and angle of the AC impedance between the bridges for each harmonic frequency of interest.

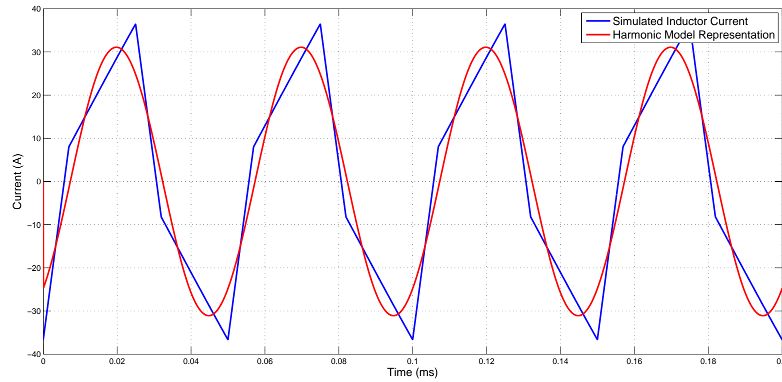
To confirm the modelling process thus far, the inductor current predicted by eq. 3.23 is matched to the results of the switched simulation in Fig. 3.10. This figure shows that the inclusion of more and more harmonics gives a more accurate representation of the AC inductor current waveform.

Since i_L has been determined in terms of the converter switching functions, the capacitor current (i_C) can also be derived. Eqns. 3.6 & 3.7 describe this current, and substituting the expressions for the inductor current (eq. 3.23), the load current (i_{load}) and the switching functions (eq. 3.14) into these equations gives:

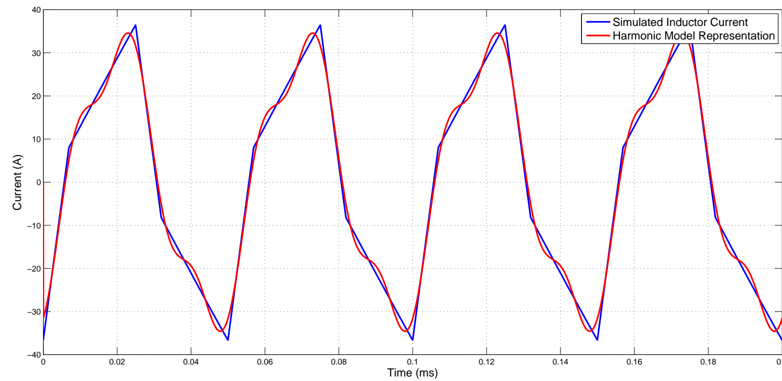
$$\begin{aligned} i_C(t) &= -i_{load}(t) + i_{DC}\left(t\right) \\ &= -i_{load}(t) + \left\{ \frac{N_p}{N_s} i_L\left(t\right)\left(S_3\left(t\right) - S_4\left(t\right)\right) \right\} \\ &= -i_{load}(t) + \\ &\left\{ \begin{array}{l} \dfrac{N_p}{N_s}\dfrac{4}{\pi}\displaystyle\sum_{m=0}^{N}\frac{1}{[2m+1]}\left\{ \begin{array}{l} \dfrac{V_{in}}{|Z[m]|}\sin\left([2m+1]\,\omega_s t - \varphi_z[m]\right) - \\ \dfrac{V_{out}(t)}{|Z[m]|}\dfrac{N_p}{N_s}\sin\left([2m+1]\left(\omega_s t - \delta\right) - \varphi_z[m]\right) \end{array} \right\} \times \\ \dfrac{4}{\pi}\displaystyle\sum_{n=0}^{N}\frac{1}{[2n+1]}\left\{\sin\left([2n+1]\left\{\omega_s t - \delta\right\}\right)\right\} \end{array} \right. \end{aligned} \tag{3.24}$$

Expanding this equation gives:

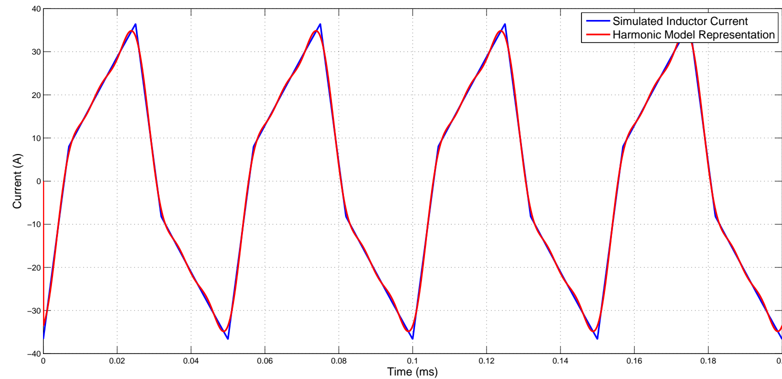
$$\begin{aligned} i_C\left(t\right) = -i_{load}\left(t\right) + \frac{8}{\pi^2}\frac{N_p}{N_s}\sum_{n=0}^{N}\sum_{m=0}^{N}\frac{1}{[2n+1]\,[2m+1]} \\ \times\left\{ \begin{array}{l} \dfrac{V_{in}}{|Z[m]|}\left[\begin{array}{l} \cos\left\{ \begin{array}{l} [2n+1]\left(\omega_s t - \delta\right) \\ \quad - [2m+1]\,\omega_s t + \varphi_z[m] \end{array} \right\} \\ \quad - \cos\left\{ \begin{array}{l} [2n+1]\left(\omega_s t - \delta\right) \\ \quad - [2m+1]\,\omega_s t - \varphi_z[m] \end{array} \right\} \end{array} \right] \\ - \dfrac{N_p}{N_s}\dfrac{V_{out}(t)}{|Z[m]|}\left[\begin{array}{l} \cos\left\{ \begin{array}{l} [2n+1]\left(\omega_s t - \delta\right) \\ \quad - [2m+1]\left(\omega_s t - \delta\right) + \varphi_z[m] \end{array} \right\} \\ \quad - \cos\left\{ \begin{array}{l} [2n+1]\left(\omega_s t - \delta\right) \\ \quad - [2m+1]\left(\omega_s t - \delta\right) - \varphi_z[m] \end{array} \right\} \end{array} \right] \end{array} \right\} \end{aligned} \tag{3.25}$$



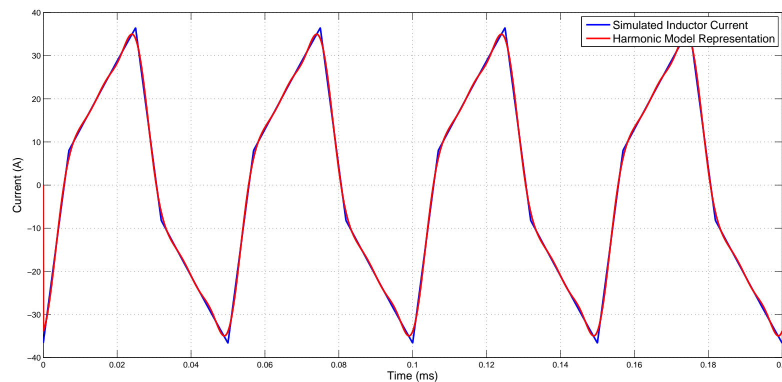
(a) N = 0 (Fundamental)



(b) N = 1



(c) N = 2



(d) N = 3

Figure 3.10: Harmonic Model Verification: Inductor Current

Eq. 3.25 is made up two components – a load current term and a series of harmonic summations that describe the current supplied by the switching bridges.

This capacitor current expression is validated by matching the simulation capacitor current to that predicted by the harmonic model. Fig. 3.11 shows the match obtained, where although the prediction of the harmonic model still includes ripples due to the contribution of each harmonic component, it provides an excellent match to the simulated capacitor current.

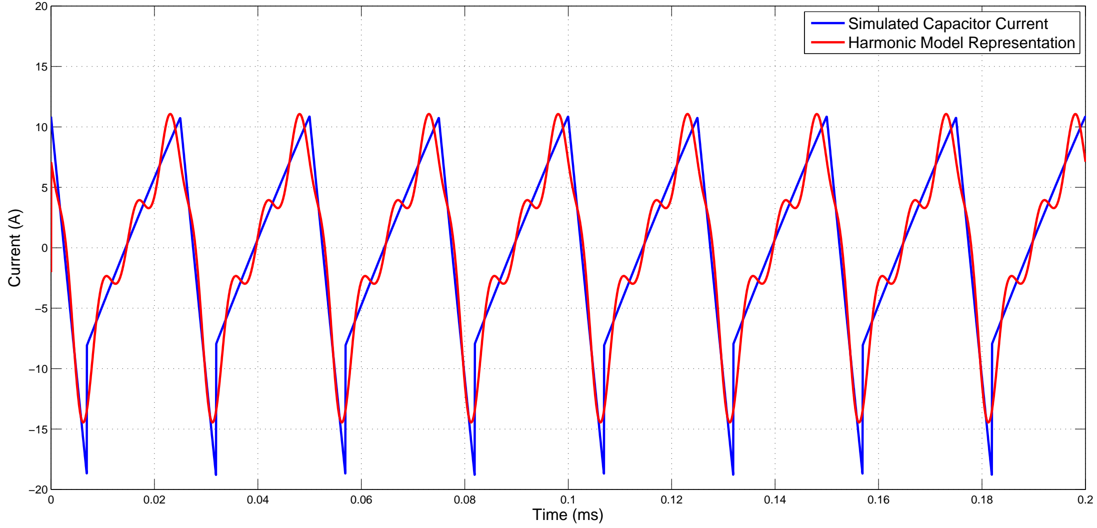


Figure 3.11: Harmonic Model Verification: Capacitor Current ($N = 3$)

Having determined the capacitor current (eq. 3.25), basic circuit theory is used to relate it to the output voltage, i.e.:

$$
\begin{aligned}
\frac{dV_{out}(t)}{dt} &= \frac{i_C(t)}{C} \\
&= -i_{load}(t) + \frac{8}{C\pi^2}\frac{N_p}{N_s}\sum_{n=0}^{N}\sum_{m=0}^{N}\frac{1}{[2n+1]\,[2m+1]} \\
&\times \left\{ \begin{array}{l}
\dfrac{V_{in}}{|Z\,[m]|}\left[\begin{array}{l}
\cos\left\{ \begin{array}{l} [2n+1]\,(\omega_s t - \delta) \\ \quad - [2m+1]\,\omega_s t + \varphi_z\,[m] \end{array} \right\} \\
\quad - \cos\left\{ \begin{array}{l} [2n+1]\,(\omega_s t - \delta) \\ \quad - [2m+1]\,\omega_s t - \varphi_z\,[m] \end{array} \right\}
\end{array} \right] \\
- \dfrac{N_p}{N_s}\dfrac{V_{out}(t)}{|Z\,[m]|}\left[\begin{array}{l}
\cos\left\{ \begin{array}{l} [2n+1]\,(\omega_s t - \delta) \\ \quad - [2m+1]\,(\omega_s t - \delta) + \varphi_z\,[m] \end{array} \right\} \\
\quad - \cos\left\{ \begin{array}{l} [2n+1]\,(\omega_s t - \delta) \\ \quad - [2m+1]\,(\omega_s t - \delta) - \varphi_z\,[m] \end{array} \right\}
\end{array} \right]
\end{array} \right\}
\end{aligned} \tag{3.26}
$$

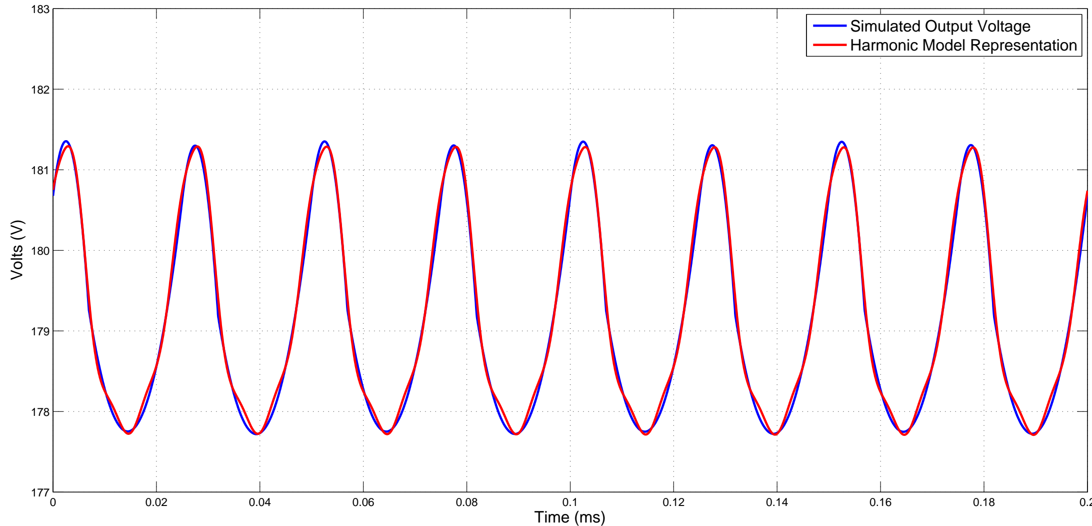


Figure 3.12: Harmonic Model Verification: Steady-state Output Voltage ($N = 3$)

This steady state output voltage expression is verified in Fig. 3.12 by comparing its result to that of the switched simulation. The error between these two waveforms is minimal, which validates the harmonic model.

With the steady-state behaviour of the DAB converter successfully modelled using harmonic analysis, the next task is to extend this model to predict converter transient behaviour. Under transient conditions, the assumption of cyclic steady-state operation to model the AC inductor current is no longer valid. Instead, classic circuit theory states that the inductor current response is made up of two parts, i.e. the *zero-state response* and the *zero-input response*[2] [60, 102].

The zero-input response corresponds to the behaviour of the system with zero input, but non-zero initial conditions, while the zero-state response is the response of a system to a non-zero input, but with an initial condition of zero. For the case of the AC inductor current of the DAB converter, the zero-input response is an exponential decay at the Resistive-Inductive ($R - L$) time constant of the AC link, while the zero-state response is the steady-state response to a particular input condition. The steady-state nature of the harmonic model allows it to predict the zero-state response, but not the zero-input response.

This is illustrated in Fig. 3.13, which shows the response of the AC inductor current to a step change in input phase shift (50° to 70° lagging phase shift). The harmonic model immediately jumps to the new steady state solution, while the exponential decay characteristic of the zero-input response is not modelled.

However, an important feature of Fig. 3.13 is the *magnitude* of the exponential decay caused by the transient step. As the figure shows, a step change in phase

[2] Also known as the *forced response* & *natural response*, respectively.

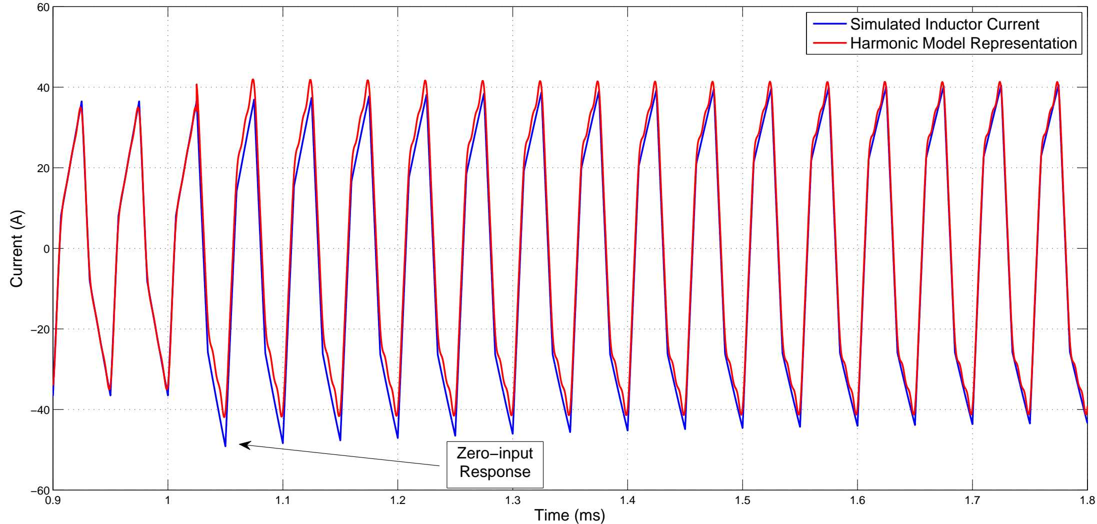


Figure 3.13: Harmonic Model Verification: AC Inductor Current (Transient step) ($N = 3$)

shift causes a relatively small change in the current magnitude, only 10 A for a peak-to-peak current of ≈40 A.

Since the zero-input response of the inductor current is relatively small in magnitude, its effect is minimal and can be ignored. As a result, the assumption of cyclic steady-state is still valid for transient as well as steady state purposes. This assumption allows the steady-state output DAB voltage expression (eq. 3.26) to also model the dynamic response of the DAB converter.

To further test this proposition, the output voltage equation (eq. 3.26) was also evaluated with a change in phase shift δ. The result of this test is plotted in Fig. 3.14, where the harmonic model prediction is compared to the response of the simulated DAB converter when subjected to the same input conditions of Fig. 3.13 (i.e. 50° to 70° lagging phase shift step). As expected, the harmonic model matches the steady-state conditions of both the inductor current and the capacitor voltage very well. Even during the transient step change though, the harmonic model still predicts the average DC component of the output voltage dynamics very well, with a discrepancy only visible in its high frequency ripple component. Since the magnitude of this discrepancy is minor, the output voltage dynamics are closely matched by the harmonic model, verifying its accuracy.

Unfortunately, although accurate, the harmonic model is non-linear in nature because it contains a state (V_{out}) that is multiplied with the input (δ). This makes this form of the model complex and unsuitable for linear closed-loop regulator design.

To make the harmonic model more tractable, eq. 3.26 must be simplified. To do this while maintaining accuracy is a two stage process. First, it can be argued that

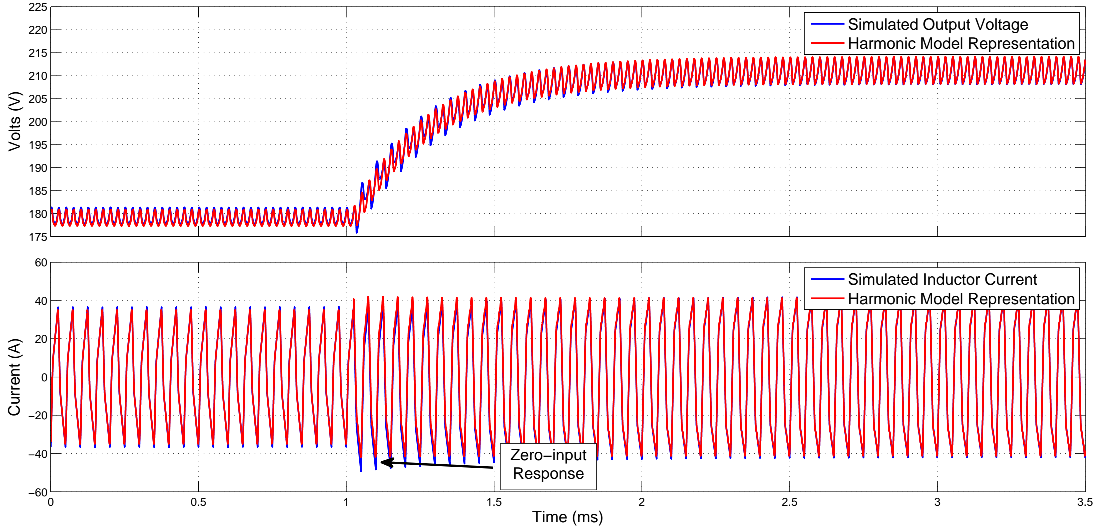


Figure 3.14: Harmonic Model Verification: Output Voltage ($N = 3$)

the high frequency ripple in the output DC voltage waveform does *not* affect overall system stability. This is because the ripple is inherent to the converter switching process, and is not caused by a controller input. Hence it is not possible to design a controller that responds to this ripple. Thus the output voltage ripple component of the output voltage can be ignored for control systems analysis, leaving only the DC average component of the waveform. Second, the non-linearities of the model can be simplified by applying standard linearisation theory. This will result in a linearised DC average model of the DAB converter output voltage dynamics, allowing classical control design techniques to then be applied [99].

In order to develop a 'low frequency' average harmonic model, the high frequency terms of eq. 3.26 must be removed. This is achieved by only considering harmonic terms where $n = m$, as this is the only condition that eliminates the high frequency $\omega_s t$ terms from the summation terms of this equation. The resulting simplified model is given as:

$$\begin{aligned}\frac{dV_{out}(t)}{dt} = f\left(V_{out}(t), \delta\right) = -i_{load}(t) + \frac{8}{C\pi^2}\frac{N_p}{N_s}\sum_{n=0}^{N}\frac{1}{\left[2n+1\right]^2} \\ \times\left\{\begin{aligned}&\frac{V_{in}}{|Z[n]|}\cos\left\{[2n+1]\,\delta - \varphi_z[n]\right\} \\ &-\frac{N_p}{N_s}\frac{V_{out}(t)}{|Z[n]|}\cos\left\{\varphi_z[n]\right\}\end{aligned}\right\}\end{aligned} \tag{3.27}$$

The validity of this step is verified in Fig. 3.15, where the response of the low frequency harmonic model is compared to the switched simulation. It shows that in spite of considerably simplifying the model, the key features of the output voltage

dynamics are still preserved. This proves that the low frequency component of the output voltage expression still accurately predict DAB converter dynamics.

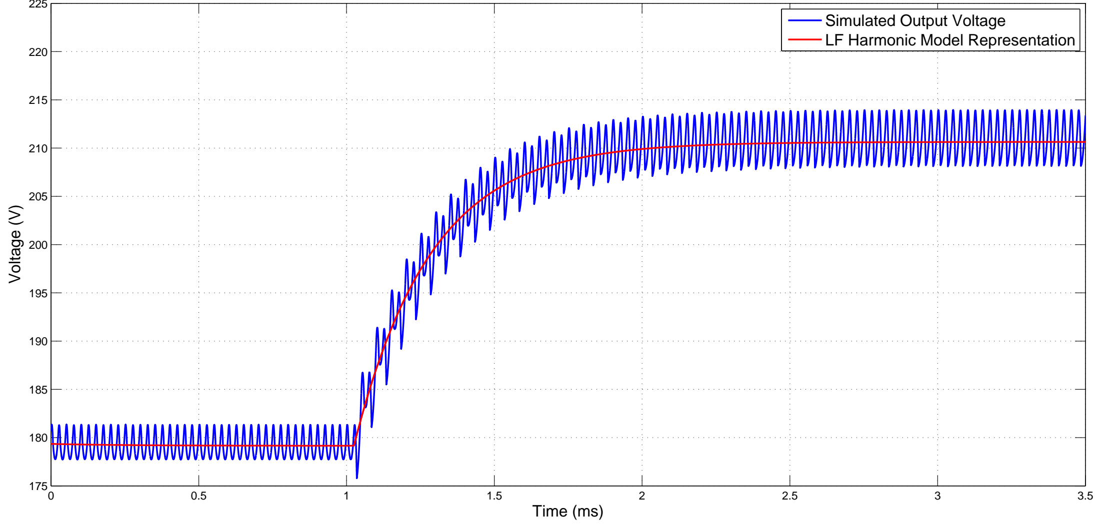


Figure 3.15: Harmonic Model Verification: Output Voltage (DC Terms Only) ($N = 3$)

Having extracted the key harmonics from the output voltage expression, the last step of model development is linearisation. Standard linearisation techniques [99] are applied to this model by developing partial differential equations that describe the variation in DAB output voltage around a steady-state operating point (V_{out_0}, δ_0, i_{load_0}) in response to small changes in phase shift and load current. This results in the following equations:

$$\begin{aligned}\frac{d\left(V_{out_0}+\Delta V_{out_0}(t)\right)}{dt} &\approx f\left(V_{out_0},\delta_0,i_{load_0}\right)\\ &\qquad + \left.\frac{\partial f}{\partial V_{out}}\right|_0 \Delta V_{out}(t) \quad + \left.\frac{\partial f}{\partial i_{load}}\right|_0 \Delta i_{load}(t) + \left.\frac{\partial f}{\partial \delta}\right|_0 \Delta\delta\end{aligned} \tag{3.28}$$

Solving these partial derivatives in terms of the low frequency non-linear dynamic output voltage expression (eq. 3.27) gives:

$$\begin{aligned}\frac{d\Delta V_{out}(t)}{dt} &= A\Delta V_{out} + B_\delta \Delta\delta + B_I \Delta i_{load}\\ &= \left\{\begin{aligned} &\left\{-\frac{8}{C\pi^2}\left(\frac{N_p}{N_s}\right)^2 \sum_{n=0}^{N}\left[\frac{\cos\left(\varphi_z[n]\right)}{[2n+1]^2\left|Z[n]\right|}\right]\right\}\Delta V_{out}\\ &\qquad\qquad\qquad + \left\{-\frac{1}{C}\right\}\Delta i_{load}\\ &+\left\{\frac{8V_{in}}{C\pi^2}\frac{N_p}{N_s}\sum_{n=0}^{N}\left[\frac{\sin\left(\varphi_z[n]-[2n+1]\,\delta_0\right)}{[2n+1]\left|Z[n]\right|}\right]\right\}\Delta\delta \end{aligned}\right\}\end{aligned} \tag{3.29}$$

This small-signal linearised harmonic model is a first-order system, with two input variables (δ & i_{load}) and a single output variable (V_{out}). Two of the model coefficients are constants (A & B_I), while B_δ varies with input phase shift, as shown in Fig. 3.16. The variation in plant characteristics seen in this figure must be accounted for when designing a closed loop regulator. Additionally, Fig. 3.16 also illustrates the difference seen for the value of the model coefficient B_δ when only the fundamental harmonic is considered, and when a summation of significant harmonics is employed. This difference proves that higher order harmonics *do* significantly affect system behavior, and thus must be included as part of an accurate dynamic model.

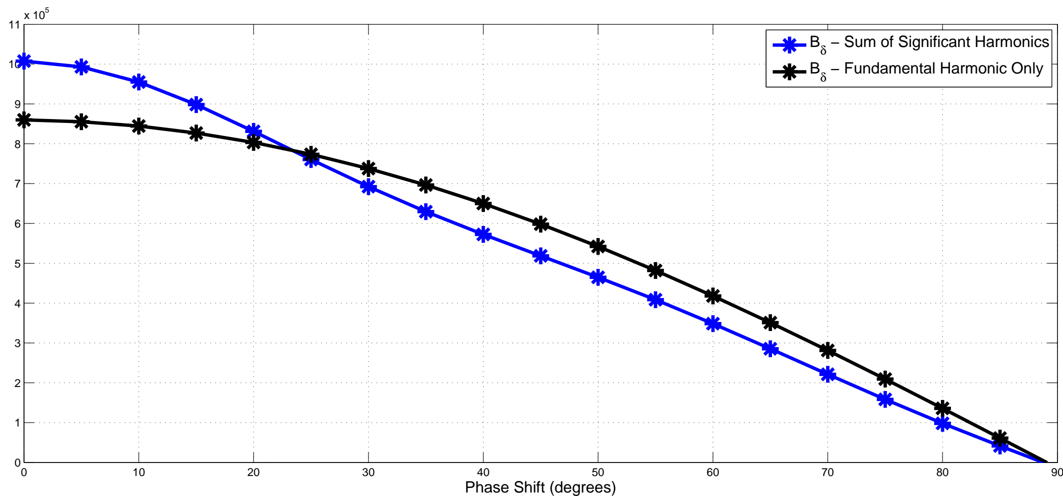
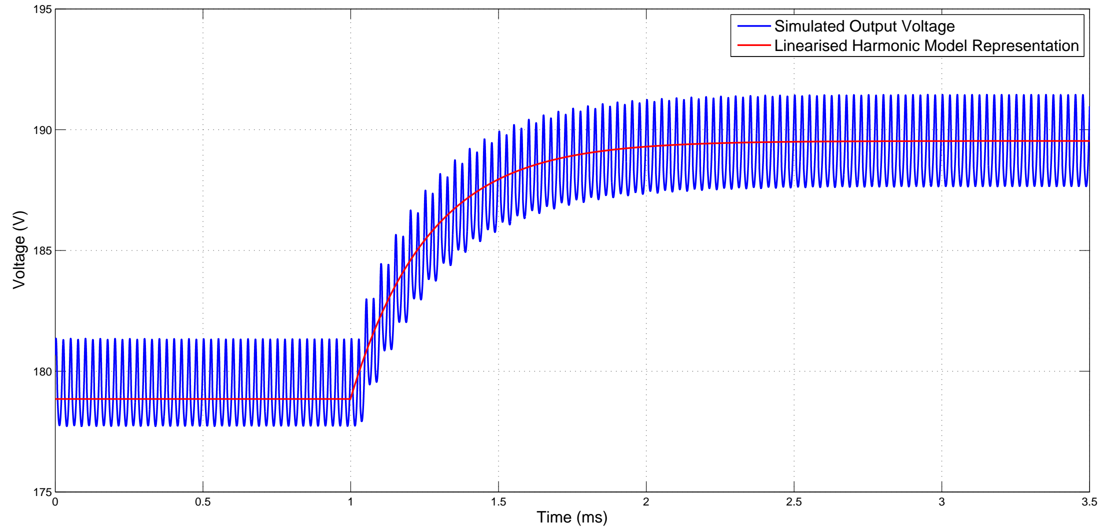
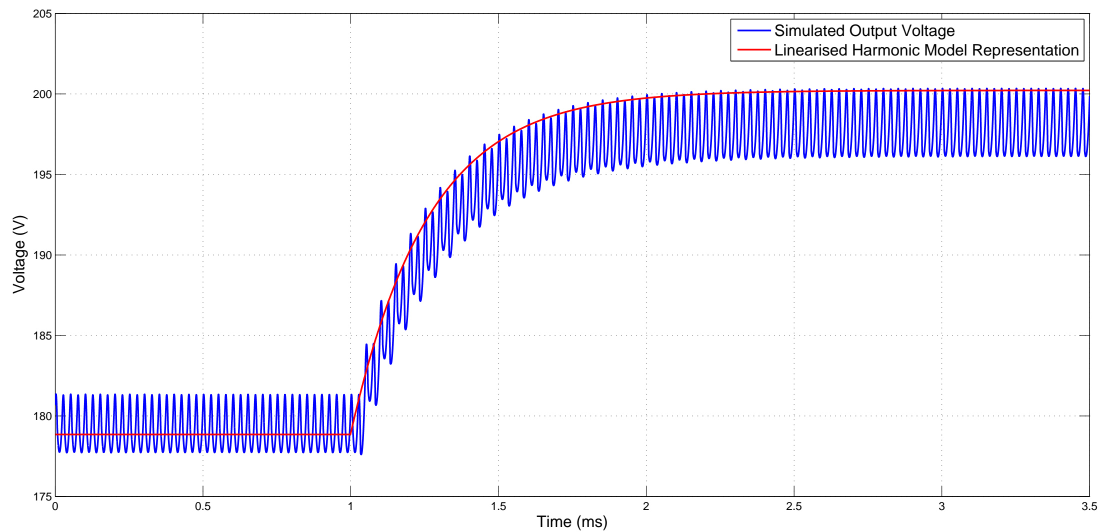


Figure 3.16: Variation in B_δ

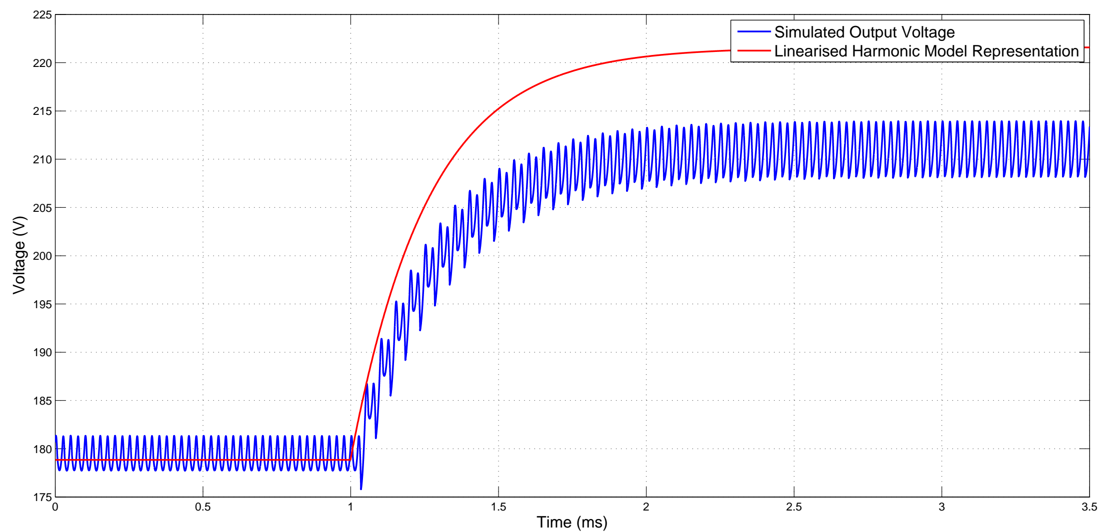
To test the validity of the linearised harmonic model, its response to a step change in δ is compared to that of the switched simulation. Like all linearised small-signal models, this model is only valid for small variations around its operating point. Hence the linearised model was tested for input step changes of varying magnitudes. The results are plotted in Fig. 3.17. When the input step is small (5°), the linearised harmonic model matches the dynamics of the switched simulation well, as shown in Fig. 3.17a. However, as expected, the quality of this match deteriorates as the step size increases, as is shown for a step change of 10° in Fig. 3.17b and 20° in Fig. 3.17c. From these plots, it can be seen that the linearised harmonic model is reasonably valid for changes of up to about 10° in operating condition. This corresponds to ≈5% of the entire 180° dynamic range. However, for larger changes in phase angle, the model will need further adaptation.



(a) 5° step change (50° → 55°)



(b) 10° step change (50° → 60°)



(c) 20° step change (50° → 70°)

Figure 3.17: Harmonic Model Verification: Output Voltage (Linearised Model) ($N = 3$)

3.6 Deadtime Compensation

In a hard-switched converter phase leg, deadtime is defined as the blanking time required between turning off an outgoing switch and turning on its complimentary incoming switch. This delay is necessary because of non-zero switch transition times, to avoid the possibility of an instantaneous phase leg short circuit (shoot-through). Deadtime is well known to affect the dynamics of the DAB converter [17,18,105,107, 111,125--127]. In this section, this effect is analysed, and a closed form expression that predicts its effect is derived.

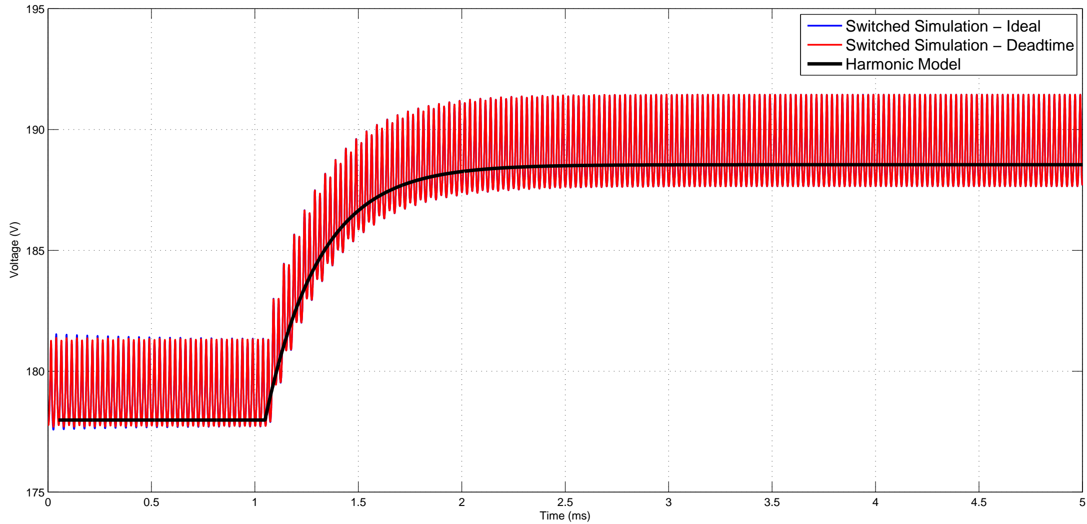
3.6.1 The Deadtime Effect

The effect of deadtime is explored for the DAB converter by comparing the responses of a switched simulation of an ideal DAB converter (without deadtime) to one with deadtime included. The input phase shift to these simulated converters was step changed by 5° at two different operating points – at a lower phase shift ($\delta = 20°\rightarrow 25°$), and at a higher phase shift ($\delta = 50°\rightarrow 55°$). These simulated responses were then compared to those predicted by the small-signal harmonic model, and are plotted in Fig. 3.18.

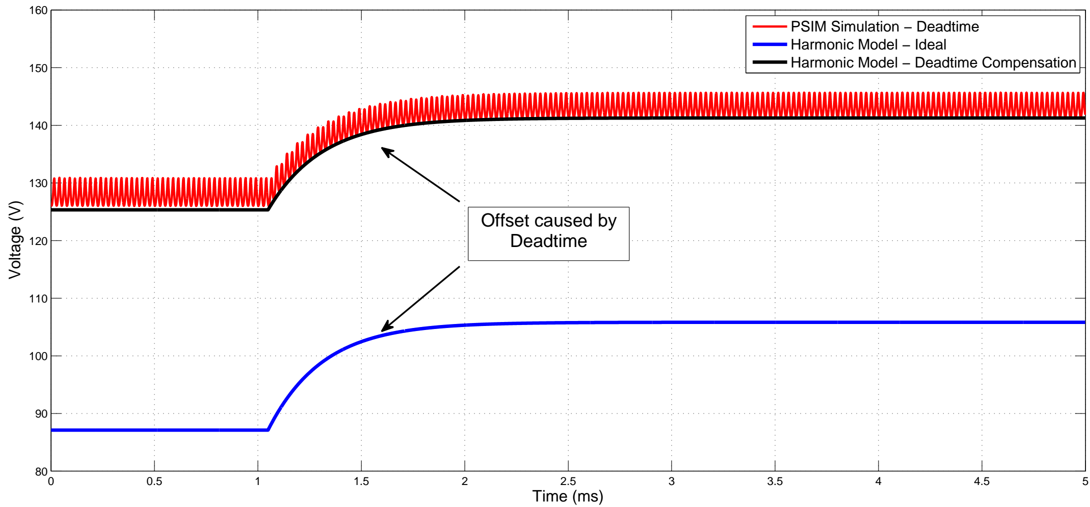
Fig. 3.18a shows that at the higher phase shift operating point ($\approx 50°$), both the ideal and the non-ideal switched simulations match well, and the harmonic model successfully predicts converter dynamics. However, this is not the case for the lower phase shift operating point (Fig. 3.18b, $\approx$20°). At this operating condition, deadtime is seen to significantly affect the converter response, where a substantial offset in the output voltage is seen. However, it is important to note that the dynamics predicted by the harmonic model still match those of the ideal simulation.

As a result of this simulation investigation, two conclusions can be drawn. First of all, deadtime only affects the behaviour of the DAB converter across some portion of the overall operating range. Second, since the harmonic model was developed based on ideal converter behaviour, its prediction is valid for the ideal system, but inadequate for the non-ideal case that includes the effect of deadtime.

The new harmonic model must therefore be extended to incorporate the deadtime effect. To do this, the behaviour of the converter during the deadtime period must first be analysed.



(a) 50° →55° step (No Deadtime Effect Visible)



(b) 20° →25° step (Deadtime Effect Visible)

Figure 3.18: Operating point dependence of the deadtime effect

3.6.2 Converter Behaviour During Deadtime

To better understand the behaviour of the DAB converter during the deadtime interval, deadtime is first analysed in the context of a single phase leg (Fig. 3.19).

During the deadtime period, both switches of the phase leg are switched off. The midpoint output voltage then no longer depends on switch conditions, but instead on external factors such as the bridge output current [10, 11]. Since the phase leg switches are not conducting, this current must flow through their antiparallel diodes. The output voltage of each phase leg during this time is therefore determined by which diode is conducting, i.e. if an upper diode conducts, the phase leg output voltage clamps to its upper DC rail, and if a lower diode conducts, the phase leg output voltage clamps to its lower DC rail. This is significant because it can cause a

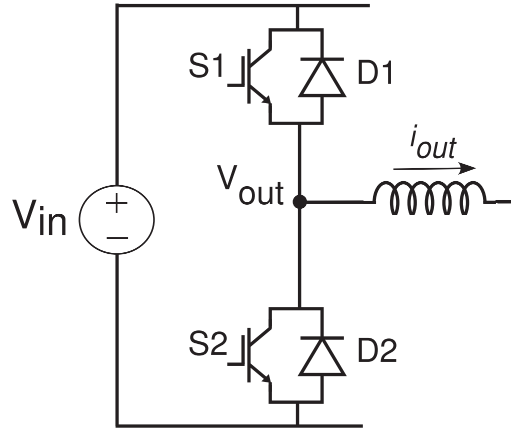


Figure 3.19: Single Phase Leg with an Inductive Load

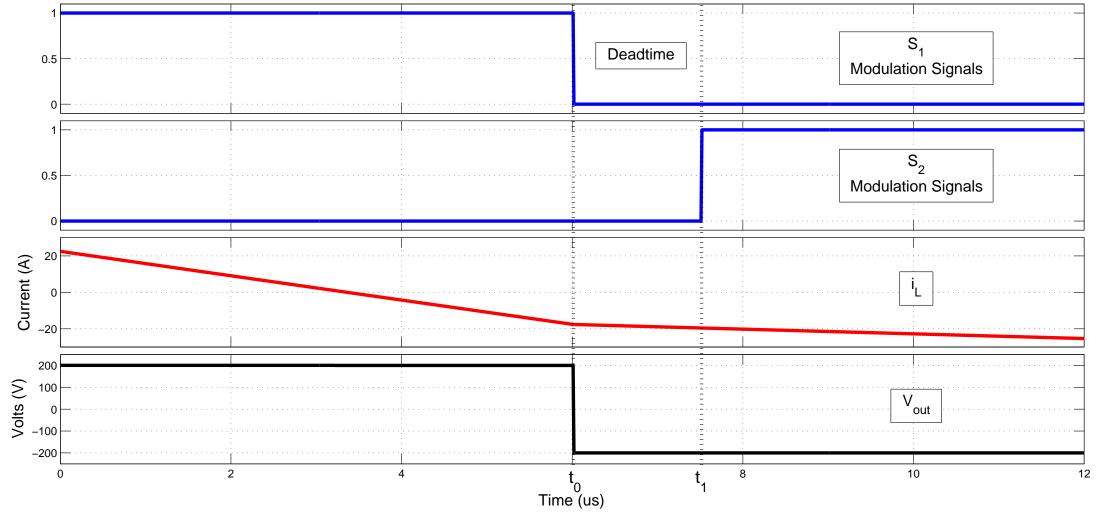
discrepancy between the signal commanded by the modulator and the true voltage that appears at the phase leg midpoint.

The output voltage error caused by deadtime is illustrated in Fig. 3.20 for a particular phase leg of the DAB converter. There are three possible conditions that exist, i.e. zero error, full error and partial error, depending on the magnitude and polarity of the phase leg output current during the deadtime interval.

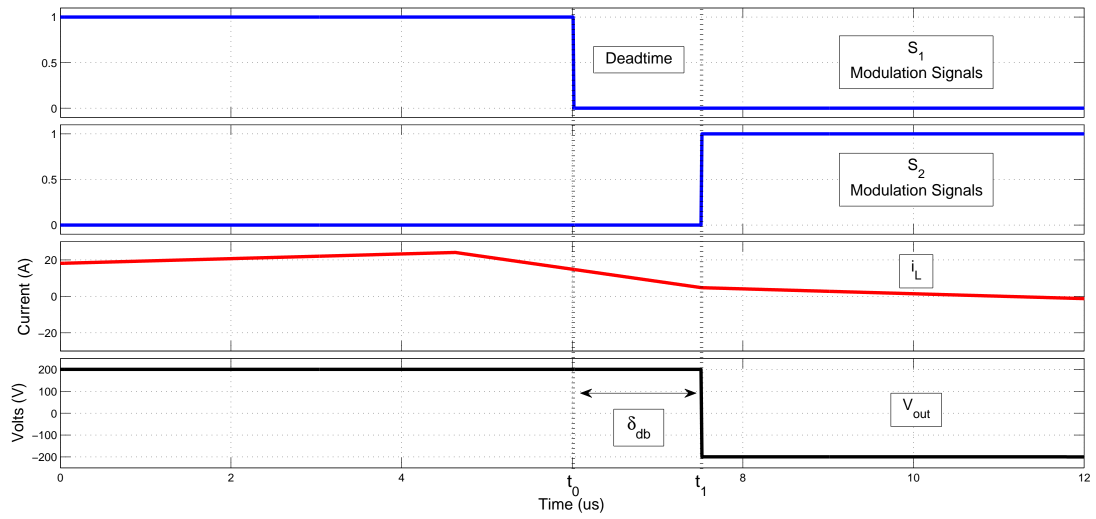
In Fig. 3.20a, the output current i_{out} is negative at the start of the deadtime interval (t_0), which means that it was conducting through switch S_1. When the deadtime interval begins, switch S turns off, so the current immediately commutes from switch S_1 to antiparallel diode D_2. The phase leg output voltage V_{out} therefore immediately changes polarity, and no voltage error effect is seen (i.e. zero error state).

In Figs. 3.20b & 3.20c, i_{out} is positive at the start of the deadtime interval (t_0). This means that in both cases, antiparallel diode D_1 is conducting. When switch S_1 turns off and the phase leg enters its deadtime interval, this current continues to flow through this diode, so the output voltage remains clamped to the positive DC bus. In Fig. 3.20b, this current is still flowing through the antiparallel diode D_1 at the end of the deadtime interval, as it has not slewed back through zero. This results in an error in the output voltage that is the length of the deadtime interval (i.e. full error state). If however, the current *does* slew through zero during this interval, as seen in Fig. 3.20c, diode D_1 stops conducting and the current commutes through diode D_2. This causes a voltage transition in the middle of the deadtime interval (a partial error state).

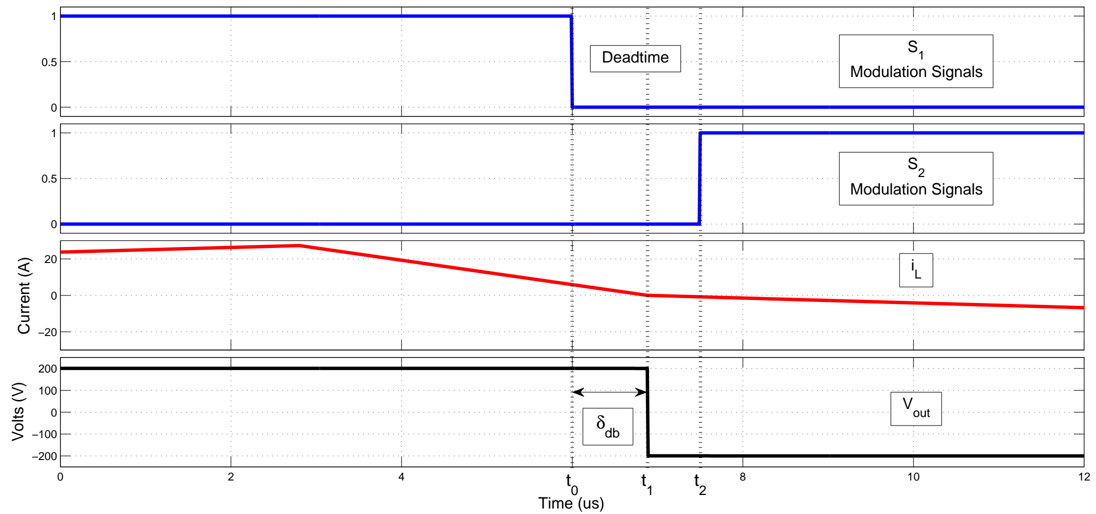
When this concept of voltage error is extended to the DAB converter, it manifests itself as an error in phase shift. This means that the phase shift commanded by the PSSW modulator does not necessarily correspond to the phase shift seen at the bridge AC output voltage terminals. In order to accurately converter dynamics, this



(a) No Deadtime Effect



(b) Full Deadtime Effect



(c) Partial Deadtime Effect

Figure 3.20: Deadtime Effect in a Phase Leg

phase shift error must be incorporated into the harmonic model. To do this, the flow of current through the converter during the deadtime period must be analytically determined.

3.6.3 Modelling the Deadtime effect

To model the flow of current through the DAB converter during the deadtime period, the operating state of the DAB converter must first be determined. Four possible converter operating states exist, i.e. the modulation signals of the primary bridge could lead or lag those of the secondary bridge, and the DC voltage level of the primary bridge could be greater than or less than that of the secondary bridge.

These four states can be reduced to just two by virtue of the symmetric converter topology, which makes the definition of the primary and secondary bridge irrelevant. Hence the two bridges can be simply defined as the *High Voltage* (HV) Bridge & the *Low Voltage* (LV) Bridge, leaving just two states, i.e.:

- HV bridge *leading* the LV bridge
- HV bridge *lagging* the LV bridge

The salient idealised switching waveforms for both operating states are simulated[3] & plotted in Figs. 3.22 & 3.22. In both cases, when the HV bridge begins its deadtime period, (point t_0 in Figs. 3.21 & 3.22), the flow of the AC inductor current immediately commutes from the active switches to the opposite antiparallel diode pair. Hence the HV bridge output bridge switches state instantaneously, and no phase shift error is observed.

A similar situation is seen for the LV bridge during high phase shift operation, as no phase shift error is observed, since the active switches carry the AC current during the deadtime interval. However this is not the case at lower phase shift operating points, where the anti-parallel diodes conduct during the deadtime period. During this interval, the bridge voltage is held high or low depending on which pair of antiparallel diodes conduct. This manifests itself as a *phase shift error*, illustrated in Figs. 3.21 & 3.22, which can be up to the full deadtime period (δ_{DT}) in duration.

[3] For the simulation investigations presented in this chapter, the following DC bus voltages were assumed:

- HV Bridge Bus Voltage (V_H): $200V$
- LV Bridge Bus Voltage (V_L): $\frac{N_p}{N_s}150V$

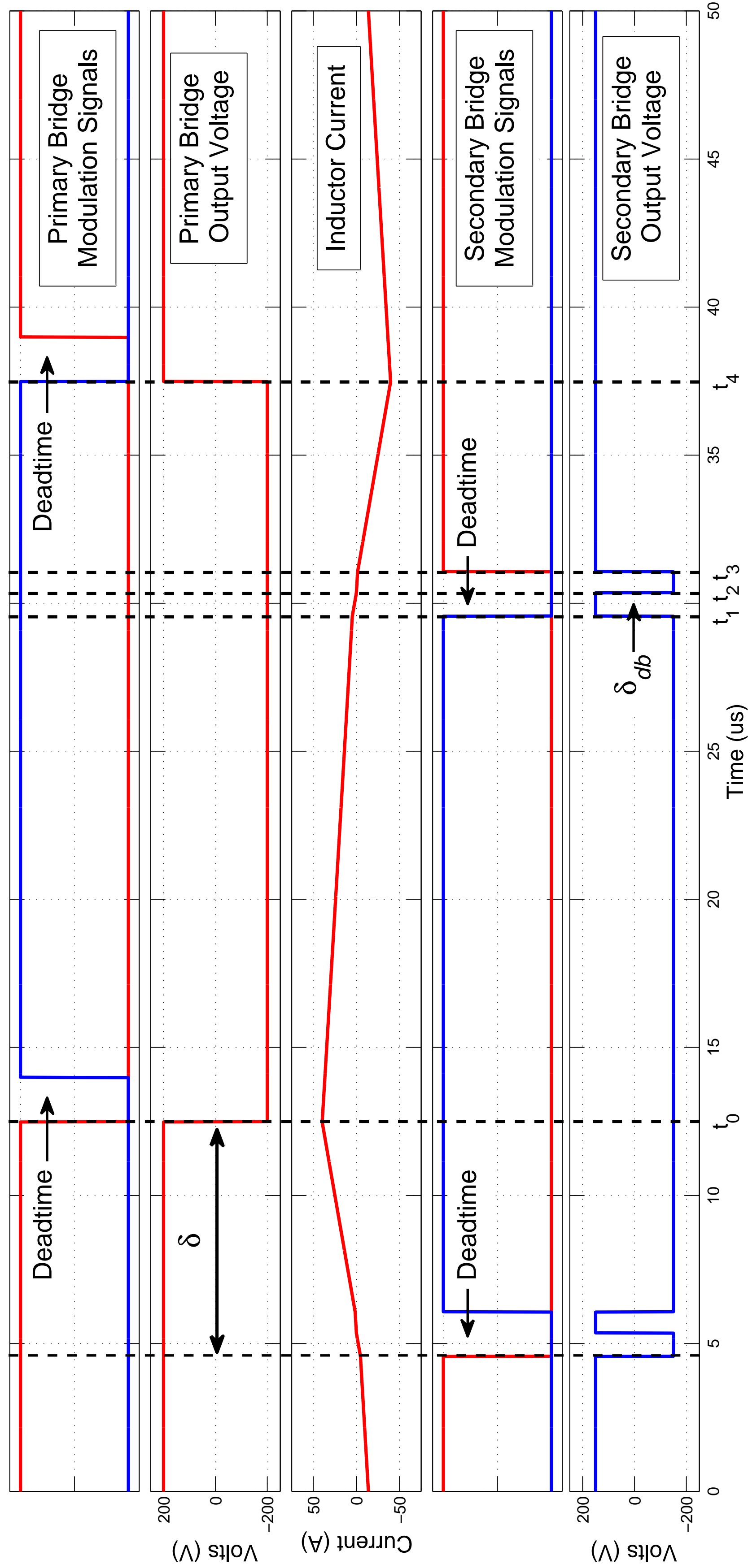


Figure 3.21: Deadtime influence - HV bridge *lags* the LV bridge

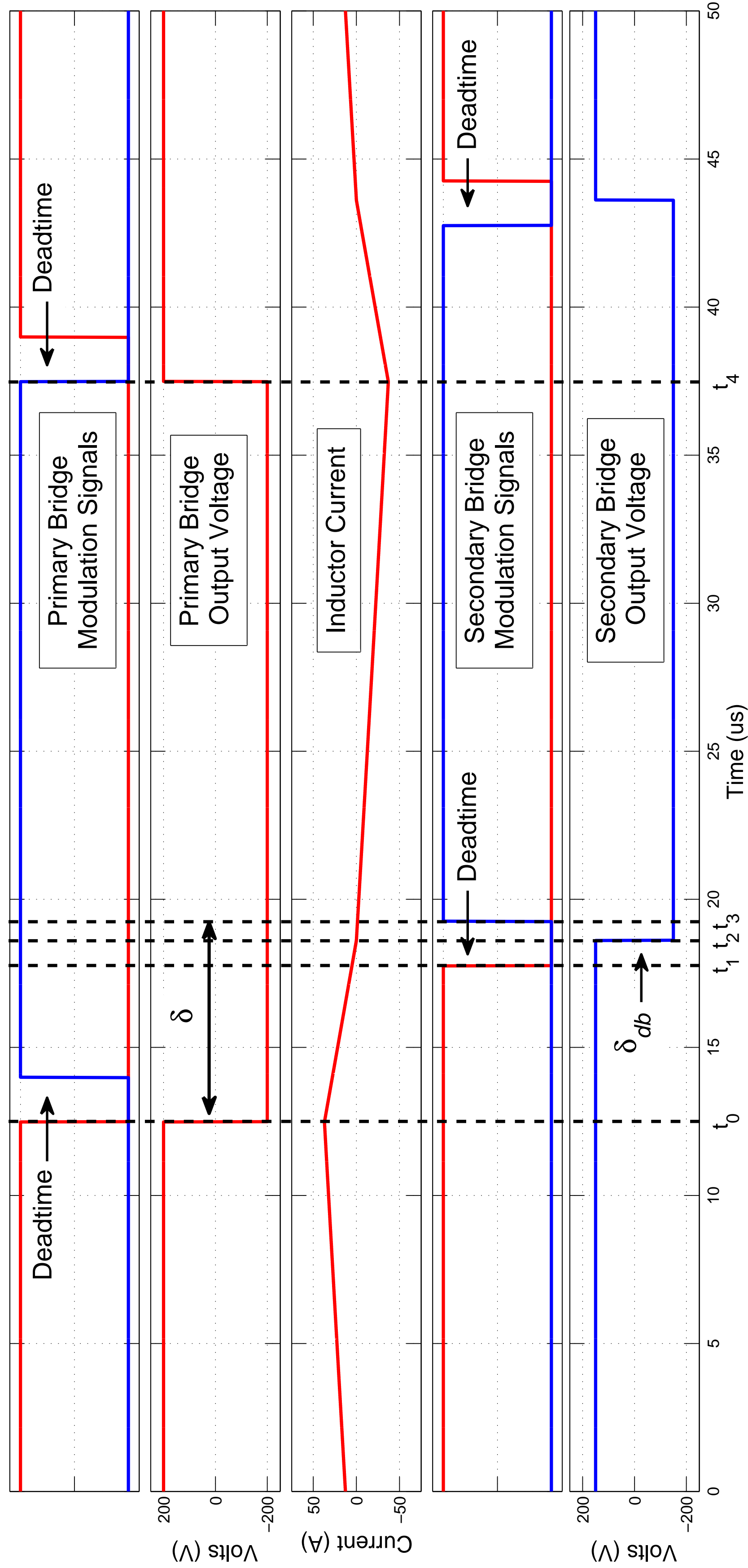


Figure 3.22: Deadtime influence - HV bridge *leads* the LV bridge

The transition between "low" & "high" phase shifts occurs when the AC inductor current changes polarity during the LV bridge deadtime period. The response of the output voltage at this operating condition needs to be considered separately for both the leading and lagging switching alternatives.

HV bridge lagging the LV bridge *(Fig. 3.21)*

At this operating point, the current in the LV bridge instantaneously commutes from the active switches to the opposite pair of antiparallel diodes at the start of the deadtime interval (t_1). This causes the output voltage to reverse polarity, and the inductor current begins to slew towards zero. When this current slews through zero (t_2), the current conduction path commutates to the opposite pair of antiparallel diodes, causing the bridge output voltage to change polarity again. The current holds the voltage until the end of the deadtime period, resulting in a short negative voltage pulse of width δ_{db} in the output voltage waveform.

The duration of this pulse is dependent on when the AC inductor current slews through zero, and effectively *reduces* the applied phase shift.

HV bridge leading the LV bridge *(Fig. 3.22)*

At this operating condition, the LV bridge output voltage does *not* change polarity when its deadtime period begins (t_1). This is because the AC inductor current is already flowing through the antiparallel diodes of the LV bridge, so the switch transition does not change the conduction path.

However, when this current slews through zero (t_2), the current commutes to the opposite pair of antiparallel diodes, and the bridge output voltage changes state. This delay in the output voltage transition is of duration δ_{db}, and *augments* the commanded phase shift.

3.6.4 Analytical calculation of the phase shift error effect

The previous subsection has identified that the distortion seen in the bridge output voltage waveform due to deadtime depends primarily on the AC inductor current during this interval. This means that the phase shift error δ_{db} can be calculated for all operating points *if* a closed form expression that describes the current waveform can be developed.

A method for deriving this expression is presented in [69, 128], which recognises that the inductor current is piecewise linear, cyclic and symmetric, as Figs. 3.22 & 3.22 illustrate for both leading and lagging switching alternatives. To model the behaviour of the inductor current, each half-cycle is divided into piecewise linear intervals based on the switching states ($t_0 \rightarrow t_4$). The applied voltage during each interval is then established and the duration of each interval determined. Basic circuit theory ($V = L\frac{di}{dt}$) is then applied to calculate the inductor current. Repeating this calculation for each switching interval gives a series of piecewise linear equations, listed in Table 3.4.

Time Period	HV bridge *lagging* LV bridge (Fig. 3.21)	HV bridge *leading* LV bridge (Fig. 3.22)
t_0 (+ve peak)	$i(t_0)$	$i(t_0)$
$t_0 \rightarrow t_1$	$i(t_1) = i(t_0) - \frac{V_H - V_L}{L}\left(\frac{\pi-\delta_c}{2\pi f_s}\right)$	$i(t_1) = i(t_0) - \frac{V_H + V_L}{L}\left(\frac{\delta_c}{2\pi f_s}\right)$
$t_1 \rightarrow t_2$	$i(t_2) = i(t_1) - \frac{V_H + V_L}{L}\left(\frac{\delta_s}{2\pi f_s}\right) = 0$	$i(t_2) = i(t_1) - \frac{V_H + V_L}{L}\left(\frac{\delta_s}{2\pi f_s}\right) = 0$
$t_2 \rightarrow t_3$	$i(t_3) = -\frac{V_H - V_L}{L}\left(\frac{\delta_{DT}-\delta_s}{2\pi f_s}\right)$	$i(t_3) = -\frac{V_H - V_L}{L}\left(\frac{\delta_{DT}-\delta_s}{2\pi f_s}\right)$
$t_3 \rightarrow t_4$ (-ve peak)	$i(t_4) = i(t_3) - \frac{V_H + V_L}{L}\left(\frac{\delta_c-\delta_{DT}}{2\pi f_s}\right) = -i(t_0)$	$i(t_4) = i(t_3) - \frac{V_H - V_L}{L}\left(\frac{\pi-\delta_c-\delta_{DT}}{2\pi f_s}\right) = -i(t_0)$

Table 3.4: Piecewise Linear solution for Inductor Current change during DAB Converter switching process.

In this table, V_H & V_L are the voltages seen by the AC inductor applied by the HV & LV bridges respectively. Additionally, the slew time δ_s defines (in radians) the time taken for the inductor current to slew to zero during the deadtime interval.

Since the phase shift error (δ_{db}) is caused by the current that slews during the deadtime period, an expression that describes the slew time (δ_s) can be derived. Also, since the AC inductor current is cyclic and half-wave symmetric, the positive peak current and the negative peak current have the same magnitude ($|i(t_0)| = |i(t_4)|$). Hence the piecewise linear equations presented in Table 3.4 completely define the inductor current during the entire half-cycle interval. This means that by setting

$i(t_4) = -i(t_0))$, an expression for δ_s can be solved for both the leading and lagging switching alternatives as:

$$\delta_s = \delta_c - \frac{V_H - V_L}{V_H}\frac{\pi}{2} - \frac{V_L}{V_H}\delta_{DT} \qquad \text{(HV leads LV)} \tag{3.30a}$$

$$\delta_s = -\delta_c + \frac{V_H - V_L}{V_H}\frac{\pi}{2} \qquad \text{(HV lags LV)} \tag{3.30b}$$

From the slew time equations (eq. 3.30), the phase shift error δ_{db} is determined by first identifying the converter operating condition. This is necessary because the phase shift error augments the applied phase shift when the HV bridge leads the LV bridge, and reduces it when the HV bridge lags the LV bridge. This allows δ_{db} to be determined based on δ_s. The relationship between δ_s and δ_{db} is therefore summarised in Table 3.5.

	$V_{in} > \frac{N_p}{N_s}V_{out}$		$V_{in} < \frac{N_p}{N_s}V_{out}$	
	Condition	δ_{db}	Condition	δ_{db}
Primary Bridge Leads Secondary	$\delta_s > \delta_{DT}$	0	$\delta_s < 0$	0
	$0 < \delta_s < \delta_{DT}$	$(\delta_{DT} - \delta_s)$	$0 < \delta_s < \delta_{DT}$	$-\delta_s$
	$\delta_s < 0$	δ_{DT}	$\delta_s > \delta_{DT}$	$-\delta_{DT}$
Primary Bridge Lags Secondary	$\delta_s > \delta_{DT}$	δ_{DT}	$\delta_s < 0$	$-\delta_{DT}$
	$0 < \delta_s < \delta_{DT}$	δ_s	$0 < \delta_s < \delta_{DT}$	$-(\delta_{DT} - \delta_s)$
	$\delta_s < 0$	0	$\delta_s > \delta_{DT}$	0

Table 3.5: Phase Shift Error Effect.

To verify this analysis, the phase shift error predicted by the new analytic model was compared to the phase shift error measured in the switched simulation of the DAB converter. The excellent match seen in Fig. 3.23 confirms the deadtime modelling techniques presented in this section for the idealised DAB converter.

The effect of non-zero device output capacitance (caused by device non-idealities or auxilliary ZVS capacitors) was experimentally explored, and found to not significantly alter the phase error from the ideal scenario (see Chapter 7). This is because the phase error is in fact an error in the applied volt-seconds, and as both rising and falling waveform edges are equally affected by the device capacitance, the applied volt-second average does not change significantly.

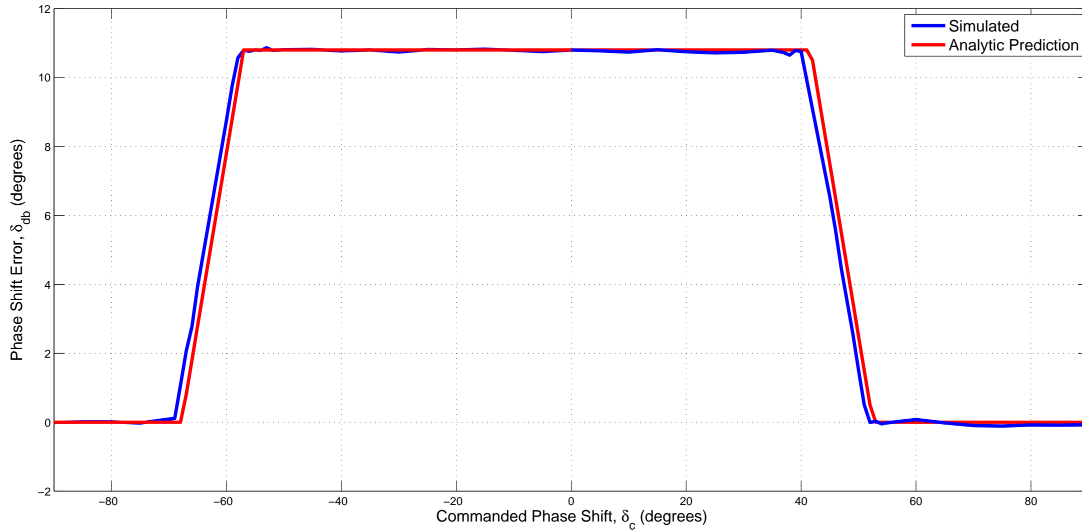


Figure 3.23: Deadtime influence in the DAB converter

3.7 Final Model Derivation & Validation

The final DAB dynamic model must include the ideal harmonic model as well as the phase shift error effect caused by deadtime. This was achieved by summing the commanded phase shift input to the harmonic model with the phase shift error predicted by the deadtime compensation algorithm, as illustrated in Fig. 3.24.

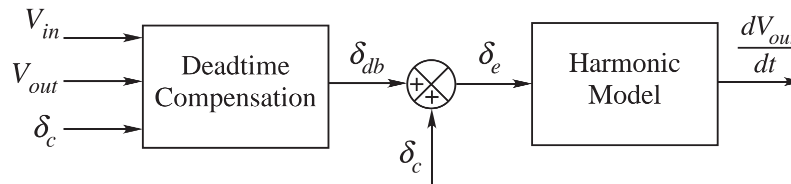


Figure 3.24: Block Diagram of Final Dynamic Model.

Fig. 3.25 validates the final dynamic model. In this figure, the response of a simulated DAB converter that included deadtime is compared to the prediction of the harmonic model. It shows that when the phase error effect of deadtime is correctly incorporated into the harmonic model, it provides a close match to the switched simulation. This validates the model and the dynamic modelling principles presented in this chapter.

3.8 Summary

This chapter has presented the derivation of a dynamic model for the DAB bi-directional DC-DC converter.

A new modelling technique was developed to derive this model, based on the switching harmonics that are present in the converter modulation waveforms. The

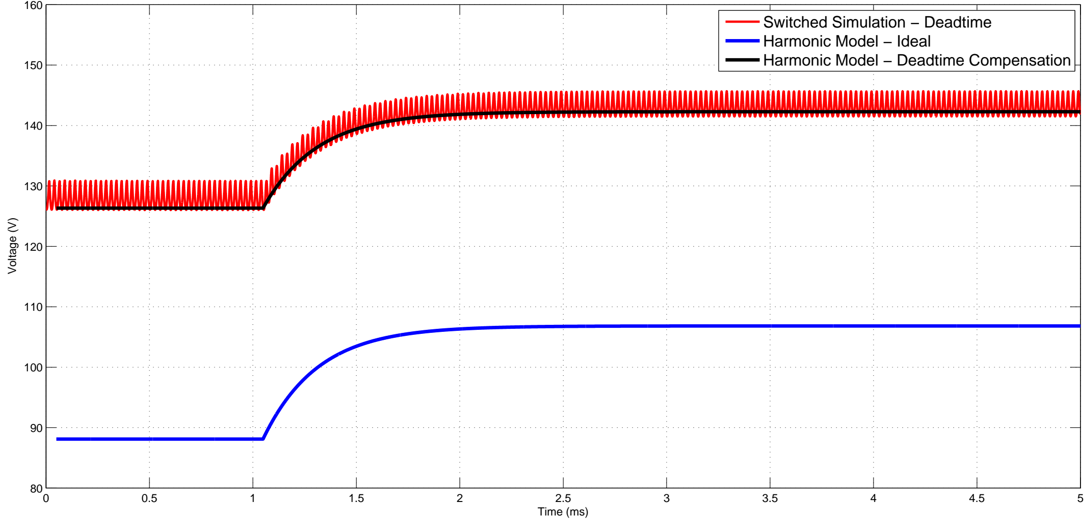


Figure 3.25: Validating the Final Model ($N = 3$)

contributions of each significant harmonic were identified and summed together to form a first-order non-linear representation of the converter dynamics, before being linearised into state space form, summarised again as eq. 3.31.

$$\begin{aligned}
\Delta \dot{V_{out}}(t) &= A\Delta V_{out} + B_\delta \Delta\delta + B_I \Delta i_{load} \\
\text{where} \quad A &= -\frac{8}{C\pi^2}\left(\frac{N_p}{N_s}\right)^2 \sum_{n=0}^{N}\left[\frac{\cos(\varphi_z[n])}{[2n+1]^2 |Z[n]|}\right] \\
B_\delta &= \frac{8V_{in}}{C\pi^2}\frac{N_p}{N_s}\sum_{n=0}^{N}\left[\frac{\sin(\varphi_z[n] - [2n+1]\delta_0)}{[2n+1]|Z[n]|}\right] \\
B_I &= -\frac{1}{C}
\end{aligned} \tag{3.31}$$

This chapter also showed that deadtime caused a phase shift offset effect in the DAB converter, which significantly affected the converter operating point and system dynamics. Since this effect strongly depends on the AC inductor current, a closed-form, piecewise linear expression for this waveform was derived, allowing a deadtime compensation algorithm to be designed to accurately predict the phase shift offset at all operating points. The operating point for the harmonic model was then updated with the predicted phase shift error to ensure a good match across the entire operating range.

Chapter 4

Closed Loop Control

To achieve high performance regulation of the DAB bi-directional DC-DC converter, the system output voltage must maintain good tracking of its reference command, despite transient events and varying operating conditions. Previous regulators that have been applied to this converter structure have three main limitations. In general, they do not guarantee maximised performance. Secondly, they do not give a consistent level of response across the entire operating range. Lastly, they do not ensure a comparable response for changes in reference command and load condition.

This chapter focuses on the design and optimisation of a new closed loop feedback controller that will resolve the issues identified in the current literature. Classic control theory states that to maximise closed loop performance, plant dynamics must be considered during controller design [99]. As such, the dynamic model of the DAB converter derived in the previous chapter will be employed to help design the new closed loop regulator. The model is first used to determine the most appropriate controller structure for the DAB converter, and its intrinsic performance limits identified. Based on these limits, techniques for maximising the closed loop regulator performance for transient changes in reference command as well as load changes are presented. Finally, the proposed control strategy is implemented and tested on the simulated DAB converter.

4.1 Choice of Feedback Controller

The controller form chosen to regulate the DAB converter must give good tracking of the reference command with no steady-state error, as well as achieve a fast transient response.

The DAB converter is to be used in a Smart Grid application, so the load seen is likely to be a DC resistance, or an AC inverter. Both these situations are best managed by output voltage regulation, so the new control strategy presented here targets the DAB converter output voltage.

A classic single-loop controller is deemed appropriate for this application because the DAB converter has one output state (V_{out}), and only one controllable input – the phase shift δ. The load current input (i_{load}) is defined as a disturbance input because it describes the load condition of the system, and thus cannot be controlled directly. The effect of this disturbance will be addressed later in this chapter. Fig. 4.1 shows the block diagram of this control structure. In this control system, regulator ($H(s)$) is used to vary the plant input (δ) such that the DC output voltage (V_{out}) tracks the reference (V_{ref}) [99].

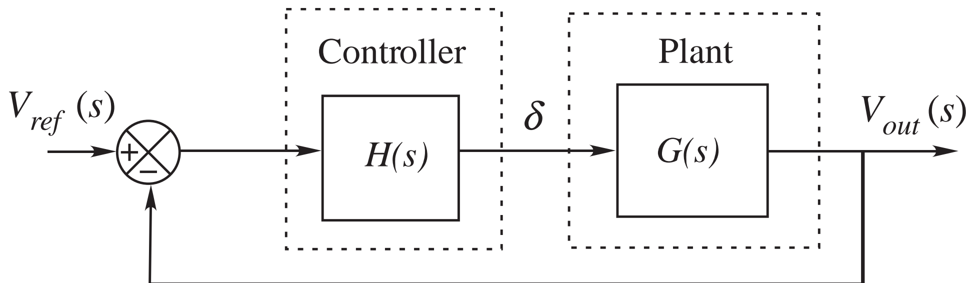


Figure 4.1: Basic closed loop block diagram of the DAB converter.

Classical control theory suggests that in order to maximise performance, the forward path transfer function of Fig. 4.1 should meet the following criteria [99]:

- **High Gain at DC** – To minimise steady-state error.
- **High Crossover Frequency** – To provide a fast transient response.

Since plant dynamics strongly affect this decision, the state-space dynamic model derived in the previous chapter is regenerated here for convenience:

$$
\begin{aligned}
\frac{d\Delta V_{out}(t)}{dt} &= A\Delta V_{out} + B_{\delta}\Delta\delta + +B_I\Delta i_{load} \\
\text{where} \quad A &= \left\{ \frac{-8}{C\pi^2}\left(\frac{N_p}{N_s}\right)^2 \sum_{n=0}^{N}\left[\frac{\cos\left(\varphi_z[n]\right)}{[2n+1]^2\left|Z[n]\right|}\right] \right\} \\
\text{and} \quad B_{\delta} &= \frac{8V_{in}}{C\pi^2}\frac{N_p}{N_s}\sum_{n=0}^{N}\left[\frac{\sin\left(\varphi_z[n]-[2n+1]\,\delta_o\right)}{[2n+1]\left|Z[n]\right|}\right] \\
\text{and} \quad B_I &= -\frac{1}{C}
\end{aligned}
\tag{4.1}
$$

This linearised model is first order in nature, with two inputs ($\Delta\delta$ & Δi_{load}) and one output (ΔV_{out}).

Since the plant model is first-order in nature and the regulator needs to regulate a DC quantity (DAB converter output voltage), a Proportional + Integral (PI) structure should be sufficient to achieve high performance output voltage regulation. The transfer function of a PI controller is given as [99]:

$$
H(s) = K_p\left(1 + \frac{1}{sT_r}\right) \tag{4.2}
$$

where the controller gains are given by K_p (proportional gain) and T_r (integrator time constant).

To justify the choice of such a simple controller, the forward path of the PI-regulated closed loop system is derived below (eq. 4.3), with its Bode plot presented in Fig. 4.2:

$$
\begin{aligned}
F(s) = H(s)G(s) &= K_p\left(1 + \frac{1}{sT_r}\right)\frac{B_{\delta}T_p}{1+sT_p} \\
&= \frac{K_pB_{\delta}}{s}\frac{T_p}{T_r}\left(\frac{1+sT_r}{1+sT_p}\right)
\end{aligned}
\tag{4.3}
$$

where $T_p = \frac{-1}{A}$ and describes the plant time constant.

The PI controller gives the forward path a pole at the origin, as seen in eq. 4.3. This makes the forward path gain asymptote to infinity as the system frequency approaches DC ($\omega \rightarrow 0$, see Fig. 4.2). This large gain eliminates steady state error, ensuring good tracking of the DC reference.

Since the forward path transfer function (eq. 4.3) contains two poles and one zero, the phase response of the forward path transfer function asymptotes to $-90°$, confirmed in Fig. 4.2. This system therefore has infinite phase margin, i.e. it is unconditionally stable, regardless of controller gains. There is therefore no theoretical

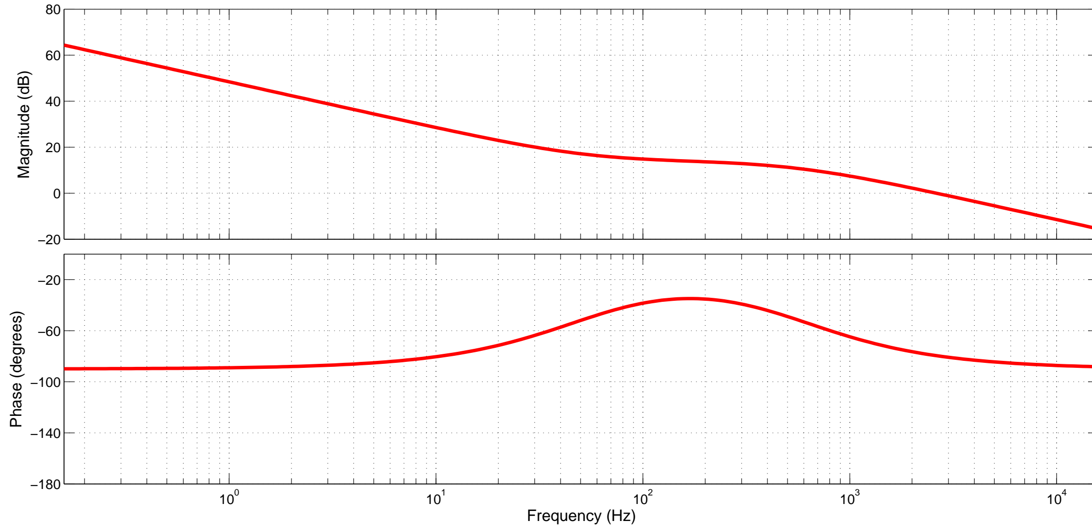


Figure 4.2: Ideal forward path Bode Plot of the Closed Loop DAB Converter

limit on controller gains, so a very high controller bandwidth and a very fast transient response can be achieved.

Unfortunately, this analysis applies only to an ideal implementation, not a practical one. It is essential to consider the implications of a practical implementation when designing modern controllers, so that realistic controller performance limits can be identified.

Modern closed loop controllers for power electronic converters are implemented digitally using powerful microprocessors (e.g. a Digital Signals Processor (DSP)). These devices are capable of managing all converter modulation, control, protection and supervisory functions in a single package, making them very attractive for modern converter implementations. However, using these digital processors means that the effects of a digital implementation on regulator performance must be considered. The vital difference between an ideal controller implementation and a digital implementation one is that digital systems include a *transport delay* effect that degrades closed loop performance [99, 100, 122, 129].

This degradation in performance is demonstrated in Fig. 4.3. This figure compares the transient response of the ideal linearised closed loop system to the digitally implemented switched simulation (with identical controller gains) to a step change in reference voltage. The performance of the digitally implemented controller is clearly poorer than that of the ideal implementation. To precisely determine the maximum achievable performance, i.e. the limits of this control architecture, the transport delay mechanism that limits performance must be understood and its effect precisely quantified. This is the focus of the following section.

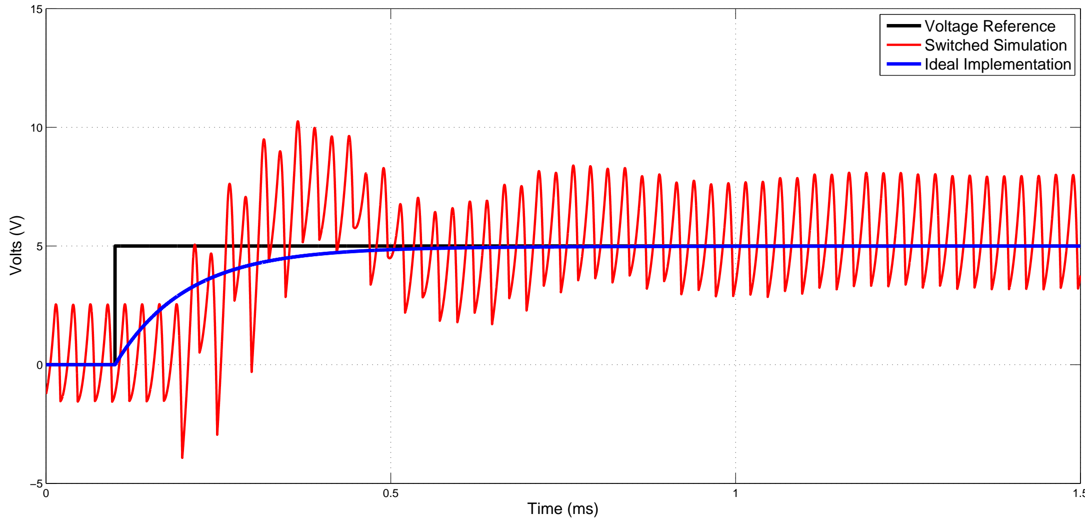


Figure 4.3: Transient Responses of Ideal & Digital implemented PI regulator. ($K_p = 4e^{-2}$, $T_r = 3e^{-4}$)

4.2 The digital modulator/PI controller & its performance limitations

The previous section has shown that a digitally implemented PI controller has an intrinsic performance limit due to the delays inherent to the digital control and modulation processes. To precisely identify this limit, this section first describes the digital controller and modulator to determine the delays associated with them. The effect of these delays is then quantified, which allows the performance limits of the digitally implemented closed loop system to be established.

4.2.1 The Digital Modulator

The digital modulator produces the turn on & turn off signals for the switching devices in the DAB converter, and is made up of a high frequency carrier wave (triangular in this case) and a modulation reference signal, as illustrated in Fig. 4.4. The gate signals are generated by *toggling* the modulator output as the carrier signal crosses the modulation reference.

This modulation reference is generated by the PI controller, and is updated every half carrier cycle - at the peak and the trough of the carrier wave. This ensures that only one switching transition occurs in each half cycle, preventing multiple switching. Multiple switching is a highly undesirable effect that occurs when the reference crosses the carrier multiple times during a single switching period, resulting in multiple undesired transitions. This can cause closed loop instability, or worse, catastrophic converter failure.

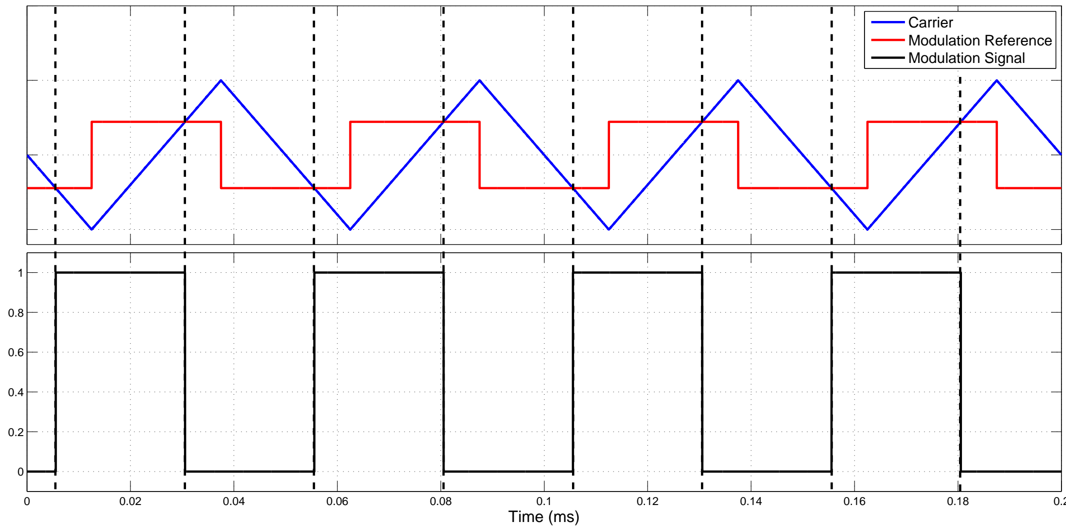


Figure 4.4: Digital PSSW Modulator

4.2.2 The Digital PI Controller

All digital control systems must interface between the continuous time domain (the plant) and the discrete time domain (the digital controller). To do this, an Analog-to-Digital Converter (ADC) is used to *sample* the continuous time plant, generating a discrete time model of its behaviour. In order to achieve high performance control, it is sensible to ensure that the sampling technique employed accurately represents the continuous time plant. The most common sampling method is a *sample-and-hold* technique[1], which freezes the sampled value until the next sampling instant, as shown in Fig. 4.5.

The output voltage of the DAB converter has a ripple component as well as an average DC value. As it is the average DC value that must be controlled by the closed-loop regulator, it is important to ensure that only this value is fed to the controller. This will prevent oscillations in the control signal caused by the DC output voltage ripple.

Synchronous sampling achieves this by timing the sampling instant such that the voltage signal is sampled at the same point of the waveform each time. This results in ripple-free voltage measurement (see Fig. 4.5).

Having developed a sampled, ripple-free representation of the DC output voltage waveform, the closed loop controller calculations are then performed based on the measured data. The control signal output from the PI regulator then becomes the reference command for the digital modulator, generating the switching pulses for the DAB converter.

[1] Also known as a Zero Order Hold (ZOH).

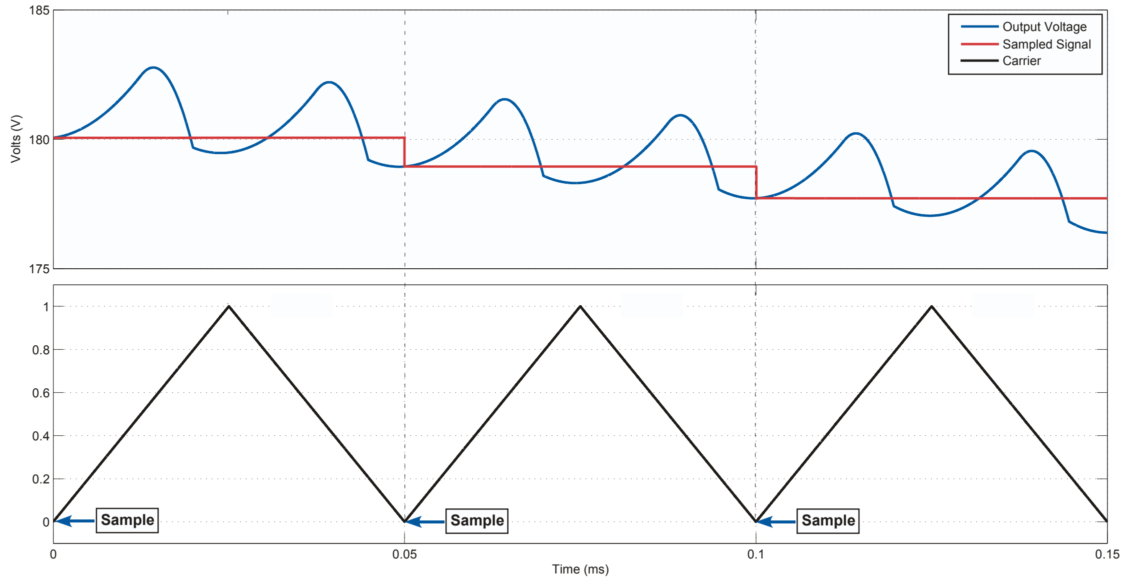


Figure 4.5: Sample & Hold

4.3 Delays in the Digital Implementation

Having analysed the digital implementation of the PI controller and the PSSW modulator, two primary delay mechanisms inherent to the design are immediately obvious, i.e.:

- **Sampling Delay**
  When the system is sampled with a ZOH, digital control theory states that this introduces a half sample period delay. This is because the average of the sampled system will lag that of the actual system by half a sample period [100]. Since the system is sampled at the carrier rate (Symmetric Sampling), this half sample period delay equates to half the switching period ($\frac{T_s}{2}$).

- **Computational Delay**
  Calculations in a microcontroller take a finite, non-zero period of time. Since the modulation reference is only updated once every carrier period, the new modulation reference generated by the PI controller after each sample only propagates to the modulator a half-carrier period later. This is illustrated in Fig. 4.6, and introduces a half-carrier period delay ($\frac{T_s}{2}$).

In total, this gives a **one carrier period transport delay** (T_d) through the digital modulator/regulator structure.

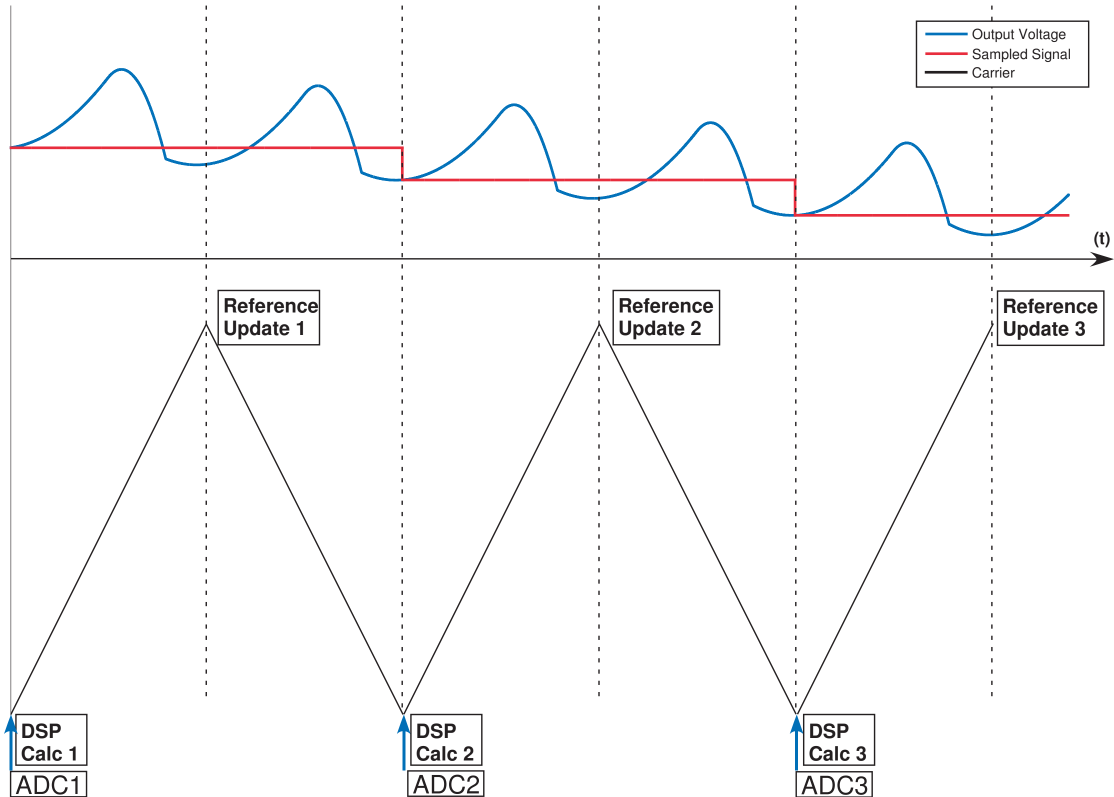


Figure 4.6: Controller Calculation & Update

4.3.1 The Effect of Transport Delay

Having identified the transport delay effect, it can now be included in the forward path transfer function as a unity gain delay function (e^{sT_d}) [99, 129]:

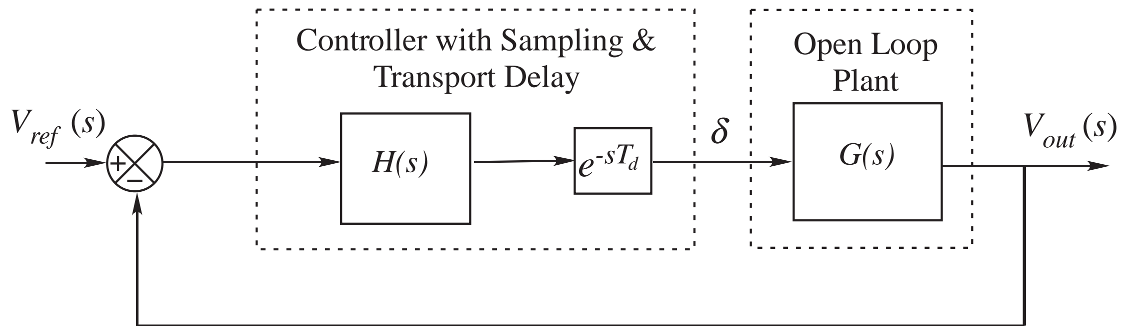


Figure 4.7: Closed loop block diagram - Including Transport Delay.

The Bode plot of this updated forward path is shown in Fig. 4.8, where transport delay causes the system phase to roll-off towards negative infinity as the frequency increases. The effect of this is that the system no longer has an infinite phase margin (see Section 4.1), and the closed loop system is no longer unconditionally stable. Unlike the ideal system, the phase margin now reduces as the gains are increased. Classic control theory states that this reduction in phase margin results in a more oscillatory closed-loop response (In fact, a negative phase margin signifies instability) [99].

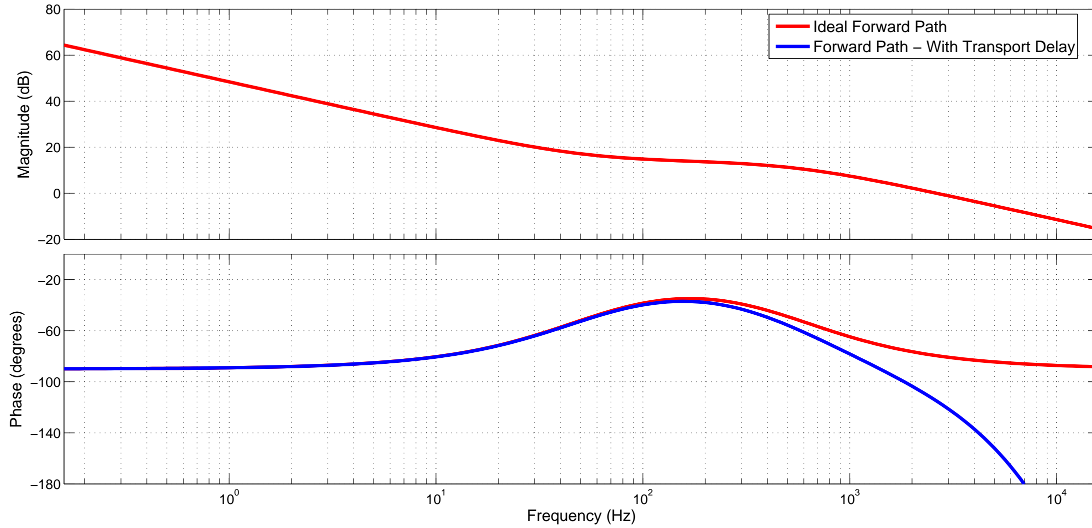


Figure 4.8: Forward Path Bode Plot - Including Transport Delay.

The effect of transport delay is now verified in simulation, and the results plotted in Fig. 4.9. This figure shows that the linearised closed loop system now successfully matches the prediction of the switched simulations, after the effects of transport delay have been accounted for.

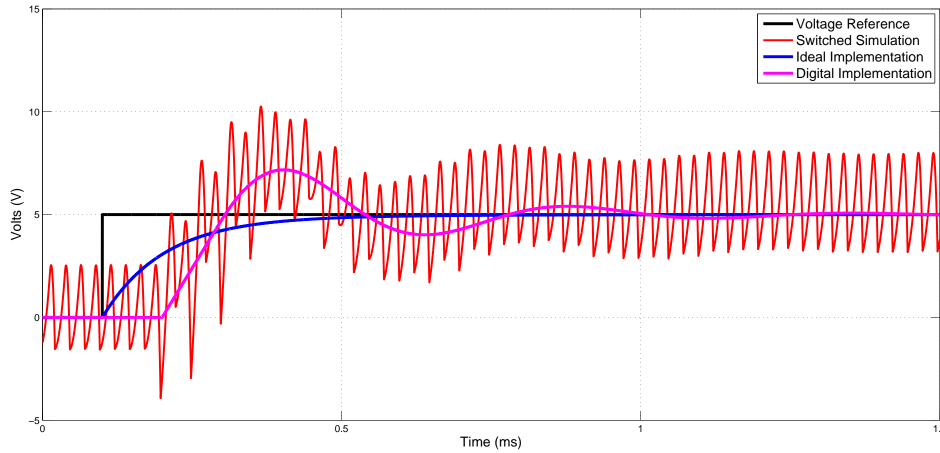


Figure 4.9: Linearised Transient Responses.

4.4 Optimising PI controller gains

The previous section identified that in digitally implemented DAB converters, transport delay is the primary mechanism that limits closed loop performance. Transport delay is a deterministic process, i.e. its duration is well known due to the regular and timely nature of the digital control & PWM processes (e.g. fixed sample & update rates). This section now calculates the maximum achievable PI controller gains while also accounting for this delay.

Classic control theory suggests that a high controller bandwidth is desirable to maximise performance [99]. Controller bandwidth is defined as the frequency at which the forward path transfer function has unity gain (ω_c). The transient performance achieved by the controller (in terms of rise time, settling time, overshoot, etc.) is governed by the available phase margin (φ_m) at this crossover frequency. In general, large phase margins give less oscillatory responses but slower rise times, while smaller phase margins give faster rise times at the cost of a more oscillatory response [99].

The controller design process therefore aims to maximise controller bandwidth while still achieving a phase margin that provides good performance. To aid the description of the controller design process, the forward path transfer function is restated here:

$$\begin{aligned} F(s) = H(s)G(s) &= K_p\left(1+\frac{1}{sT_r}\right)e^{sT_d}\frac{B_\delta T_p}{1+sT_p} \\ &= \frac{K_p B_\delta}{s}\frac{T_p}{T_r}\left(\frac{1+sT_r}{1+sT_p}\right)e^{sT_d} \end{aligned} \tag{4.4}$$

To calculate the maximum bandwidth (ω_c), it is recognised that the phase of the system at this frequency must be equal to the desired phase margin (φ_m). Therefore, the phase component of eq. 4.4 is derived below and solved for ω_c:

$$\begin{aligned} \angle F(j\omega_c) &= \angle\left(\frac{1+j\omega_c T_r}{j\omega_c Tr}\exp^{j\omega_c T_d}\frac{1}{1+j\omega_c T_p}\right) \\ &= -\pi+\varphi_m \end{aligned} \tag{4.5}$$

which can be restated as:

$$-\pi+\varphi_m = tan^{-1}(\omega_c T_r) - \frac{\pi}{2} - \omega_c T_d - tan^{-1}(\omega_c T_p) \tag{4.6}$$

This equation is further simplified by recognising that ω_c is invariably much higher than the frequency of the plant pole (i.e. $\omega_c \gg \frac{1}{T_p}$). This makes the angular contribution of the plant pole ($tan^{-1}(\omega_c T_p)$) approximately equal to $\frac{\pi}{2}$, further simplifying eq. 4.6 to:

$$\varphi_m = tan^{-1}(\omega_c T_r) - \omega_c T_d \tag{4.7}$$

From this equation, it can be seen that the maximum value of ω_c is achieved when the phase contribution of the integrator is maximised ($tan^{-1}(\omega_c T_r) \approx \frac{\pi}{2}$). To achieve

this phase contribution while still maximising integrator gain, the integrator time constant must be set approximately a decade below ω_c [129], i.e.:

$$T_r = \frac{10}{\omega_c} \tag{4.8}$$

This allows eq. 4.7 to be solved for ω_c in terms of the transport delay (T_d) and the desired phase margin (φ_m), giving:

$$\omega_c = \frac{\frac{\pi}{2} - \phi_m}{T_d} \tag{4.9}$$

The proportional gain K_p that gives the desired phase margin φ_m at this crossover frequency (i.e. the maximum K_p) can now be calculated by determining the value of K_p for which the magnitude of the forward path transfer function (eq. 4.4) is unity at the crossover frequency, ω_c [17, 18, 122, 129]. This gives:

$$\begin{aligned} 1 = |G(j\omega_c)| &= \left| \frac{K_p B_\delta}{j\omega_c} \frac{T_p}{T_r} \left(\frac{1 + j\omega_c T_r}{1 + j\omega_c T_p} \right) \exp^{-j\omega_c T_d} \right| \\ &= \frac{K_p B_\delta}{\omega_c} \\ \therefore K_p &= \frac{\omega_c}{B_\delta} \end{aligned} \tag{4.10}$$

The proportional gain is therefore heavily dependent upon the B_δ term from the state-space dynamic model, whose formula was derived in the previous chapter is restated here for convenience:

$$B_\delta = \frac{8V_{in}}{C\pi^2} \frac{N_p}{N_s} \sum_{n=0}^{N} \left[\frac{\sin(\varphi_z[n] - [2n+1]\,\delta_o)}{[2n+1]\,|Z[n]|} \right] \tag{4.11}$$

The δ_o term in eq. 4.11 suggests that B_δ varies significantly with the phase shift operating point, illustrated in Fig. 4.10. This means that a proportional gain calculated to give optimised performance at the nominal phase shift will not give an equivalent level of performance across the entire operating range.

The effect of the varying plant characteristics is illustrated in simulation by plotting the transient responses of the closed-loop DAB converter with fixed PI gains to step changes in reference voltage at different operating conditions. The controller gains employed for this simulation are listed in Table 4.1, and correspond to a 40° phase margin at the the nominal operating point of 190V. This phase margin is chosen because classical control theory suggests that it will give a good trade-off between speed of response and damping (15% overshoot, 2 oscillations) [99].

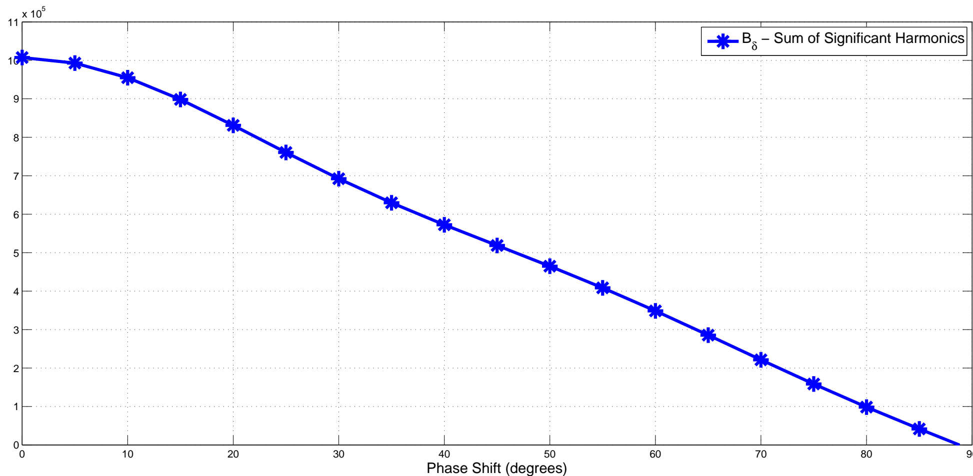


Figure 4.10: B_δ term variation with operating phase shift

Fig. 4.12 plots the resulting transient responses. The DAB converter output voltage waveform is synchronously sampled at the trough of its ripple, so it is the bottom of the voltage waveform that is regulated to track its reference. The upper trace in Fig. 4.12a shows that the desired phase margin is indeed achieved at this operating condition. However, at the 90V operating point, performance has degraded considerably, as a far more oscillatory response is seen.

The solution to this problem is to vary the proportional gain with operating phase shift, such that consistent performance is achieved across all operating conditions. Thus K_p is adaptively recalculated at every sample point as part of the control loop calculations. The closed loop block diagram for this system is shown in Fig. 4.11, where the applied phase shift is used to calculate the optimal gains for the current operating point. Since the controller gains are inversely proportional to the applied phase, the gain calculation system has negative feedback, which is stable.

The same transient steps of Fig. 4.12a are repeated with this new ***Adaptive PI controller***, and the results shown in Fig. 4.12b. Consistent performance is now achieved at all operating conditions.

Circuit Parameter		**Value**
Desired Phase Margin	(φ_m)	40°
Transport Delay Time	(T_d)	50 μs
Controller Bandwidth	(ω_c)	1667 Hz
Fixed PI Prop. Gain	(K_p)	0.0499
Maximum Adaptive Prop. Gain	$(K_{p_{Adapt_{max}}})$	0.0499
Minimum Adaptive Prop. Gain	$(K_{p_{Adapt_{min}}})$	0.247
Integrator Time Constant	(T_r)	6 ms

Table 4.1: DAB Converter PI Controller Parameters

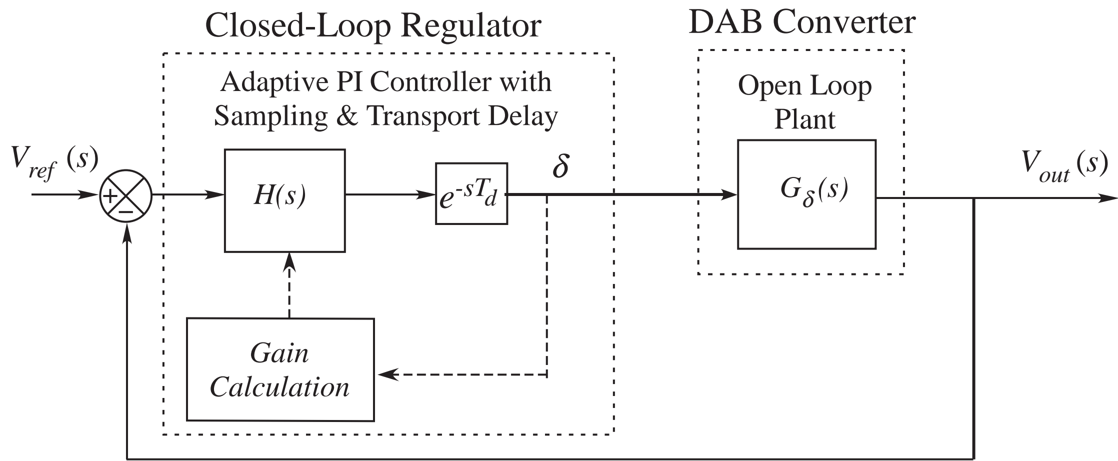
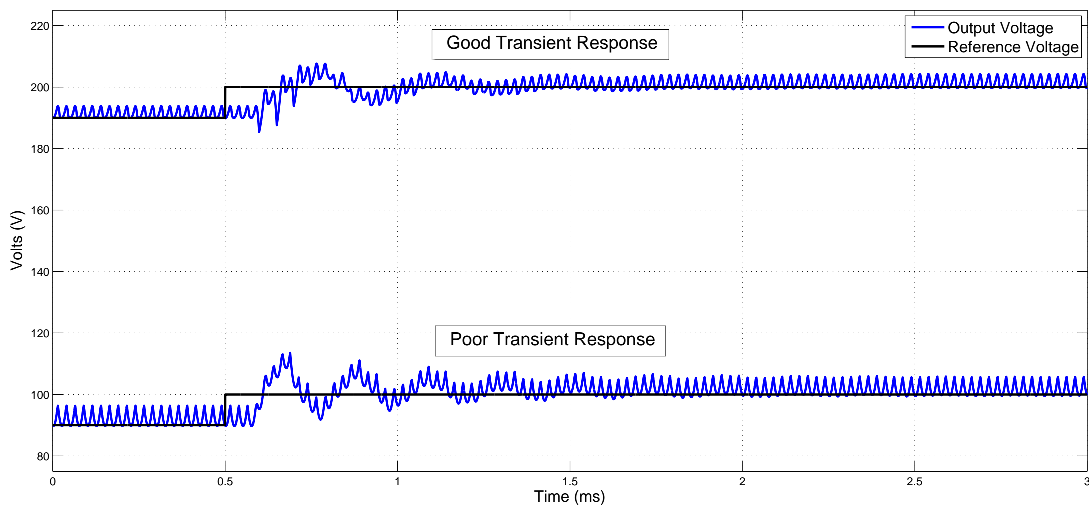
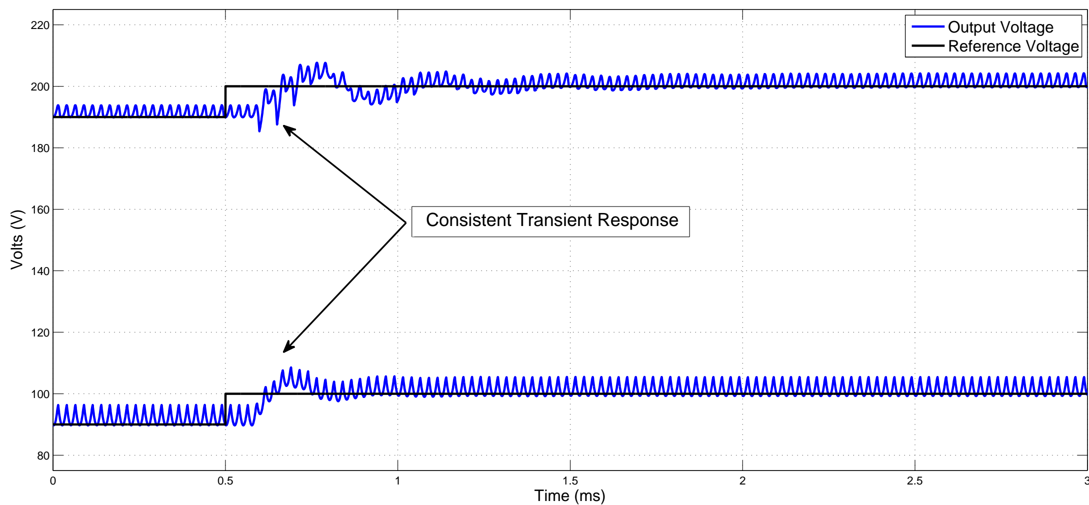


Figure 4.11: Closed Loop Block diagram of the DAB converter with an Adaptive PI controller



(a) Fixed PI Controller Gains



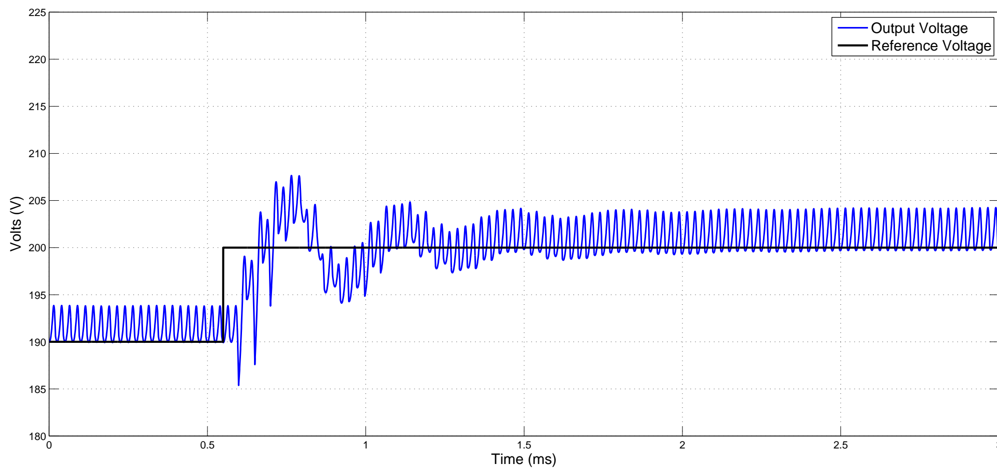
(b) Adaptive PI Controller Gains

Figure 4.12: Closed loop Step Response Comparison

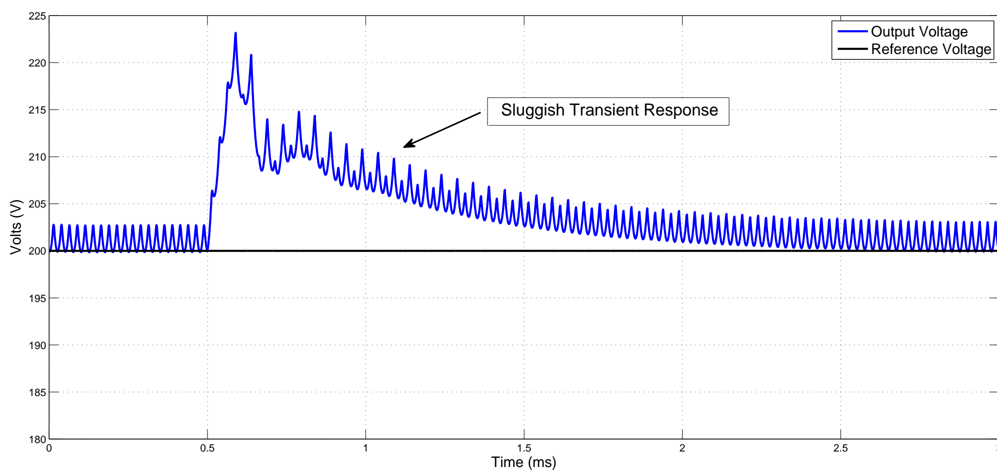
4.5 Load Step Performance

Since DAB converters commonly face changing load conditions, the closed loop controller must provide good load transient regulation. In fact, a high performance controller should provide equivalent performance for both reference and load transients.

However, this is generally not the case for the DAB converter. Fig. 4.13 compares the transient responses of the closed loop voltage regulated DAB converter to a reference and a load transient. Although the closed loop regulator is maximally tuned based on the ideas presented in the previous section, the load transient is clearly sluggish compared to its reference step counterpart.



(a) Reference Transient



(b) Load Transient

Figure 4.13: Comparison of Load & Reference Transient Responses

This section investigates the reasons behind this suboptimal load transient response and presents a solution, which is verified in simulation.

4.5.1 Exploring the load transient

The cause of this poor load transient is best understood by re-examining the harmonic model, restated here for convenience:

$$\begin{aligned} \frac{d\Delta V_{out}\left(t\right)}{dt} &= A\Delta V_{out} + B_{\delta}\Delta\delta + B_{I}\Delta i_{load} \\ \text{where } A &= \frac{-8}{C\pi^2}\left(\frac{N_p}{N_s}\right)^2 \sum_{n=0}^{N}\left[\frac{\cos\left(\varphi_z\left[n\right]\right)}{\left[2n+1\right]^2\left|Z\left[n\right]\right|}\right], \\ B_{\delta} &= \frac{8V_{in}}{C\pi^2}\frac{N_p}{N_s}\sum_{n=0}^{N}\left[\frac{\sin\left(\varphi_z\left[n\right]-\left[2n+1\right]\delta_o\right)}{\left[2n+1\right]\left|Z\left[n\right]\right|}\right] \\ \text{and } B_I &= -\frac{1}{C} \end{aligned} \tag{4.12}$$

The model can be separated into two parts, i.e. a harmonic summation term that defines the current injected into the output capacitor, and a load current term, drawn from the output capacitor. The load current variable can therefore be extracted as a disturbance term, resulting in a two-input, single-output (MISO) system. The block diagram of the system is presented in Fig. 4.14 [99], and the transfer functions that relate each input to the output are:

$$G_{\delta}(s) = \frac{T_p B_{\delta}}{1+sT_p} \tag{4.13a}$$

$$G_{I}(s) = \frac{T_p B_{I}}{1+sT_p} \tag{4.13b}$$

To explore the reason for the poor load step response, it is instructive to derive the transfer functions that relate each input (V_{ref} & I_{load}) to the output voltage (V_{out}), as follows:

$$\begin{aligned} \left.\frac{\Delta V_{out}}{\Delta V_{ref}}\right|_{\Delta I_{load}=0} &= \frac{H(s)G_{\delta}(s)}{1+H(s)G_{\delta}(s)} \\ &= \frac{K_p e^{sT_d} T_p B_{\delta}(1+sT_r)}{sT_r(1+sT_p)+K_p e^{sT_d} T_p B_{\delta}(1+sT_r)} \end{aligned} \tag{4.14a}$$

$$\begin{aligned} \left.\frac{\Delta V_{out}}{\Delta I_{load}}\right|_{\Delta V_{ref}=0} &= \frac{G_I(s)}{1+H(s)G_{\delta}(s)} \\ &= \frac{-sT_r T_p/C}{sT_r(1+sT_p)+K_p e^{sT_d} T_p B_{\delta}(1+sT_r)} \end{aligned} \tag{4.14b}$$

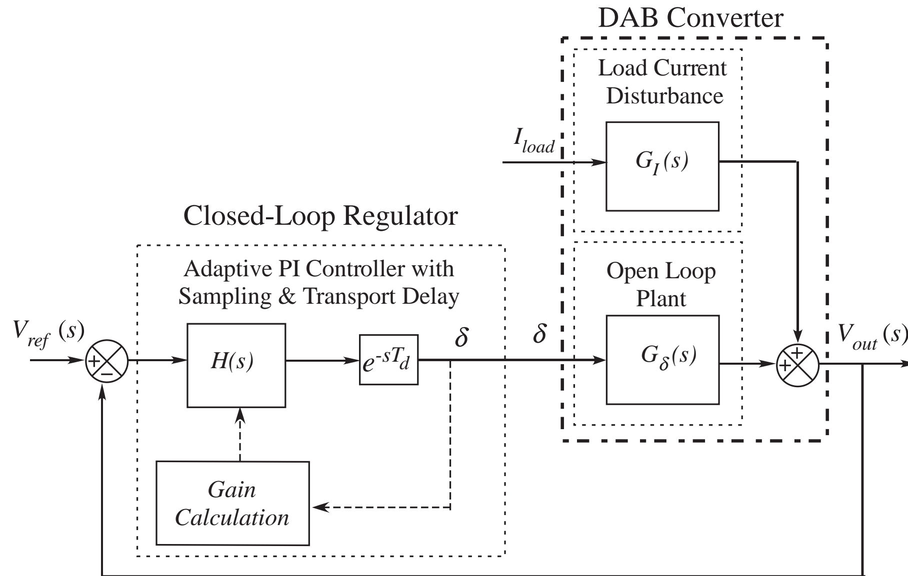


Figure 4.14: DAB Closed Loop Block Diagram - with Load Current Disturbance

The pole zero maps of these two functions are shown in Fig. 4.15, which helps identify the cause of the poor load transient performances. The response of the voltage reference transfer function (eq. 4.14(a), Fig. 4.15(a)) is dominated by the high frequency pole (≈ -16krad/s), because the low frequency pole is largely cancelled out by the nearby low frequency zero (≈ -2krad/s). However, this zero does not exist in the load change transfer function (eq. 4.14b, Fig. 4.15b), so the overall response is dominated by the slower low frequency pole, causing the slow load transient [16].

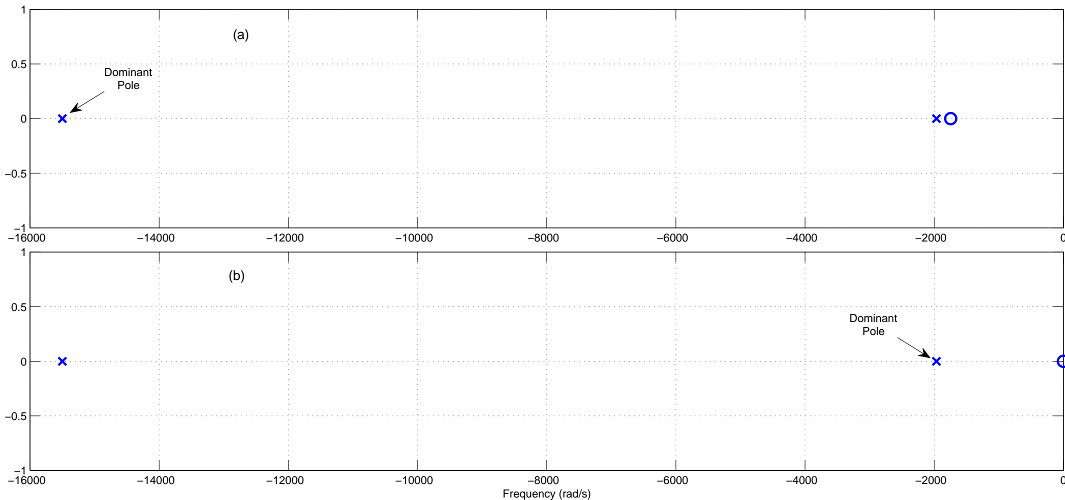


Figure 4.15: Pole Zero map of Closed Loop Transfer Functions

The traditional solution to a sluggish transient response is to increase controller gains, but this is impossible, since the controller gains have already been set to their maximum allowable values. An alternative solution is to compensate for the effect

of the load current disturbance. This technique is known as disturbance rejection [99, 129], and is the focus of the following section.

4.5.2 Disturbance Rejection

Classical control theory states that if a disturbance can be measured, its effect can be rejected by using feed-forward compensation [99]. This means that since the load current disturbance can be measured, a phase shift correction factor δ_{FF} can be used to adjust the DAB operating point to compensate for its effect as the load changes [16]. This concept is illustrated in the updated control block diagram presented in Fig. 4.16.

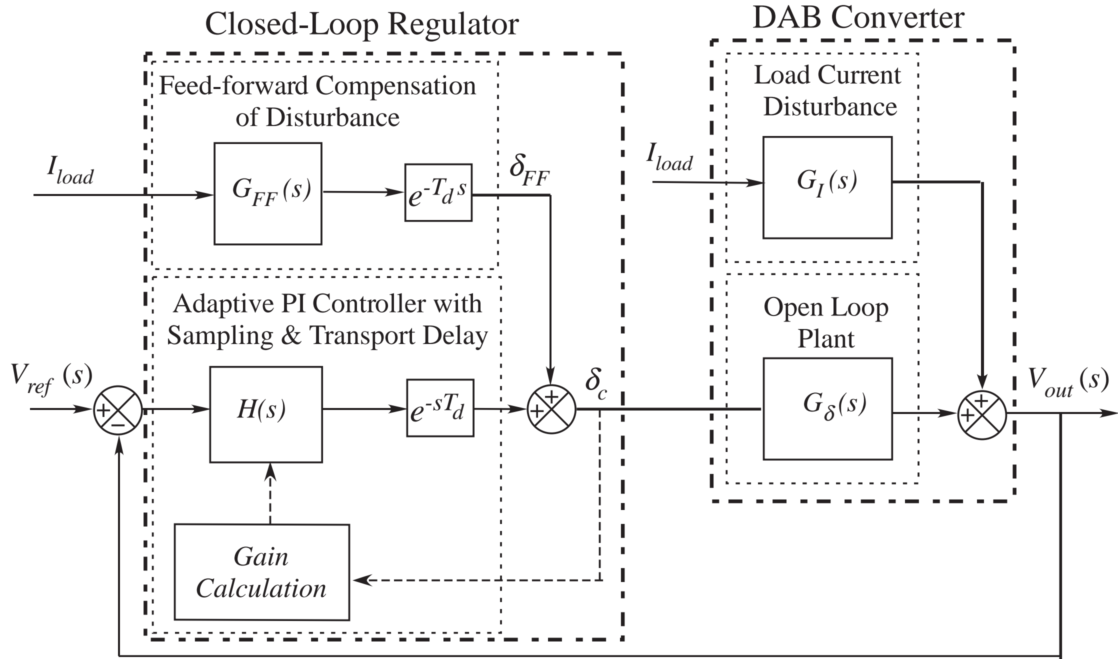


Figure 4.16: Closed Loop Block Diagram of the DAB Converter with Feed-forward Disturbance Rejection

This implies the need for a relationship between the load current and the commanded phase shift (δ). This relationship can be determined based on the steady-state DAB power transfer equations presented in eq. 3.19, restated here for convenience:

$$P = V_{out} I_{load} = \frac{8}{\pi^2} V_{in} V_{out} \frac{N_p}{N_s} \sum_{n=0}^{N} \left(\frac{1}{[2n+1]^3} \frac{sin\left([2n+1]\,\delta\right)}{\omega_s L} \right) \tag{4.15}$$

This expression can then be solved for I_{load} so the load current is known for any phase shift δ. To cope with variation in input voltage (V_{in}), which can strongly affect this calculation, the power expression is solved for $\frac{I_{load}}{V_{in}}$ as shown below:

$$\frac{I_{load}}{V_{in}} = \frac{8}{\pi^2}\frac{N_p}{N_s}\sum_{n=0}^{N}\left(\frac{1}{[2n+1]^3}\frac{sin\left([2n+1]\,\delta\right)}{\omega_s L}\right) \tag{4.16}$$

This equation is complex, so it is implemented as a pre-calculated lookup table that relates the measured $\frac{I_{load}}{V_{in}}$ to a feed-forward command δ_{FF}.

It is important to realise that the effect of deadtime must also be taken into account when attempting to reject the load current disturbance. This is because the phase shift distortion caused by deadtime can cause an error in the feed-forward phase shift, reducing the effectiveness of the disturbance rejection [16, 126].

The solution to this problem is simple. The phase shift error (δ_{db}) predicted by the deadtime compensation algorithm derived in Chapter 3 is simply summed with the feed-forward compensation signal to ensure the accuracy of the feed-forward command.

4.5.3 Improvement in Load Transient Performance

The final closed loop controller is an Adaptive PI controller that ensures maximum gain and consistent performance regardless of operating point, along with load current disturbance rejection and feed-forward deadtime phase shift error compensation. The block diagram of the final closed loop system is shown below:

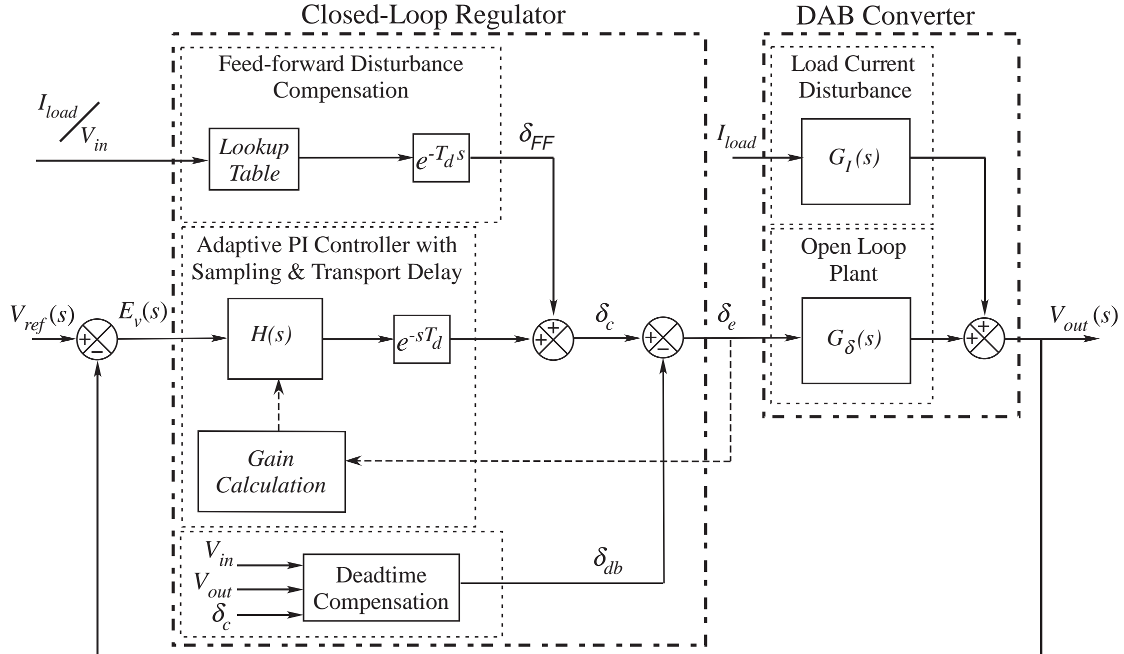
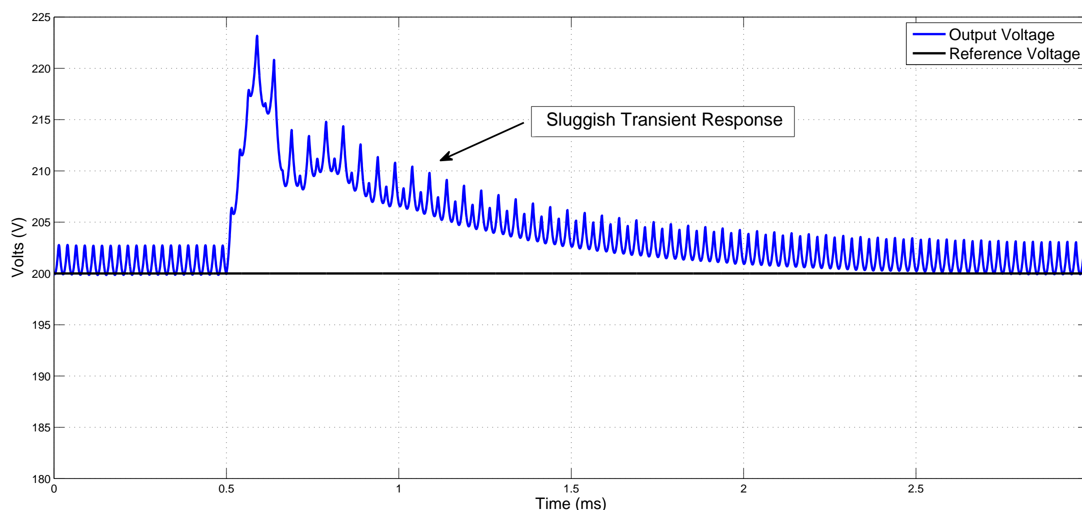
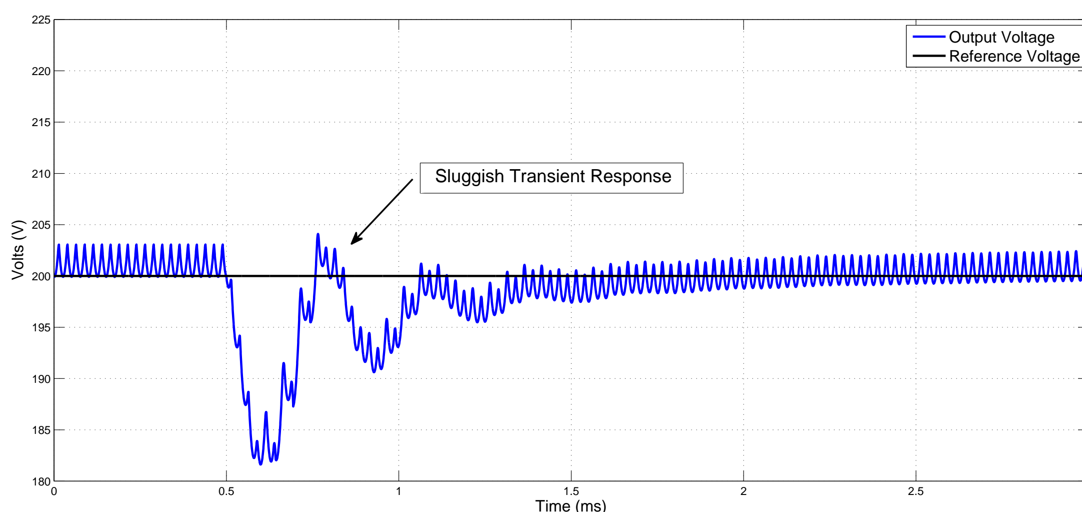


Figure 4.17: Final Closed Loop Block Diagram of the DAB Converter

This regulator was then implemented and tested in simulation, and the results plotted in Figs. 4.18 & 4.19. In both cases, the output voltage slews for approximately



(a) Decrease in Load



(b) Increase in Load

Figure 4.18: Load Step Response - Without Feed-forward

one switching cycle (20 kHz) before the controller takes effect. This is due to transport delay, as the regulator is unable to respond to a transient within this time period. As such, there is a minimum voltage deviation that occurs for a given transient, regardless of the closed-loop control technique.

Fig. 4.18 shows the load transient response for the Adaptive PI regulator without feed-forward. As predicted, the disturbance of the change in load current cannot be directly regulated by the controller, so the voltage returns to steady state slowly, with a clearly visible long 'tail'. A transient increase in DAB converter load condition also appears more oscillatory than a decrease. This occurs because as operating phase shift increases with load, a higher PI controller gain is applied. During this transient event, the dramatic increase in operating point and controller gains tend to cause a more oscillatory response.

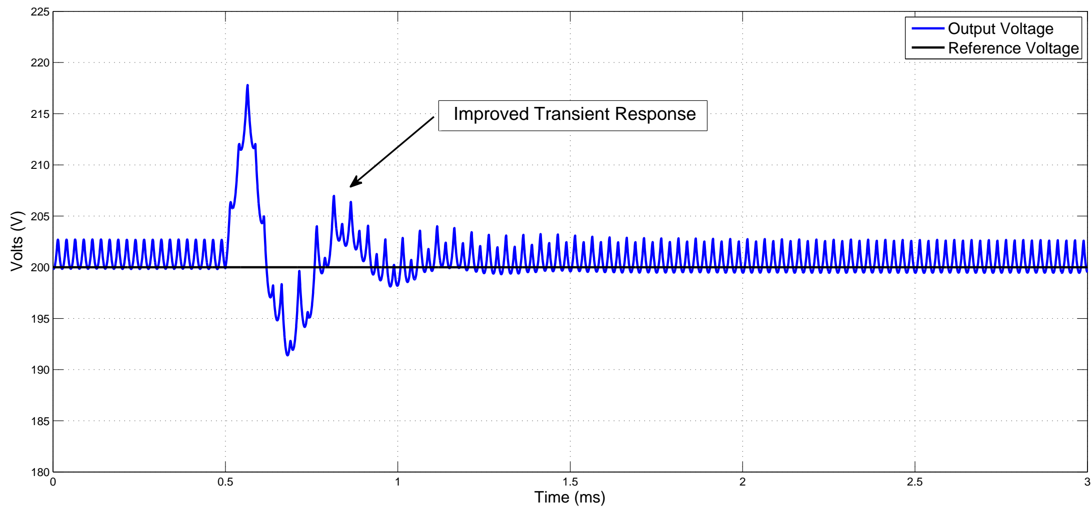
With feed-forward injection, the load current disturbance is compensated, presented in Fig. 4.19. The long 'tail' in the output voltage waveform is eliminated, and the controller now responds quickly to the change in load, significantly improving load transient performance. Additionally, the phase shift excursion during the transient too is smaller, so the variation in gain during the transient event is minimised. This results in a more consistent response for both an increase and a decrease in load condition.

4.6 Summary

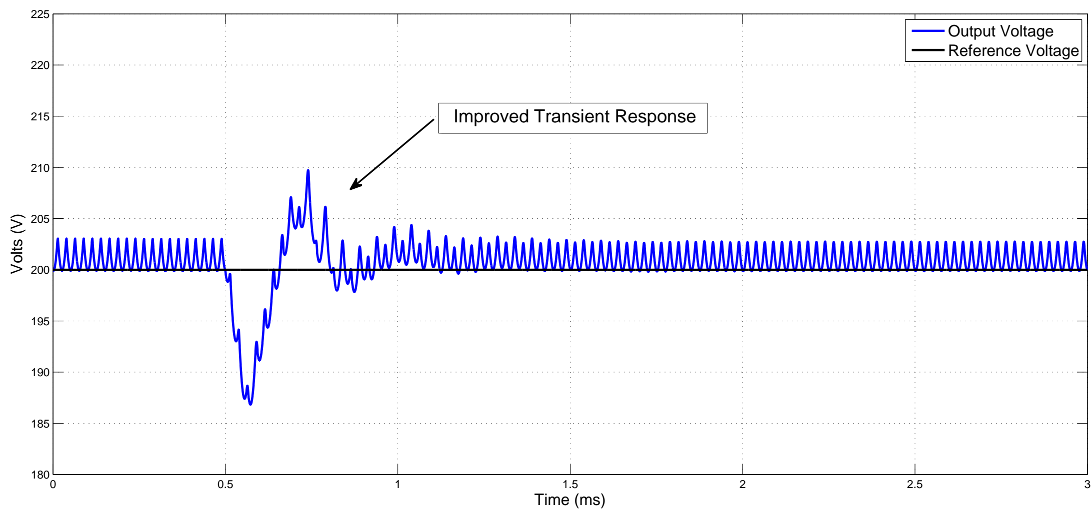
This chapter has presented the development of a new high performance closed loop regulator for the DAB converter.

Transport delay was identified as the primary factor that limits regulator gains in a digitally implemented controller. Accounting for the effect of this delay allowed the maximum achievable gains to be calculated, resulting in a fast transient response. To maintain the same level of performance across the converter operating range, the accurate dynamic model derived in Chapter 3 was used to develop a gain calculation algorithm that adapted controller gains as operating point varied to ensure consistent performance.

This chapter also identified that the DAB converter load current acts as a disturbance to the closed-loop system, degrading load transient performance. Feed-forward compensation has been proposed to reject the effect of this disturbance, so comparable performance for both reference command transients as well as changes in load condition is achieved.



(a) Decrease in Load



(b) Increase in Load

Figure 4.19: Load Step Response - With Feed-forward

Chapter 5

System Performance with an AC Load

Grid stability and performance is dependent on regulation of power flow (Chapter 1), high performance control algorithms for converters that interface to the Smart Grid are required. For applications that require bi-directional DC-DC power flow with galvanic isolation, this thesis has presented the DAB converter as the most appropriate topology at higher power levels (Chapter 2). Next, a high performance closed loop control architecture to regulate its output voltage (Chapter 4) has been developed based on the highly accurate dynamic model that was presented in Chapter 3.

Since energy in the Smart Grid is AC in nature, a DC-AC inverter must be connected to the DC output terminals of the DAB bi-directional DC-DC converter to form an isolated, bi-directional DC-AC converter. This chapter will first describe these converters in detail before identifying some of the challenges encountered with their design. It will also address the implications of the AC load seen by the DAB converter in this context. It will then show that most of these issues can be overcome by the high performance closed loop regulation algorithm presented in this thesis. The new control algorithms presented in this thesis are applied to this context to provide fast, precise power flow regulation for a Smart Grid appliance while also potentially overcoming some of these issues. These ideas are validated with detailed switched simulations, and the results of this investigation presented.

5.1 Challenges of Smart Grid Converter Design

An excellent topology choice for linking the AC Smart Grid to DC energy storage is a bi-directional AC-DC converter with galvanic isolation. The functional circuit diagram of this topology is shown in Fig. 5.1, and is a two-stage converter where the first stage is a single-phase, grid-connected Voltage Source Inverter (VSI) and the second stage is a DAB bi-directional DC-DC converter [17]. The DAB converter provides voltage level translation (if necessary) as well as high frequency galvanic isolation, while the VSI provides the connection to the AC grid. Both stages can handle bi-directional power flow – the VSI implicitly, and the DAB by design.

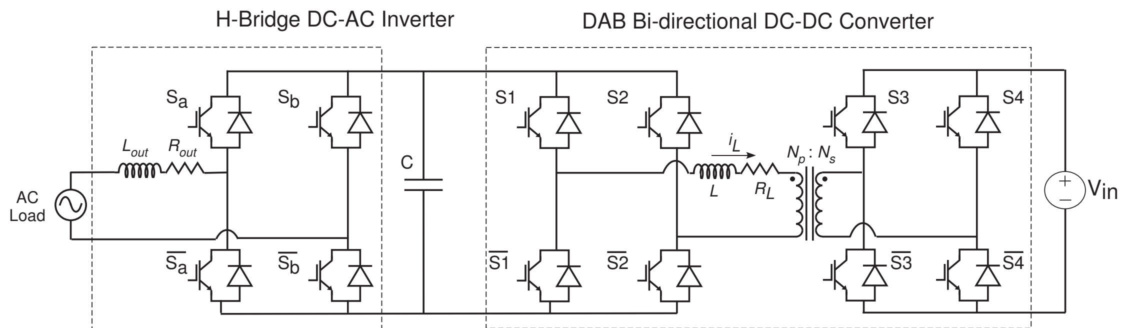


Figure 5.1: Topology of the two-stage isolated Bi-directional AC-DC converter

In order for stable operation, the instantaneous energy power flow between the energy storage elements and the Smart Grid must be matched by each converter stage. Mismatch in power flow will causes the intermediate DC bus to fluctuate, degrading overall performance. In the extreme case, this can lead to a loss of regulation and possibly even catastrophic converter failure. To avoid this scenario, the intermediate DC link capacitor provides energy storage to balance the instantaneous energy flow between the two converters, minimising DC bus voltage excursions [12, 17].

However, using the intermediate capacitor to absorb the mismatch in power flow tends to require a large capacitance. This usually implies that an electrolytic capacitor is needed, which is a severe limitation, as these devices have a limited lifetime. The electrolyte within these capacitors dries out with time, and they therefore need to be replaced every five years or so [130]. Hence there is a strong interest to reduce the required bus capacitance, so that more reliable alternatives such as film capacitors (which do not dry out [131]) can be used.

Since the required DC link capacitance is directly related to the mismatch in energy flow between the two converter stages, it is highly desirable to keep this mismatch to a minimum. This will help to reduce the required capacitance, potentially allowing the use of film capacitors. Accurately matching this power flow can be achieved in two ways – by employing converter control algorithms that can accommodate

the complex power flow dynamics of the system, and by maximising the closed loop dynamic performance of each converter stage in order to precisely control instantaneous energy flow.

This requires a detailed understanding of the energy flow through each converter stage, which will be explored in the following section. This understanding is then used to evaluate the feasibility of the proposed control architectures to minimise converter capacitance.

5.2 Converter Principles of Operation

In this section, the basic operating principles of the two converter stages (VSI & DAB) are reviewed, so that the flow of power through each stage can be understood.

5.2.1 Single-phase Voltage Source Inverter (VSI)

Modulation

The single-phase VSI shown in Fig. 5.2a below is almost invariably modulated with using sine-triangle PWM (Fig. 5.2b) because it gives the best quality output waveform (minimised Total Harmonic Distortion[1]) [12].

Power Flow

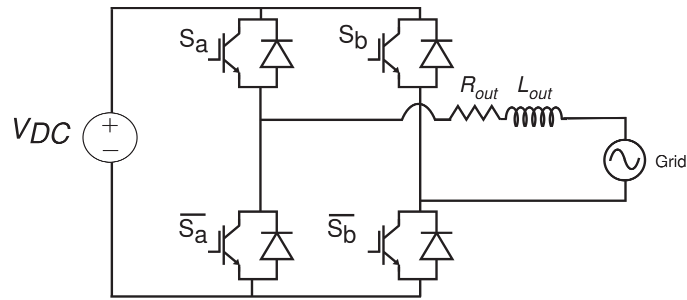
The averaged AC circuit model of the grid-connected VSI is shown in Fig. 5.3, where both the grid and the inverter are represented as sinusoidal voltage sources. This approximation is valid for the VSI because of the low THD produced by the chosen PWM modulation technique. The two sources are linked via an impedance L. V_g defines the peak grid voltage, while mV_{DC} defines the peak inverter AC output, where m is the modulation depth and V_{DC} is the inverter bus voltage.

The voltages in this system can be defined using phasor concepts as:

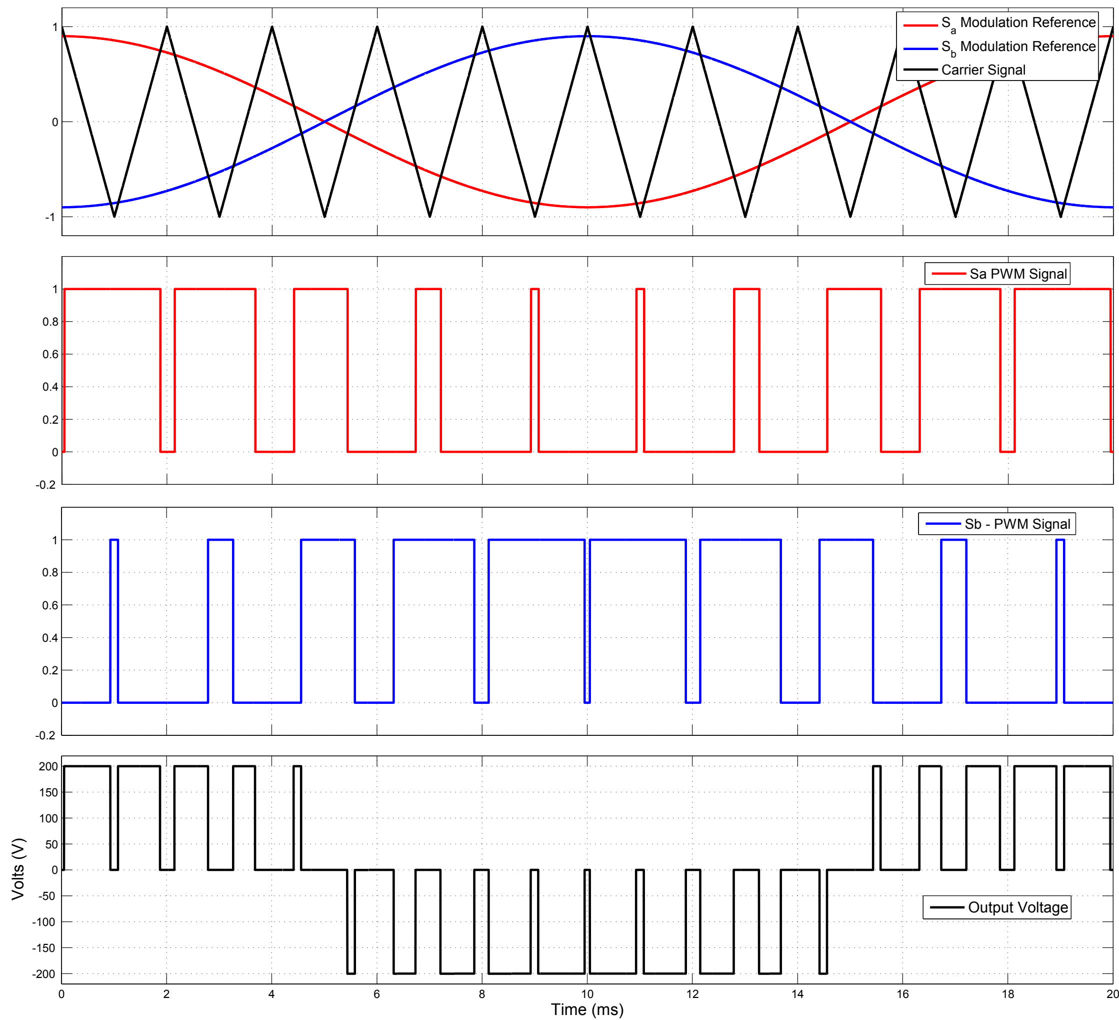
$$V_g \angle 0 = V_g \cos\{\omega_o t\} \tag{5.1a}$$

$$mV_{DC} \angle \varphi = mV_{DC} \cos\{\omega_o t + \varphi\} \tag{5.1b}$$

[1] Total Harmonic Distortion (THD) is a ratio of the energy in undesired harmonics of a waveform to the energy in its fundamental, and is used as a measure of waveform quality.



(a) Single-phase VSI Topology



(b) Sine-triangle Modulation

Figure 5.2: Topology & Modulation of a Single-phase VSI

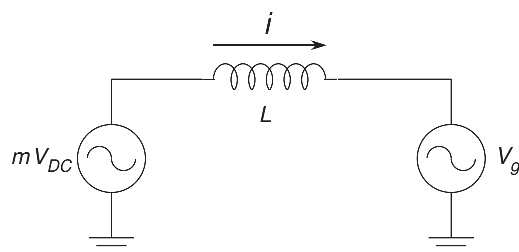


Figure 5.3: Fundamental equivalent circuit model of the grid-connected VSI

where ω_o is the fundamental frequency (50Hz in this case) expressed in rad/s and φ is the relative angle between the inverter and the grid. Assuming that the power factor angle of the AC impedance is ≈90°, as is the case when losses are small enough to be neglected, the current that flows between the two sources also be defined using phasor theory as:

$$\begin{aligned} i\left(t\right) &= \frac{mV_{DC}\angle\varphi - V_g\angle 0}{j\omega_o L} \\ &= \frac{mV_{DC}\sin\left\{\omega_o t + \varphi\right\} - V_g \sin\left\{\omega_o t\right\}}{\omega_o L} \\ &= \frac{V_g sin\varphi}{\omega_o L}\cos\left\{\omega_o t + \varphi\right\} \\ &\quad + \frac{\left[mV_{DC} - V_g\cos\varphi\right]}{\omega_o L}\sin\left\{\omega_o t + \varphi\right\} \end{aligned} \tag{5.2}$$

The flow of power from the VSI into the grid is therefore simply given as the product of the inverter AC voltage and the current, $i\left(t\right)$, i.e.:

$$\begin{aligned} P_{VSI}\left(t\right) &= mV_{DC}\cos\left\{\omega_o t + \varphi\right\} i\left(t\right) \\ &= mV_{DC}\cos\left\{\omega_o t + \varphi\right\}\left\{\begin{array}{l} \frac{V_g sin\varphi}{\omega_o L}\cos\left\{\omega_o t + \varphi\right\} \\ \quad + \frac{\left[mV_{DC} - V_g\cos\varphi\right]}{\omega_o L}\sin\left\{\omega_o t + \varphi\right\} \end{array}\right\} \\ &= \frac{mV_{DC}V_g sin\varphi}{\omega_o L}\cos^2\left\{\omega_o t + \varphi\right\} \\ &\quad + \frac{mV_{DC}\left[mV_{DC} - V_g\cos\varphi\right]}{\omega_o L}\sin\left\{\omega_o t + \varphi\right\}\cos\left\{\omega_o t + \varphi\right\} \\ &= \begin{array}{l} \frac{mV_{DC}V_g sin\varphi}{2\omega_o L} + \frac{mV_{DC}V_g sin\varphi}{2\omega_o L}\cos\left\{2(\omega_o t + \varphi)\right\} \\ \quad + \frac{mV_{DC}\left[mV_{DC} - V_g\cos\varphi\right]}{2\omega_o L}\sin\left\{2(\omega_o t + \varphi)\right\} \end{array} \end{aligned} \tag{5.3}$$

This equation shows that the power flow through the VSI has a constant average DC real power offset, as well as double fundamental frequency (100 Hz) oscillating real and reactive power terms, as shown in Fig. 5.4. This oscillating power flow is the cause of DC bus voltage fluctuation, and must either be absorbed by the DC bus capacitor, or by the second stage DC-DC converter by transferring the varying power flow directly to the battery without requiring intermediate energy storage.

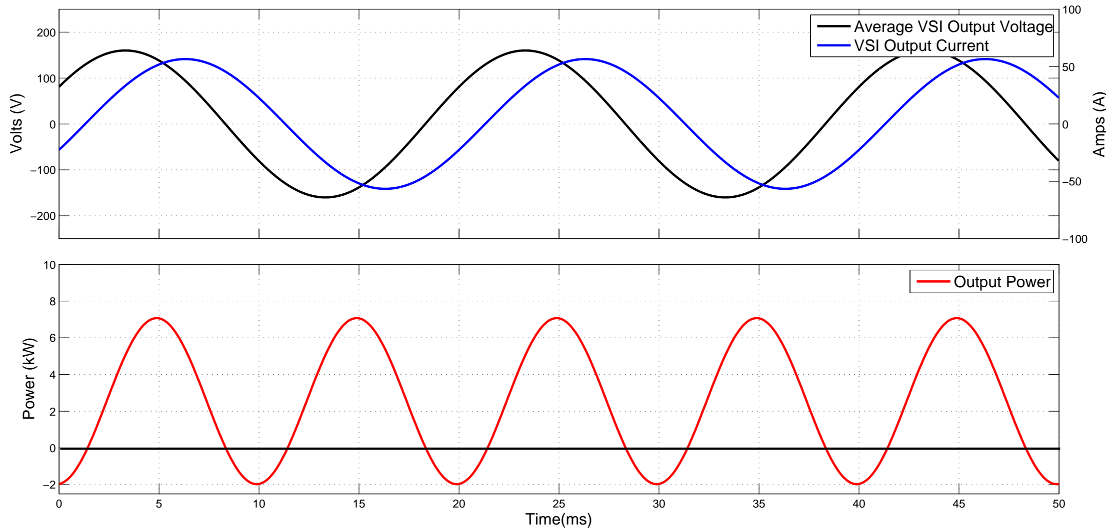


Figure 5.4: VSI fundamental average Voltage, Current & Power Flow
$[V_{DC} = 200$ V, $V_g = 100$ V , $L = 1$ mH, $m = 0.8$, $\varphi = 30°]$

5.2.2 DAB Bi-directional DC-DC Converter

The principles of operation that apply to the DAB converter have already been discussed in detail in previous chapters, so they will only be briefly outlined here. The chosen modulation scheme is a PSSW pattern to give the best dynamic performance, and the power flow dynamics of this scheme were derived in Chapter 3. The equation that governs the instantaneous steady-state power flow in this converter is restated here for convenience:

$$P_{DAB} = \frac{8}{\pi^2} V_{in} V_{DC} \frac{N_p}{N_s} \sum_{n=0}^{N} \left(\frac{1}{[2n+1]^3} \frac{\sin([2n+1]\,\delta)}{\omega_s L} \right) \tag{5.4}$$

In order for the DAB converter to match the power that flows between the grid and the VSI, it must transfer both an average real power component as well as an oscillating instantaneous power component. To achieve this, the input phase shift δ of the DAB converter must change rapidly. This requires a closed loop regulator, so the new high performance Adaptive PI controller described in Chapter 4 is applied, and its key features are restated briefly in the following Section.

5.3 Closed loop controller design

The challenge for the closed-loop control of this system is to transfer the desired average real power between the VSI and the grid while simultaneously ensuring that the DAB power flow matches the oscillatory instantaneous power flow seen by

the VSI. Achieving this will considerably reduce the required DC link capacitance because the DC link capacitor does not have to handle the large, low frequency oscillations in current caused by the oscillatory power flow. This leaves only the currents caused by the high frequency switching harmonics inherent to the PWM process, which are far smaller in magnitude, and so absorbing them requires far less capacitance.

This section first describes an overall control architecture that can achieve this target, and then presents controller design principles that ensure maximised performance.

5.3.1 Choice of controller architecture

The control architecture for this converter must control two variables simultaneously – the power flow through the system and the intermediate DC link voltage. Conventionally this is achieved by using the VSI to regulate the DC link voltage (i.e. as an Active Rectifier [10]), while power flow regulation is achieved by regulating the current through the DC-DC converter.

However, the performance of this architecture is limited. The current reference required by the DAB is complex, since it must include both the oscillating AC component as well as the average DC component. Also, a voltage-regulated VSI traditionally employs a dual-loop structure, with an inner current and outer voltage loop control structure. Typically the outer loop is designed to be ten times slower than the inner loop. This means that a large DC bus capacitance is required to maintain overall stability.

To avoid these complications and to achieve better closed-loop performance, an alternative control structure is proposed here, where the roles of the two controllers are reversed. The proposed strategy controls power flow by current-regulating the VSI, while the DC bus voltage is maintained by regulating the output voltage of the DAB DC-DC converter. The major advantage of this architecture is that the instantaneous power flow between the two stages is implicitly matched, given the assumption that the DC bus is held constant [17]. The AC current reference magnitude will be generated by an overarching system controller, based on criteria such as battery charge/discharge profiles, grid support needs, etc. [16--18].

To test and validate the closed loop regulation strategies employed under this new control architecture, a bi-directional AC-DC converter was designed in simulation, whose salient circuit parameters are shown in Table 5.1. The following section now describes these strategies in detail, looking to maximise overall system performance.

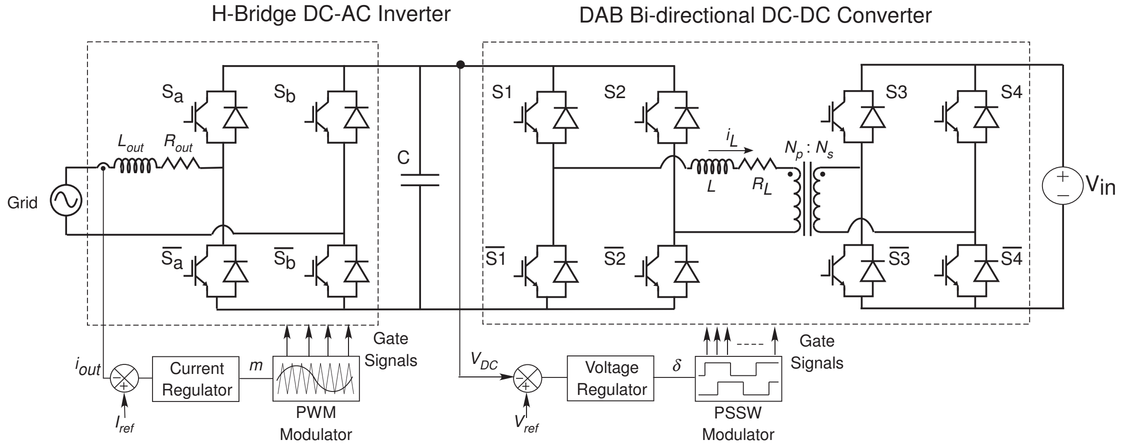


Figure 5.5: Proposed Closed Loop DC-AC Converter Architecture

5.3.2 VSI current regulator

The structure of the current-regulated VSI is shown in Fig. 5.6, which presents a single-phase VSI feeding an AC grid via a Resistive-Inductive $(R - L)$ impedance. The block diagram of the proposed closed loop control structure is shown in Fig. 5.7, where the inverter power stage is modelled as a linear amplifier of gain V_{DC}[2], and a PI controller is used to regulate the output current. This simple control structure can give excellent closed-loop performance, provided that the controller gains are high [129].

To maximise the gains of this digitally implemented PI controller, Holmes et al. [129] identifies that the effect of transport delay must be accounted for. The maximum achievable controller bandwidth, $\omega_{c_{VSI}}$, is therefore calculated for the desired phase margin (φ_m) as:

Circuit Parameter		Value
DC Input Voltage	(V_{in})	200 V
DC Output Voltage	(V_{out})	200 V
Peak AC Grid Voltage	(V_g)	100 V
Transformer Turns Ratio	$(N_{Pri} : N_{Sec})$	10 : 15
VSI Switching Frequency	(f_{VSI})	5 kHz
DAB Switching Frequency	(f_{DAB})	20 kHz
DC Capacitance	(C)	20 μF
AC Inductor Inductance	(L)	50 μH
AC Inductor Resistance	(R_L)	0.1 Ω
Output Inductance	(L_{out})	5 mH
Output Load Resistance	(R_{out})	0.5 Ω
Nominal Output Power	(P_{out})	3 kW

Table 5.1: DC-AC Converter Parameters

[2] This assumption is valid as long as the modulator operates in the linear region.

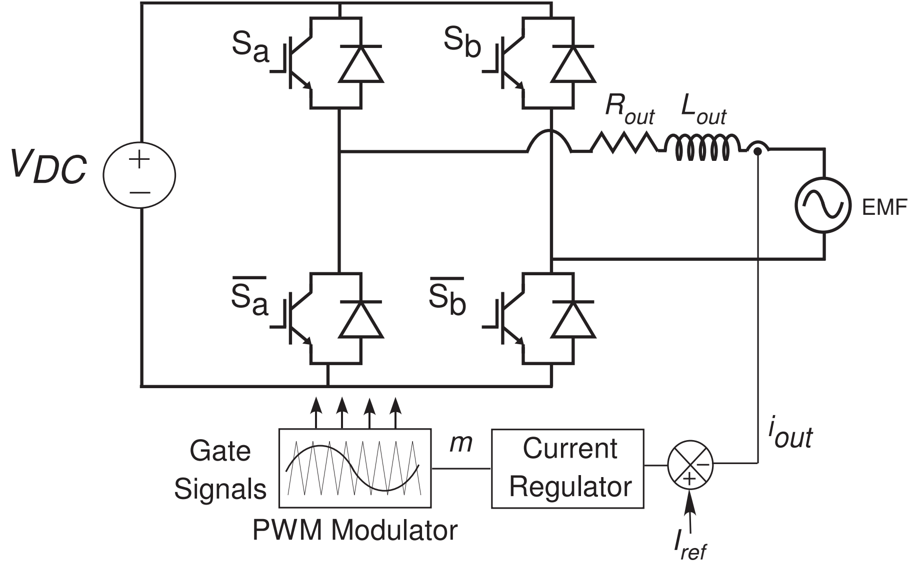


Figure 5.6: Structure of the Current Regulated VSI

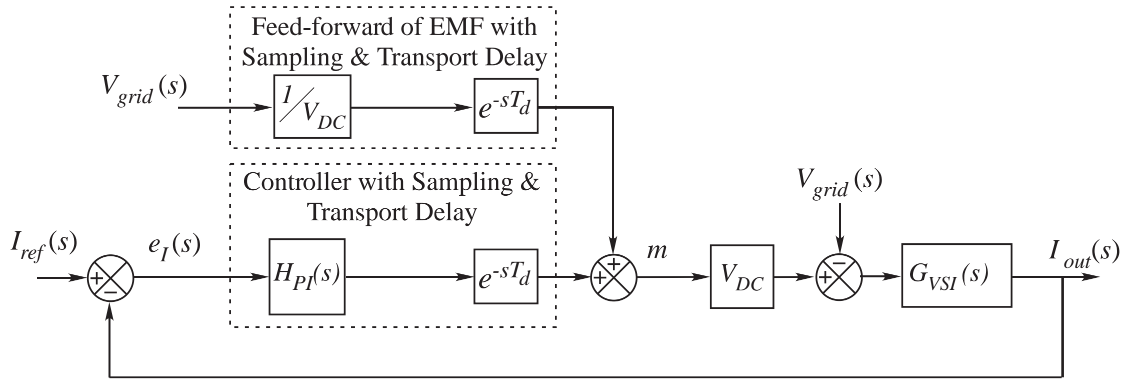


Figure 5.7: Closed-loop block diagram of the Current Regulated VSI

$$\omega_{c_{VSI}} = \frac{\left(\frac{\pi}{2} - \varphi_m\right)}{T_d} \tag{5.5}$$

Controller gains can now be calculated as [129]:

$$K_{p_{VSI}} = \frac{\omega_{c_{VSI}} L_{out}}{V_{DC}} \tag{5.6a}$$

$$T_{r_{VSI}} = \frac{10}{\omega_{c_{VSI}}} \tag{5.6b}$$

To minimise tracking error, [129] also suggests rejecting the effect of the grid voltage disturbance using feed-forward compensation, as incorporated into Fig. 5.7.

The design of this controller was then validated in simulation. Table 5.2 lists the parameters and gain values calculated for this controller, while Fig. 5.8 shows the

forward-path Bode plot of the closed-loop system. The response of this system to a step change in current reference is presented in Fig. 5.9 to show the fast transient response that was achieved.

Circuit Parameter		Value
Desired Phase Margin	$(\varphi_{m_{VSI}})$	40°
VSI Transport Delay Time	$(T_{d_{VSI}})$	150 μs
VSI Controller Bandwidth	$(\omega_{c_{VSI}})$	926 Hz
VSI Proportional Gain	$(K_{p_{VSI}})$	0.1454
VSI Integrator Time Constant	$(T_{r_{VSI}})$	10.8 ms

Table 5.2: VSI Current Regulator Parameters

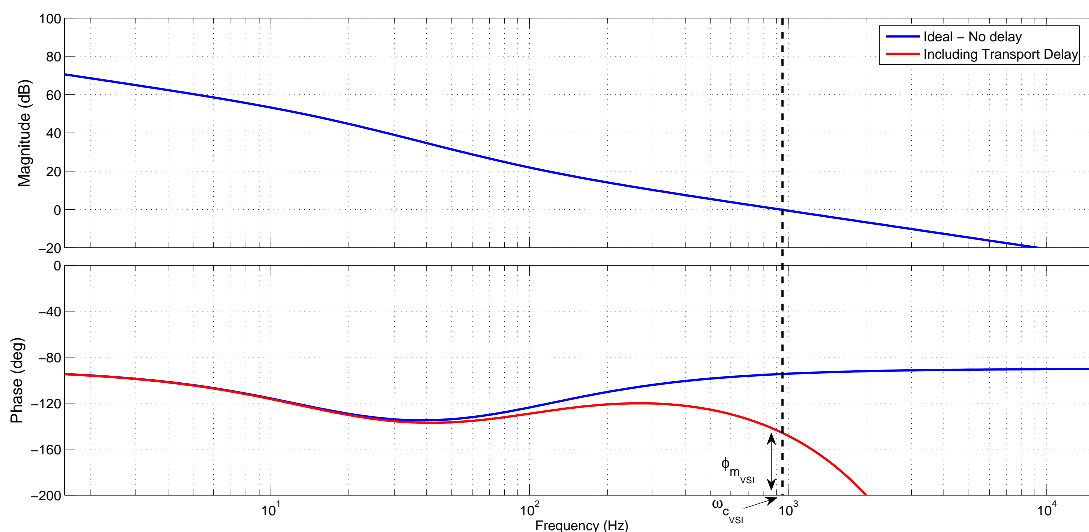


Figure 5.8: Forward path Bode plot of the Current-regulated VSI

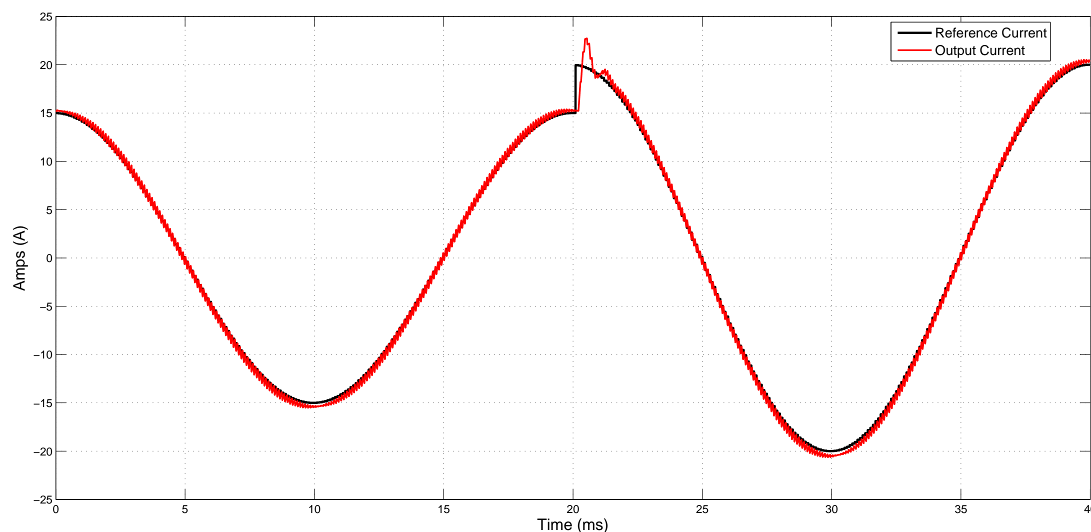


Figure 5.9: Step response of the current regulated VSI
[15A → 20A step]

5.3.3 DAB voltage regulator

The high performance voltage regulator structure designed in Chapter 4 is implemented on the DAB converter, whose closed-loop block diagram is restated in Fig. 5.10 for convenience. Table 5.3 lists the controller gains calculated for the simulated system, and Fig. 5.11 shows the forward path Bode plot of the closed-loop system. The response of the converter to a step change in voltage reference is shown in Fig. 5.12, which shows a fast transient response with no steady-state error.

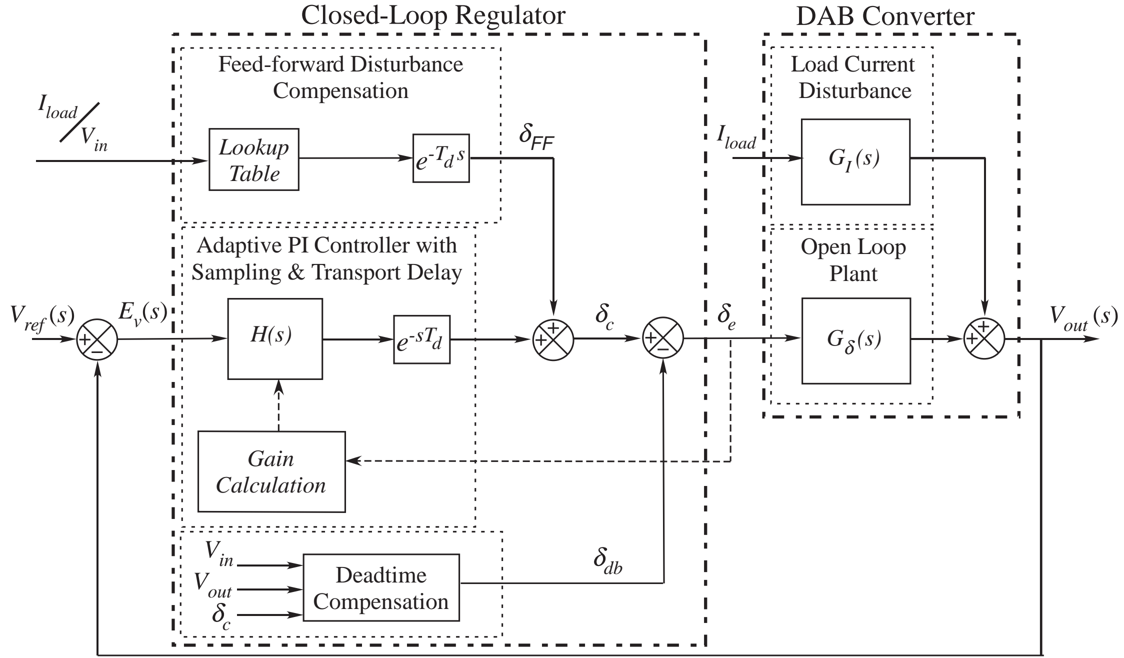


Figure 5.10: Closed-loop block diagram of the Voltage Regulated DAB

Load Current Variation

Since the converter feeds a continuously varying AC current to the grid, the load current seen by the DAB is also a continuously varying quantity. Chapter 4 has identified that the load current acts as a disturbance to the DAB converter, degrading

Circuit Parameter		Value
Desired Phase Margin	$(\varphi_{m_{DAB}})$	60°
DAB Transport Delay Time	$(T_{d_{DAB}})$	50 μs
DAB Controller Bandwidth	$(\omega_{c_{DAB}})$	1667 Hz
Maximum DAB Prop. Gain	$(K_{p_{DAB_{max}}})$	0.0102
Minimum DAB Prop. Gain	$(K_{p_{DAB_{min}}})$	0.2427
DAB Integrator Time Constant	$(T_{r_{DAB}})$	6 ms

Table 5.3: DAB Voltage Regulator Controller Parameters

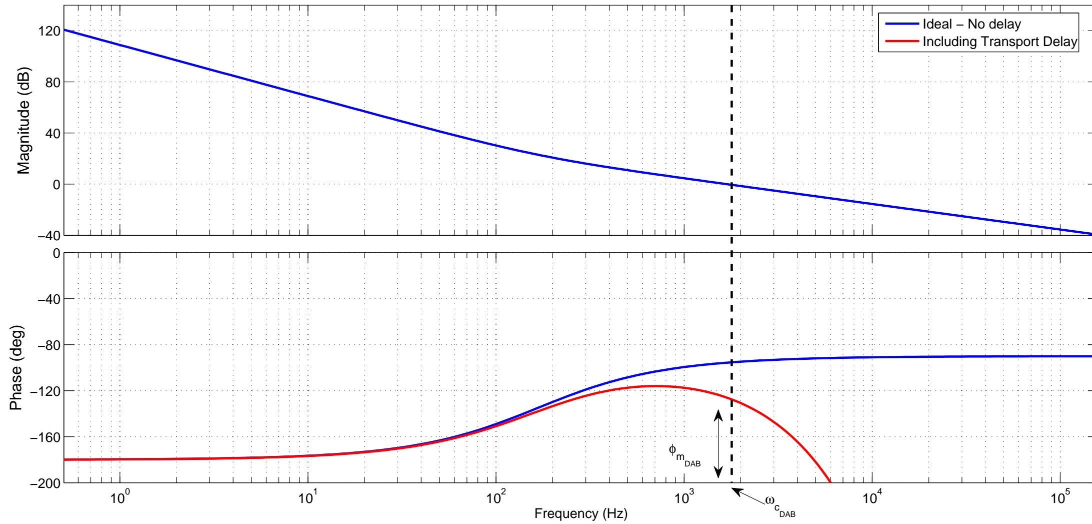


Figure 5.11: Forward path Bode plot of the Voltage-regulated DAB

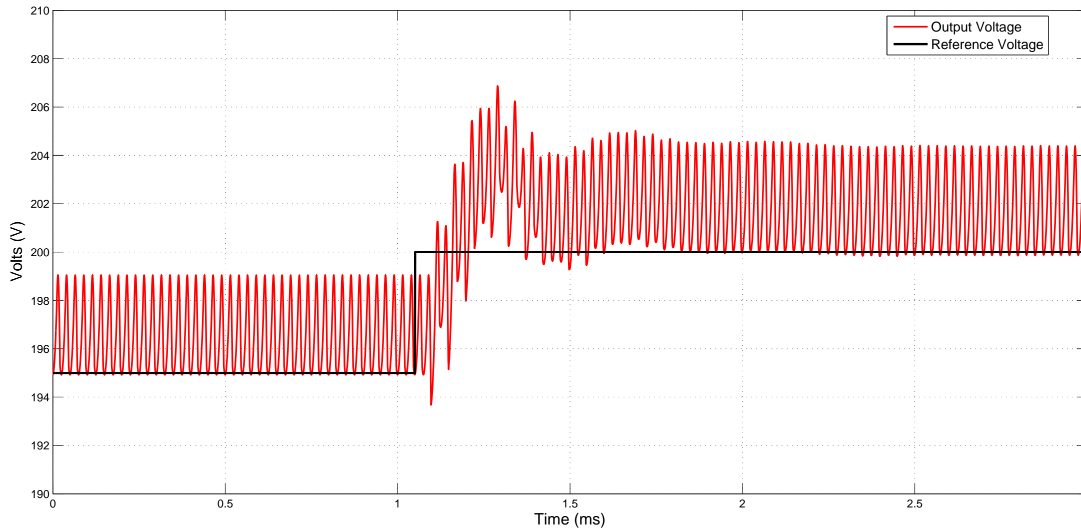


Figure 5.12: Step response of the voltage regulated DAB
[195V → 200V step]

the closed loop response in the face of a varying load current. Fig. 5.13 shows that ignoring the effect of the load current disturbance results in significant oscillations on the output DC bus, and Fig. 5.14 plots the frequency domain representation[3] of the voltage error. The error spectrum is clearly dominated by the harmonic term at twice the fundamental frequency (100Hz), caused by the oscillating power flow drawn by the VSI (see eq. 5.3). To improve the quality of the voltage waveform without increasing the required capacitance, feed-forward compensation of the load current disturbance is proposed.

To correctly implement disturbance compensation for this system, it is essential to first observe the load current waveform seen by the DAB. Unlike the continuous load current seen with a resistive load, the current drawn by an AC inverter is a train of switched pulses, as shown in Fig. 5.15. Each pulse has the same peak magnitude as

[3] Single-sided magnitude spectrum.

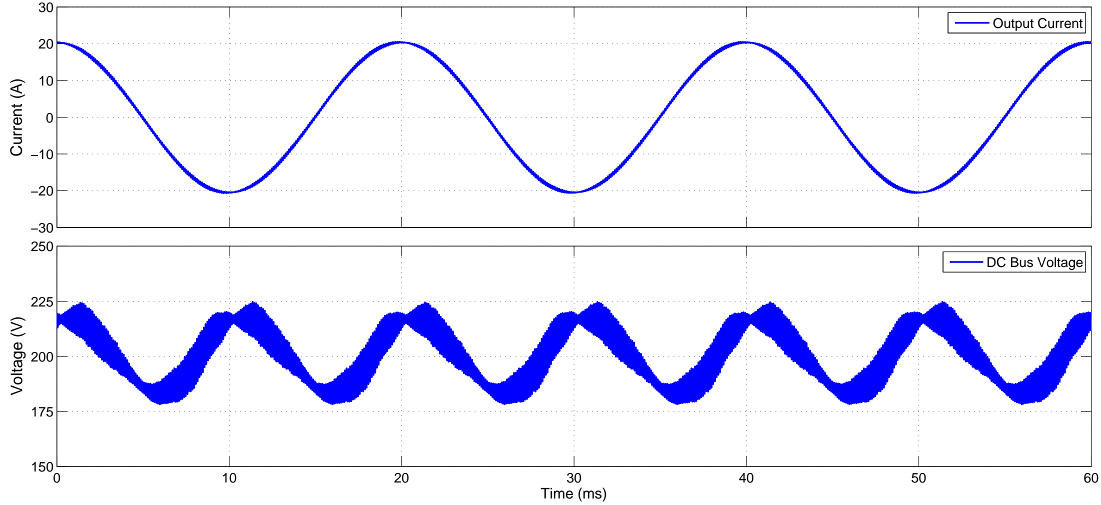
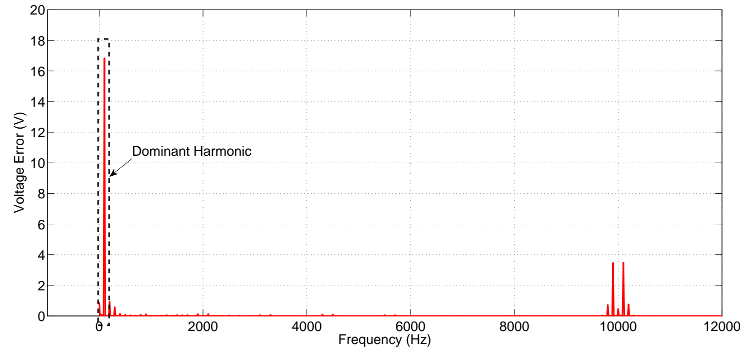
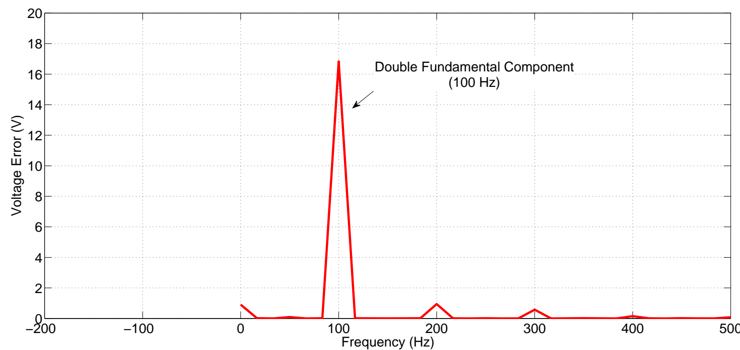


Figure 5.13: Output Voltage of the DAB converter with an AC load – No feed-forward compensation



(a) Full Spectrum

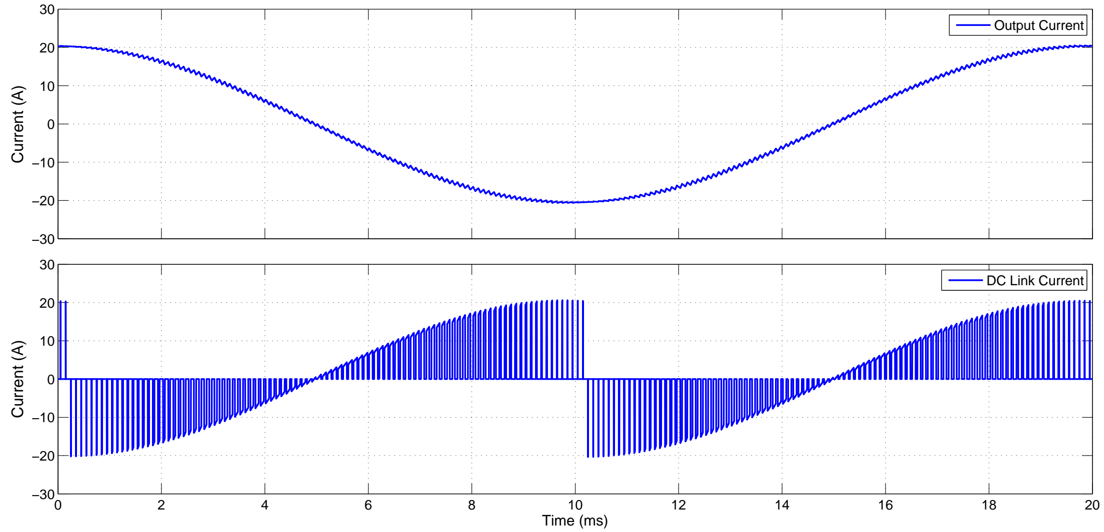


(b) Magnified

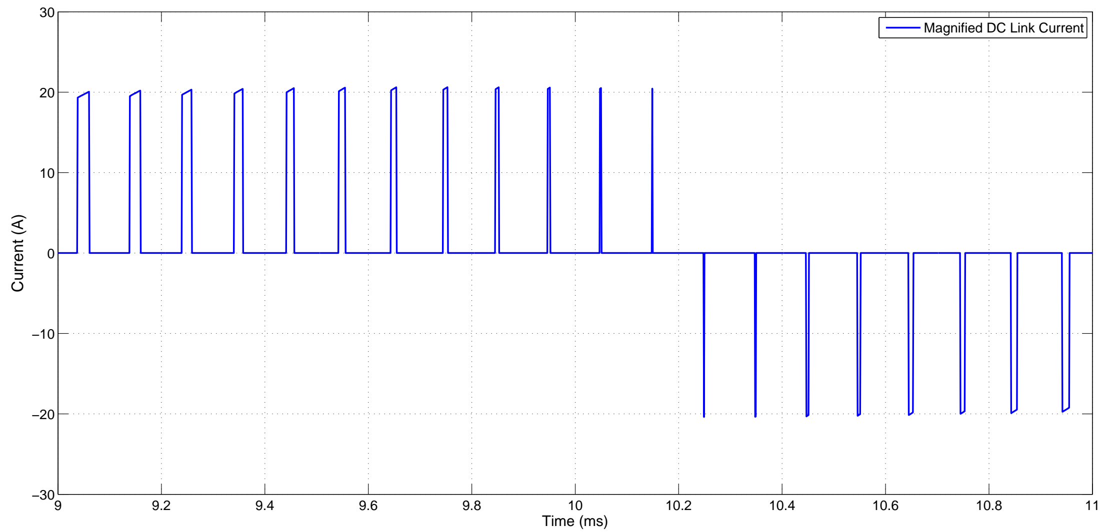
Figure 5.14: Harmonic Spectrum of Voltage Error - Without Feed-forward Compensation

the output AC current, and its duration depends on the instantaneous modulation depth (m) of the VSI, while the polarity of the current pulse is dependent on the load power factor.

Therefore, the load current disturbance that must be compensated is not the peak current current that flows during each switching cycle, but its *average*. This is easily



(a) DC link current



(b) Magnified DC Link Current

Figure 5.15: DC link current of the DAB converter with an AC load

approximated by scaling the sampled peak current by the instantaneous modulation depth of the VSI, as:

$$I_{load_{avg}} = mI_{load} \tag{5.7}$$

Having determined the average load current, feed-forward is implemented as proposed in Chapter 4 to correct for this disturbance, as shown in Fig. 5.10.

The transient response of this controller is illustrated in Figs. 5.16 & 5.17, which present the time domain voltage waveforms of the DC bus and the frequency spectrum of the error signal respectively. The double fundamental frequency oscillation in the frequency spectrum has been eliminated, leaving only the much higher frequency

terms caused by the PWM switching process. These terms cannot be removed by closed loop control because they exceed the controller bandwidth in frequency.

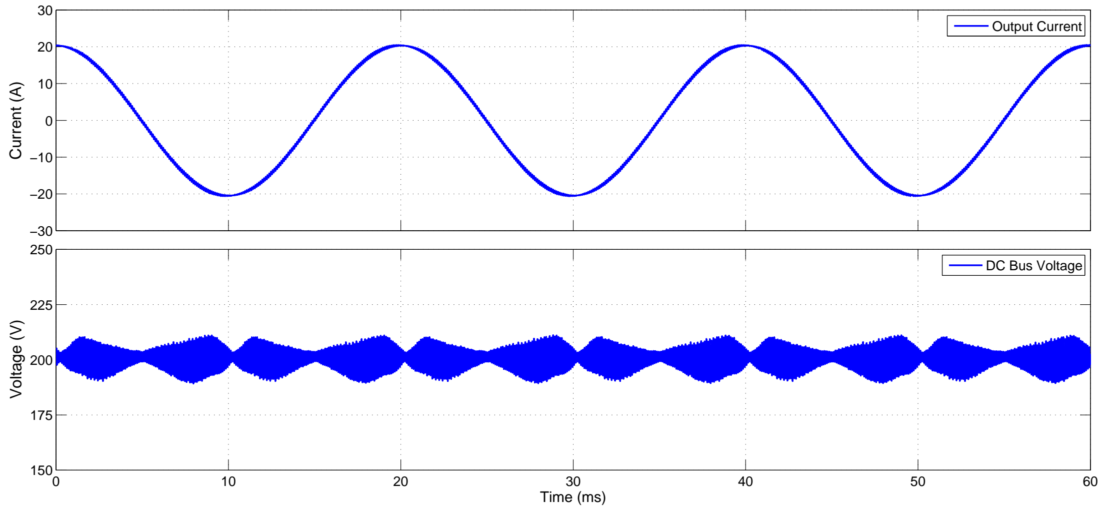


Figure 5.16: Output Voltage of the DAB converter with an AC load - With Feed-forward

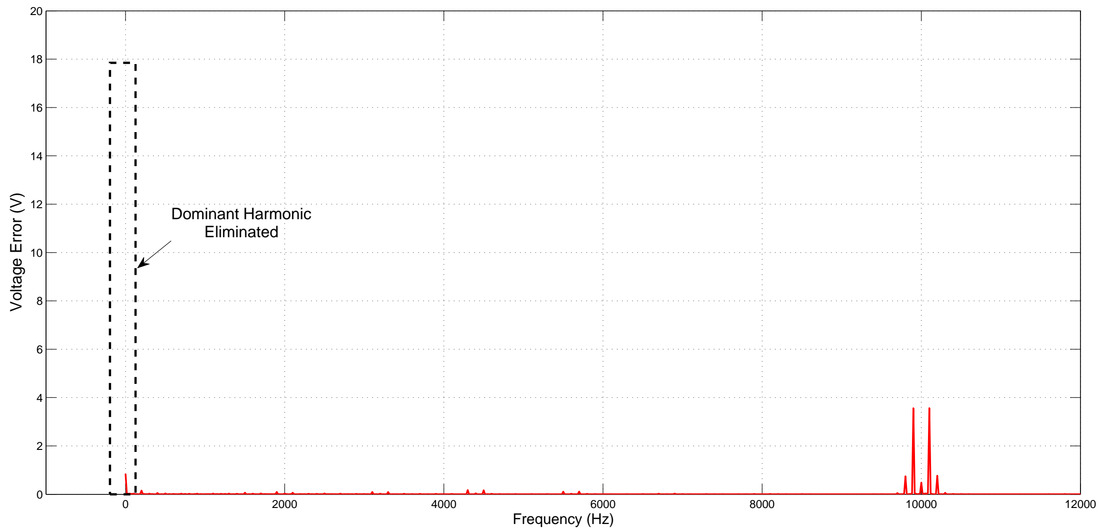
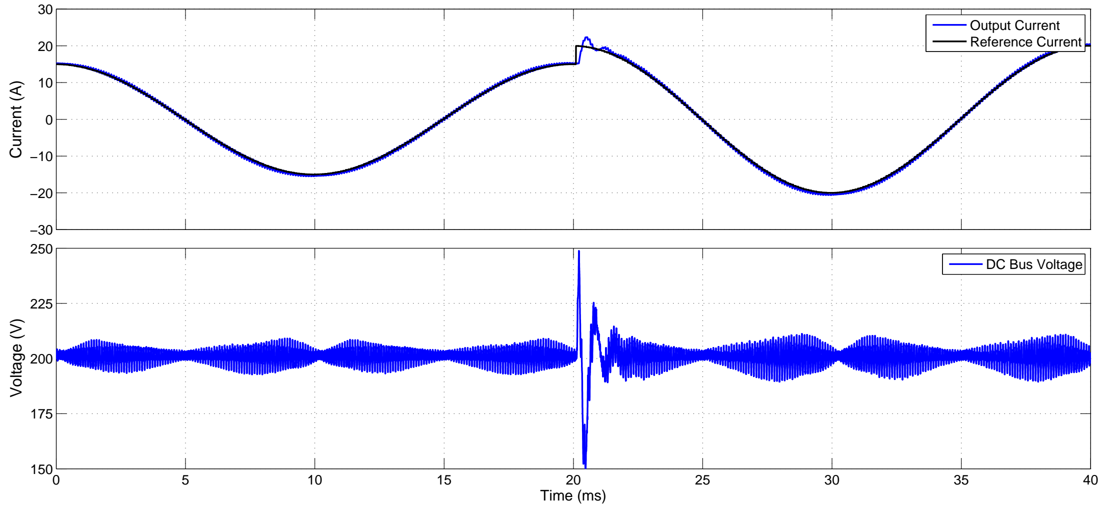


Figure 5.17: Harmonic Spectrum of Voltage Error - With Feed-forward

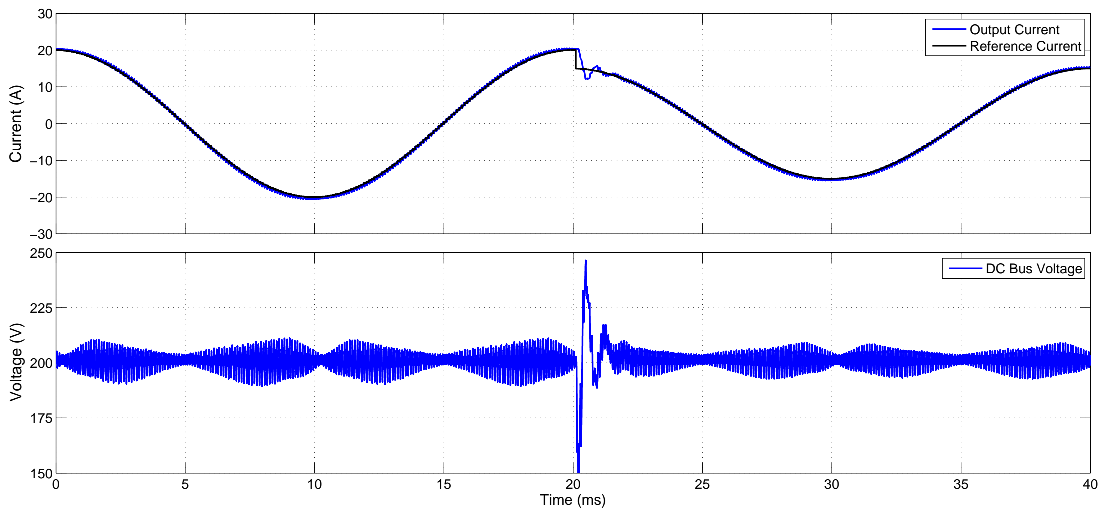
5.4 Results

The new closed-loop control techniques described in this chapter were tested by applying a step change to the converter AC output current reference, and monitoring the response of the intermediate DC bus voltage. Fig. 5.18 plots this transient response. The high performance current regulator gives a very rapid response, showing two oscillations before tracking the new current reference. This is consistent with the designed 40° phase margin. The rapid recovery of the DC bus to this

transient event is also clear. It first oscillates with the rapidly varying AC current before achieving steady-state. The speed of recovery is extremely fast, as stability is reached within 5 VSI switching cycles. The excursion of the DC bus voltage too is minimal, as 5% DC bus voltage ripple is achieved despite the low DC bus capacitance employed (20μF).



(a) Current Reference Step Up (15A $\rightarrow$ 20A)



(b) Current Reference Step Down (20A $\rightarrow$ 15A)

Figure 5.18: Converter output waveforms - Step change in current reference

5.5 Summary

This chapter has presented the design of a high-performance bi-directional AC-DC converter to interface energy storage elements to the Smart Grid. To optimise its transient response, this system made use of a new high performance closed loop control strategy that matched the oscillating AC energy flow of the grid without

the need for a large intermediate DC bus capacitor. This potentially eliminates the traditional electrolytic capacitor, replacing it with a film capacitor instead, achieving the goal of an electrolytic-free converter.

Chapter 6

Description of Simulated & Experimental Systems

During the course of this research, the ideas generated were extensively explored in simulation before being validated on the experimental prototype. This allowed each stage of the work to be verified, providing support for the overall results. This chapter describes these simulated and experimental systems were used for this exploration.

6.1 Simulated Systems

To simulate the behaviour of the DAB converter, the simulation package PowerSim (PSIM) was used. PSIM is a circuit simulation package created by PowerSim Inc. It specialises in simulating the behaviour of switched systems, which makes it a very powerful tool for power electronic converter analysis. In addition to being able to simulate basic circuit models, it also allows the effect of numerous non-ideal features to be included as part of the simulation (e.g. device voltage drops, deadtime, parasitic impedances, etc.), which allows the constructed simulations to closely match reality [132]. This ability to use the simulations to accurately predict the behaviour of physical systems is highly desirable because it allows the exploration of new ideas to be conducted in simulation with confidence that equivalent results will be achieved in practice. This saves time, and has significant safety benefits as well.

This section presents a functional overview of the simulation arrangement, followed by a description of all major simulation components.

6.1.1 Overview

The PSIM simulation used to examine the DAB dynamics can be divided into three parts, i.e.:

- **Power stages**
  The power stages cover the main switching converters, i.e. the DAB converter itself and its associated supply, loads and measurement circuitry.
- **Modulators**
  The modulators produce the commanded switching signals which control the states of the power stage switches.
- **Controllers**
  The term controller is used here to encompass not just the closed loop regulators employed by the system, but also the reference generation for these controllers and the operating mode selection. This allowed many different ideas to be tested on one simulation setup, which helped ensure consistent results.

6.1.2 Power Stage

The main power stage of the PSIM simulation is divided into two main components – the DAB converter and its load.

DAB Converter

The simulated DAB converter is shown in Fig. 6.1, and is made up of IGBT devices[1], supplied from a DC voltage supply. The system also includes salient voltage and current measurements (e.g. bridge output voltages, inductor & load currents, etc.).

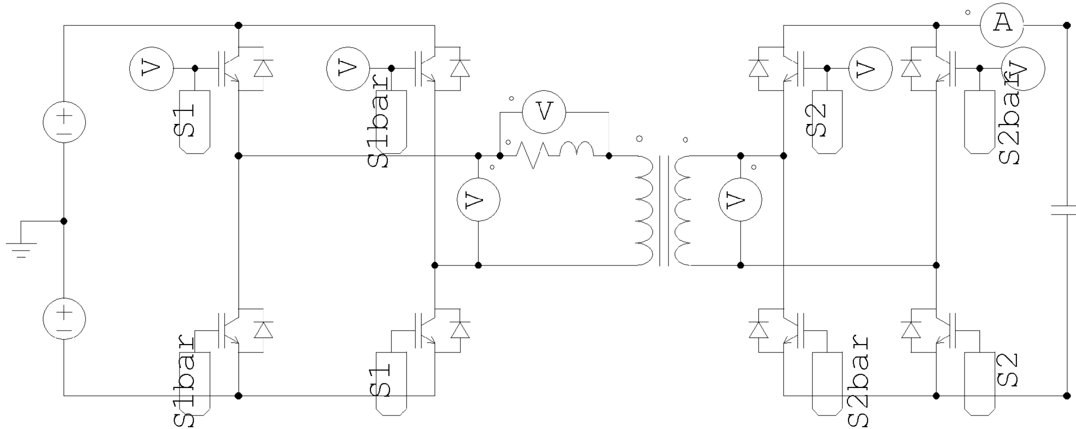


Figure 6.1: PSIM Power Stage - DAB Converter

Load

The complexity of the DAB load reflects the diversity of investigations that have been carried out during the course of this research. Fig. 6.2 shows the load system used for the simulation investigations. This simulation is designed to manage three possible load conditions, i.e.:

- **Constant load**
  This is the simplest possible load, i.e. a single load resistance (R_{const}).
- **Switched load**
  This is a load resistance that can be switched in or out of the circuit, and is used to explore the response of the DAB converter to step changes in load (R_{switch}).
- **Voltage Source Inverter (VS) load**
  This is the most complex load condition, made up of a H-bridge connected to a 50Hz AC grid via an Resistive/Inductive load (L_{VSI} & R_{VSI}). This was used to explore the effects of the AC load (Chapter 5).

[1] To match the experimental prototype.

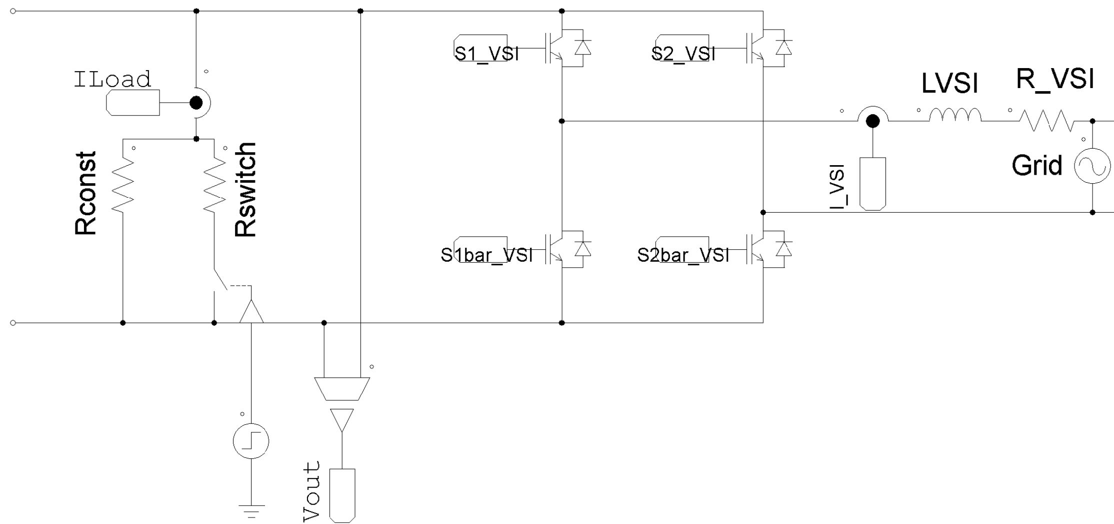


Figure 6.2: PSIM Power Stage - DAB Load

6.1.3 Modulators

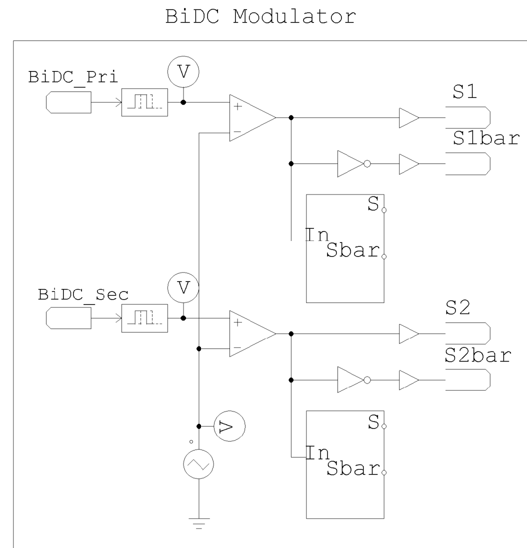
The modulators (shown in Fig. 6.3) are used to generate the switching signals needed by the power stage. There are two switched converters in the simulation power stage (the DAB & the load VSI), so each one has its own modulator. The DAB modulator produces PSSW modulation signals while the VSI modulator produces PWM.

The modulators use a comparator that compares the input modulation reference signal to a carrier wave to determine the condition of the output switch. The simulation also includes a time delay block (Fig. 6.4a) to account for computation delays in the digital modulator/controller (Section 4.2), and a deadtime generation block, shown in Fig. 6.4b so the effect deadtime has on the converter can be simulated (Section 3.6).

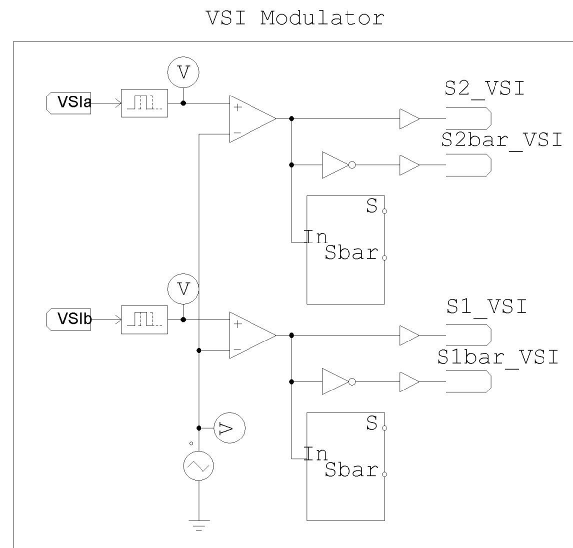
6.1.4 Controllers

Control of the simulation was achieved using the Dynamic Link Library (DLL) feature of PSIM (see Fig. 6.5). This allows C code to be embedded into the circuit simulation. Since the experimental prototype is also programmed in C (Section 6.2.3), this feature is very powerful because once a simulation is constructed, the same algorithms can be implemented on the experimental prototype with little or no modification. The code used to generate this DLL is included in Appendix A.

The inputs to the DLL include all the measurements necessary to regulate the DAB converter and the load H-bridge, such as the DC output voltage (V_{out}), DC

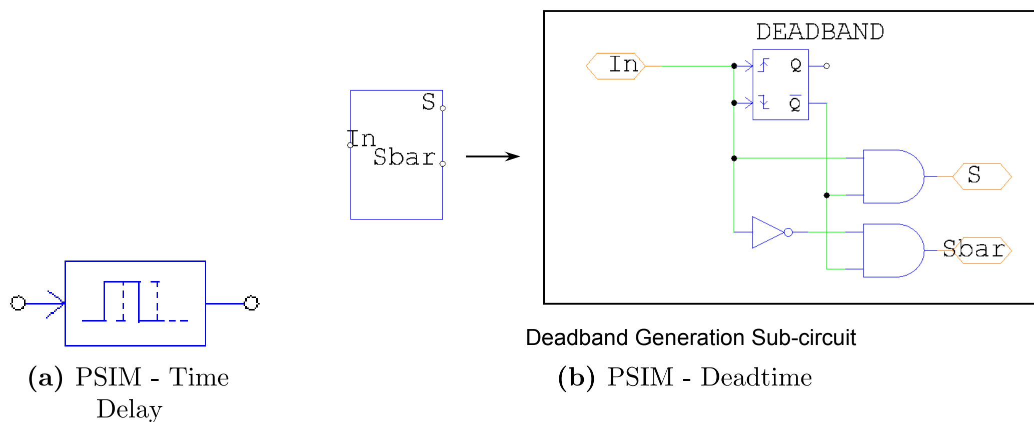


(a) DAB modulator



(b) VSI modulator

Figure 6.3: PSIM Simulation - Modulators



(a) PSIM - Time Delay

(b) PSIM - Deadtime

Figure 6.4: Modulator Features

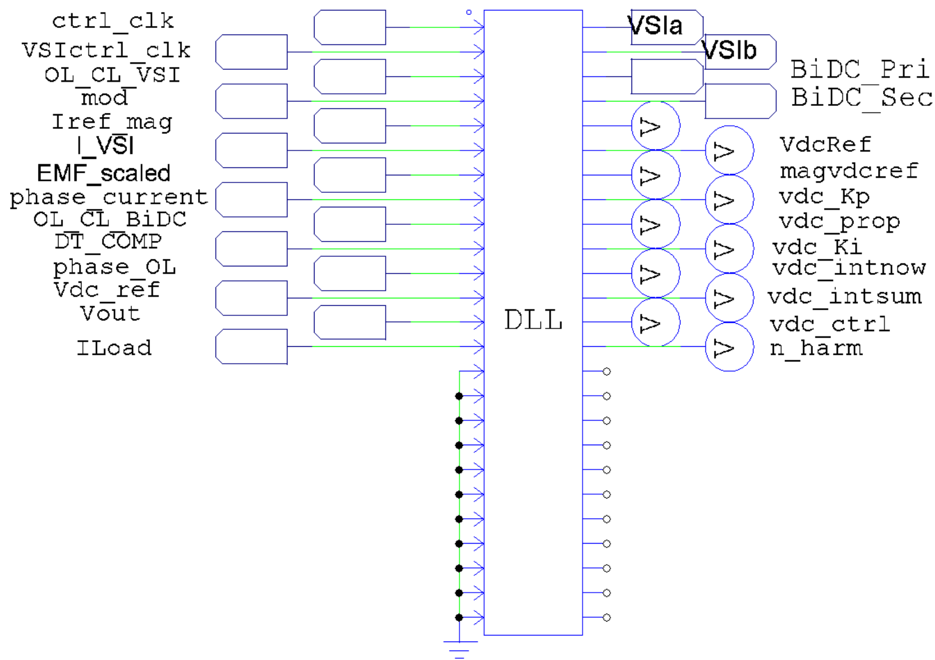


Figure 6.5: PSIM Simulation - DLL Block

load current (I_{Load}), VSI output current (I_{VSI}), etc.), as well as mode-setting inputs such as OL_CL_VSI, OL_CL_DAB & DT_COMP, which select the active features for a particular simulation run, as Table 6.1 shows.

Having selected a mode of operation and read all necessary system measurements, the DLL block then performs the required closed-loop calculations (e.g. the PI controllers, feed-forward compensation, deadtime compensation factors, etc.) in the discrete time domain. The results of these calculations are the final modulation references, which are set as DLL block outputs (e.g. VSIa & VSIb, which are the modulation references for the output H-bridge, etc.), and their values passed to the modulators. However, DLL outputs are not limited to just closed-loop regulator results. In fact, any variable within the C code can become an output, simplifying the debug process.

Circuit Parameter	**Value**	**Effect**
OL_CL_VSI	0	Open-loop Modulated VSI
	1	Closed-loop PI Current Regulated VSI
	2	Closed-loop PI Current Regulated VSI with Feed-forward Grid Disturbance Compensation
OL_CL_BiDC	0	Open-loop Modulated DAB
	1	Closed-loop Adaptive PI Voltage Regulated DAB
	2	Closed-loop Adaptive PI Voltage Regulated DAB with Feed-forward Load Current Disturbance Compensation (DC)
	3	Closed-loop Adaptive PI Voltage Regulated DAB with Feed-forward Load Current Disturbance Compensation (AC)
	4	Closed-loop Fixed PI Voltage Regulated DAB
DT_COMP	0	Deadtime Compensation Inactive
	1	Deadtime Compensation Active

Table 6.1: DAB Voltage Regulator Controller Parameters

6.2 Experimental Prototype

This section describes the experimental prototype. It first presents a functional overview of the system before detailing its salient components and their functionality.

6.2.1 Overview

Fig. 6.6 shows the circuit diagram of the experimental setup, and Table 6.2 lists its salient parameters. The system can be divided into two parts, i.e.:

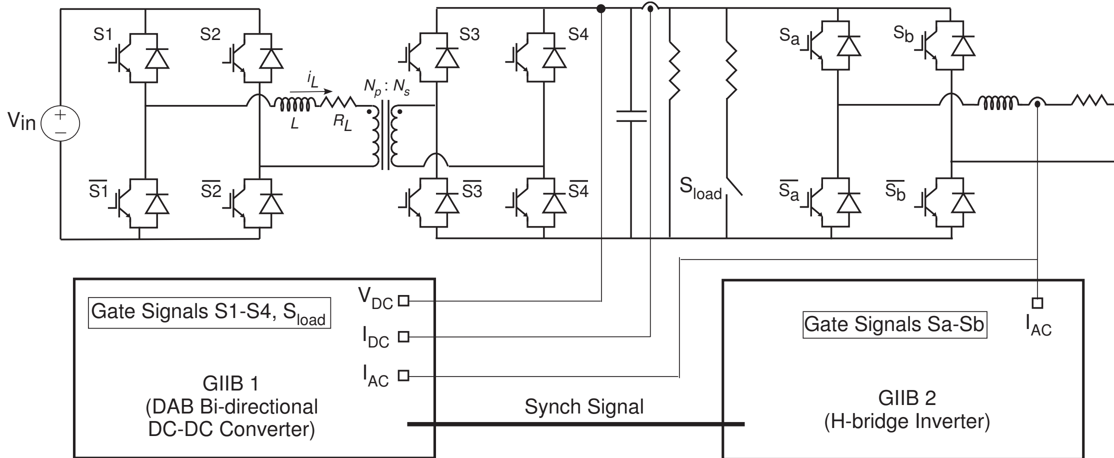


Figure 6.6: Experimental Setup Circuit Diagram

- **Power stage**
  Includes the incoming DC voltage supply, both switching converters (DAB & VSI) and their load impedances.

- **Controllers**
  Comprises the microprocessor-based converter control boards.

A photograph of the prototype is presented in Fig. 6.7, and the details of its construction and implementation are the focus of the following two sections.

6.2.2 Power Stage

Input Supply

The power supply to the system (see Fig. 6.8) is a MagnaPower XR250-8 current-limited DC supply, capable of supplying up to 250 Volts at 8 Amps. This provided the stiff voltage source necessary for system operation.

Circuit Parameter		Value
DC Input Voltage	(V_{in})	200 V
DC Output Voltage	(V_{out})	200 V
Transformer Turns Ratio	$(N_p : N_s)$	10 : 11
VSI Switching Frequency	(f_{VSI})	5 kHz
DAB Switching Frequency	(f_{DAB})	20 kHz
DC Capacitance	(C)	12 μF
AC Inductor Inductance	(L)	132 μH
AC Inductor Resistance	(R_L)	0.1 Ω
Output Inductance	(L_{out})	8 mH
Output Load Resistance	(R_{out})	16.5 Ω
Nominal Output Power	(P_{out})	1 kW

Table 6.2: DC-AC Experimental Converter Parameters

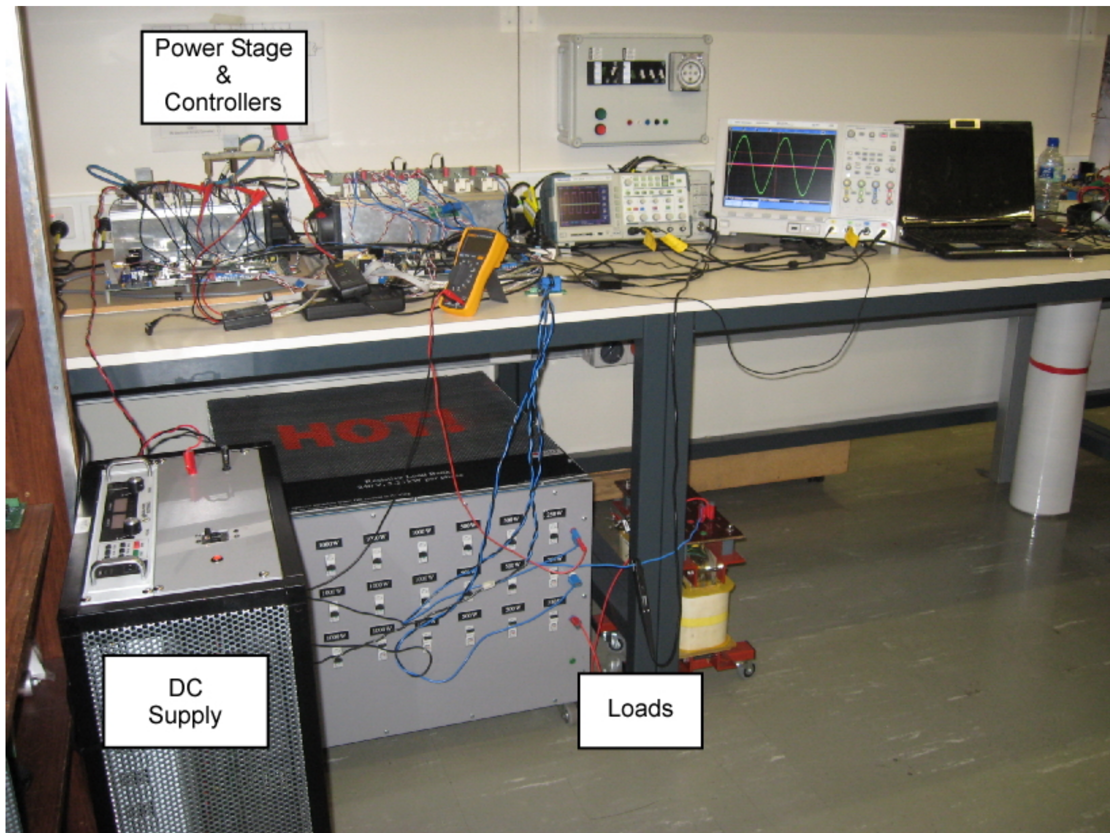


Figure 6.7: Laboratory Setup

Figure 6.8: MagnaPower DC Supply

Power Converter

As described in Section 2.1, the DAB converter is made up of two H-bridge converters connected across an AC inductor and a transformer. A photograph of the experimental DAB converter is shown in Fig. 6.9.

As Fig. 6.9 shows, each H-bridge is made up of two BSM50GB120DLC IGBT phase legs. These are 1200V, 50A devices, and are bolted to an aluminium heatsink. The DC bus is made of copper bars, and the DC bus voltage is supported by film capacitors. For maximum performance, these capacitors are bolted directly to the copper bars. All current-carrying wires to and from the primary & secondary side DC buses are kept short and twisted to minimise stray DC inductance, which helps improve system performance.

The High Frequency air-cored AC inductor for the DAB converter (Fig. 6.10) is wound using 2mm diameter copper wire (rated for ≈20A DC), hand wound on a PVC tube 60mm in diameter. To achieve the desired 1 kW power level, it was calculated that $\approx 132\mu$H of inductance was required, which required 83 turns.

The transformer (see Fig. 6.11) is a TR-MODU-T1 product from Creative Power Technologies, and uses a U80 core made of ferrite material 3C81, which helps minimise loss in the transformer. The entire transformer is mounted in 'potting mix' which

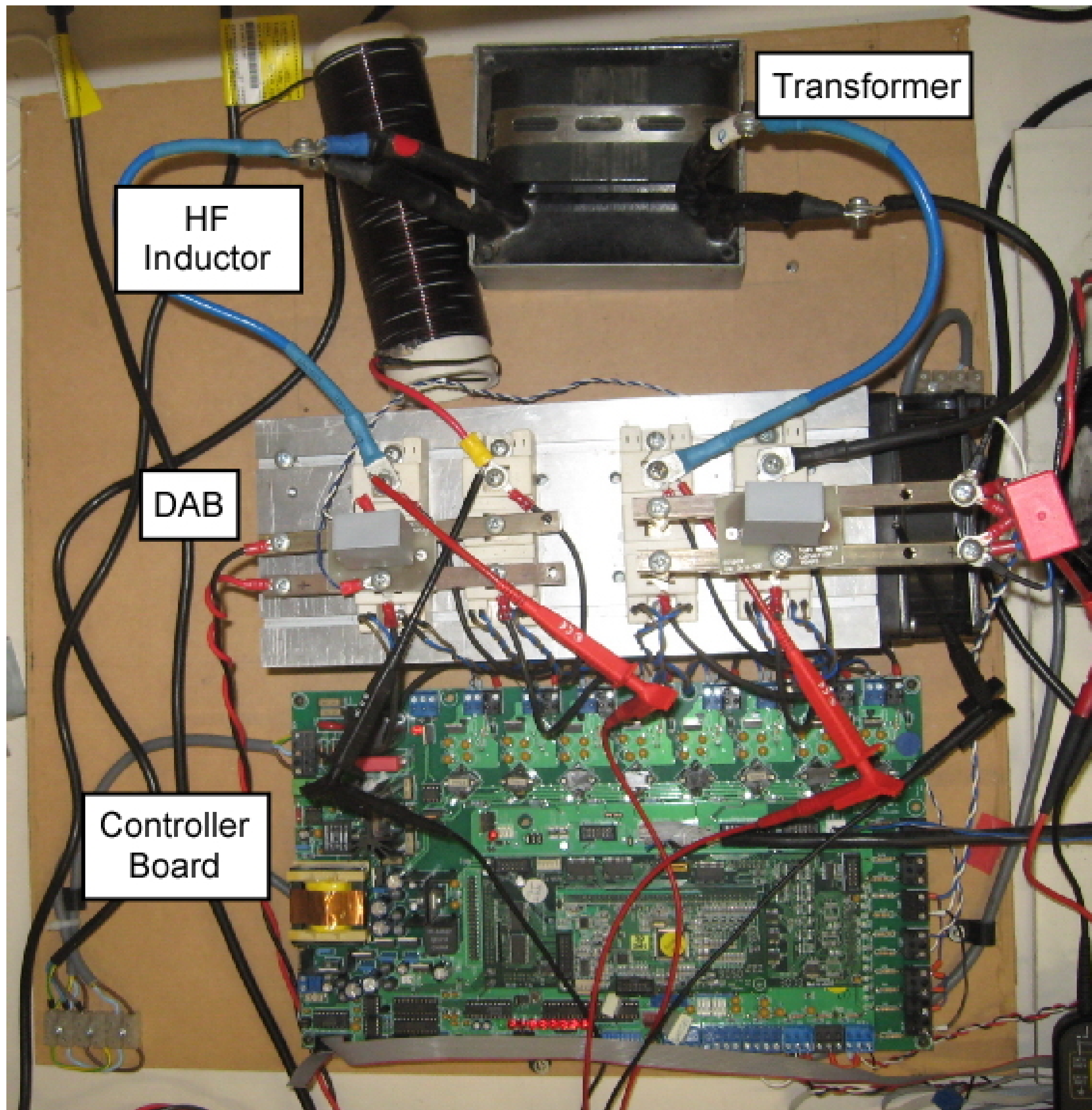


Figure 6.9: Experimental DAB Converter

helps to extract heat from the core and its associated windings. More details on the construction of this transformer can be found in [133].

Load

The converter load is complex, because this single experimental prototype had to handle three possible load conditions, i.e.:

- **Constant Load**
- **Switched Load**
- **VSI Load**

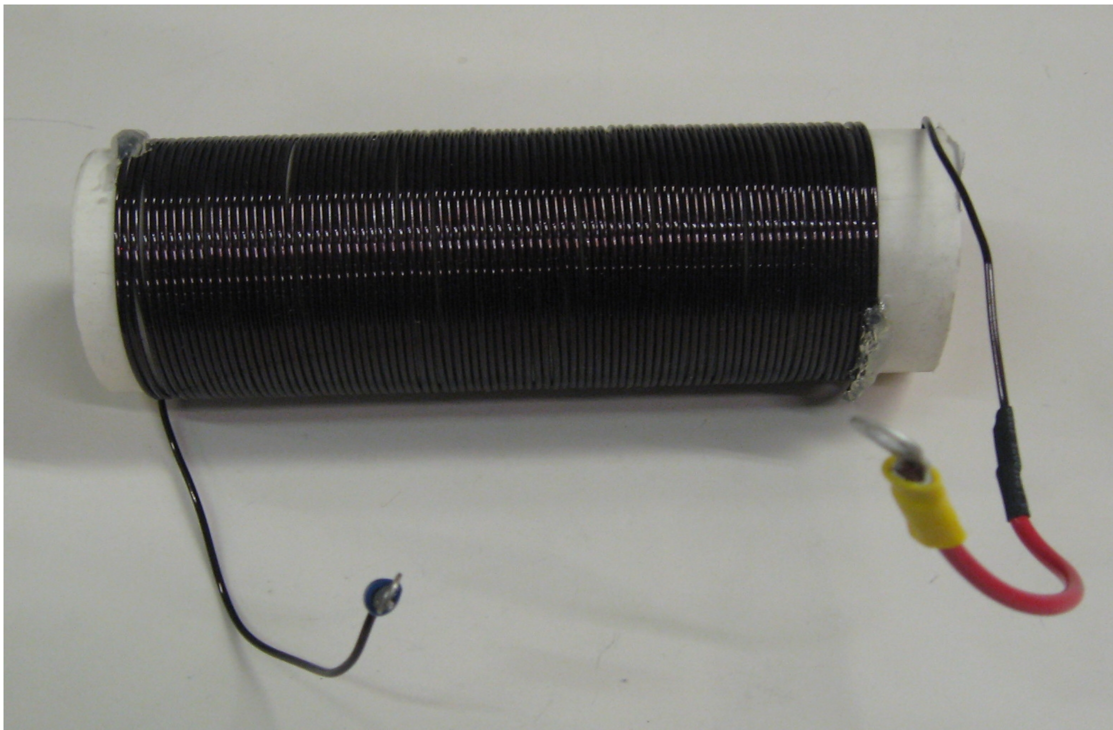


Figure 6.10: Experimental High Frequency AC Inductor

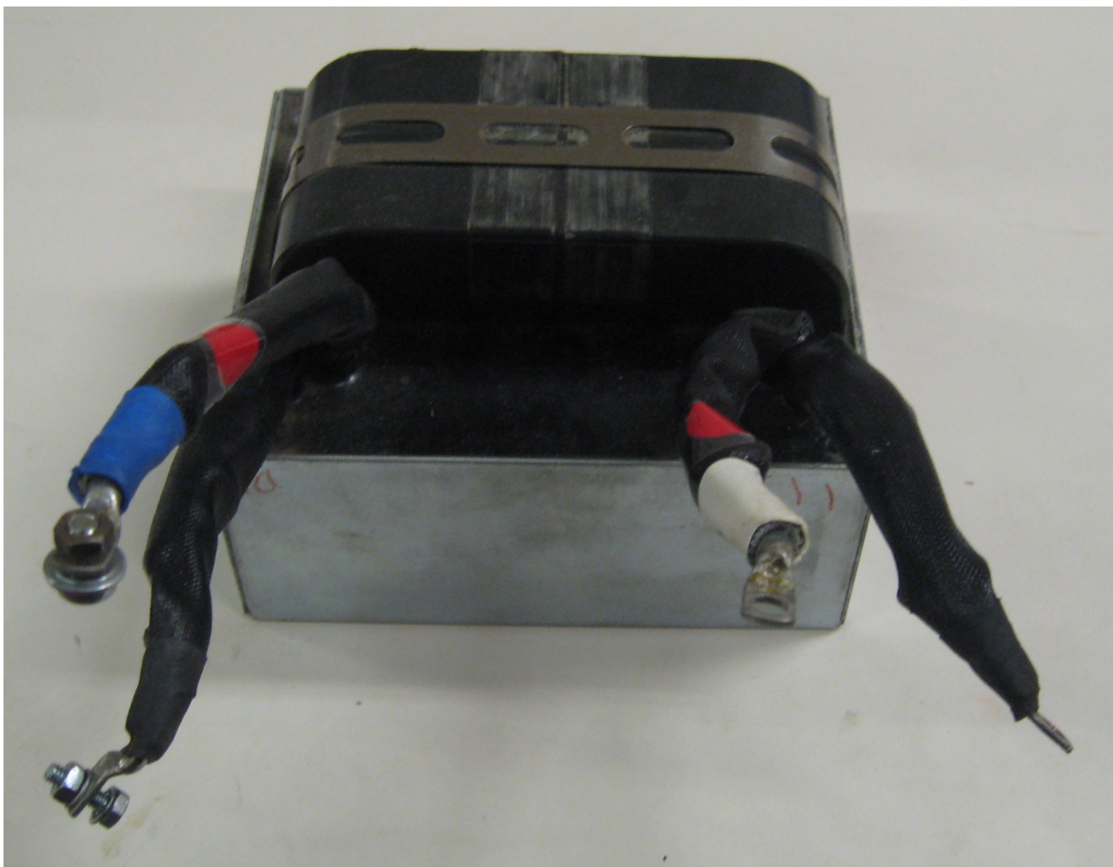


Figure 6.11: Experimental High Frequency Transformer

The circuit diagram of the load is shown in Fig. 6.12. The impedances used are standard laboratory resistors and inductors, shown in Fig. 6.13, but additional circuitry was needed for operation of the switched and VSI loads.

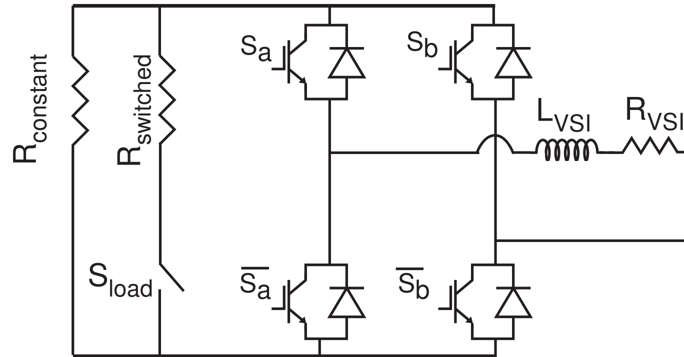


Figure 6.12: Experimental Load Circuit Configuration

(a) Resistive Load Bank

(b) AC Inductor

Figure 6.13: Experimental Load Elements

Specifically, the VSI load requires a single phase H-bridge, while the switched load requires a fast-acting switch in series with the load resistance to quickly and safely connect/disconnect it from the circuit. Both these tasks were achieved using a set of IGBT switches connected as shown in Fig. 6.14. This was constructed by using a set of six BSM100GB120DLCK IGBTs bolted to an aluminium heatsink. The interconnections between these IGBTs was achieved using a Printed Circuit Board (PCB) DC bus structure (see Fig. 6.15).

The final constructed IGBT platform is shown in Fig. 6.16. Only three of the six available phase legs are used – $\bar{S}1$ of the first phase leg provides the switch for the switched load (S_{load} in Fig. 6.12), while the next two phase legs ($S2, \bar{S}2, S3, \bar{S}3$) make up the single-phase H-bridge.

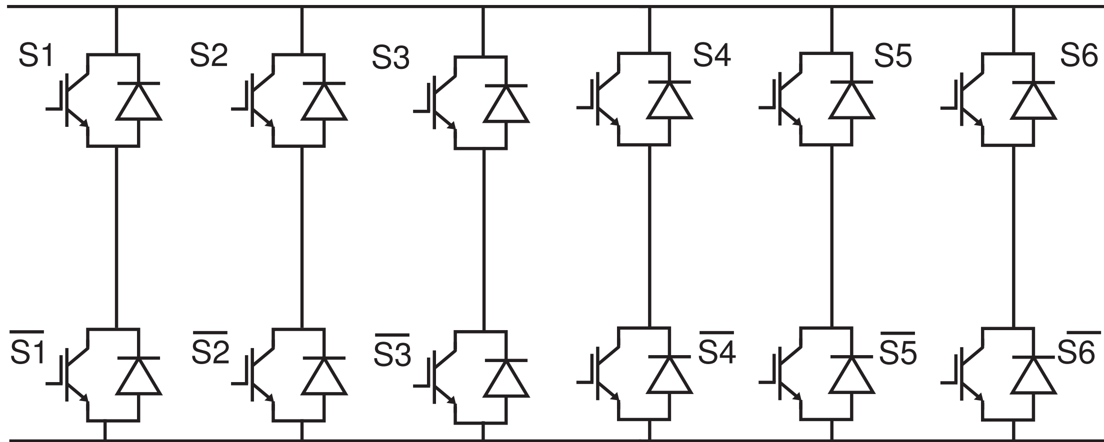


Figure 6.14: Circuit Diagram of Experimental 6 phase leg converter

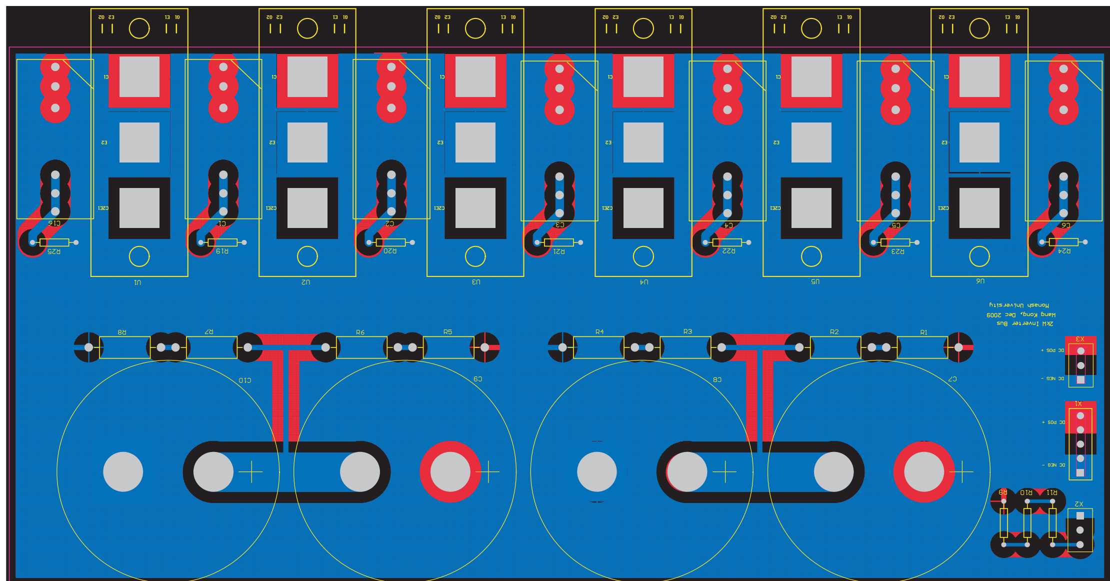


Figure 6.15: PCB DC Bus Structure

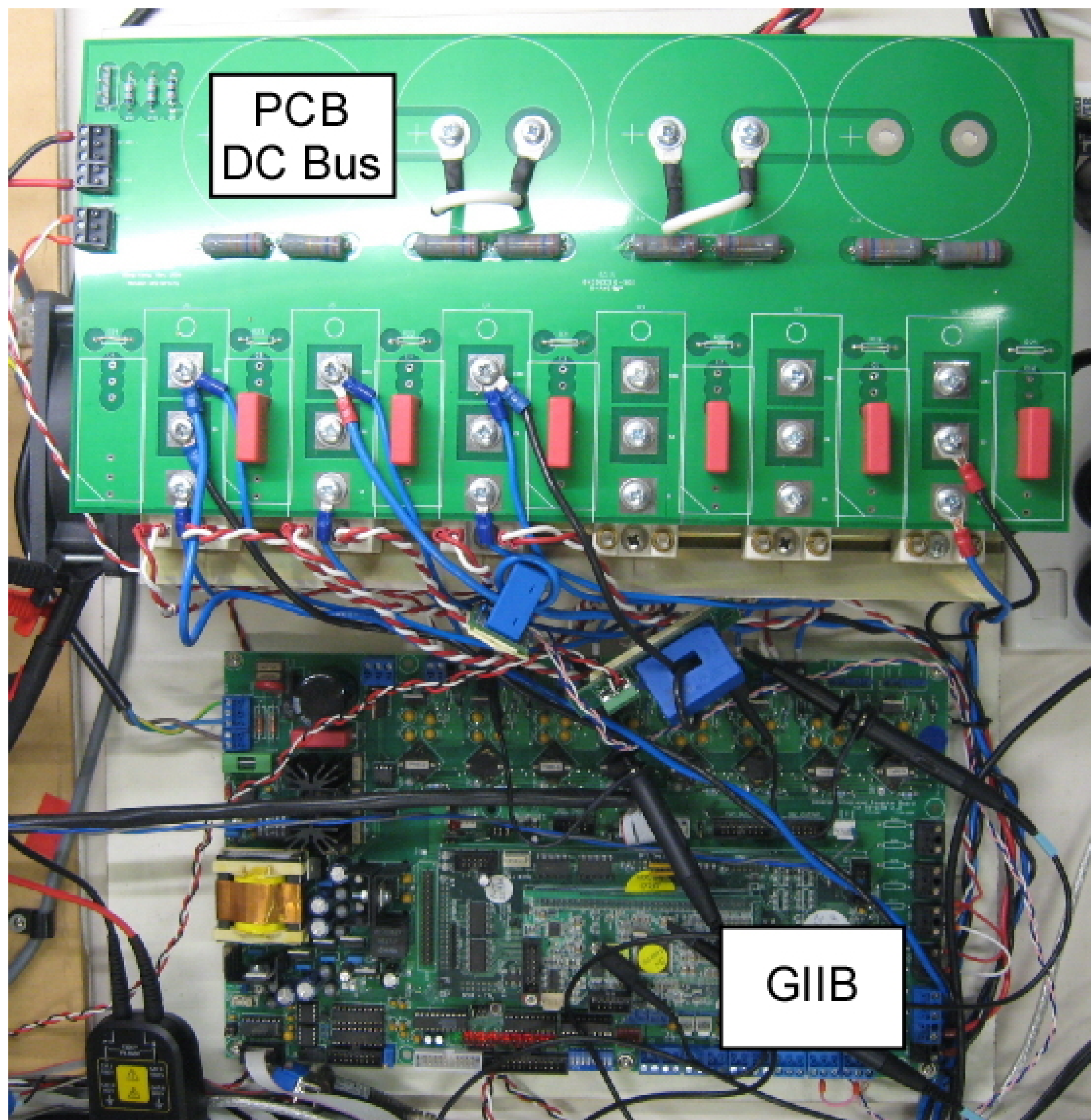


Figure 6.16: Experimental 6 phase leg IGBT platform.

6.2.3 Controller Hardware

The power stage converters (DAB & VSI) were each controlled using an inverter control board produced by Creative Power Technologies (CPT) [134--136].

Each inverter control board is based on the Texas Instruments TMS320F2810 Digital Signal processor (DSP). This powerful microprocessor handles all converter control tasks, and interfaces to the power stage via three daughter boards, all produced by the company CPT. These boards are the DA-2810, the MINI-2810 and the GIIB (Generalised Integrated Inverter Board), respectively. The functionality of the DSP as well as each board is described in the sections below, followed by a description of the inter-GIIB communication that was employed for this work.

2810 DSP

The TMS320F2810 Digital Signals Processor (hereafter referred to as the '2810') is a product of Texas Instruments, and is designed specifically for motor control and power electronic applications. It includes numerous features, which include, but are not limited to:

- Analog-to-Digital Converters (ADCs) for measurements & sensing
- Event Managers capable of generating many kinds of modulation signals (e.g. PWM & PSSW)
- Serial Peripheral Interfaces (SPI) for communication and user interface
- Transition logging functionality (Capture Ports)
- Digital-to-Analog Converters (DACs)

To correctly perform calculations using the 2810, it is important to recognise that it is designed for fixed-point calculations, i.e. only integer variables. Floating-point (decimal) calculations can be performed, but are quite expensive in terms of computation time, and should therefore be avoided. Hence, to achieve the high precision demanded by the closed-loop calculations, a technique called *Floating-point Emulation* is used [137]. This technique artificially scales fixed-point numbers such that they can represent floating point values before performing the necessary calculations. This allows the accuracy of a floating-point calculation to be emulated with only fixed-point variables, keeping computation time to a minimum [137].

DA-2810 Board

The DA-2810 is a standardised DSP controller board produced by CPT (Fig. 6.17). It is designed to provide a fully flexible interface between the 2810 DSP and the subsequent daughter boards (MINI-2810 & GIIB, in this case). This board therefore brings features of the 2810 out to physical ports (e.g. a Molex header for serial RS-232 communications, a JTAG header for chip programming, etc.), while also providing all necessary auxiliary circuitry for DSP functionality (e.g. power supply, etc.). The technical manual for this board is available as [134].

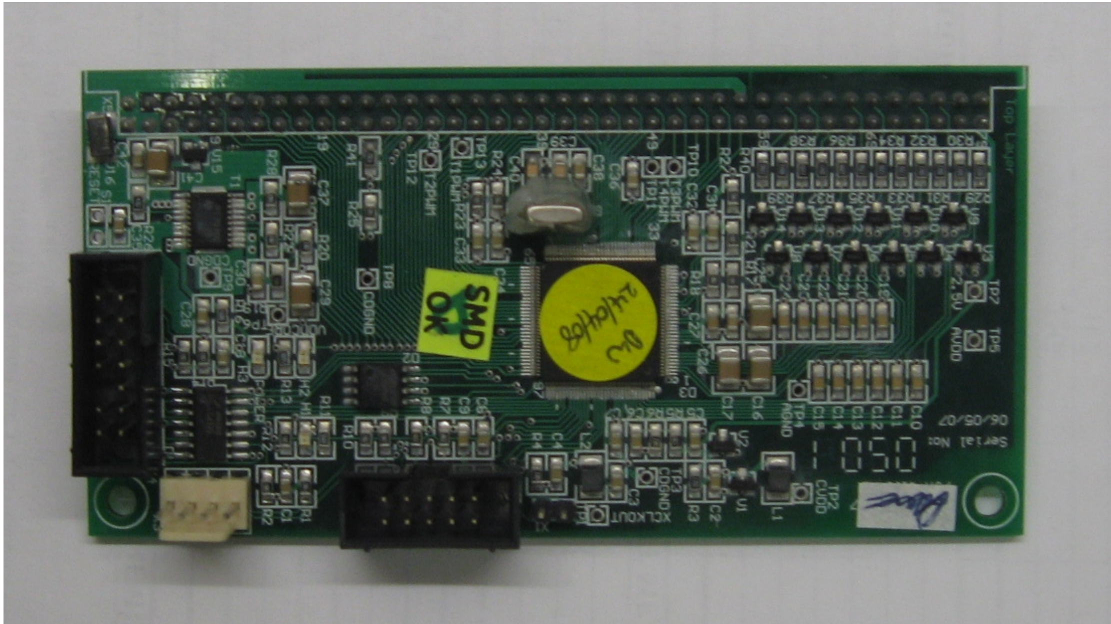


Figure 6.17: DA2810 DSP Controller Board

MINI-2810 Board

As Fig. 6.18 shows, the DA-2810 plugs directly into the MINI-2810 [135]. The MINI-2810 acts as an interface board between the DA-2810 and the GIIB, and is based on the Altera MAX II EPM570T100C5N Complex Programmable Logic Device (CPLD). For this project, this board performed three basic functions, i.e.:

- **Signal Routing & Protection**
  The MINI-2810 takes signals from the DA-2810 & GIIB boards (e.g. PWM modulation signals, Capture port signals, ADC signals, etc.) and routes them between the two boards as needed. It also includes a set of input buffer chips which help protect the DA-2810 board.

- **SPI - MiniBus translation**
  MiniBus is a proprietary communication protocol used by CPT to communicate

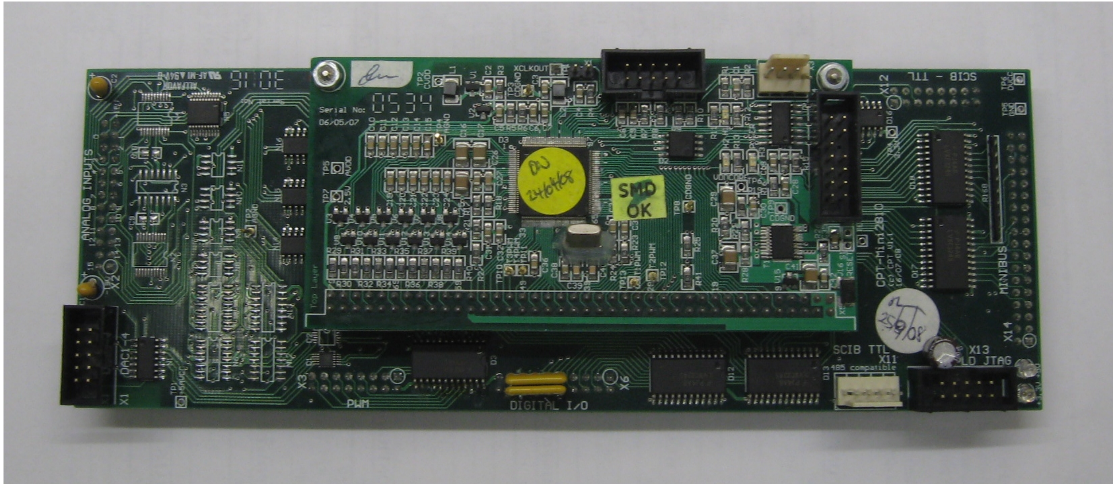


Figure 6.18: Mini2810 Controller Board

between the 2810 and the external functionality of the converter board(s). For example, a MiniBus command is used to operate the Digital to Analog Converters (DACs) on the GIIB. The Mini2810 translates the serial commands from the 2810 (SPI) into MiniBus commands for the GIIB board.

- **PWM Lockout**
  As an additional safety feature, the MINI-2810 provides a lockout mechanism for PWM modulation signals. Functionally, this means that switching signals cannot propagate to the converter power stage before the MINI-2810 is correctly enabled. This prevents spurious switching signals upon start-up.

Generalised Integrated Inverter Board (GIIB)

The GIIB board (see Fig. 6.19)is the primary interface between the high voltages and currents of the converter power stage and the logic level control signals of the MINI-2810 & the DA-2810. For a full description of the GIIB functionality, its technical manual is available as [136]. In the context of this research, it performs four basic functions, i.e.:

- **Power Supply**
  The GIIB includes a Switch Mode Power Supply (SMPS) that connects the incoming AC mains (240V) mains AC and converts it to the various DC voltage levels required[2] by the GIIB, MINI-2810 and DA-2810 (Fig. 6.19).

- **Driving Power Devices**
  Driving the IGBT switches that make up the power stage takes specialised gate drive circuitry, which is provided on the GIIB board (Fig. 6.19).

[2] e.g. +24VDC, +12VDC, +/-15VDC, etc.

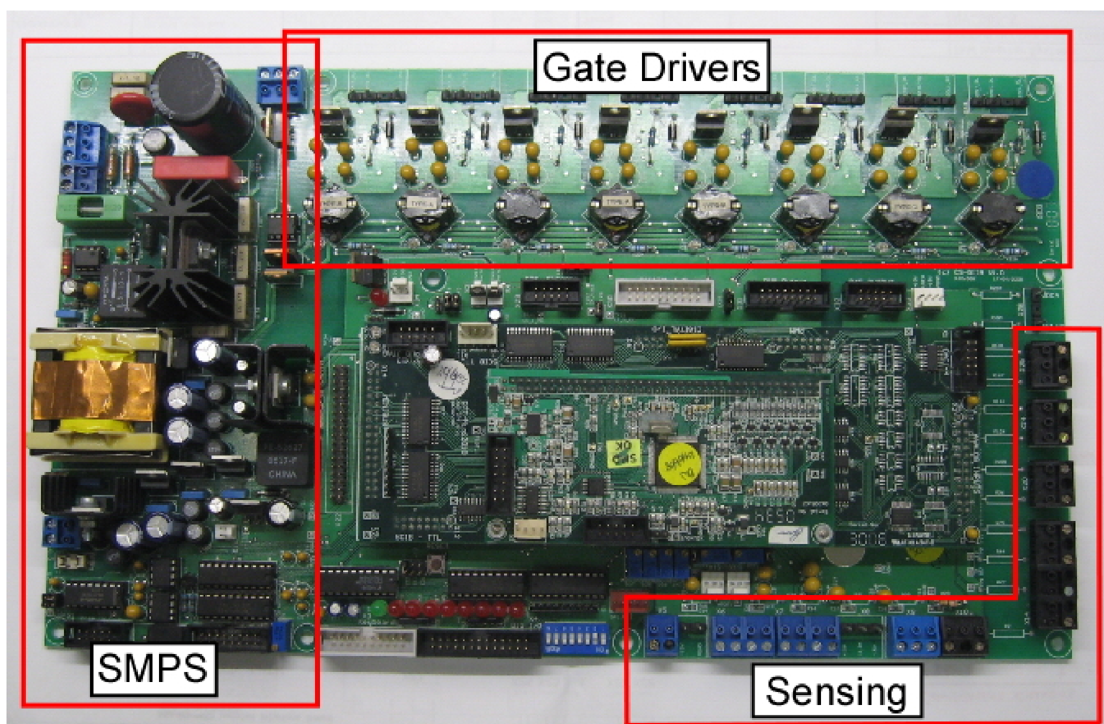


Figure 6.19: GIIB Inverter Board

- **Sensing**
  All voltage and current measurements link to the analog measurement circuitry on the GIIB. This measurement circuitry is primarily made up of op-amp based differential amplifiers [134--136], and is used to scale the incoming analog measurements to levels that the ADCs on the DA-2810 can safely read (0 – 3V). Translating these ADC results back into sensible voltage readings is then done in software.

- **Isolated Serial Communication**
  The user interface to the 2810 DSP is based on serial communications. The GIIB therefore provides TTL/RS-232 voltage level translation as well as optical isolation so that user communications can be achieved safely.

6.2.4 Inter-GIIB Communication

In order for the two GIIB boards to control the prototype DAB converter, inter-processor communication was required. Specifically, the switching signals of the two boards needed to be synchronised, and modulation depth information from the load H-bridge needed to be passed to the DAB converter to help with load current feed-forward. Both communication methods employed are outlined here.

Synchronisation

Synchronising the switching signals between both boards was achieved using a simple Phase Locked Loop (PLL) algorithm. To achieve this, the two boards were set up in Master/Slave configuration, with the DAB control board acting as Master and the VSI control board as slave.

The GIIB board that controlled the VSI generated its own 5kHz triangular carrier for its PWM waveform generation, while the Master DAB control board generated a 5kHz strobe signal based on its own timers. To ensure synchronisation, these two waveforms had to match in both frequency and phase. Therefore, the Master strobe signal was passed to the slave board via shielded ribbon cable, shown in Fig. 6.20. This type of cable was used to help prevent the synchronising signal from being polluted by the switching noise of the converter.

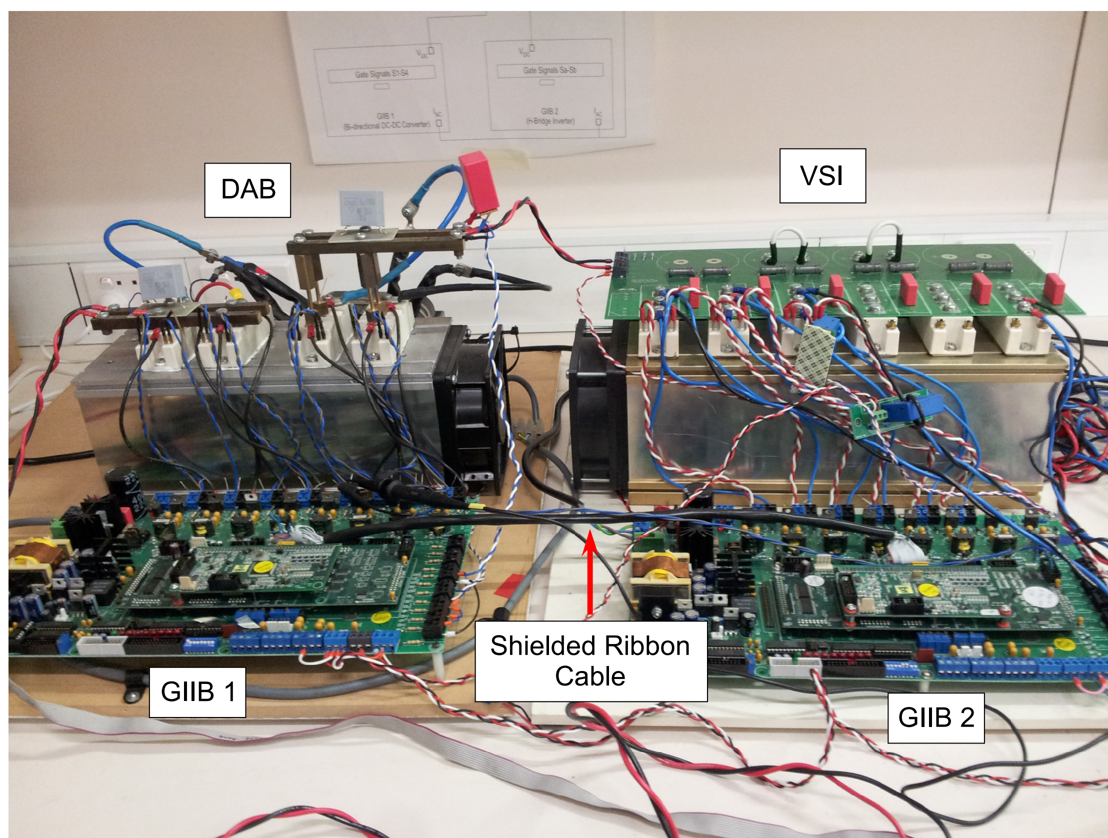


Figure 6.20: Linked GIIB Boards

A Capture port on the slave board logged the timing of the incoming transitions, and used it to determine whether the slave carrier signal was leading or lagging the master strobe signal. The slave would then adjust its timer period as necessary to ensure that the phase of the two waveforms would always match. For example, if the slave lagged behind the master strobe, it would decrease its timer value (increasing frequency), allowing the slave carrier to 'catch up' to the master. The opposite

occurs when the slave leads the master strobe, for the slave timer value would be increased (reducing frequency), until the master strobe 'caught up' with the slave.

This simple method gave a highly stable, well synchronised signal, with less than 800ns of jitter in the carrier waveforms generated by the two boards (see Fig. 6.21). This jitter is less than 0.5% of the full 5kHz carrier interval, which was more than adequate for this system.

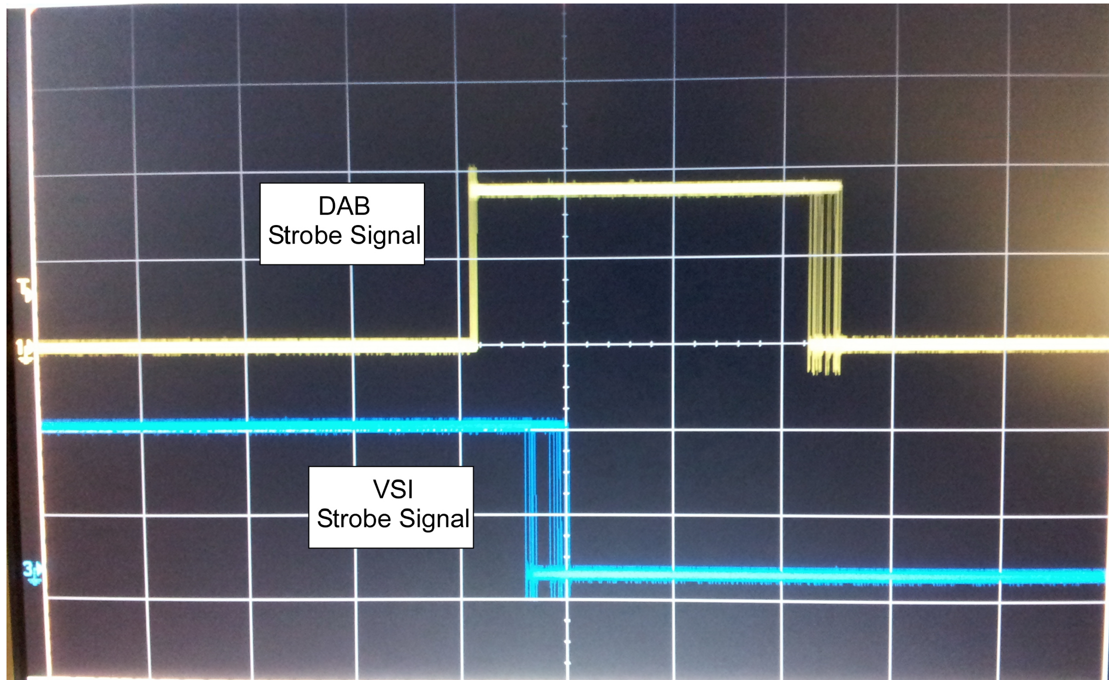


Figure 6.21: Synchronisation Quality between GIIB boards
(x-axis: $2\mu s$/div, y-axis: 2 V/div)

Modulation Depth Information

In Section 4.5.2, it was shown that feed-forward compensation of the load current disturbance was essential to obtain a good load transient response. Measuring this load current correctly is complex when a single-phase H-bridge inverter load is used, because while the sampling technique employed samples the average of the AC current waveform, this is very different to the average DC load current seen by the DAB (see Fig. 5.15).

To determine the average DC load current, the sampled AC current must be scaled by the modulation depth, according to:

$$I_{load_{avg}} = mI_{load} \tag{6.1}$$

Since the DAB and the H-bridge were controlled using two separate GIIB boards, the modulation depth information had to be passed from the H-bridge control board to the DAB. This was achieved by making the VSI control board generate a voltage that was proportional to the generated modulation depth. The Digital-to-Analog Converter (DAC) functionality provided by the 2810 was used to generate this voltage. This voltage was sensed by a voltage sensor on the DAB control board, and the voltage reading scaled back to a modulation depth in software. This allowed the instantaneous VSI modulation depth to be very simply and easily passed to the DAB closed-loop voltage controller, allowing high performance regulation to be achieved.

6.3 Summary

The main features of the PSIM simulations as well as the experimental prototype constructed during the course of this thesis have been presented in this chapter. The building blocks that make up these systems are described, along with the key algorithms that have been implemented to facilitate operation. The results obtained from these simulation and experimental investigations are presented in the following chapter.

Chapter 7

Simulation & Experimental Results

The previous chapter described the simulation and experimental systems used to verify the ideas developed in this thesis. In this chapter, the match between the simulation & experimental results are presented. This validates the major concepts of this thesis as well as the simulation studies that have been presented in this thesis. Some of these results have already been included in previous chapters, but are restated here to provide a complete record of the results obtained.

7.1 Overview

The prototype converter is a two-stage converter made up of a DAB bi-directional DC-DC converter and a single phase VSI that share a common intermediate DC bus, as shown in Fig. 7.1. The salient parameters of this converter are presented in Table 7.1.

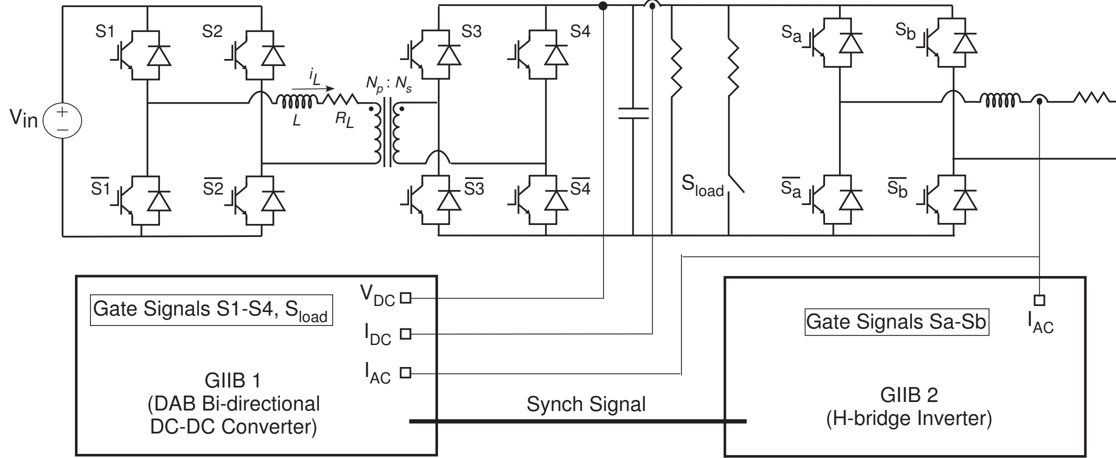


Figure 7.1: Circuit Diagram of the Experimental Prototype

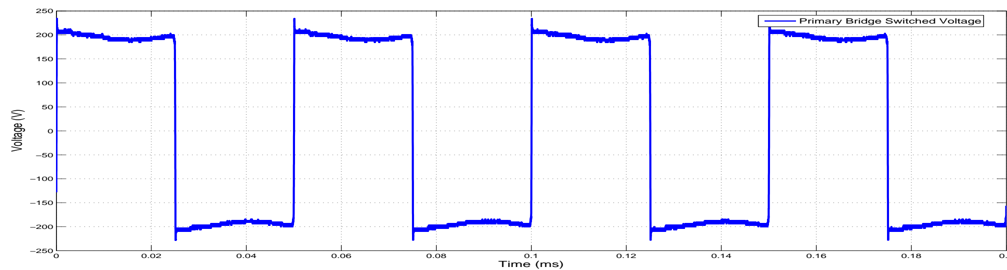
Circuit Parameter		Value
DC Input Voltage	(V_{in})	200 V
DC Output Voltage	(V_{out})	200 V
Transformer Turns Ratio	$(N_p : N_s)$	10 : 11
VSI Switching Frequency	(f_{VSI})	5 kHz
DAB Switching Frequency	(f_{DAB})	20 kHz
DC Capacitance	(C)	12 μF
AC Inductor Inductance	(L)	132 μH
AC Inductor Resistance	(R_L)	0.1 Ω
Output Inductance	(L_{out})	8 mH
Output Load Resistance	(R_{out})	16.5 Ω
Nominal Output Power	(P_{out})	1 kW

Table 7.1: DC-AC Experimental Converter Parameters

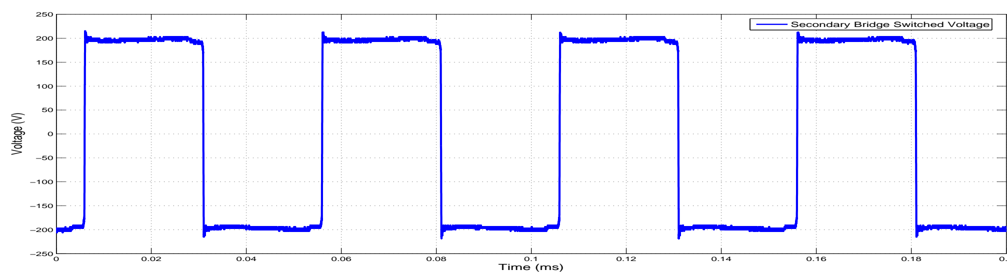
7.2 Steady-state Operating Waveforms

This section presents the essential switching waveforms of the DAB converter. Figs. 7.2a and 7.2b show the PSSW modulation signals employed, with a lagging phase shift clearly visible between the primary and secondary bridges. This matches well with the simulated waveforms of Fig. 3.6. The resulting inductor current (Fig. 7.2c) also has the same features of its simulated counterparts (Fig. 3.6 & 3.10).

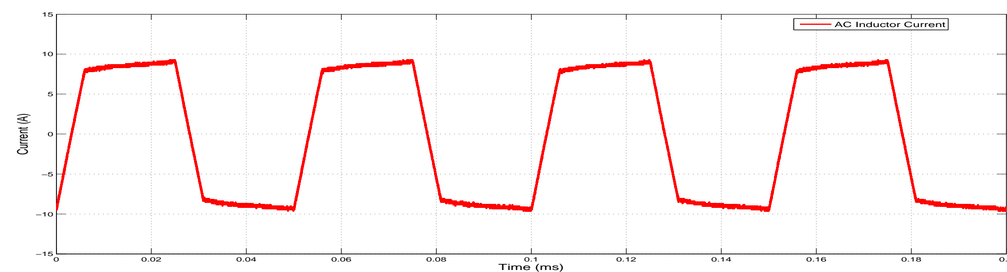
The experimental inductor current (Fig. 7.2c) does differ slightly to the simulated current of Fig. 3.6 & 3.10, but this is only because the parameter differences between the simulated and experimental systems cause a different volt-second average to be applied to the inductor, changing the rate of current change. The experimental DC link voltage is also shown in Fig. 7.2d.



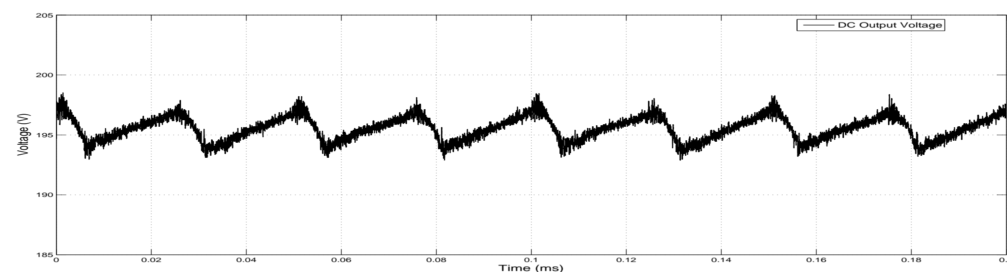
(a) Steady-state Primary Bridge Output Voltage



(b) Steady-state Secondary Bridge Output Voltage



(c) Steady-state AC Inductor Current



(d) Steady-state DC Output Voltage

Figure 7.2: DAB Steady State Operating Waveforms

The waveforms of Fig. 7.3 & 7.4 experimentally demonstrate the effect of deadtime on DAB converter modulation. In both figures, the output voltage of the secondary bridge does not match its modulation signal. Instead the voltage depends on the polarity of the inductor current during the deadtime interval, as predicted in Section 3.6.

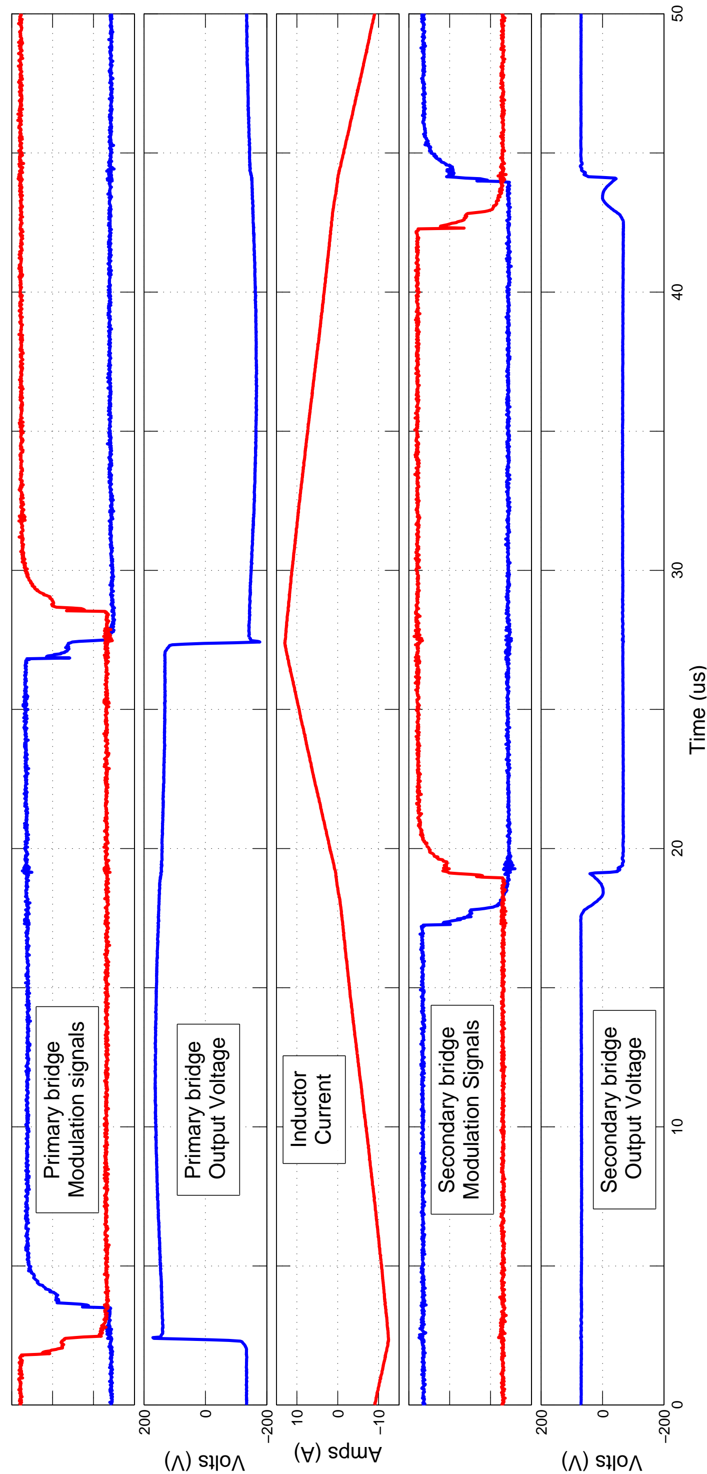


Figure 7.3: Deadtime Effect - HV bridge *Lagging* the LV bridge

These waveforms do not precisely match those of Fig. 3.21 & 3.22 as the analysis of Section 3.6 does not include the effect of IGBT output capacitance. However, the effect of this non-ideal feature is not significant as the device capacitance affects both the rising and falling waveform edges equally, so the applied volt-second average is not significantly altered from the ideal scenario.

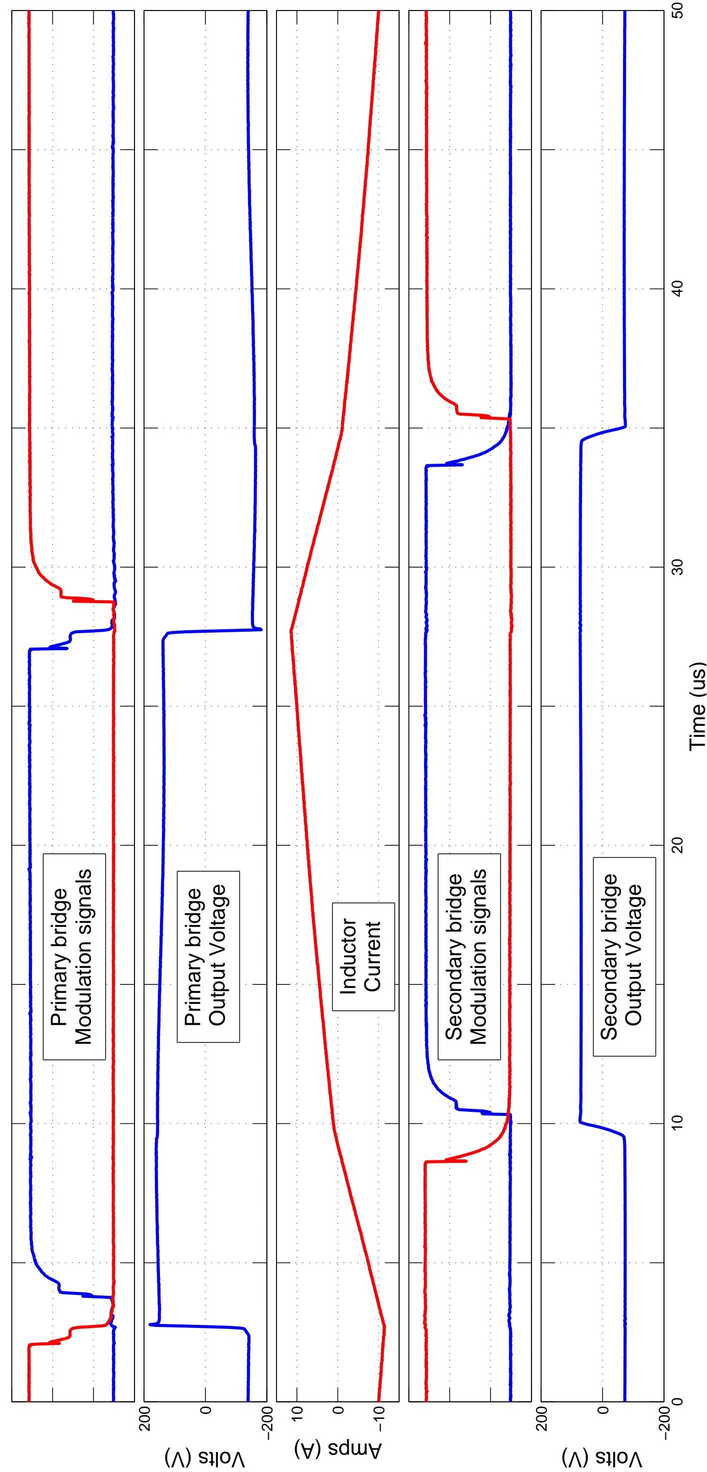
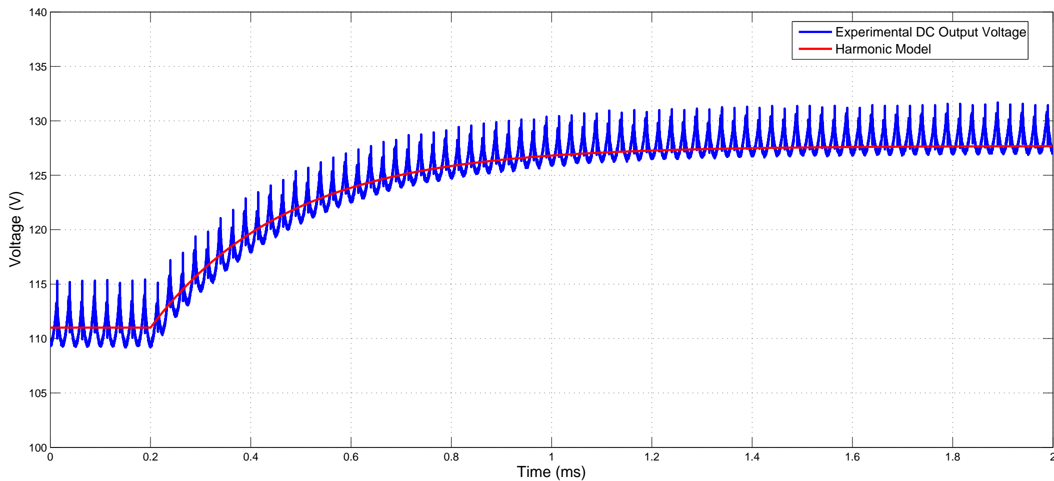


Figure 7.4: Deadtime Effect - HV bridge *Leading* the LV bridge

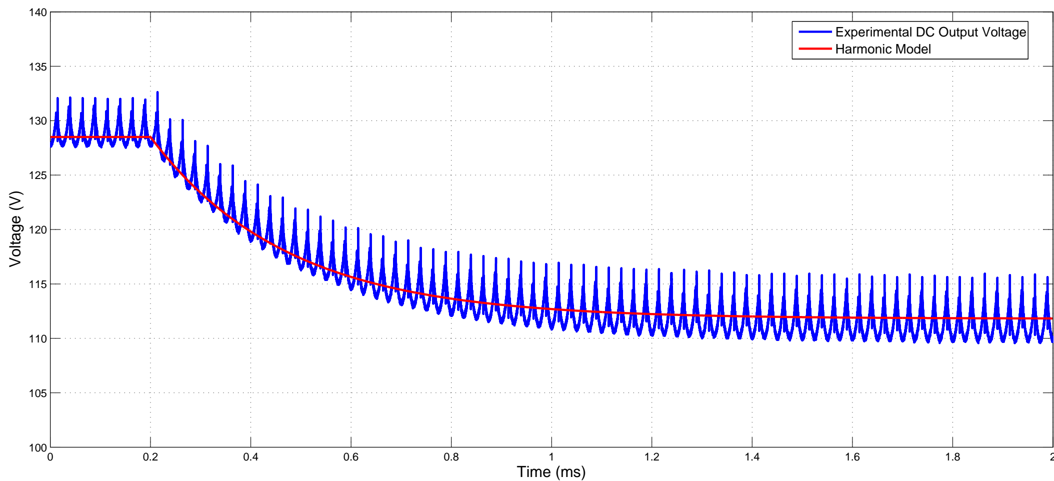
7.3 Open Loop Transients

In this section, the DAB converter is open-loop modulated and fed a step change in phase shift input δ. In each case a step change of $5°$ is applied, and the resulting transient response is compared to the predicted response of the dynamic converter model developed in Chapter 3.

The first set of transient responses are presented in Fig. 7.5, and correspond to an operating point affected by deadtime. The good match between the experimental result and the new dynamic model helps verify its accuracy.



(a) $\delta = 10° \rightarrow 15°$

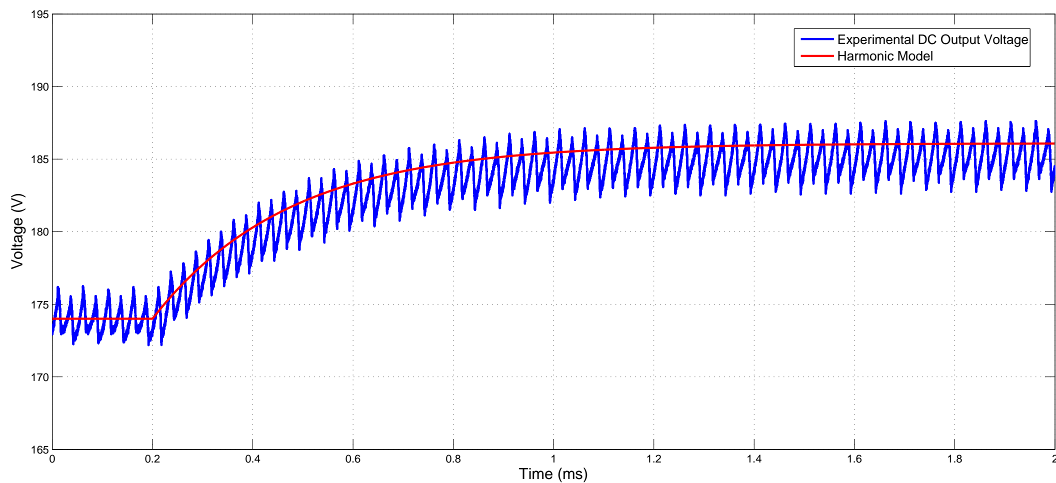


(b) $\delta = 15° \rightarrow 10°$

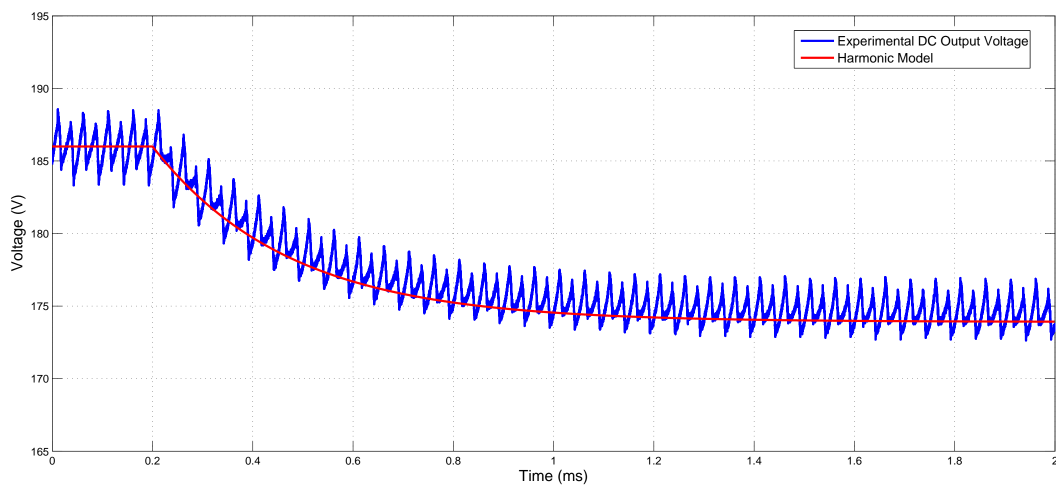
Figure 7.5: DAB Converter Open Loop Step Response – Affected by Deadtime

The second set of open loop transients are obtained at an operating point that is *not* affected by deadtime. Fig. 7.6 presents these responses, and once again the harmonic model provides a good match to these transients.

These transient responses verify the accuracy of the harmonic model and the deadtime compensation algorithm proposed in this thesis, as the resulting dynamic model is able to predict system dynamics at a wide variety of operating conditions.



(a) $\delta = 40° \rightarrow 45°$



(b) $\delta = 45° \rightarrow 40°$

Figure 7.6: DAB Converter Open Loop Step Response – Unaffected by Deadtime

7.4 Closed Loop Transients

In order to verify the performance of the new closed loop Adaptive PI voltage regulator developed in Chapter 4, its response to three types of closed loop transient events is tested, i.e.:

- Voltage Reference Step
- Load Change
- AC Load

The controller gains used for the DAB converter were calculated using the methods presented in Chapter 4, and are summarised in Table 7.2:

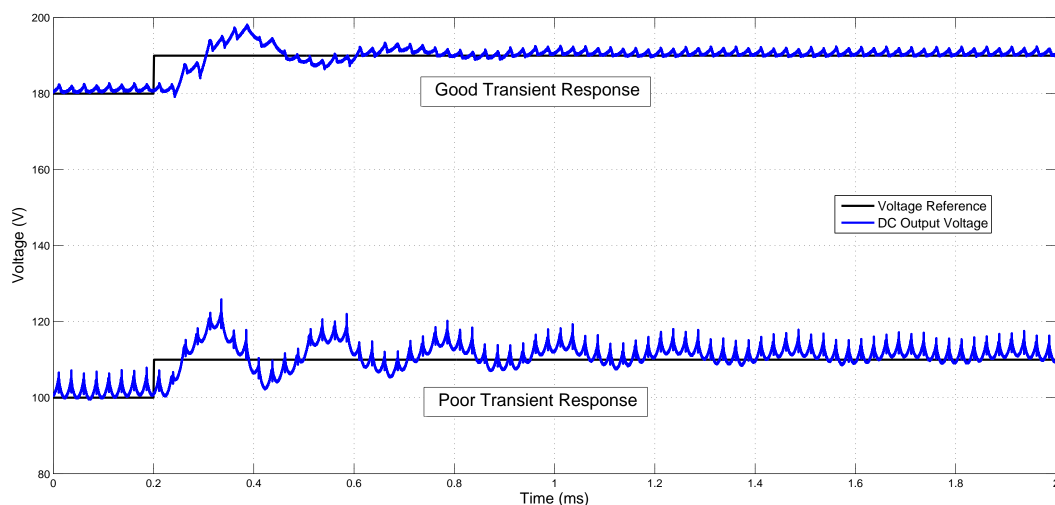
Circuit Parameter		**Value**
Desired Phase Margin	$(\varphi_{m_{DAB}})$	40°
DAB Transport Delay Time	$(T_{d_{DAB}})$	50 μs
DAB Controller Bandwidth	$(\omega_{c_{DAB}})$	2778 Hz
Maximum DAB Prop. Gain	$(K_{p_{DAB_{max}}})$	0.0236
Minimum DAB Prop. Gain	$(K_{p_{DAB_{min}}})$	0.2905
DAB Integrator Time Constant	$(T_{r_{DAB}})$	3.6 ms

Table 7.2: DAB Voltage Regulator Controller Parameters

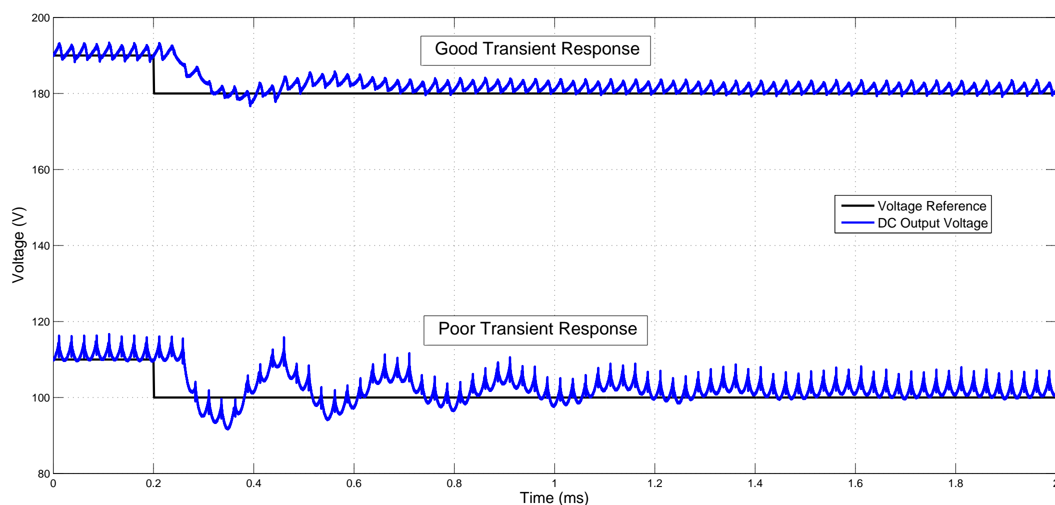
7.4.1 Voltage Reference Step

The transient response of a DAB converter feeding a fixed load resistance (29Ω) was recorded when a step change in voltage reference is applied.

To illustrate the need for an adaptive gain, the variation in performance that occurs when fixed controller gains are used is shown in Fig. 7.7. These waveforms show the output voltage response when a PI controller with fixed gains is employed, and show that although a good transient performance is achieved for the 190 V step change, clear degradation in performance is observed at the 100 V step change.



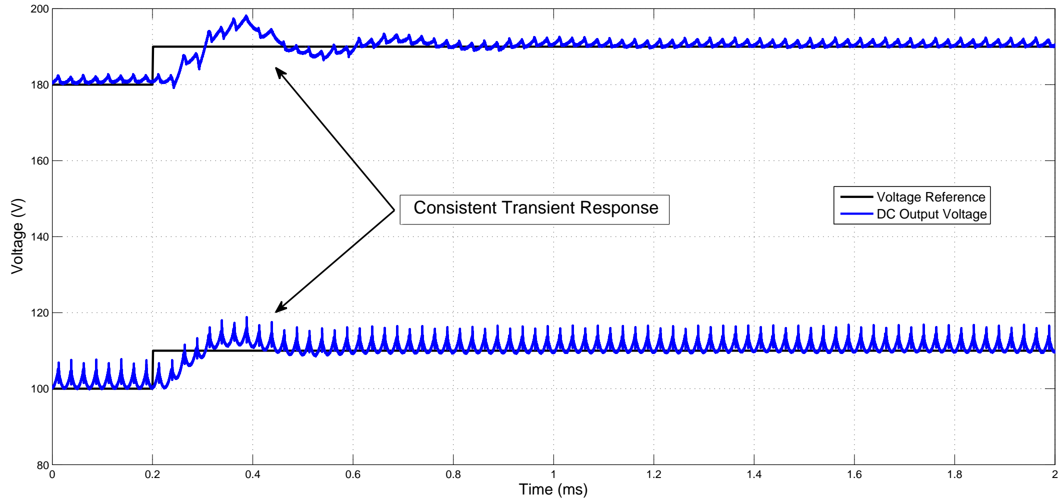
(a) Increasing Voltage Reference



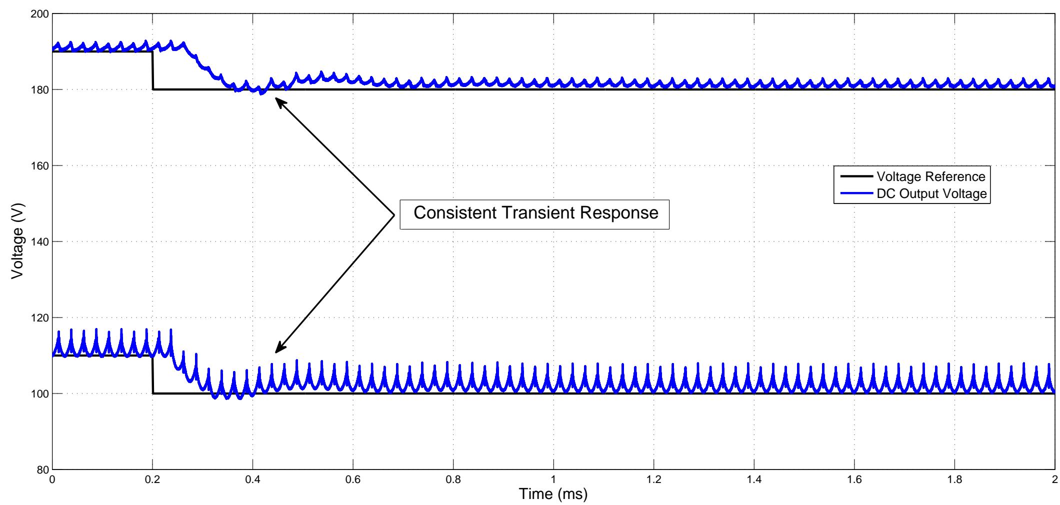
(b) Decreasing Voltage Reference

Figure 7.7: DAB Closed loop Transient Response - Fixed PI gains

The proposed Adaptive PI voltage regulator significantly improves this response. The transient responses of Fig. 7.8 show a consistent level of performance at all operating points, unlike those in Fig. 7.7.



(a) Increasing Voltage Reference



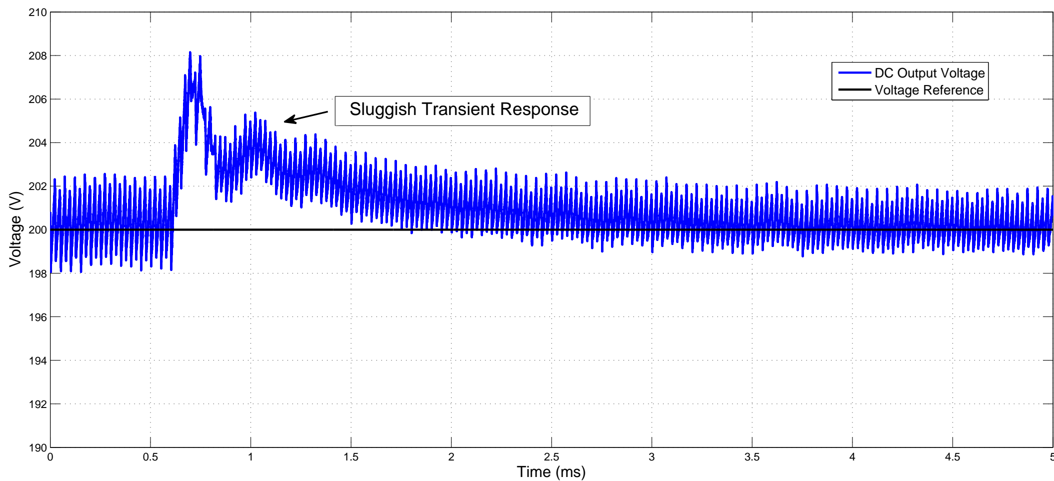
(b) Decreasing Voltage Reference

Figure 7.8: DAB Closed loop Transient Response - Adaptive PI gains

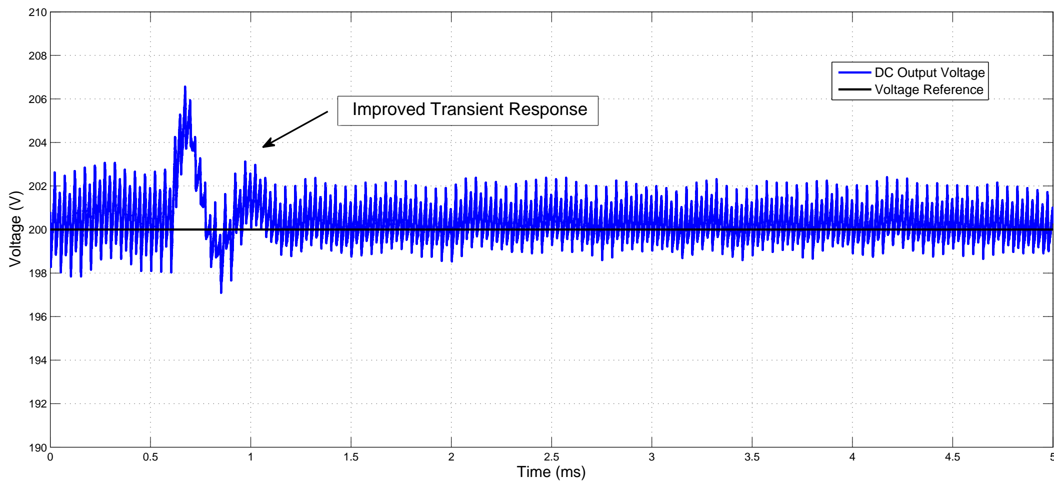
7.4.2 Load Change

The analysis presented in Chapter 4 suggests that an Adaptive PI regulator is insufficient to manage a load transient event, and proposes feed-forward compensation to improve converter response. To test this, the second type of transient event that was applied to the closed loop DAB converter was a change in load resistance. For these tests, a constant output voltage was commanded, and a step change in DC load resistance was applied ($38.4\Omega \leftrightharpoons 32.9\Omega$).

Fig. 7.9 shows the transient responses caused by a load decrease ($32.9\Omega \rightarrow 38.4\Omega$). Fig. 7.9a shows the sluggish output voltage transient response achieved without feed-forward, while Fig. 7.9b shows the significant improvement achieved when feed-forward compensation is included, giving a faster dynamic response.



(a) Adaptive PI Control

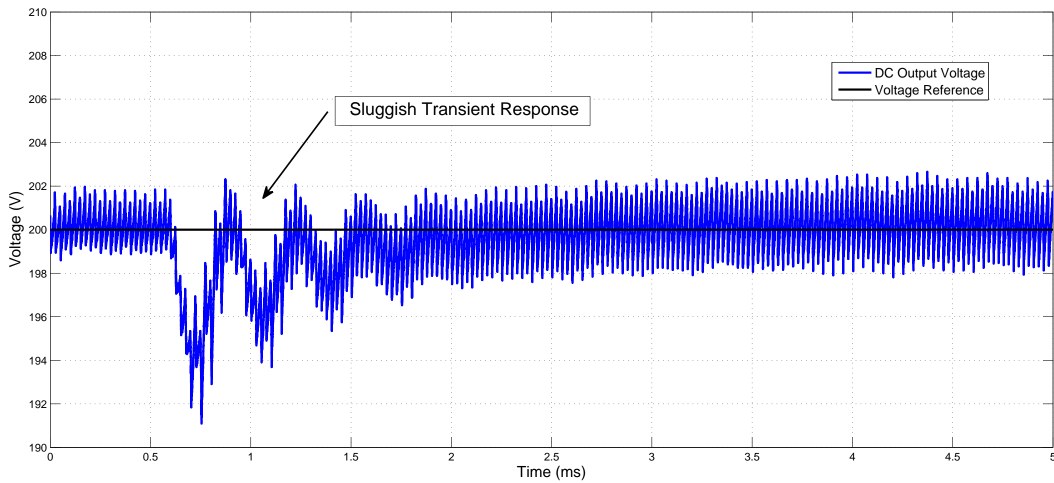


(b) Adaptive PI + Feed-forward Control

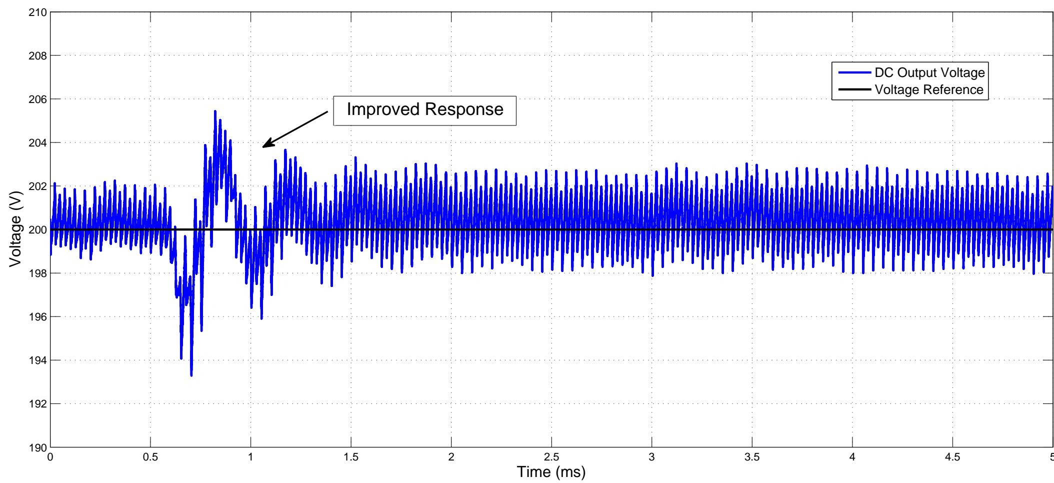
Figure 7.9: DAB Closed loop Transient Response - Load Reduction

These transient tests were repeated for a load increase (38.4Ω → 32.9Ω). Once again, a sluggish output response is seen without feed-forward (Fig. 7.10a), but this response is significantly improved when feed-forward is included (see Fig. 7.10b).

Additionally, the more oscillatory response predicted in Chapter 4 for an increase in load is also visible in Fig. 7.10. This undesirable response, caused by the large variation in plant and controller gains during the transient event, is also minimised when feed-forward compensation is applied.



(a) Adaptive PI Control



(b) Adaptive PI + Feed-forward Control

Figure 7.10: DAB Closed loop Transient Response - Load Increase

7.4.3 AC Load

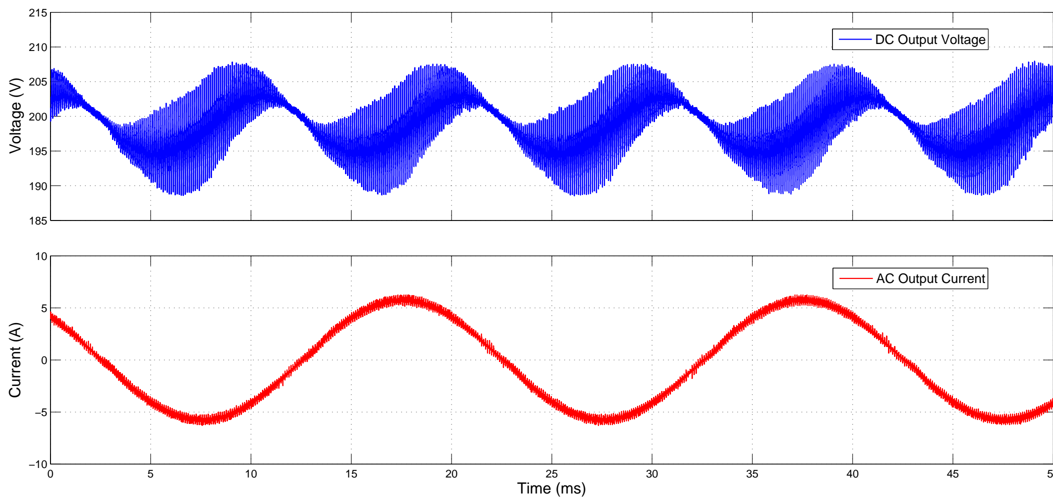
The final load condition applied to the DAB converter was an AC load. This was described in Chapter 5, and achieved by connecting the DC output of the DAB converter to a H-bridge inverter that fed an R-L load (see Fig. 7.1). To emulate a grid-connected system, a relatively large load resistance was used so that the modulation depth achieved by the converter would be comparable to those demonstrated in Chapter 5. This H-bridge was controlled with a PI current regulator, whose parameters are listed in Table 7.3.

Circuit Parameter		**Value**
Desired Phase Margin	$(\varphi_{m_{VSI}})$	40°
VSI Transport Delay Time	$(T_{d_{VSI}})$	150 μs
VSI Controller Bandwidth	$(\omega_{c_{VSI}})$	926 Hz
VSI Proportional Gain	$(K_{p_{VSI}})$	0.1454
VSI Integrator Time Constant	$(T_{r_{VSI}})$	10.8 ms

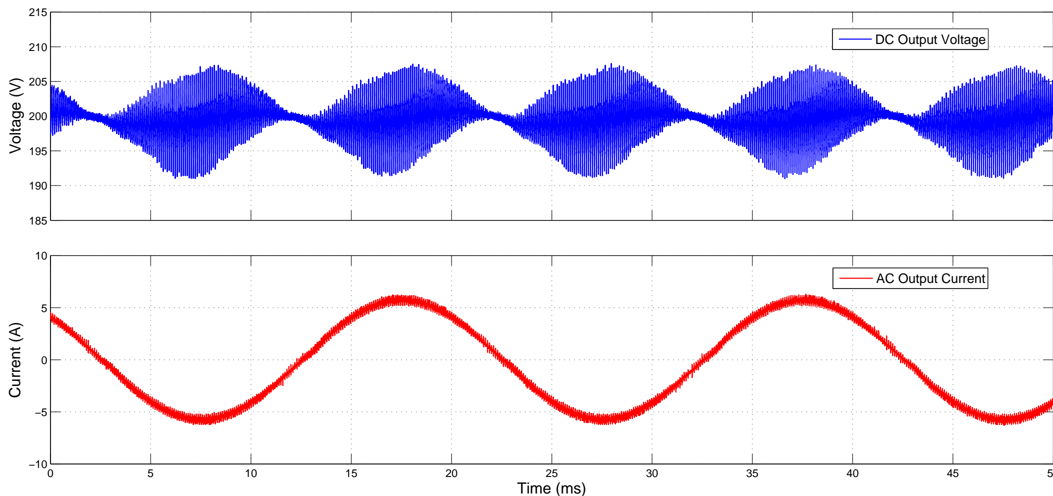
Table 7.3: H-bridge Current Regulator Parameters

The first AC load test applied to the DAB converter was a constant AC output current. The new Adaptive PI controller maintained a constant 200 V DAB converter output voltage, and a constant 6 A peak AC load current was drawn from the H-bridge. Fig. 7.11a shows the DC bus voltage rippling due to oscillations in the load current, as predicted by the analysis presented in Chapter 5. The envelope of the experimental voltage ripple differs slightly to the simulation analysis presented in Chapter 5 because the simulation results presented were obtained at the worst-case scenario of near zero power factor, while the experimental results were obtained at a more realistic power factor that is closer to unity.

Feed-forward compensation was then used to reject the disturbance caused by the load current, and the improved performance is plotted in Fig. 7.11b. The low frequency AC oscillations in the DAB output voltage are eliminated, verifying the control ideas presented in Chapter 4.



(a) Adaptive PI Control

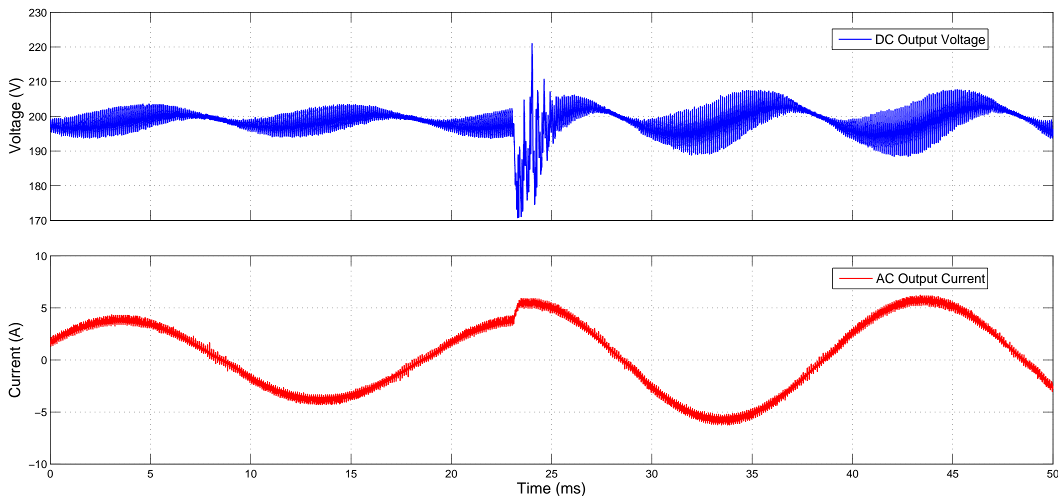


(b) Adaptive PI + Feed-forward Control

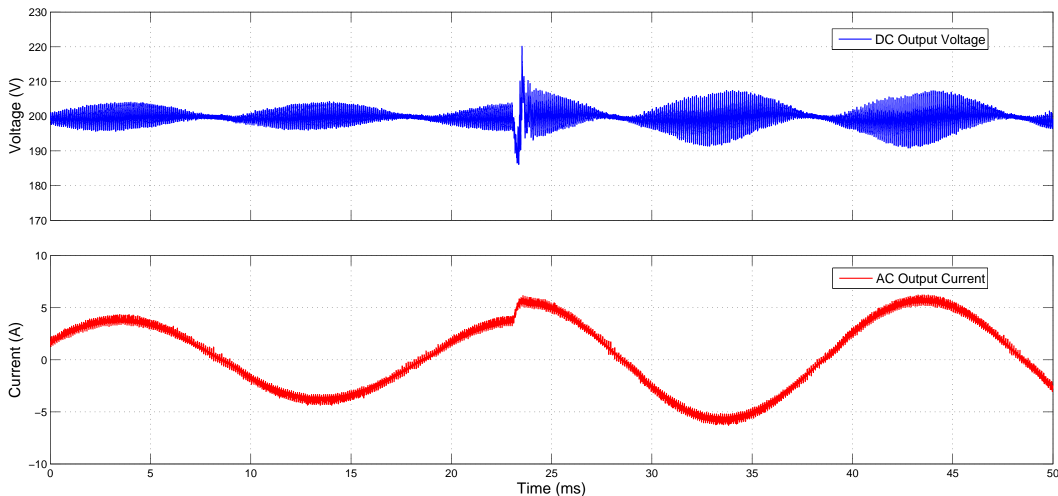
Figure 7.11: Experimental Steady State Waveforms - Steady State AC Load

The final load test applied to the DAB converter was a step change in AC load, achieved by commanding a step change in the output AC current (4 A $\leftrightharpoons$ 6A).

The effect of a step change of (4 A $\rightarrow$ 6A) is presented in Fig. 7.12. The effect of the oscillating AC load current on the DC output voltage (apparent in Fig. 7.12a), is once again eliminated using feed-forward compensation in Fig. 7.12b. The transient response caused by the step change in AC current is also shown in these figures. A relatively oscillatory response is seen without feed-forward compensation (Fig. 7.12a). These oscillations are due to the large variations in plant characteristics and controller gains seen during a load transient event, and are clearly inadequate. When feed-forward compensation is enabled, this response improves considerably, as the output voltage is far less oscillatory, and returns to steady state within 5 H-bridge switching cycles.



(a) Adaptive PI Control



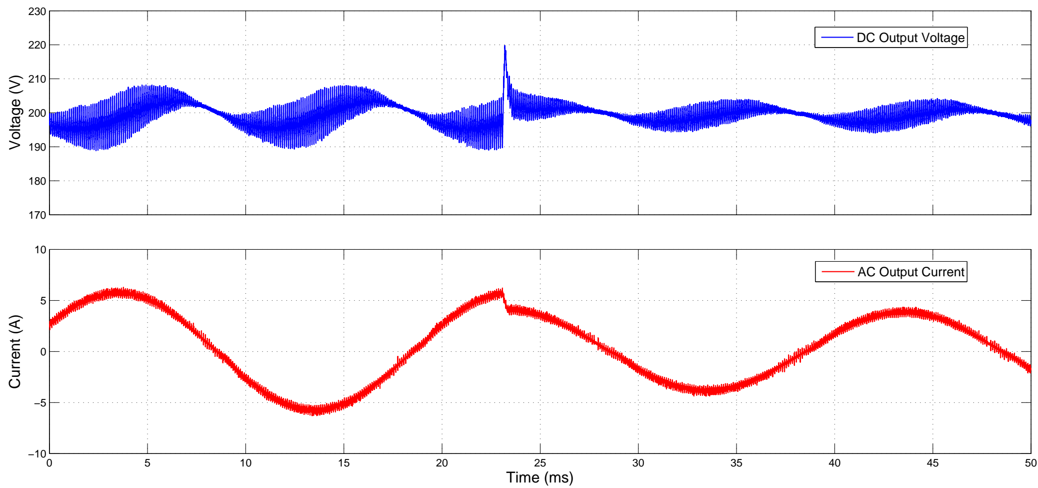
(b) Adaptive PI + Feed-forward Control

Figure 7.12: Experimental Transient Waveforms - AC Load Step (4A $\leftrightharpoons$ 6A)

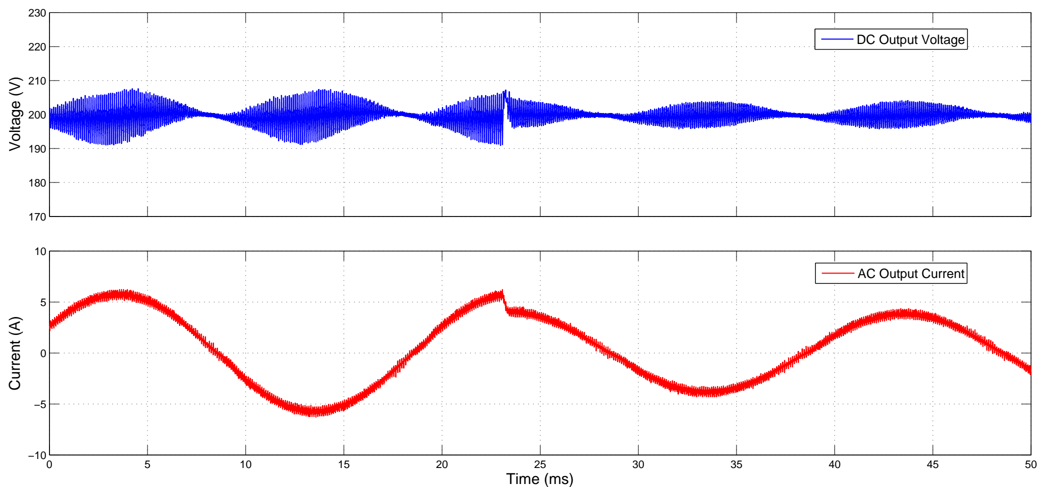
The response of a 6 A → 4 A step is also observed, and plotted in Fig. 7.13. Once again, feed-forward compensation minimises the transient voltage excursion validating the control principles presented in this thesis.

7.5 Summary

The experimental results in this chapter verify the ideas and algorithms presented in this thesis. The new harmonic model is proven to accurately predict DAB converter dynamic behaviour, and the Adaptive PI controller achieves consistently high performance across the entire dynamic range. The proposed load current feed-forward strategy also ensures a fast load transient response for both DC and AC loads. These excellent results prove that this thesis has attained its objective – achieving high performance bi-directional DC-DC conversion for a grid-connected application.



(a) Adaptive PI Control



(b) Adaptive PI + Feed-forward Control

Figure 7.13: Experimental Transient Waveforms - AC Load Step (6A ⇋ 4A)

Chapter 8

Conclusions

Bi-directional DC-DC converters have been the focus of power electronic research for over twenty years, but more recently they have been identified as a key technology for the emerging Smart Grid. Optimising grid operation requires high performance regulation of these converters, but existing literature is yet to address this issue, and no clear definition of the maximum achievable performance has been made.

Existing closed loop control strategies for DC-DC converters do not usually guarantee maximised performance, and do not ensure a consistent transient response across the operating range. The models that have been used to design these controllers are also limited as they are often inaccurate, and do not accommodate the effects of non-ideal converter features such as deadtime.

The work in this thesis presents significant advances in these fields by developing a new dynamic model for this converter based on harmonic analysis and using it to derive a better closed loop regulator. The simplicity of this new harmonic model makes it attractive for closed loop design purposes, and its accurate prediction of converter dynamics that also include the effect of deadtime make it extremely powerful. This model is then employed to support the derivation of a new high performance closed loop regulator. This regulator can achieve high performance for transient changes in both reference command and load condition, and ensures consistent performance across the entire dynamic range.

This concluding chapter summarises the main findings and outcomes of this research and also presents a discussion of avenues for future work.

8.1 Contributions

8.1.1 Harmonic Model

The first major contribution of this thesis is the new harmonic model, which accurately determines the dynamic response of the DAB converter based on its switching functions. The contribution of each significant harmonic of the modulation function is summed together to give the overall converter dynamic response. From this model, it is shown that the DAB converter can be modelled as a linear first-order system. However, since the model coefficients are operating point dependent, plant characteristics vary significantly across the operating range.

8.1.2 Deadtime Modelling

The second major contribution of this thesis is the analytic prediction of the effect deadtime has on DAB dynamics. This is achieved by first identifying that the AC inductor current that flows during the deadtime period can cause the phase shift seen between the two bridges of the DAB converter to differ from the commanded phase shift. This phase shift error changes the converter operating point, altering the dynamic response.

Since the phase shift error is dependent on the AC inductor current, a series of piecewise linear equations that describe its behaviour across the entire switching cycle were developed. Since the current is cyclic and symmetric in nature, these equations form a closed-form expression for this current that is used to analytically determine the phase shift error effect caused by deadtime. Including the predicted effect of deadtime in the harmonic model gives a highly accurate dynamic model of the DAB converter that was verified both in simulation as well as on the experimental prototype.

8.1.3 Maximised controller gains

The third major contribution of this thesis is the identification that controller gains for the DAB converter are primarily limited by transport delay. Transport delay is a feature of the digital implementation of the converter modulator and regulator. Specifically, the sampled nature of digital control systems and the non-zero computation times of control loop calculations both introduce a delay into the regulator that degrades controller performance. Since these delays are deterministic in nature, the effect they have on controller gains is precisely identified in this

thesis, allowing controller gains to be calculated that achieves the best possible performance.

8.1.4 Adaptive controller gains

The fourth contribution of this thesis is an adaptive gain calculation algorithm that ensures consistent performance across the operating range. The harmonic model predicts significant variation in plant characteristics as operating point changes, which can cause closed loop performance to vary as well. Adapting controller gains with operating point allows consistent transient performance to be achieved across the entire dynamic range.

8.1.5 Improved Load Transient Response

The fifth contribution of this thesis is the use of feed-forward compensation to improve the converter response to a load transient event. It was identified that the load current acted as a disturbance to the closed loop system, which significantly degraded load transient performance. Disturbance rejection in the form of a feed-forward command was used to compensate for the effect of this disturbance, resulting in a significantly improved load transient response.

8.1.6 AC Load Condition

The sixth contribution of this thesis is the application of the new closed loop controller to AC load conditions. Usually the AC oscillating power flow is absorbed by the intermediate capacitor, so the DAB sees constant DC power flow. However, the bulk capacitance this requires usually means this capacitor is an electrolytic, and the short lifetime of this component is a significant disadvantage. This thesis shows how high performance voltage regulation can be used maintain the DC bus voltage, reducing the required capacitance. This means that the electrolytic capacitor can be eliminated and replaced with longer lasting film capacitors. This has significant size and lifetime benefits.

8.2 Future Work

This thesis has dealt with the optimised modelling and closed loop regulation of the DAB converter, but there is still significant scope for further research in this area.

8.2.1 Multiport Converters

The modelling and closed loop regulation ideas presented in this thesis have only been applied to a Dual Active Bridge topology, so an extension to multiport converters is a clear direction for future research. Regulation of these systems is more complex than the standard DAB topology because additional control objectives must also be met, such as guaranteed current sharing and minimised circulating energy between each port. Applying the harmonic model to these converters and maximising their closed loop performance has not yet been considered.

8.2.2 Magnetics Design

A major limitation for practical bi-directional DC-DC converters is the design and construction of its magnetic components. Although a popular research area, the design of high-powered, high frequency inductors and transformers is still very complex. There is therefore considerable scope for developing magnetic component design criteria to achieve an optimised design in terms of weight, size, efficiency and cost.

8.2.3 Extending the Harmonic Model

The ideas presented in this thesis are limited to a two-level, hard-switched PSSW modulation scheme. Three-level modulation strategies and soft-switching modulation techniques have been presented in the literature and predict possible efficiency benefits, but have not been considered in this thesis. There is therefore significant scope for research in this area, as the harmonic modelling technique has not yet been applied to converters that employ these modulation schemes.

Additionally, the dynamic model presented in this thesis assumes ideal switching devices, but practical devices include non-ideal features such as device voltage drops and diode reverse recovery effects. A clear research path therefore exists to enhance the model by including these non-ideal effects in the harmonic model and the deadtime compensation algorithm.

8.2.4 Controller Performance

This thesis has identified that transport delay is the primary limiting factor for DAB controller performance. Several techniques for minimising transport delay exist, such as asymmetric sampling and multi-sampling, but have not been applied in this thesis [122, 129]. Applying these techniques to the DAB converter is an obvious direction for future research as it has the potential to increase the achievable controller bandwidth, further improving closed loop performance.

8.3 Closure

High performance closed loop regulation of bi-directional DC-DC converters is a key requirement for the modern Smart Grid. This need for a fast transient response and good steady-state tracking across the entire operating range has driven this research towards maximising this performance.

A new powerful dynamic modelling technique has been presented based on converter switching harmonics. This technique can be applied to any switching converter, and successfully applied in this thesis to the DAB converter. The non-linear effect of deadtime on this converter has also been analytically modelled, allowing dynamic behaviour during this period to be precisely determined. The effects of a digital control implementation have also been identified, and the maximum controller gains that can achieved by these controllers calculated. A novel strategy for ensuring consistent transient performance for changes in both reference command and load condition is also developed and tested in a variety of conditions.

This closed loop regulator has met the goal of this thesis – high performance bi-directional DC-DC conversion for a Smart Grid application.

Appendix A

Simulation & Experimental Code

This appendix presents the program code that was used to control the simulated and experimental systems that were developed during the course of this thesis.

This chapter is divided into two sections, i.e. the simulation program code & the experimental program code. The simulation code is used in the Dynamic Link Library (DLL) blocks employed by PSIM, while the experimental code comprises the 'C' code used by the Texas Instruments TMS320F2810 Digital Signals Processor as well as the VHDL code used by the Altera MAX II EPM570T100C5N CPLD.

A.1 Simulation Code

The code used in the PSIM simulations of the DAB converter is included here:

```
1 /****************** Standard PSim DLL readme **********************************/
2
3 // This is a sample C program for Microsoft C/C++ 5.0 or 6.0.
4 // The generated DLL is to be linked to PSIM.
5
6 // To compile the program into DLL, you can open the workspace file "msvc_dll.dsw"
7 // as provided. Or you can create a new project by following the procedure below:
8
9 //    - Create a directory called "C:\msvc_dll", and copy the file "msvc_dll.c"
10 //      that comes with the PSIM software into the directory C:\msvc_dll.
11 //
12 //    - Start Visual C++. From the "File" menu, choose "New". In the "Projects"
13 //      page, select "Win32 Dynamic-Link Library", and set "Project name" as
14 //      "ms_user0", and "Location" as "C:\msvc_dll". Make sure that
15 //      "Create new workspace" is selected, and "Win32" is selected under
16 //      "Platform",
17 //      .
18 //
19 //    - [for Version 6.0] When asked "What kind of DLL would you like to create?",
20 //      select "An empty DLL project.".
21 //
22 //    - From the "Project" menu, go to "Add to Project"/"Files...", and select
23 //      "msvc_dll.c".
24 //
25 //    - Add your own code as needed.
26 //
27 //    - From the "Build" menu, go to "Set Active Configurations...", and select
28 //      "Win32 Release". From the "Build" menu, choose "Rebuild All" to generate
```

```
29 //      the DLL file "msvc_dll.dll". The DLL file will be stored under the
30 //      directory "C:\msvc_dll\release".
31 //
32 //    - Give a unique name to the DLL file. For example, if your schematic file
33 //      is called "test msvc_dll.sch", you can call it "test_msvc_dll.dll".
34 //
35 //    - Copy the renamed DLL file into the same directory as the schematic file.
36 //      In the circuit, specify the external DLL block file name as the one
37 //      you specified (for example, "test_msvc_dll.dll" in this case). You are
38 //      then ready to run PSIM with your own DLL.
39
40 // This sample program calculates the rms of a 60-Hz input in[0], and
41 // stores the output in out[0].
42
43 // Activate (enable) the following line if the file is a C++ file (i.e. "msvc_dll.cpp")
44 //extern "C"
45
46 // You may change the variable names (say from "t" to "Time").
47 // But DO NOT change the function name, number of variables,  variable type, and sequence.
48
49 // Variables:
50 //      t: Time, passed from PSIM by value
51 //   delt: Time step, passed from PSIM by value
52 //     in: input array, passed from PSIM by reference
53 //    out: output array, sent back to PSIM (Note: the values of out[*] can
54 //         be modified in PSIM)
55
56 // The maximum length of the input and output array "in" and "out" is 20.
57
58 // Warning: Global variables above the function ms_user0 (t,delt,in,out)
59 //          are not allowed!!!
60
61
62 /****************************** Read me for this file **********************************/
63
64 //Controller simulation for Grid Connected Bidirectional DC-DC Converter
65 //02/03/2011
66
67 //The topology in question is a single phase Bidirectional DC-DC Converter which feeds the DC link
68 //of a H-bridge connected to the grid.
69
70 //The Bidirectional DC-DC Converter is PI controlled with an Adaptive PI controller with
71 //Feed-forward compensation of load current.
72
73 //The Hbridge is current regulated, and can change power factor on command. This is done by changing
74 //the phase of the desired output, referenced to the Grid AC.
75
76 //This code will modulate, sense and control
77
78 //DLL of the code which is used by PSim is generated in the "debug" folder,
79 //so the corresponding simulation should also be placed in that folder.
80
81 #include <math.h>
82
83 /*******************
84 _hash_definitions()
85 *******************/
86
87 //For Fixed Point
88 #define int16 short
89 #define Uint16 unsigned short
90 #define int32 long
91 #define Uint32 unsigned long
92
93 //fxed point scaling
94 #define FIXED_Q            10    //11
95 #define FIXED_Q_SCALE      1024.0//2048
96 #define SMALL_Q             14
97 #define SMALL_Q_SCALE       16384.0
98
99 //constants
100 #define SQRT3_ON2           FIXED_Q_SCALE*(0.866025403784439)               // 65536*sqrt(3)/2
101 #define INV_SQRT3           FIXED_Q_SCALE*(0.577350269189626)               // 65536/sqrt(3)
102 #define PI                 3.14159265358979
103 #define _2PI               2*PI
104 #define PI_2               1.57079632679489
105 #define INV_PI             0.31830988618379
106 #define INV2_PI             0.636619772367581
107 #define  PI_FIXED           (long)(PI*FIXED_Q_SCALE)
108 #define  PI_2_FIXED         (PI_FIXED>>1)
```

```
109 #define DEG_TO_RAD          PI/180.0
110
111 //For DSP emulation
112 #define HSPCLK               (150e6)
113 #define MIN_VSI_TIME        1e-6
114 #define MIN_VSI_COUNT        (HSPCLK*MIN_VSI_TIME)
115 #define MAX_VSI_TIME        (int16)(PERIOD_2_VSI-MIN_VSI_COUNT)
116 #define SIN_TABLE_SIZE      512
117
118 //vsi parameters
119 #define SW_FREQ_VSI          (5000.0)
120 #define PERIOD_2_VSI        ((Uint16)((HSPCLK/SW_FREQ_VSI)/4.0))
121 #define PERIOD_VSI          ((Uint16)(2*PERIOD_2_VSI))
122 #define FSAMPLE_VSI          (SW_FREQ_VSI*2.0)
123 #define TSAMPLE_VSI          (1.0/FSAMPLE_VSI)
124 #define T_DELAY_VSI          (1.5*TSAMPLE_VSI)
125 #define F_FUND              50.0
126 #define OMEGA_FUND          (2*PI*F_FUND)
127 #define OMEGA_C_VSI          (PI_2-(40*DEG_TO_RAD))/(T_DELAY_VSI)
128 #define KP_VSI              (OMEGA_C_VSI*LVSI/VIN)
129 #define KI_VSI              (OMEGA_C_VSI/10.0)
130
131 //BiDC parameters
132 #define SW_FREQ_BIDC        (20000.0)
133 #define TS_BIDC              ((double)(1.0/SW_FREQ_BIDC))
134 #define OMEGA_BIDC          2.0*PI*SW_FREQ_BIDC
135 #define PERIOD_2_BIDC        ((Uint16)((HSPCLK/SW_FREQ_BIDC)/4.0))
136 #define PERIOD_BIDC          (2*PERIOD_2_BIDC)
137 #define FSAMPLE_BIDC        (1.0*SW_FREQ_BIDC)
138 #define TSAMPLE_BIDC        (1.0/FSAMPLE_BIDC)
139 #define MAX_PHASE            (PERIOD_2_BIDC-1)
140 #define T_DELAY_BIDC        (1.0*TSAMPLE_BIDC)
141 #define OMEGA_C_BIDC        (PI_2-(60*DEG_TO_RAD))/(T_DELAY_BIDC)    //50 deg phase margin
142 #define OMEGA_C_10_BIDC      (OMEGA_C_BIDC/10.0)
143 #define OMEGA_C_BIDC_FIXED  ((int32)(OMEGA_C_BIDC*SMALL_Q_SCALE))
144 #define PERIOD_SCALE_BIDC   ((int32)(PERIOD_2_BIDC*INV2_PI))
145
146 //Topology parameters
147 #define C                  20e-6
148 #define L                  50e-6
149 #define R_L                0.1
150 #define R_L_2              R_L*R_L
151 #define LVSI              (5e-3)
152 #define KP_CONST          (int32)(LVSI*OMEGA_C_VSI*FIXED_Q_SCALE)
153 #define OMEGA_BIDC_L      (OMEGA_BIDC*L)
154 #define OMEGA_BIDC_L_2    (OMEGA_BIDC_L*OMEGA_BIDC_L)
155 #define NPRI              (10.0)
156 #define NSEC              (15.0)
157 #define NPRI_NSEC          ((double)(NPRI/NSEC))
158 #define NPRI_NSEC_FIXED   ((int32)(NPRI_NSEC*FIXED_Q_SCALE))
159 #define VIN                (200.0)
160 #define _4VIN              (4.0*VIN)
161 #define VIN_FIXED          ((int32)(VIN*FIXED_Q_SCALE))
162 #define VP_FIXED          ((int32)(VIN*FIXED_Q_SCALE/2.0))
163 #define VDCPRI            (VIN/2.0)
164 #define VBUS_NOM          VIN
165 #define VBUS_NOM_FIXED    ((int32)(VIN*FIXED_Q_SCALE))
166
167 //Adaptive controller parameters
168 #define VDC_KP_INIT        0.005
169 #define VDC_KP_MAX        0.04
170 #define VDC_KP_MIN        0.001
171 #define VDC_KP_MAX_FIXED  (int32)(VDC_KP_MAX*SMALL_Q_SCALE)
172 #define VDC_KP_MIN_FIXED  (int32)(VDC_KP_MIN*SMALL_Q_SCALE)
173 #define VDC_KP_INIT_FIXED  (int32)(VDC_KP_INIT*SMALL_Q_SCALE)
174 #define DELF_DELU_CONST    (VDCPRI*NPRI_NSEC/(C*PI*PI))    //divide by 16.0 is for scaling purposes
175 #define DELF_DELX_CONST    ((-8.0*NPRI_NSEC*NPRI_NSEC)/(C*PI*PI))
176 #define BIDC_FF_CONST      ((16.0*VDCPRI*NPRI_NSEC/(PI*PI))/OMEGA_BIDC_L)
177
178 //the phase step is the difference in phase between two switching cycles.
179 //That is a 50Hz sin wave, switched at 5kHz, sampled at 10kHz. so the switching is 10kHz/50Hz faster.
180 //the switching is therefore 200x faster than the fundamental. so the phase step is 360 degrees/200.
181 //so in each switching cycle, the phase has advanced by 360*VSI_SW_FREQ/f_fund (in degrees)
182 /* the phase is scaled so that one fundamental is 2^32 counts. */
183 #define PHASE_STEP    (Uint32)(4294967296.0*(double)F_FUND/(double)SW_FREQ_VSI/2.0)
184 //#define PHASE_STEP    (Uint16)(65536.0*(double)F_FUND/(double)SW_FREQ_BIDC/2.0)
185
186 //deadtime compensation parameters
187 #define INV_NP_NS_VIN_FIXED  (long)((NPRI*32768)/(NSEC*VIN)) // is shifted by FIXED_Q+4 (15)
188 #define DEADBAND_BIDC        1e-6
```

```
189 #define DB_DEG_BIDC          (360.0*SW_FREQ_BIDC*DEADBAND_BIDC)
190 #define DB_RAD_BIDC          (DB_DEG_BIDC*DEG_TO_RAD)
191 #define DEADBAND_COUNT_BIDC  ((int16)(DEADBAND_BIDC*HSPCLK))
192 #define DB_RATIO_BIDC        ((double)DEADBAND_COUNT_BIDC/(PERIOD_BIDC*2))
193
194 //sine table hash definitions
195 #define COUNT_TO_SINTABLE (4294967296.0/(PERIOD_BIDC*2))
196 #define COUNT_TO_RAD      PI/(PERIOD_BIDC)
197 #define COUNT_TO_RATIO    1.0/(2*3750.0)
198 #define RAD_TO_COUNT      3750.0/PI
199
200 /********
201 _MACROS()
202 ********/
203
204 #define SIN_TABLE_READ(PHASE,SIN_VAL){\
205         SIN_VAL = sin_table[((Uint32)PHASE>>23)];\
206         VAL_DIFF = (sin_table[((Uint32)PHASE>>23)+1]) - SIN_VAL;\
207         SIN_VAL += (int16)( ((int32)((Uint32)PHASE&0x3FFFFF)*(int32)VAL_DIFF)>>23 );}
208 // phase is a 16bit number, but the index is only 10 (513 values). The whole sine wave is represented in 16bits (0-65536),
209 // shift right by 6 to know where to aim in the sine table. interpolate using the last 7 bits.
210
211 void __declspec(dllexport) simuser (double t, double delt, double *in, double *out)
212 {
213
214 /***********
215 _inputs()
216 ***********/
217
218   double ctrlclk                   = in[0];
219   double VSI_ctrlclk               = in[1];
220   int16  OL_CL_VSI                 = (int16)in[2];
221   double mod                       = in[3];
222   double mag_Iref                   = in[4];
223   double Iout                       = in[5];
224   double emf_scaled                 = in[6];
225   int32  phase_current             = (int32)in[7];
226   int16  OL_CL_BiDC                 = (int16)in[8];
227   int32  DT_COMP                   = (int32)in[9];
228   double phase_OL                   = in[10];
229   double mag_VDCref                 = in[11];
230   double Vout                       = in[12];
231   double Iload                     =  in[13];
232
233 //Variable_Declarations
234 /**************************
235 _DSP_Emulation_Variables()
236 **************************/
237 // Sine & Cos Tables - Calculated in Initialisation
238   static int16       sin_table[SIN_TABLE_SIZE+1],
239                     cos_table[SIN_TABLE_SIZE+1];
240   static int16      init_table=0;
241
242   static int16      UF,
243                     UF_VSI,
244                     int_count,  //to tell which interrupt to run in.
245                     prev_ctrlclk,
246                     prev_VSI_ctrlclk;
247
248 /*****************
249 _Macro_Variables()
250 *****************/
251 //sin table read variables
252   static Uint32   PHASE;
253   static int16    SIN_VAL,
254                   VAL_DIFF;        // interpolation temp variable
255
256 /***************************
257 _DSP_Modulation_Variables()
258 ***************************/
259 static Uint16  VSIa,
260               VSIb,
261               BiDC_Pri,
262               BiDC_Sec,
263               CMPR1,
264               CMPR2,
265               CMPR_Pri,
266               CMPR_Sec,
267               V_Asat = 0,
268               V_Bsat = 0;
```

```
269
270 /***************************
271 _VSI_Modulation_Variables()
272 ***************************/
273 static int16   va,
274               va_temp,
275               max_time,
276               t_A,
277               t_B,
278               sin_val,
279               cos_val,
280               val_diff;
281
282 static int16  mod_fixed;
283 static Uint16  index;
284 static Uint32  vsiphase = 0,
285               phase_offset;
286 static double phase_init;
287
288 /**************************
289 _BiDC_Modulation_Variables()
290 **************************/
291 static int16 phase_shift=0;
292
293 /**********************
294 _Controller_Variables()
295 **********************/
296 typedef struct
297 {
298   double
299     Kp,
300     Ki,
301     ref,
302     error,
303     prop,
304     intnow,
305     intsum,
306     ctrl;
307 }type_pi_dbl;
308
309 /*********************
310 _VSI_Curreg_Variables()
311 *********************/
312 ////floating point implementation
313 //static double Iout_float,
314 //              Iref_float,
315 //              VSIerror_float,
316 //              Kp_VSI_float,
317 //              Ki_VSI_float,
318 //              VSIprop_float,
319 //              VSI_intnow_float,
320 //              VSI_int_float,
321 //              VSI_ctrl_float;
322
323 //fixed point implementation
324 static long   Iref_mag_fixed,
325               I_VSI_fixed,
326               Iref_fixed,
327               VSIerror_fixed,
328               Kp_VSI_fixed,
329               Ki_VSI_fixed,
330               VSIprop_fixed,
331               VSI_intnow_fixed,
332               VSI_int_fixed,
333               VSI_ctrl_fixed,
334               emf_scaled_fixed;
335
336 //floating point version
337 static type_pi_dbl I_PI_DBL;
338
339 //DC Bus compensation
340 static long   VDC_VSI_fixed;
341
342 /**********************
343 _Grid_Synch_Variables()
344 **********************/
345 static int16  prev_ZX,
346               test_point;
347
348 //current phase step variables
```

```
349 static int32 prev_phase_current =0;
350
351 /*******************
352 _DT_Comp_Variables()
353 *******************/
354
355 // New version. Unified DT compensation
356 //floating point
357 static double    VDCout_txscaled,
358                 Vs_Vp_4Vs,
359                 Vp_Vs_DB;
360
361 //fixed point
362 static int32    phase_rad_ratio_fixed,
363                 VDCout_txscaled_fixed,
364                 Vp_Vs_4Vp_fixed,
365                 Vs_Vp_4Vp_fixed,
366                 Vs_Vp_4Vs_fixed,
367                 Vs_Vp_DB_fixed,
368                 Vp_Vs_DB_fixed;
369
370 static int16    Tslew_count,
371                 phase_aug_DT_fixed;
372
373 /*******************
374 _Adaptive_Variables()
375 *******************/
376 typedef struct
377 {
378   double
379     delta0_aug,
380     delf_delu,
381     sin_val,
382     sin,
383     sum,
384     Kp;
385 }type_adapt;
386
387 static type_adapt Kp_float;
388 static double delf_delx,
389               delf_delx_temp,
390               delf_delx_scaled,
391               delf_delu_temp,
392               delf_delu,
393               delf_delu_scaled,
394               Kp_adapt,
395 //              phi_z=PI/2.0,
396               Z_harm[7],
397               phi_z[7];
398
399 static int16   phase_shift_avrg,
400               phase_shift_record[4],
401               counter_avrg,
402               n_harm;
403
404 //in fixed point
405 static int16  phi_z_fixed[7],
406               harm[7]={1,3,5,7,9,11,13},
407               harm_sq[7]={1,9,25,49,81,121,169},
408               sin_val_adapt;
409
410 static double sin_val_adapt_double;
411
412 static int32  delta0_aug_fixed,
413               sin_count,
414               inv_Z_harm_fixed[7],
415               delf_delu_temp_fixed,
416               delf_delu_fixed,
417               delf_delu_fixed_scaled,
418               Kp_adapt_fixed;
419 //              Kp_adapt_fixed_prev;
420
421 /***********************
422 _BiDC_PI_Control_Variables()
423 ***********************/
424
425 static type_pi_dbl VDC_PI_DBL;
426 //floating point version
427 static double VDCout_float,
428               VDC_Kp_float=0.02,
```

```
429              VDC_Ki_float,
430              VDCerror_float,
431              VDC_prop_float,
432              VDC_intnow_float,
433              VDC_int_float,
434              VDC_cont_signal_float;
435
436 //fixed point version
437 static int32  VDCout_fixed,
438              VDCout_avrg,
439              VDCref_fixed=0,
440              VDCerror_fixed,
441              VDC_Kp_fixed,
442              VDC_Ki_fixed,
443              VDC_prop_fixed,
444              VDC_intnow_fixed,
445              VDC_int_fixed=0,
446              VDC_cont_signal_fixed;
447
448 static int16  saturated;
449 /*******************
450 _BIDC_FF_Variables()
451 *******************/
452 static double Iload_FF_double;
453
454 static int32  Iload_fixed,
455              Iload_abs;
456
457 static int32  BIDC_FF,
458              Iload_FF_fixed[PERIOD_2_BIDC];
459
460 static int16  hi,
461              lo,
462              mid,
463              harm_3[7]={1,27,125,343,729,1331,2197};
464
465 /*************************
466 _BIDC_Resonant_Variables()
467 *************************/
468 //Canonical representation
469 static double Kemfint2,
470              int2,
471              Kemf,
472              Sum1,
473              int1,
474              int3,
475              PPlaceCan_out,
476              Komega1,
477              Komega2,
478              Komega3;
479
480
481 // Discretised version
482 static double  A1dig,
483              A2dig,
484              A3dig,
485              B1dig,
486              B2dig,
487              B3dig;
488
489 static double VerrorKp,
490              VerrorKp_1delay,
491              VerrorKp_2delay,
492              VerrorKp_3delay,
493              PPlace_out,
494              PPlace_out_1delay,
495              PPlace_out_2delay,
496              PPlace_out_3delay;
497
498 //END DECLARATIONS
499
500 //CODE STARTS HERE
501 /**********************
502 _TIMER_INTERRUPT_TASKS()
503 ***********************/
504 //  03/03/2011   - The interrupt runs at 10kHz, and open loop modulates a Single Phase VSI
505 //  03/03/2011   - 10kHz interrupt, PI Current regulate a Single Phase VSI + FF compensation of the load EMF
506 //               - update to 40kHz interrupt. Current still sampled at 10kHz.
507 //                 Just a counter that makes it only work on 1 in 4 cycles. (int_count)
508 //  04/03/2011   - Determine the phase of the back emf - very miniature grid synch
```

```
509 //              - Included Bi-Directional DC-DC Converter. (Open loop)
510 //  07/03/2011  -  Closed loop control of the Bi-directional DC-DC Converter. Optimised Integrator
511 //              - Synchronous Sampled Adaptive PI Controller (Floating Point)
512 //              - Asynchronous Sampling - Why is it peak & trough now????
513 //  08/03/2011  - Fixed Point Adaptive Controller
514 //  09/03/2011  -  Feed Forward of load current, based on power.
515 //  10/03/2011  - Debugging Feed-Forward and Adaptive Controller.
516 //  11/03/2011  - Match BiDC controller pole to cancel out the plant pole - bad idea?
517 //  12/03/2011  -  Included Deadband and Device Drops - no deadtime comp yet - may not be necessary
518 //  14/03/2011  -  Resonant Controller - Floating Point - attempted
519 //              -  DC Bus compensation
520 //  21/03/2011  - Emulate ADCs
521
522 // PORTED OVER TO OPEN GIIB
523 //  14/04/2011  - Reduced DC bus voltage
524
525 // PORTED OVER TO PHD THESIS - to generate plots for Harm model
526 //  19/10/2011  - Removed DSP Compare. DLL block generates CMPR values, modulation & DT generated externally
527
528 /******************
529 _Initialisations()
530 ******************/
531 //Setting initial conditions for the simulation
532   if (t==delt)
533   {
534     //set up sine & cos tables
535     for(init_table=0;init_table<(SIN_TABLE_SIZE+1);init_table++)
536     {
537       sin_table[init_table] = (int16)(16384*sin((double)init_table/(double)SIN_TABLE_SIZE*2.0*PI));
538       cos_table[init_table] = (int16)(16384*cos((double)init_table/(double)SIN_TABLE_SIZE*2.0*PI));
539     }
540     int_count = 0;
541
542     //VSI initialisations
543     max_time = MAX_VSI_TIME;
544     phase_init = -90.0;
545     vsiphase  =  (int32)(phase_init*(4294967296.0/360.0));  //initial phase of current
546
547     //determine gains
548     I_PI_DBL.Kp  =  KP_VSI;
549     I_PI_DBL.Ki = KI_VSI/FSAMPLE_VSI;
550     I_PI_DBL.intsum = 0;
551     Kp_VSI_fixed=(int32)(KP_VSI*FIXED_Q_SCALE);
552     Ki_VSI_fixed=(long)((KI_VSI/FSAMPLE_VSI)*FIXED_Q_SCALE);
553
554     //BiDC initialisations
555     CMPR_Pri        = PERIOD_2_BIDC;
556
557     //Adaptive initialisations
558     Z_harm[0]=  sqrt(R_L_2 + OMEGA_BIDC_L_2);
559     Z_harm[1]=   sqrt(R_L_2 + 3.0*3.0*OMEGA_BIDC_L_2);
560     Z_harm[2]=  sqrt(R_L_2 + 5.0*5.0*OMEGA_BIDC_L_2);
561     Z_harm[3]=  sqrt(R_L_2 + 7.0*7.0*OMEGA_BIDC_L_2);
562     Z_harm[4]=  sqrt(R_L_2 + 9.0*9.0*OMEGA_BIDC_L_2);
563     Z_harm[5]=  sqrt(R_L_2 + 11.0*11.0*OMEGA_BIDC_L_2);
564     Z_harm[6]=  sqrt(R_L_2 + 13.0*13.0*OMEGA_BIDC_L_2);
565
566     phi_z[0] =  atan2(OMEGA_BIDC_L,R_L);
567     phi_z[1] =  atan2(OMEGA_BIDC_L*3.0,R_L);
568     phi_z[2] =  atan2(OMEGA_BIDC_L*5.0,R_L);
569     phi_z[3] =  atan2(OMEGA_BIDC_L*7.0,R_L);
570     phi_z[4] =  atan2(OMEGA_BIDC_L*9.0,R_L);
571     phi_z[5] =  atan2(OMEGA_BIDC_L*11.0,R_L);
572     phi_z[6] =  atan2(OMEGA_BIDC_L*13.0,R_L);
573
574     phi_z_fixed[0] =  (int16)(phi_z[0]*RAD_TO_COUNT);
575     phi_z_fixed[1] =  (int16)(phi_z[1]*RAD_TO_COUNT);
576     phi_z_fixed[2] =  (int16)(phi_z[2]*RAD_TO_COUNT);
577     phi_z_fixed[3] =  (int16)(phi_z[3]*RAD_TO_COUNT);
578     phi_z_fixed[4] =  (int16)(phi_z[4]*RAD_TO_COUNT);
579     phi_z_fixed[5] =  (int16)(phi_z[5]*RAD_TO_COUNT);
580     phi_z_fixed[6] =  (int16)(phi_z[6]*RAD_TO_COUNT);
581
582     inv_Z_harm_fixed[0]=  (int32)(32768.0/(1.0*Z_harm[0]));
583     inv_Z_harm_fixed[1]=   (int32)(32768.0/(3.0*Z_harm[1]));
584     inv_Z_harm_fixed[2]=  (int32)(32768.0/(5.0*Z_harm[2]));
585     inv_Z_harm_fixed[3]=  (int32)(32768.0/(7.0*Z_harm[3]));
586     inv_Z_harm_fixed[4]=  (int32)(32768.0/(9.0*Z_harm[4]));
587     inv_Z_harm_fixed[5]=  (int32)(32768.0/(11.0*Z_harm[5]));
588     inv_Z_harm_fixed[6]=  (int32)(32768.0/(13.0*Z_harm[6]));
```

```
589     //scaled by 32768 = 2^15
590
591     //BiDC integrator initialisation
592     VDC_Ki_fixed    = (int32)((OMEGA_C_10_BIDC/FSAMPLE_BIDC)*FIXED_Q_SCALE);
593     VDC_PI_DBL.Ki    =  OMEGA_C_10_BIDC/FSAMPLE_BIDC;
594     VDC_PI_DBL.intsum = 0;
595
596     //Iload Feed forward initialisations
597     //Generate a lookup table of the steady state load current based on operating phase shift.
598     //I_load_FF = 16/pi^2 *Vp * Np/Ns * sum(1/(2n+1)^3 * sin((2n+1)delta)/(omega*L)
599     //done in floating point, converted to fixed point at the last step
600
601     for (init_table=0;init_table<=PERIOD_2_BIDC;init_table++)
602     {
603       Iload_FF_double=0.0;
604       for (n_harm=0;n_harm<6;n_harm++)
605       {
606         Iload_FF_double  +=   (1.0/harm_3[n_harm])*sin(harm[n_harm]*(init_table*COUNT_TO_RAD));
607       }
608       Iload_FF_fixed[init_table] = (int32)(BIDC_FF_CONST*Iload_FF_double*FIXED_Q_SCALE);
609     }
610
611     //to help with initialisations
612     VDCout_fixed=0;
613   }
614
615 /*******************
616 _TIMER_INTERRUPT()
617 *******************/
618 //Now we run the TIMER interrupts
619 /**********
620 _VSI_INT()
621 **********/
622     if ((prev_VSI_ctrlclk <=0 && VSI_ctrlclk >=1)||(prev_VSI_ctrlclk >=1 && VSI_ctrlclk <=0))  //Sampling clock
623     {
624       if (UF_VSI==0)   UF_VSI = 1;
625       else             UF_VSI = 0;
626
627 //      Calculate current sin table value:
628       vsiphase+=PHASE_STEP;
629
630       //check for step change in phase
631       if (phase_current!=prev_phase_current)
632       {
633         vsiphase = phase_current;
634       }
635
636       SIN_TABLE_READ(vsiphase,sin_val);
637
638 //      We also want the ability to step change the flow of power.
639 //      this is a step change in phase, with reference to the back emf.
640 //      So the phase of the back emf must be determined
641
642     //Run the ADCs
643     I_VSI_fixed = (int32)(Iout*FIXED_Q_SCALE);
644     emf_scaled_fixed = (int32)(emf_scaled*FIXED_Q_SCALE);
645
646   //the whole point of closed loop control is to determine the required magnitude & phase
647   //that will give the desired output current.
648
649 /**********************
650 _VSI_Current_Regulator()
651 **********************/
652     //in fixed point - scaled by FIXED_Q
653     //first, generate reference
654       Iref_mag_fixed = (long)(mag_Iref*FIXED_Q_SCALE);
655       Iref_fixed = (long)(Iref_mag_fixed*sin_val)>>14;//scaled by 2^14 from sin table
656
657       //scale KP by DC Bus
658 //      Kp_VSI_fixed=(long)((KP_CONST*2.0*FIXED_Q_SCALE)/VDCout_fixed);    //scaled by dc
659
660       //determine error
661        VSIerror_fixed = (Iref_fixed - I_VSI_fixed);
662       //proportional control
663       VSIprop_fixed = (VSIerror_fixed*Kp_VSI_fixed)>>FIXED_Q;
664       //integrator
665       VSI_intnow_fixed = (VSIprop_fixed*Ki_VSI_fixed)>>FIXED_Q;
666       VSI_int_fixed += VSI_intnow_fixed;
667       VSI_ctrl_fixed = VSIprop_fixed + VSI_int_fixed;
668
```

```
669       //DC Bus compensation
670 //      VSI_ctrl_fixed = (int16)((VSI_ctrl_fixed*VBUS_NOM_FIXED)/VDCout_fixed);
671 /***************
672 _Switch_Times()
673 ***************/
674   //for the VSI
675       if (OL_CL_VSI==0)
676       {
677         mod_fixed = (int16)(mod*FIXED_Q_SCALE);
678         va = (int16)((((int32)(mod_fixed*sin_val)>>FIXED_Q)*(int32)PERIOD_2_VSI)>>14);//(int16)(((int32)(mod*sin_val*PERIOD_2_VSI))>>15);
679 //        va = (int16)((int32)(mod_fixed*PERIOD_2_VSI)>>FIXED_Q);
680       }
681       else if (OL_CL_VSI==1)
682       {
683         va = (int16)((VSI_ctrl_fixed*PERIOD_2_VSI)>>FIXED_Q);
684       }
685       else if (OL_CL_VSI==2)
686       {
687         va = (int16)(((VSI_ctrl_fixed+emf_scaled_fixed)*PERIOD_2_VSI)>>FIXED_Q);
688       }
689       /* Switching duty cycles */
690       t_A = va;
691       t_B = -t_A;
692
693 /************
694 _VSI_DESAT()
695 ************/
696       /* clamp switch times for pulse deletion and saturation */
697       // UF flags underflow interrupt
698
699       // A phase
700       if (t_A > max_time)
701       {
702         CMPR1 = 1;
703       }
704       else if (t_A < (-max_time))
705       {
706         if (!V_Asat && UF_VSI)
707           CMPR1 = PERIOD_VSI - 1;
708         else
709           CMPR1 = PERIOD_VSI;
710         V_Asat = 1;
711       }
712       else
713       {
714         if (V_Asat && UF_VSI)
715           CMPR1 = PERIOD_VSI-1;
716         else
717           CMPR1 = (Uint16)(PERIOD_2_VSI - t_A);
718         V_Asat = 0;
719       }
720
721       // B phase
722       if (t_B > max_time)
723       {
724         CMPR2 = 1;
725       }
726       else if (t_B < (-max_time))
727       {
728         if (!V_Bsat && UF_VSI)
729           CMPR2 = PERIOD_VSI - 1;
730         else
731           CMPR2 = PERIOD_VSI-1;
732         V_Bsat = 1;
733       }
734       else
735       {
736         if (V_Bsat && UF_VSI)
737           CMPR2 = PERIOD_VSI-1;
738         else
739           CMPR2 = (Uint16)(PERIOD_2_VSI - t_B);
740         V_Bsat = 0;
741       }
742
743       prev_phase_current  = phase_current;
744     }
745
746   //CONTROL LOOP INTERRUPT - BIDC
747   if ((prev_ctrlclk <=0 && ctrlclk >=1)||(prev_ctrlclk >=1 && ctrlclk <=0))  //Sampling clock
748   {
```

```
749     //identify interrupt
750     if (prev_ctrlclk <=0 && ctrlclk >=1) UF = 1;  //  UNDERFLOW
751     if (prev_ctrlclk >=1 && ctrlclk <=0) UF = 0;  //  PERIOD MATCH
752
753     int_count++;
754     if (int_count>=4) int_count = 0;
755 /***********
756 _BiDC_INT()
757 ***********/
758     //This section of code looks after the Bi-directional DC-DC Converter.
759     //It needs to take the output voltage and regulate it using the
760     //Adaptive PI Controller with FF compensation of the load.
761
762     //uses Asynchronous Sampling with Asynchronous update
763
764     //Run the ADCs
765     VDCout_fixed = (int32)(Vout*FIXED_Q_SCALE);
766     VDCout_avrg = (VDCout_avrg>>1)+ (VDCout_fixed>>1);  //rolling avrg of last two samples
767     va_temp = va>>2;  //simulates the transfer of va via the DAC
768
769     //Output DC Current FF
770 //    if(OL_CL_BiDC==2)    Iload_fixed = (int32)(Iload*FIXED_Q_SCALE);
771     //Output H-bridge Current FF
772     if(OL_CL_BiDC==3)    Iload_fixed = (int32)((I_VSI_fixed*((int32)va_temp<<2))/PERIOD_2_VSI); // scaled by mod depth*2
773     else                Iload_fixed = (int32)(Iload*FIXED_Q_SCALE);
774
775 //    if ((OL_CL_BiDC==2)||(OL_CL_BiDC==3)) Iload_fixed = ((int32)(Iload*FIXED_Q_SCALE))+ ((int32)((I_VSI_fixed*((int32)va_temp<<2))/PERIOD_2_VSI));
776
777     //first determine the Average operating phase_shift (moving average of the last 4 phaseshifts)
778     phase_shift_avrg=0;  //in counts
779     counter_avrg=0;
780     phase_shift_record[int_count] = abs(phase_shift);
781     while(counter_avrg<=3)
782     {
783       phase_shift_avrg += phase_shift_record[counter_avrg]>>2;
784       counter_avrg++;
785     }
786
787 /*****************************
788 __Deadtime_Compensation()     *
789 *****************************/
790 //lifted from MATLAB code - works in rad. floating point
791
792     phase_rad_ratio_fixed  = (abs(phase_shift)<<FIXED_Q)/(PERIOD_BIDC<<1);
793 //    VDCout_txscaled_fixed = (VDCref_fixed*NPRI_NSEC_FIXED)>>FIXED_Q;
794     VDCout_txscaled_fixed = (VDCout_fixed*NPRI_NSEC_FIXED)>>FIXED_Q;
795     if (VDCout_txscaled_fixed==0) VDCout_txscaled_fixed=1;
796     VDCout_txscaled = (double)VDCout_txscaled_fixed/FIXED_Q_SCALE;
797
798     Vp_Vs_4Vp_fixed = ((VIN_FIXED-VDCout_txscaled_fixed)<<(FIXED_Q-2))/VIN_FIXED;
799     Vs_Vp_4Vp_fixed = ((VDCout_txscaled_fixed-VIN_FIXED)<<(FIXED_Q-2))/VIN_FIXED;
800
801     Vs_Vp_4Vs = (VDCout_txscaled-VIN)/(4*VDCout_txscaled);
802     Vs_Vp_4Vs_fixed = ((VDCout_txscaled_fixed-VIN_FIXED)<<(FIXED_Q-2))/VDCout_txscaled_fixed;
803     Vs_Vp_DB_fixed = VDCout_txscaled_fixed*DEADBAND_COUNT_BIDC/((int32)VIN*(PERIOD_BIDC<<1));
804     Vp_Vs_DB =  (VIN/VDCout_txscaled)*DB_RATIO_BIDC; //DB_RATIO_BIDC=DEADBAND_COUNT_BIDC/(PERIOD_BIDC<<1)
805     Vp_Vs_DB_fixed   = (((VIN_FIXED/(PERIOD_BIDC<<1))*DEADBAND_COUNT_BIDC)<<FIXED_Q)/VDCout_txscaled_fixed;
806
807 // First, calculate slew time
808     if (VIN_FIXED>VDCout_txscaled_fixed)    //Vp>Vs
809     {
810       if (phase_shift_avrg<0)  //leading
811       {
812         Tslew_count = (int16)((phase_rad_ratio_fixed - Vp_Vs_4Vp_fixed  - Vs_Vp_DB_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
813
814         //then calculate phase augmentation
815         if ((VIN_FIXED-VDCout_txscaled_fixed)>(5<<FIXED_Q))
816         {
817           if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = 0;
818           else if (Tslew_count<0)                  phase_aug_DT_fixed = DEADBAND_COUNT_BIDC;
819 //          else                                  phase_aug_DT_fixed = DEADBAND_COUNT_BIDC;
820           else                                    phase_aug_DT_fixed = DEADBAND_COUNT_BIDC-Tslew_count;  //in counts
821         }
822         else
823         {
824           if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = 0;
825           else                                    phase_aug_DT_fixed = DEADBAND_COUNT_BIDC;
826         }
827       }
828       else  //lagging
```

```
829       {
830         Tslew_count = (int16)((Vp_Vs_4Vp_fixed - phase_rad_ratio_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
831
832         if ((VIN_FIXED - VDCout_txscaled_fixed)>(5<<FIXED_Q))
833         {
834           if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = DEADBAND_COUNT_BIDC;
835           else if (Tslew_count<0)                  phase_aug_DT_fixed = 0;
836 //          else                                  phase_aug_DT_fixed = DEADBAND_COUNT_BIDC;
837           else                                    phase_aug_DT_fixed = Tslew_count;  //in counts
838         }
839         else
840         {
841           if (Tslew_count>0)                      phase_aug_DT_fixed = DEADBAND_COUNT_BIDC;
842           else                                     phase_aug_DT_fixed = 0;
843         }
844       }
845     }
846     else  //Vp<Vs
847     {
848       if (phase_shift_avrg<0)  //leading
849       {
850         Tslew_count = (int16)((Vs_Vp_4Vp_fixed - phase_rad_ratio_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
851
852         if ((VDCout_txscaled_fixed-VIN_FIXED)>(5<<FIXED_Q))
853         {
854           if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
855           else if (Tslew_count<0)                  phase_aug_DT_fixed = 0;
856 //          else                                  phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
857           else                                    phase_aug_DT_fixed = -Tslew_count;  //in counts
858         }
859         else
860         {
861           if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
862           else                                     phase_aug_DT_fixed = 0;
863         }
864       }
865       else  //lagging
866       {
867         Tslew_count = (int16)((phase_rad_ratio_fixed - Vs_Vp_4Vs_fixed  - Vp_Vs_DB_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
868
869         if ((VDCout_txscaled_fixed-VIN_FIXED)>(5<<FIXED_Q))
870         {
871           if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = 0;
872           else if (Tslew_count<0)                  phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
873 //          else                                  phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
874           else                                    phase_aug_DT_fixed = -(DEADBAND_COUNT_BIDC-Tslew_count);  //in counts
875         }
876         else
877         {
878           if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = 0;
879           else                                     phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
880         }
881       }
882     }
883
884 /*******************
885 _Adaptive_Gain_Calc()
886 *******************/
887     //floating point first - make sure it works
888     //Then, include the deadtime compensation
889
890     if(DT_COMP)
891       delta0_aug_fixed=abs(phase_shift_avrg-phase_aug_DT_fixed);
892     else
893       delta0_aug_fixed=abs(phase_shift_avrg);
894
895     if (delta0_aug_fixed >= (MAX_PHASE-100))
896       delta0_aug_fixed = MAX_PHASE-100;
897
898       //floating point
899       Kp_float.delta0_aug = (double)delta0_aug_fixed*COUNT_TO_RAD;
900
901       n_harm=0;
902       Kp_float.sum = 0.0;
903
904       for (n_harm=0;n_harm<6;n_harm++)
905       {
906
907         Kp_float.sin_val        =   phi_z[n_harm] - harm[n_harm]*Kp_float.delta0_aug;
908         Kp_float.sin            =    sin(Kp_float.sin_val);
```

```
909       Kp_float.sum            +=   Kp_float.sin/(harm[n_harm]*Z_harm[n_harm]);
910
911     }
912       Kp_float.delf_delu      =   (16.0*VDCPRI/(C*PI*PI))*(NPRI/NSEC)*Kp_float.sum;
913       if (Kp_float.delf_delu==0)
914         Kp_float.delf_delu=1e-4;
915       Kp_float.Kp             =   OMEGA_C_BIDC/Kp_float.delf_delu;
916
917     if (Kp_float.Kp >=VDC_KP_MAX) Kp_float.Kp  = VDC_KP_MAX;
918     if (Kp_float.Kp <=VDC_KP_MIN) Kp_float.Kp  = VDC_KP_MIN;
919
920     //fixed point
921     //then determine the B value
922     delf_delu_fixed = 0;
923     n_harm=0;
924
925     for (n_harm = 0;n_harm<6;n_harm++)
926     {
927       //fixed point
928       sin_count             = (int32)((phi_z_fixed[n_harm]-harm[n_harm]*delta0_aug_fixed)*COUNT_TO_SINTABLE);
929       sin_val_adapt         = sin_table[(Uint32)(sin_count>>22)];
930       sin_val_adapt_double   = (int16)(16384.0*sin(phi_z[n_harm]-harm[n_harm]*(double)(delta0_aug_fixed*COUNT_TO_RAD)));
931
932       //Determine B_delta value - for proportional term
933       delf_delu_temp_fixed  =  (int32)(sin_val_adapt_double*inv_Z_harm_fixed[n_harm])>>(14+15-SMALL_Q);
934       //shift right because Z-harm_fixed is scaled by 15 and 14 for the sine table, we want to leave it scaled to small_Q
935       delf_delu_fixed       += delf_delu_temp_fixed;
936     }
937
938     //scale by constants
939     delf_delu_fixed_scaled   = (int32)(delf_delu_fixed*(int32)DELF_DELU_CONST)>>(SMALL_Q-4);
940     delf_delu_fixed_scaled   = (delf_delu_fixed_scaled*VP_FIXED)>>FIXED_Q;
941                               //further shift by 4 is needed because delf_delu_const has been scaled by 4 earlier
942     if (delf_delu_fixed_scaled==0) delf_delu_fixed_scaled=1;
943
944     //scale the proportional gain
945     Kp_adapt_fixed=((OMEGA_C_BIDC_FIXED)/delf_delu_fixed_scaled);
946     if (Kp_adapt_fixed>=VDC_KP_MAX_FIXED) Kp_adapt_fixed = VDC_KP_MAX_FIXED;
947     if (Kp_adapt_fixed<=VDC_KP_MIN_FIXED) Kp_adapt_fixed = VDC_KP_MIN_FIXED;
948
949     Kp_adapt_fixed   =  (int32)(Kp_float.Kp*SMALL_Q_SCALE);
950
951     if (OL_CL_BiDC == 4)  //fixed PI gains
952     {
953       VDC_PI_DBL.Kp   =  VDC_KP_MAX;
954       VDC_Kp_fixed     =  VDC_KP_MAX_FIXED;//(int32)(0.0196*SMALL_Q_SCALE);
955     }
956     else
957     {
958       VDC_PI_DBL.Kp   =  Kp_float.Kp;
959       VDC_Kp_fixed     =  Kp_adapt_fixed;
960     }
961
962 /*********
963 _BIDC_FF()
964 *********/
965 //    if (UF)
966 //    {
967     Iload_abs=abs(Iload_fixed);
968
969     //Iload Feedforward - search algorithm
970     lo=0;
971     hi=PERIOD_2_BIDC-1;
972     while (hi>lo)
973     {
974       mid = ((hi-lo)/2)+lo;
975       if (Iload_abs<Iload_FF_fixed[mid])        hi=mid-1; //in the bottom half
976       else if (Iload_abs>Iload_FF_fixed[mid])    lo=mid+1;
977       else if (Iload_abs==Iload_FF_fixed[mid])
978         {
979           lo=mid;
980           break;
981         }
982       else if ((hi-lo)<10) break;
983     }
984
985     if (saturated==1) BIDC_FF=0;
986     else
987     {
988       if  (Iload_fixed>0) BIDC_FF = lo;
```

```
989         else BIDC_FF = -lo;
990       }
991 //    }
992 /*******************
993 _BIDC_PI_Control_Loop()
994 *******************/
995     //Now in fixed point
996     if (UF)
997     {
998       VDC_PI_DBL.ref         =    mag_VDCref;
999       VDC_PI_DBL.error       =    VDC_PI_DBL.ref - Vout;
1000      VDC_PI_DBL.prop       =   VDC_PI_DBL.error*VDC_PI_DBL.Kp;
1001      VDC_PI_DBL.intnow      =   VDC_PI_DBL.prop*VDC_PI_DBL.Ki;
1002
1003      //fixed point
1004      VDCref_fixed          =    (int32)(mag_VDCref*FIXED_Q_SCALE);
1005       VDCerror_fixed         =   VDCref_fixed-VDCout_fixed;
1006      VDC_prop_fixed         =   (VDCerror_fixed*VDC_Kp_fixed)>>SMALL_Q;
1007      VDC_intnow_fixed       =   (VDC_prop_fixed*VDC_Ki_fixed)>>FIXED_Q;
1008      VDC_cont_signal_fixed =   (((VDC_prop_fixed + VDC_int_fixed)*PERIOD_SCALE_BIDC)>>FIXED_Q);
1009
1010      if (saturated==0)
1011      {
1012        VDC_PI_DBL.intsum    +=   VDC_PI_DBL.intnow;
1013        VDC_int_fixed        +=   VDC_intnow_fixed;
1014      }
1015
1016      VDC_PI_DBL.ctrl       =   (VDC_PI_DBL.prop+VDC_PI_DBL.intsum)*(double)PERIOD_SCALE_BIDC;
1017 //      VDC_cont_signal_fixed =   (((VDC_prop_fixed + VDC_int_fixed)*PERIOD_SCALE_BIDC)>>FIXED_Q);
1018    }
1019
1020 /****************
1021 _BIDC_SET_PHASE()
1022 ****************/
1023    if (OL_CL_BiDC==0)        phase_shift = (int16)((-phase_OL*PERIOD_BIDC)/180);  //Open Loop
1024    else
1025    {
1026      if ((OL_CL_BiDC==1)||(OL_CL_BiDC==4))  //no FF
1027      {
1028        phase_shift =   (int16)(-VDC_cont_signal_fixed);    //fixed point
1029      }
1030      else
1031        phase_shift =   (int16)(-(VDC_cont_signal_fixed + BIDC_FF));  //CL with Feedforward
1032
1033      if(DT_COMP)            phase_shift =   phase_shift+phase_aug_DT_fixed;
1034    }
1035 /**************
1036 _BIDC_DESAT()    *
1037 ***************/
1038    if (abs(phase_shift)>(MAX_PHASE-1))
1039      {
1040        saturated=1; // desat bit
1041        VDC_int_fixed -= VDC_intnow_fixed;
1042        if (phase_shift>0)
1043          {
1044            phase_shift = MAX_PHASE;
1045          }
1046        if (phase_shift<0)
1047          {
1048            phase_shift = -MAX_PHASE;
1049          }
1050      }
1051    else
1052      {
1053        saturated=0;
1054      }
1055 /*****************
1056 _BiDC_Modulator()
1057 *****************/
1058    CMPR_Pri        = PERIOD_2_BIDC;
1059    if (UF)
1060    {
1061 //    The Bidirectional DC-DC Converter is comprised of 2 PSSW single phase bridges.
1062      CMPR_Sec = (Uint16)(PERIOD_2_BIDC+phase_shift);
1063    }
1064    else
1065    {
1066      CMPR_Sec = (Uint16)(PERIOD_2_BIDC-phase_shift);
1067    }
1068    /* --------- Finish Experimental Interrupt Code --------- */
```

```
1069   } /* End Interrupt Code */
1070
1071 /*******************
1072 To Do: Grid Synch
1073 *******************/
1074 //this interrupt will determine the phase & frequency of the backemf (grid) to allow synchronising.
1075
1076
1077 /*********
1078 _OUTPUTS()
1079 *********/
1080
1081   out[0] = CMPR1;
1082   out[1] = CMPR2;
1083
1084   out[2] = CMPR_Pri;
1085   out[3] = CMPR_Sec;
1086
1087   //references
1088   out[4] = Iref_fixed/FIXED_Q_SCALE;
1089   out[5] = VDC_PI_DBL.ref;
1090
1091   //Debug BiDC FF
1092   out[6]   = VDC_prop_fixed;
1093   out[7]   = VDC_int_fixed;
1094   out[8]   = VDC_cont_signal_fixed;
1095   out[9]   = BIDC_FF;
1096   out[10]  = phase_shift;// VSI_intnow_fixed/FIXED_Q_SCALE;
1097   out[11]  = I_PI_DBL.intsum;// VSI_int_fixed/FIXED_Q_SCALE;
1098   out[12] = VSI_ctrl_fixed/FIXED_Q_SCALE;
1099   out[13] = va;
1100   out[14] = (int16)((VSI_ctrl_fixed*PERIOD_2_VSI)>>FIXED_Q);
1101   out[15] = UF_VSI;
1102
1103   prev_ctrlclk      =  (int16)ctrlclk;
1104   prev_VSI_ctrlclk  =  (int16)VSI_ctrlclk;
1105
1106 }
```

A.2 Experimental Code

The experimental code was developed from the base code written by Mr. Andrew McIver and Mr. Sorrel Grogan of Creative Power Technologies. Since the experimental setup included a DAB converter with a VSI load, this section is separated as follows:

- CPLD Code – Dual Active Bridge
- DSP Code – Dual Active Bridge
- DSP Code – Voltage Source Inverter

A.2.1 CPLD Code – Dual Active Bridge

```
 1 -- CPT-Mini2810 EPM570T100C5N CPLD Base Program
 2 -- Developed By:
 3 --  Power Electronics Group, Monash University
 4 --  Creative Power Technologies, (C) Copyright 2008
 5 -- Written by: S.Grogan
 6 -- Date: 25/09/08 Initial Release to customer
 7 -- Modified: D. Segaran 2009
 8
 9 -- assumptions are that the nSYNC line goes low first, then the sclk begins from a high state
10 -- data is read from SPI on the falling edge of sclk
11 -- data is sent to SPI on the rising edge of sclk
12
13 -- interrupts need to be fully tested
14 -- a /1 clock option is NOT a possibility in an EPLD, as any register change can happen on a
15 -- rising XOR falling edge, not both.
16
17 -- adding a reset signal pre clock to all processes introduces a timing delay significant enough to
18 -- adversely affect SPI comms.
19
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22 use IEEE.STD_LOGIC_ARITH.ALL;
23 use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25 entity spi_to_bus_v2 is
26
27   port(
28       -- CPLD requirements:
29       clock      : IN STD_LOGIC;
30       nRESET      : IN STD_LOGIC;
31
32       -- SPI interface:
33       sclk      : IN STD_LOGIC;    -- clock from the 2810
34       nSYNC      : IN STD_LOGIC;    -- chip select from the 2810, not the boy band
35       mosi      : IN STD_LOGIC;    -- data from the 2810
36       miso      : INOUT STD_LOGIC;  -- data to the 2810
37
38       debug      : OUT STD_LOGIC;
39
40       -- Minibus interface:
41       nMINIBUS    : OUT STD_LOGIC;              -- minibus enable       (active low)
42       nRD        : OUT STD_LOGIC;              -- minibus read       (active low)
43       nWR        : OUT STD_LOGIC;              -- minibus write       (active low)
44       nCS        : OUT STD_LOGIC_VECTOR (2 downto 0);    -- minibus chip selects   (active low)
45       MA        : OUT STD_LOGIC_VECTOR (2 downto 0);    -- minibus memory addresses  (active low)
46       minibus      : INOUT STD_LOGIC_VECTOR (7 downto 0);    -- minibus bus (debugging)
47
48       -- Communications:
49       SCIBMODE    : OUT STD_LOGIC;              -- direction select
50
51       -- CAPQEP:
52       DIGIN      : IN STD_LOGIC_VECTOR (3 downto 0);
53       INDEX      : IN STD_LOGIC_VECTOR (1 downto 0);
```

```
54      CAP        : OUT STD_LOGIC_VECTOR (5 downto 0);
55
56      -- INTSEL:
57      XINT1A      : IN STD_LOGIC;
58      XINT1B      : IN STD_LOGIC;
59      XINT1      : OUT STD_LOGIC;
60
61      -- PWM EVA:
62      PWMIN      : IN STD_LOGIC_VECTOR (5 downto 0);
63      TxPWM      : IN STD_LOGIC_VECTOR (1 downto 0);
64      PWMOUT      : OUT STD_LOGIC_VECTOR (7 downto 0);
65
66      -- PWM EVB:
67      PWMB7      : OUT STD_LOGIC;
68      PWMB8      : OUT STD_LOGIC;
69      T3PWM      : IN STD_LOGIC;
70      T4PWM      : IN STD_LOGIC;
71
72      -- Analog Switch:
73      ANIN      : OUT STD_LOGIC_VECTOR (6 downto 0);    -- MSB [IN1  IN2  IN3  INA  INB  INC  IND] LSB
74
75      -- GPIO Interface
76      GPIO      : INOUT STD_LOGIC_VECTOR (1 downto 0);  -- generic digital IO
77
78      -- Error Signals
79      PDPINTA    : IN STD_LOGIC;
80
81      -- PWM output enable
82      PWMen      : OUT STD_LOGIC);      -- S.G. updated 09/09/09
83
84 end spi_to_bus_v2;
85
86 architecture behaviour of spi_to_bus_v2 is
87
88   signal neg_clock      : STD_LOGIC;
89   signal out_clock      : STD_LOGIC;
90
91   signal mosi_reg        : STD_LOGIC_VECTOR (23 downto 0);  -- data read from the 2810 SPI is fed into here
92   --( [ 8 bits command ] [ 8 bits data to write] [ 8 bits data to read ]  )
93   --( [23,22,21,20,19,18,17,16] [15,14,13,12,11,10,9,8] [7,6,5,4,3,2,1,0] )
94   signal miso_reg        : STD_LOGIC_VECTOR (7 downto 0);  -- data sent to the 2810 SPI is loaded into here
95
96   signal spi_pos_count    : STD_LOGIC_VECTOR (4 downto 0);  -- overall position of the SPI registers
97
98   signal command        : STD_LOGIC_VECTOR (7 downto 0);  -- command fed from SPI
99   signal data_1        : STD_LOGIC_VECTOR (7 downto 0);  -- data byte 1 fed from SPI
100
101   signal minibus_data      : STD_LOGIC_VECTOR (7 downto 0);  -- data read from minibus
102   signal mb_wait        : STD_LOGIC_VECTOR (2 downto 0);  -- to implement a delay for minibus I/O
103
104   signal data_1_ok      : STD_LOGIC;
105   signal data_2_ok      : STD_LOGIC;
106   signal command_ok      : STD_LOGIC;
107   signal read_point      : STD_LOGIC;
108   signal write_point      : STD_LOGIC;
109
110   signal SCIBMODE_reg      : STD_LOGIC;
111
112   signal CAPQEP_reg      : STD_LOGIC;
113
114   -- interrupt routine registers:
115   signal INTSEL_reg      : STD_LOGIC_VECTOR (7 downto 0);
116   signal INTSRC_reg      : STD_LOGIC_VECTOR (7 downto 0);
117   signal int_clear      : STD_LOGIC;
118
119   signal XINT1A_prev      : STD_LOGIC;
120   signal XINT1B_prev      : STD_LOGIC;
121
122   signal XINT1A_int      : STD_LOGIC;
123   signal XINT1A1_reg      : STD_LOGIC;
124   signal XINT1A2_reg      : STD_LOGIC;
125   signal XINT1A3_reg      : STD_LOGIC;
126   signal XINT1A4_reg      : STD_LOGIC;
127
128   signal XINT1B_int      : STD_LOGIC;
129   signal XINT1B1_reg      : STD_LOGIC;
130   signal XINT1B2_reg      : STD_LOGIC;
131   signal XINT1B3_reg      : STD_LOGIC;
132   signal XINT1B4_reg      : STD_LOGIC;
133   -- end interrupt routine registers
```

```
134
135   signal PWMOUT_reg      : STD_LOGIC_VECTOR (7 downto 0);
136   signal PWMprev        : STD_LOGIC;
137   signal PWMreq        : STD_LOGIC_VECTOR (1 downto 0);
138   signal PWMdb_state      : STD_LOGIC;
139
140   signal EVB_ENABLE      : STD_LOGIC;
141   signal PWMBprev        : STD_LOGIC;
142   signal PWMBreq        : STD_LOGIC_VECTOR (1 downto 0);
143   signal PWMBdb_state      : STD_LOGIC;
144
145   signal slow_clockA_per    : STD_LOGIC_VECTOR (4 downto 0);  -- period register
146   signal slow_clockB_per    : STD_LOGIC_VECTOR (4 downto 0);
147   signal slow_clockA      : STD_LOGIC;            -- the actual clock
148   signal slow_clockB      : STD_LOGIC;
149   signal slow_clockA_prev    : STD_LOGIC;            -- previous value
150   signal slow_clockB_prev    : STD_LOGIC;
151
152   signal EVACOMCON_reg    : STD_LOGIC_VECTOR (7 downto 0) := "00000000";
153   signal EVACONDB_reg      : STD_LOGIC_VECTOR (7 downto 0);
154   signal EVBCOMCON_reg    : STD_LOGIC_VECTOR (7 downto 0);
155   signal EVBCONDB_reg      : STD_LOGIC_VECTOR (7 downto 0);
156
157   signal CAP_reg        : STD_LOGIC_VECTOR (5 downto 0);
158
159   signal ANLGSW_reg      : STD_LOGIC_VECTOR (7 downto 0);
160
161   signal GPIO_reg        : STD_LOGIC_VECTOR (7 downto 0);
162
163   signal debug_reg      : STD_LOGIC_VECTOR (7 downto 0);  -- used to read/write for debug purposes
164
165   begin
166
167   spi: process( sclk,nSYNC,mosi,command_ok,command,data_1_ok,data_2_ok,data_1,write_point,spi_pos_count,minibus,GPIO,read_point,
168           SCIBMODE_reg,CAPQEP_reg,XINT1A_int,XINT1B_int,DIGIN,INDEX,INTSEL_reg,EVACOMCON_reg,EVACONDB_reg,EVBCOMCON_reg,
169           EVBCONDB_reg,ANLGSW_reg,debug_reg,PDPINTA)
170     begin
171
172 ------------------------------------------------------------------------------------------------------------------
173 -- spi, instruction decode and minibus comms:
174
175     --if(nRESET = '1') then
176
177       if(nSYNC = '0') then  -- chip set active
178
179       -- the use of flags to signify data ready points is not optimal. It can be adjusted so that within the
180       -- main program looking at the spi_pos_count determines when something is ready
181
182       ------------------------------------------------------------------------------------------------------
183       -- Read the SPI:
184         if(falling_edge(sclk)) then
185
186           mosi_reg(conv_integer(spi_pos_count)) <= mosi;  -- read in the value, MSB first
187           spi_pos_count <= spi_pos_count - 1;        -- decrement counter (starts at 23)
188
189       ---------------------------
190           if((spi_pos_count >= "10000")) then        -- (>16) if we're not as yet at the point of having valid data
191             command_ok <= '0';              -- ensure we don't go off doing a command
192             data_1_ok <= '0';
193             data_2_ok <= '0';
194             read_point <= '0';
195             write_point <= '0';
196           end if;
197
198           if(spi_pos_count = "01111") then        -- (15) at this point, we've read in enough to determine the command
199             command <= mosi_reg(23 downto 16);      -- copy to the command register
200             command_ok <= '1';
201           end if;
202
203           if(spi_pos_count = "01100") then        -- (12) READ POINT definition (currently set at pos=12 (3 clocks after the command
204             read_point <= '1';              -- register has been decoded)
205           end if;
206
207           if(spi_pos_count = "00111") then        -- (7) at this point, we've read in enough to write data
208             data_1 <= mosi_reg(15 downto 8);      -- copy to the data register
209             data_1_ok <= '1';
210           end if;
211
212           if(spi_pos_count = "00101") then        -- WRITE POINT definition(currently set at pos=5 (2 clocks after the data_1
213             write_point <= '1';              -- register has been read in)
```

```
214          end if;
215
216          if(spi_pos_count = "00000") then        -- DEPENDING ON HOW FAST IT DOES THIS, THESE NUMBERS MAY NEED TO BE CHANGED
217            -- data_2 is irrelevant and hence is not implemented
218            data_2_ok <= ’1’;
219          end if;
220
221        end if;  -- end SPI falling clock case
222      ------------------------------------------------------------------------------------------------
223      -- Write to SPI:
224        if(rising_edge(sclk)) then
225
226          if(spi_pos_count <= "01000") then            -- if we’re down to 7, time to put data out
227            miso <= miso_reg(conv_integer(spi_pos_count-1));  -- send out the value, MSB first
228                                 -- the -1  with the offset of 8 needs to be there for
229                                 -- it to work for some weird reason
230          end if;
231
232        end if;  -- end SPI rising clock case
233      ------------------------------------------------------------------------------------------------
234      -- Chip Select inactive:
235
236      else
237        miso <= ’Z’;                      -- high impedance 09/09/09 S.G
238        spi_pos_count <= "10111";          -- reset the counter (set to 23)
239
240        -- reset all registers
241        -- active high, hence clear them
242        command_ok <= ’0’;              -- ensure we don’t go off doing a command
243        data_1_ok <= ’0’;
244        data_2_ok <= ’0’;
245        read_point <= ’0’;
246        write_point <= ’0’;
247
248        -- minibus comms don’t occur outside the nSYNC low, so reset all
249        minibus <= "ZZZZZZZZ";
250        nWR <= ’1’;
251        nRD <= ’1’;
252        nMINIBUS <= ’1’;
253
254      end if;  -- end chip select case
255
256 --------------------------------------------------------------------------------------------------------------
257 -- Minibus communications:
258
259      if(command_ok = ’1’) then    -- if we’re allowed to operate (this flag is set when a new SPI command comes in)
260
261        if((command(7)=’0’ or command(6)=’0’)) then  -- check it’s a minibus command
262
263          case command (7 downto 6) is
264
265            when "00" =>
266              nCS <= "110";
267            when "01" =>
268              nCS <= "101";
269            when "10" =>
270              nCS <= "011";
271            when others =>    -- will never happen
272              nCS <= "111";
273
274          end case;
275
276          MA   <= command (3 downto 1);  -- map MA bits across
277
278          if(command(0)=’0’)  then      -- if it’s a write
279
280            if(data_2_ok = ’1’) then    -- (0) if the write command has completed
281              nWR <= ’1’;          -- reset all that we’ve changed
282              nMINIBUS <= ’1’;
283
284            elsif(write_point=’1’) then    -- (5) wait 2 SPI clocks before saying OK to write (setup time)
285              nWR <= ’0’;          -- setup write on the minibus
286
287            elsif(data_1_ok = ’1’) then    -- (7) if the data is now ready
288              minibus   <= data_1;    -- place the data on the bus
289              miso_reg   <= "00000000";  -- zero the read register
290              nRD <= ’1’;
291              nMINIBUS <= ’0’;      -- signal "go"
292
293            else              -- not doing anything yet
```

```
294             minibus <= "ZZZZZZZZ";
295             nWR <= '1';
296             nRD <= '1';
297 --              nCS <= "111";         -- reset chip selects high (PJM ADDED)
298             nMINIBUS <= '1';
299
300
301           end if;  -- end delay waited case
302
303
304         else                 -- else it's a read
305
306           -- following needs to take precedence over the reading point
307           if(data_1_ok = '1') then      -- (7) we've waited long enough to read it in
308             nRD <= '1';
309             nCS <= "111";         -- reset chip selects high (PJM ADDED)
310             nMINIBUS <= '1';
311
312           elsif(read_point = '1') then    -- (12) wait 3 SPI clocks before reading in the data (setup time)
313             miso_reg <= minibus;      -- read the data in from the bus
314
315           elsif(command_ok = '1') then    -- (15) kinda redundant but needed to clean up nicely
316             nWR <= '1';            -- signal a read
317             nRD <= '0';
318             nMINIBUS <= '0';        -- signal "go"
319
320           else                 -- not doing anything yet
321             minibus <= "ZZZZZZZZ";
322             nWR <= '1';
323             nRD <= '1';
324             nMINIBUS <= '1';
325
326           end if;  -- end delay waited case
327
328         end if; -- end read/write case
329
330       else   -- else, it's not to do with the minibus
331
332         -- leave the minibus comms in a known state
333         minibus <= "ZZZZZZZZ";
334         nWR <= '1';
335         nRD <= '1';
336 ---           nCS <= "111";         -- reset chip selects high (PJM ADDED)
337         nMINIBUS <= '1';
338 ---------------------------------------------------------------------------------------------------------------------------------------------
339 -- Peripheral device setup:
340
341         case(command(5 downto 1)) is      -- determine what peripheral/reg we're writing to
342
343           ----------------------------------------------------------------------
344           when "01101" =>            -- GPIO (0xDA)
345
346             if(command(0)='0') then      -- write command
347
348               if(data_1_ok = '1') then      -- if our write data is valid
349                 GPIO_reg <= data_1;        -- place it on the output
350
351               end if;  -- end data_1 valid if
352
353             else              -- else it must be a read
354               if(read_point='1') then        -- (13) wait 2 SPI clicks until sampling the DigIO
355                 miso_reg <= "000000" & GPIO;        -- place it in the register ready for output
356
357               end if;  -- end waited enough to sample DigIO
358
359             end if;  -- end read/write command if
360
361           ----------------------------------------------------------------------
362           when "00001" =>            -- SCIBMODE (0xC2)
363
364             if(command(0)='0') then      -- write command
365
366               if(data_1_ok = '1') then      -- if our write data is valid
367                 SCIBMODE_reg <= data_1(0);      -- place the written command
368               end if;  -- end data_1 valid case
369
370             else
371               if(read_point='1') then
372                 miso_reg <= "0000000" & SCIBMODE_reg;
373               end if;
```

```
374
375             end if;  -- end read/write command if
376
377           ----------------------------------------------------------------------
378           when "00010" =>            -- CAPQEP (0xC4)
379
380             if(command(0)='0') then      -- write command
381               if(data_1_ok = '1') then  -- if our write data is valid
382                 CAPQEP_reg <= data_1(0);
383
384               end if; -- end data_1 valid if
385
386             else              -- else, read command
387               if(read_point='1') then
388                 miso_reg <= "0000000" & CAPQEP_reg;
389               end if; -- end read point reached if
390
391             end if; -- end read/write command if
392
393           ----------------------------------------------------------------------
394           when "00011" =>            -- INTSEL (0xC6)
395
396             if(command(0)='0') then      -- write command
397               if(data_1_ok = '1') then  -- if our write data is valid
398                 INTSEL_reg <= data_1;
399               end if; -- end data_1 valid if
400
401             else              -- else, read command -- SPECIAL, corresponds to (0xC7)
402               if(read_point='1') then
403                 miso_reg <= "000000" & XINT1B_int & XINT1A_int;    -- pass the interrupt that occured up to the DSP
404
405                 if(data_2_ok = '1') then             -- wait until the SPI command is done, then reset interrupts
406                   int_clear <= '1';                -- clear the interrupt registers (this is done in the int process below)
407                 else
408                   int_clear <= '0';                -- provide means to let the interrupt registers to be re-filled
409                 end if;
410
411               end if; -- end read point reached if
412
413             end if; -- end read/write command if
414
415           ----------------------------------------------------------------------
416           when "00100" =>            -- DEBUG ONLY (0xC8)
417             if(command(0)='1') then      -- read command
418               if(read_point='1') then    -- if our read data is valid
419                 miso_reg <= INTSEL_reg;
420               end if; -- end data_1 valid if
421             end if;
422
423           ----------------------------------------------------------------------
424           --  Mod Dinesh 20th November 2009
425           --  This code will run when this command is written to the address
426           --  ADD_EVB: 11 00101 0 0000 0000 0000 000(0/1)
427           --  It will do the following things:
428           --  1) activate EVA_enable, which will
429           --  route EVB signals (first 4 only, single phase) to the gate drivers
430
431           when "00101" =>
432             if(command(0)='0') then  -- write command
433               if(data_1_ok = '1') then      -- if our write data is valid
434                 if (data_1(0) = '1') then      -- this means that you activate EVB
435                   EVB_enable <= '1';
436                 else
437                   EVB_enable <= '0';
438                 end if;
439               end if;
440             end if;
441           ----------------------------------------------------------------------
442           when "01000" =>          -- EVACOMCON (0xD0)
443
444             if(command(0)='0') then      -- write command
445               if(data_1_ok = '1') then    -- if our write data is valid
446
447                 -- Additional case added 21/10/2009 by S.G
448                 if(PDPINTA = '1') then      -- if there's no fault
449                   EVACOMCON_reg <= data_1;    -- copy in the data
450                 else                        -- otherwise
451                   EVACOMCON_reg <= (data_1 and "11111110");  -- zero the LSB
452                 end if;
453
```

```
454             end if; -- end data_1 valid if
455
456           else              -- else, read command
457             if(read_point='1') then
458               miso_reg <= EVACOMCON_reg;
459             end if; -- end read point reached if
460
461           end if; -- end read/write command if
462
463         ---------------------------------------------------------------------
464         when "01001" =>          -- EVACONDB (0xD2)
465
466           if(command(0)='0') then      -- write command
467             if(data_1_ok  = '1') then  -- if our write data is valid
468               EVACONDB_reg <= data_1;
469             end if; -- end data_1 valid if
470
471           else              -- else, read command
472             if(read_point='1') then
473               miso_reg <= EVACONDB_reg;
474             end if; -- end read point reached if
475
476           end if; -- end read/write command if
477
478         ---------------------------------------------------------------------
479         when "01010" =>          -- EVBCOMCON (0xD4)
480
481           if(command(0)='0') then      -- write command
482             if(data_1_ok = '1') then  -- if our write data is valid
483               EVBCOMCON_reg <= data_1;
484             end if; -- end data_1 valid if
485
486           else              -- else, read command
487             if(read_point='1') then
488               miso_reg <= EVBCOMCON_reg;
489             end if; -- end read point reached if
490
491           end if; -- end read/write command if
492
493         ---------------------------------------------------------------------
494         when "01011" =>          -- EVACONDB (0xD6)
495
496           if(command(0)='0') then      -- write command
497             if(data_1_ok = '1') then  -- if our write data is valid
498               EVBCONDB_reg <= data_1;
499             end if; -- end data_1 valid if
500
501           else              -- else, read command
502             if(read_point='1') then
503               miso_reg <= EVBCONDB_reg;
504             end if; -- end read point reached if
505
506           end if; -- end read/write command if
507
508         ---------------------------------------------------------------------
509         when "01100" =>          -- ANLGSW (0xD8)
510
511           if(command(0)='0') then      -- write command
512             if(data_1_ok = '1') then  -- if our write data is valid
513               ANLGSW_reg <= data_1;
514
515               -- MSB [IN1  IN2  IN3  INA  INB  INC  IND] LSB
516               ANIN(6) <= ANLGSW_reg(4);
517               ANIN(5) <= ANLGSW_reg(5);
518               ANIN(4) <= ANLGSW_reg(6);
519               ANIN(3) <= ANLGSW_reg(0);
520               ANIN(2) <= ANLGSW_reg(1);
521               ANIN(1) <= ANLGSW_reg(2);
522               ANIN(0) <= ANLGSW_reg(3);
523
524             end if; -- end data_1 valid if
525
526           else              -- else, read command
527             if(read_point='1') then
528               miso_reg <= ANLGSW_reg;
529             end if; -- end read point reached if
530
531           end if; -- end read/write command if
532
533         ---------------------------------------------------------------------
```

```
534
535             when "11111" =>             -- DEBUG (temp register)
536
537                 if(command(0)='0') then       -- write command
538                   if(data_1_ok = '1') then         -- if our write data is valid
539                     debug_reg <= data_1;           -- place it on the output
540                   end if;  -- end data_1 valid if
541
542                 else                   -- else it must be a read
543                   if(read_point='1') then
544                     miso_reg <= debug_reg;         -- place it in the register ready for output
545                   end if;
546
547                 end if;  -- end read/write command if
548
549             ----------------------------------------------------------------------
550
551             when others =>
552                 miso_reg <= "00000000";
553           end case;
554
555         end if;  -- end minibus check case
556
557       else  -- command is not okay yet
558
559         minibus <= "ZZZZZZZZ";
560         nWR <= '1';      -- reset all the minibus paraphernalia
561         nRD <= '1';
562         nCS <= "111";        -- reset chip selects high (PJM ADDED)
563         nMINIBUS <= '1';
564
565       end if;  -- end command not ready case
566
567     --end if;  -- end reset if
568
569   end process spi;
570 --------------------------------------------------------------------------------------------------------------
571 -- CAPQEP passthrough:
572 -- regrettably, needs to be synchronous
573
574   CAPQEP_proc: process(clock,DIGIN,INDEX)
575
576     begin
577
578       if(rising_edge(clock)) then
579
580           if(CAPQEP_reg='1') then    -- defined in page 31 of the manual
581             CAP_reg(0) <= DIGIN(0);
582             CAP_reg(1) <= DIGIN(1);
583             CAP_reg(3) <= DIGIN(2);
584             CAP_reg(4) <= DIGIN(3);
585             CAP_reg(2) <= INDEX(0);
586             CAP_reg(5) <= INDEX(1);
587           else
588             CAP_reg(3) <= DIGIN(0);
589             CAP_reg(4) <= DIGIN(1);
590             CAP_reg(0) <= DIGIN(2);
591             CAP_reg(1) <= DIGIN(3);
592             CAP_reg(5) <= INDEX(0);
593             CAP_reg(2) <= INDEX(1);
594
595           end if;  -- end CAPQEP register selection
596
597       end if;  -- end rising clock edge
598
599   end process CAPQEP_proc;
600
601 --------------------------------------------------------------------------------------------------------------
602
603   int_sel: process(XINT1A,XINT1B,INTSEL_reg,INTSRC_reg,XINT1A_int,int_clear,XINT1A1_reg,XINT1A2_reg,XINT1A3_reg,XINT1A4_reg,
604            XINT1B1_reg,XINT1B2_reg,XINT1B3_reg,XINT1B4_reg)
605
606     begin
607
608     -- major assumption is that XINT1 is asserted when an interrupt occurs
609
610   -- interrupt A:
611
612       if(INTSEL_reg(2 downto 0) = "011") then    -- rising edge (and enabled)
613
```

```
614         --if(rising_edge(XINT1A)) then
615         --  XINT1A1_reg <= '1';
616
617         --end if;  -- end XINT1A rising edge
618
619       end if; -- end if
620
621       if(INTSEL_reg(2 downto 0) = "101") then    -- falling edge (and enabled)
622
623         if(falling_edge(XINT1A)) then
624           XINT1A2_reg <= '1';
625
626         end if;  -- end XINT1A falling edge
627
628       end if; -- end if
629
630       if(INTSEL_reg(2 downto 0) = "001") then   -- active low (and enabled)
631
632         if(XINT1A = '0') then
633           XINT1A3_reg <= '1';
634         end if;
635
636       end if;
637
638       if(INTSEL_reg(2 downto 0) = "111") then    -- active high (and enabled)
639
640         if(XINT1A = '1') then
641           XINT1A4_reg <= '1';
642         end if;
643
644       end if;
645
646       if((int_clear = '1') or (INTSEL_reg(0) = '0')) then    -- activated if the user reads the INTSRC register
647         XINT1A1_reg <= '0';                  -- or if it's been disabled
648         XINT1A2_reg <= '0';
649         XINT1A3_reg <= '0';
650         XINT1A4_reg <= '0';
651       end if;
652
653       XINT1A_int <= XINT1A1_reg or XINT1A2_reg or XINT1A3_reg or XINT1A4_reg;
654
655
656   -- interrupt B:
657
658       if(INTSEL_reg(6 downto 4) = "011") then    -- rising edge (and enabled)
659
660         --if(rising_edge(XINT1B)) then
661         --  XINT1B1_reg <= '1';
662
663         --end if;  -- end XINT1B rising edge
664
665       end if; -- end if
666
667       if(INTSEL_reg(6 downto 4) = "101") then    -- falling edge (and enabled)
668
669         if(falling_edge(XINT1B)) then
670           XINT1B2_reg <= '1';
671
672         end if;  -- end XINT1B falling edge
673
674       end if; -- end if
675
676       if(INTSEL_reg(6 downto 4) = "001") then   -- active low (and enabled)
677
678         if(XINT1B = '0') then
679           XINT1B3_reg <= '1';
680         end if;
681
682       end if;
683
684       if(INTSEL_reg(6 downto 4) = "111") then    -- active high (and enabled)
685
686         if(XINT1B = '1') then
687           XINT1B4_reg <= '1';
688         end if;
689
690       end if;
691
692       if((int_clear = '1') or (INTSEL_reg(0) = '0')) then    -- activated if the user reads the INTSRC register
693         XINT1B1_reg <= '0';                  -- of if it's been disabled
```

```
694         XINT1B2_reg <= '0';
695         XINT1B3_reg <= '0';
696         XINT1B4_reg <= '0';
697       end if;
698
699       XINT1B_int <= XINT1B1_reg or XINT1B2_reg or XINT1B3_reg or XINT1B4_reg;
700
701   end process int_sel;
702
703 ------------------------------------------------------------------------------------------------------------
704 -- UNTESTED!
705
706   pwmA_sys: process(clock,TxPWM,PWMIN,EVACOMCON_reg,EVACONDB_reg,slow_clockA,slow_clockA_prev)
707
708     variable wait_reg    : STD_LOGIC_VECTOR (3 downto 0);
709     variable period      : STD_LOGIC_VECTOR (3 downto 0);
710
711   begin
712
713     -- ASSUMPTION: T1PWM and T2PWM are the passthrough digital IO for PWM7/8
714
715     -- no hysteresis implemented as yet, passthrough 6 PWM outputs
716
717     -- this is a messy implementation as we can't look for a rising AND falling edge on the
718     -- PWM input, which means we need to clock it and compare between clock cycles - but because
719     -- we're doing this, we can't clock off the clock that's been divided down (can't write to
720     -- the same variables in diffent clock edges) so yeah, that's why it has this structure.
721
722       if (EVB_ENABLE = '0') then
723         PWMOUT(5 downto 0) <= PWMIN;
724
725         period := EVACONDB_reg(6 downto 3);  -- map period register across (add a zero to match reg sizes)
726
727         if(EVACOMCON_reg(7) = '1') then    -- setup complimentary deadtime legs sources from T1PWM (t1 is MSB)
728
729           if(rising_edge(clock)) then
730
731             if(PWMdb_state = '0') then      -- the change detection state
732
733               if(TxPWM(1) /= PWMprev) then    -- if a state change has occurred
734
735                 PWMOUT(7 downto 6) <= "00";      -- turn off the legs
736                 PWMprev <= TxPWM(1);        -- record what our new value is
737
738                 if(TxPWM(1)='1') then        -- if we've requested to go high
739                   PWMreq <= "10";            -- make a note of what the final state should be
740                 else              -- else we've requested to go low
741                   PWMreq <= "01";            -- make a note of what the final state should be
742                 end if;
743
744                 PWMdb_state <= '1';          -- jump to the other state to wait
745
746               end if; --end state change if
747
748             else              -- the waiting and implementation state
749
750               if(slow_clockA /= slow_clockA_prev) then  -- if our slow clock has toggled
751
752                 slow_clockA_prev <= slow_clockA;    -- record what our new value is
753                 wait_reg := wait_reg + 1;        -- increment period counter
754
755                 if(wait_reg = period) then    -- if we've waited long enough
756
757                   PWMOUT(6) <= PWMreq(1);      -- implement the requested state
758                   PWMOUT(7) <= PWMreq(0);
759                   PWMdb_state <= '0';          -- go back to waiting for a change
760                   wait_reg := "0000";          -- reset the waiting register
761
762                 end if;
763
764               end if;
765
766             end if;  -- end deadband state toggle
767
768           end if;  -- end main clock rising edge
769
770         else                  -- else, pass through the TxPWM legs
771           PWMOUT(7 downto 6) <= TxPWM;
772
773         end if;
```

```
774     else
775
776       PWMOUT(3 downto 0) <= PWMIN(3 downto 0);
777       PWMOUT(7 downto 4) <= DIGIN(3 downto 0);
778     end if;
779   end process pwmA_sys;
780
781 ----------------------------------------------------------------------------------------------------------------
782
783   -- PWMB7 is the primary
784
785   pwmB_sys: process(clock,EVBCOMCON_reg,EVBCONDB_reg,T3PWM,T4PWM,slow_clockB,slow_clockB_prev)
786     variable wait_reg    : STD_LOGIC_VECTOR (3 downto 0);
787     variable period      : STD_LOGIC_VECTOR (3 downto 0);
788
789   begin
790
791     if(EVACOMCON_reg(0) = '0') then    -- S.G added 20/10/2009 to protect PWM outputs on startup
792
793       PWMB7 <= '0';
794       PWMB8 <= '0';
795
796     else
797
798       period := EVBCONDB_reg(6 downto 3);  -- map period register across (add a zero to match reg sizes)
799
800       if(EVBCOMCON_reg(7) = '1') then    -- setup complimentary deadtime legs sources from T1PWM (t1 is MSB)
801
802         if(rising_edge(clock)) then
803
804           if(PWMBdb_state = '0') then      -- the change detection state
805
806             if(T3PWM /= PWMBprev) then      -- if a state change has occurred
807
808               PWMB7 <= '0';            -- turn off the legs
809               PWMB8 <= '0';
810               PWMBprev <= T3PWM;          -- record what our new value is
811
812               if(T3PWM='1') then          -- if we've requested to go high
813                 PWMBreq <= "10";          -- make a note of what the final state should be
814               else                -- else we've requested to go low
815                 PWMBreq <= "01";          -- make a note of what the final state should be
816               end if;
817
818               PWMBdb_state <= '1';        -- jump to the other state to wait
819
820             end if; --end state change if
821
822           else              -- the waiting and implementation state
823
824             if(slow_clockB /= slow_clockB_prev) then  -- if our slow clock has toggled
825
826               slow_clockB_prev <= slow_clockB;    -- record what our new value is
827               wait_reg := wait_reg + 1;        -- increment period counter
828
829               if(wait_reg = period) then        -- if we've waited long enough
830
831                 PWMB7 <= PWMBreq(1);          -- implement the requested state
832                 PWMB8 <= PWMBreq(0);
833                 PWMBdb_state <= '0';            -- go back to waiting for a change
834                 wait_reg := "0000";          -- reset the waiting register
835
836               end if;
837
838             end if;
839
840           end if;  -- end deadband state toggle
841
842         end if;  -- end main clock rising edge
843
844       else  -- else, pass through the TxPWM legs
845
846         PWMB7 <= T3PWM;
847         PWMB8 <= T4PWM;
848
849       end if;
850
851     end if; --end PWM output protection if
852
853   end process pwmB_sys;
```

```
854
855 ------------------------------------------------------------------------------------------------------------
856 -- TESTED OK (with exception of /1 clock which freezes the DSP)
857
858   clock_div: process(clock,EVACONDB_reg,EVBCONDB_reg)
859
860     variable slow_clock_regA  : STD_LOGIC_VECTOR(7 downto 0);
861     variable slow_clock_regB  : STD_LOGIC_VECTOR(7 downto 0);
862
863     begin
864
865     -- for PWM set A:
866
867       if (EVACONDB_reg(2 downto 0) = "000") then  -- special /1 case
868
869         --slow_clockA <= clock;      -- INCLUSION OF THIS LINE WILL HANG THE DSP! (soft clock output?)
870
871       else
872
873         if(rising_edge(clock)) then
874
875           slow_clock_regA := slow_clock_regA + 1;
876
877           case EVACONDB_reg(2 downto 0) is  -- switch on clock scaling values
878
879           when "001" =>            -- /2 clock
880             if(slow_clock_regA = "00000001") then
881               slow_clockA <= not slow_clockA;
882               slow_clock_regA := "00000000";
883             end if;
884
885           when "010" =>            -- /4 clock
886             if(slow_clock_regA = "00000010") then
887               slow_clockA <= not slow_clockA;
888               slow_clock_regA := "00000000";
889             end if;
890
891           when "011" =>            -- /8 clock
892             if(slow_clock_regA = "00000100") then
893               slow_clockA <= not slow_clockA;
894               slow_clock_regA := "00000000";
895             end if;
896
897           when "100" =>            -- /16 clock
898             if(slow_clock_regA = "00001000") then
899               slow_clockA <= not slow_clockA;
900               slow_clock_regA := "00000000";
901             end if;
902
903           when others =>            -- /32 clock
904             if(slow_clock_regA = "00010000") then
905               slow_clockA <= not slow_clockA;
906               slow_clock_regA := "00000000";
907             end if;
908
909           end case;
910
911         end if; -- end clock rising
912
913       end if;  -- end special /1 case
914
915
916     -- for PWM set B:
917
918       if (EVBCONDB_reg(2 downto 0) = "000") then  -- special /1 case
919
920         --slow_clockB <= clock;
921
922       else
923
924         if(rising_edge(clock)) then
925
926           slow_clock_regB := slow_clock_regB + 1;
927
928           case EVBCONDB_reg(2 downto 0) is  -- switch on clock scaling values
929
930           when "001" =>            -- /2 clock
931             if(slow_clock_regB = "00000001") then
932               slow_clockB <= not slow_clockB;
933               slow_clock_regB := "00000000";
```

```
934           end if;
935
936         when "010" =>           -- /4 clock
937           if(slow_clock_regB = "00000010") then
938             slow_clockB <= not slow_clockB;
939             slow_clock_regB := "00000000";
940           end if;
941
942         when "011" =>           -- /8 clock
943           if(slow_clock_regB = "00000100") then
944             slow_clockB <= not slow_clockB;
945             slow_clock_regB := "00000000";
946           end if;
947
948         when "100" =>           -- /16 clock
949           if(slow_clock_regB = "00001000") then
950             slow_clockB <= not slow_clockB;
951             slow_clock_regB := "00000000";
952           end if;
953
954         when others =>           -- /32 clock
955           if(slow_clock_regB = "00010000") then
956             slow_clockB <= not slow_clockB;
957             slow_clock_regB := "00000000";
958           end if;
959
960         end case;
961
962       end if; -- end clock rising
963
964     end if;  -- end special /1 case
965
966   end process clock_div;
967 ------------------------------------------------------------------------------------------------------------
968 -- process to ensure a known startup value and a known fault value. Added 20/10/2009 by S.G.
969   pwm_en: process(EVACOMCON_reg,PDPINTA)
970     begin
971
972     if((EVACOMCON_reg(0) = '0') or (PDPINTA = '0')) then   -- if the control register is zeroed or a hardware fault exists
973       PWMen <= '0';    -- drive the PWM outputs low
974     else
975       PWMen <= '1';    -- else, enable the passthrough
976     end if;
977
978   end process pwm_en;
979
980 ------------------------------------------------------------------------------------------------------------
981
982   -- asynchronous declarations
983   GPIO <= GPIO_reg(1 downto 0);
984   SCIBMODE <= SCIBMODE_reg;
985   XINT1 <= (XINT1A_int or XINT1B_int);
986   CAP <= CAP_reg;
987   debug <= slow_clockA;
988
989
990 ------------------------------------------------------------------------------------------------------------
991
992 end behaviour;
```

A.2.2 DSP Code – Dual Active Bridge

```
1 /**
2 \file
3 \brief Main system definitions
4
5 \par Developed By:
6   Creative Power Technologies, (C) Copyright 2009
7 \author A.McIver
8 \par History:
9 \li  23/04/09 AM - initial creation
10 \ Modified Dinesh Segaran
11 \    26/08/10 DS - Fixed Point implementation of the Adaptive Controlled
12 \                  Bidirectional DC-DC Converter
13 */
14
15
16 /* =======================================================================
17 __Definitions()
18 ========================================================================== */
19
20 #define __SQRT2          1.4142135624
21 #define __SQRT3          1.7320508075
22 #define __PI            3.1415926535
23 #define __PI_2          __PI/2.0
24 #define __INVPI         1/__PI
25 #define __INVPI_2       1/__PI_2
26
27 #define SYSCLK_OUT        (150e6)
28 #define  HSPCLK          (SYSCLK_OUT)
29 #define LSPCLK          (SYSCLK_OUT/4)
30
31 /* =======================================================================
32 __State_Simple_Definitions()
33 ========================================================================== */
34
35 /** Simple State Machine Type */
36  typedef void (* funcPtr)(void);
37 typedef struct
38 {
39   funcPtr f;
40   unsigned int call_count;
41   unsigned char first;
42 } type_state;
43
44
45 /* Simple State Handling Macros */
46 #define SS_NEXT(_s_,_f_)    { _s_.f = (funcPtr)_f_; \
47                   _s_.call_count = 0; \
48                   _s_.first = 1; }
49 #define SS_IS_FIRST(_s_)    (_s_.first == 1)
50 #define SS_DONE(_s_)      { _s_.first = 0; }
51 #define SS_DO(_s_)          { _s_.call_count++; \
52                   ((*(_s_.f))()); }
53 #define SS_IS_PRESENT(_s_,_f_)  (_s_.f == (funcPtr)_f_)
54
55
56 /* =======================================================================
57 __Grab_Code_Definitions()
58 ========================================================================== */
59 /**/
60 #define GRAB_INCLUDE
61
62 #ifdef GRAB_INCLUDE
63
64 //#define GRAB_LONG
65 #define GRAB_DOUBLE
66
67 // grab array size
68 #define GRAB_LENGTH        20
69 #define GRAB_WIDTH        5
70
71 // modes
72 #define GRAB_GO            0
73 #define GRAB_WAIT          1
74 #define GRAB_TRIGGER      2
75 #define GRAB_STOPPED      3
76 #define GRAB_SHOW          4
77
```

```
 78 // macros
 79 #define GrabStart()        grab_mode = GRAB_TRIGGER;
 80 #define GrabStop()        grab_mode = GRAB_STOPPED;
 81 #define GrabRun()          grab_mode = GRAB_GO;
 82 #define GrabShow()        grab_mode = GRAB_SHOW;
 83
 84 #define GrabClear()        { grab_mode = GRAB_WAIT; \
 85                             grab_index = 0; }
 86
 87 #define GrabTriggered()      (grab_mode == GRAB_TRIGGER)
 88 #define GrabRunning()        (grab_mode == GRAB_GO)
 89 #define GrabStopped()        (grab_mode == GRAB_STOPPED)
 90 #define GrabAvail()          (grab_mode >= GRAB_STOPPED)
 91 #define GrabShowTrigger()    (grab_mode == GRAB_SHOW)
 92
 93 #define GrabStore(_loc_,_data_)  grab_array[grab_index][_loc_] = _data_;
 94
 95 #define GrabStep()        { grab_index++; \
 96                   if (grab_index >= GRAB_LENGTH) \
 97                       grab_mode = GRAB_STOPPED; }
 98
 99 // variables
100 extern int16
101   step,
102   grab_mode,
103   grab_index,
104   set_vref;
105
106 extern long
107   volt_req,wo;
108
109 #ifdef GRAB_DOUBLE
110 extern double      //call this double normally
111   grab_array[GRAB_LENGTH][GRAB_WIDTH];
112 #endif
113
114 #ifdef GRAB_LONG
115 extern long      //call this double normally
116   grab_array[GRAB_LENGTH][GRAB_WIDTH];
117 #endif
118
119
120 // functions
121 void GrabDisplay(int16 index);
122 void GrabInit(void);
123
124 #endif
125 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

```
1  /**
2 \file
3 \brief System software for the DA-2810 Demo code
4
5
6 \par Developed By:
7   Creative Power Technologies, (C) Copyright 2009
8 \author A.McIver
9 \par History:
10 \li  23/04/09 AM - initial creation
11 \    26/08/10 DS - Fixed Point implementation of the Adaptive Controlled
12 \                 Bidirectional DC-DC Converter
13
14 */
15
16 // compiler standard include files
17 #include <stdlib.h>
18 #include <stdio.h>
19 #include <math.h>
20
21 // processor standard include files
22 #include <DSP281x_Device.h>
23 #include <DSP281x_Examples.h>
24
25 #ifdef COM0_CONSOLE
26 #include <bios0.h>
27 #endif
28 #ifdef COM1_CONSOLE
29 #include <bios1.h>
30 #endif
31
32 // board standard include files
33 #include <lib_mini2810.h>
34 #include <dac_ad56.h>
35 #include <lib_cpld.h>
36 #include <lib_giib.h>
37
38 // common project include files
39
40 // local include files
41 #include "main.h"
42 #include "conio.h"
43 #include "vsi.h"
44 //IqMath toolbox
45 //#include <IQmathLib.h>
46
47 /* ========================================================================
48 __Definitions()
49 ======================================================================== */
50 // Serial step in frequency
51 #define FREQ_STEP 100
52
53 //Serial step in phase
54 #define PHASE_STEP_LARGE 10
55 #define PHASE_STEP_SMALL 1
56
57 /* ========================================================================
58 __Typedefs()
59 ======================================================================== */
60
61 /// Time related flag type
62 /** This structure holds flags used in background timing. */
63 typedef struct
64 {
65   Uint16
66     msec:1,    ///< millisecond flag
67     msec10:1,  ///< 10ms flag
68     sec0_1:1,  ///< tenth of a second flag
69     sec:1;    ///< second flag
70 } type_time_flag;
71
72
73 /* ========================================================================
74 __Variables()
75 ======================================================================== */
76
77 #ifndef BUILD_RAM
78 // These are defined by the linker (see F2812.cmd)
79 extern Uint16 RamfuncsLoadStart;
80 extern Uint16 RamfuncsLoadEnd;
```

```
 81 extern Uint16 RamfuncsRunStart;
 82 #endif
 83
 84 // Background variables
 85 Uint16
 86   quit = 0;  ///< exit flag
 87
 88
 89 /// timing variable
 90 type_time_flag
 91   time =
 92   {
 93     0,0,0,0 // flags
 94   };
 95
 96 Uint32
 97   idle_count = 0,        ///< count of idle time in the background
 98   idle_count_old = 0,      ///< previous count of idle time
 99   idle_diff = 0;        ///< change in idle time btwn low speed tasks
100
101 char
102   str[40];        // string for displays
103
104 //to display correctly
105 int initial=0;
106
107 /********************
108 _External_Variables()
109 ********************/
110 //debug variables. so they can be displayed
111 extern int16   FF_ENABLE,
112               AC_FF,
113               DT_COMP,
114               phase_aug_DT_fixed;
115 extern int32  I3_fixed,
116               I4_fixed;
117 /* =====================================================================
118 __Local_Function_Prototypes()
119 ======================================================================== */
120
121 /* 1 second interrupt for display */
122 interrupt void isr_cpu_timer0(void);
123
124 /// display operating info
125 void com_display(void);
126
127 /// display help
128 void display_help(void);
129
130 /* process keyboard input */
131 void com_keyboard(void);
132
133 /* =====================================================================
134 __Grab_Variables()
135 ======================================================================== */
136
137 #ifdef GRAB_INCLUDE
138 //#pragma DATA_SECTION(grab_array, "bss_grab")
139 int16
140   step=0,
141   grab_mode = GRAB_STOPPED,
142   grab_index,
143   set_vref=0;
144 long
145   volt_req=10,
146   wo=314;
147   #ifdef GRAB_DOUBLE
148   double
149     grab_array[GRAB_LENGTH][GRAB_WIDTH];
150   #endif
151   #ifdef GRAB_LONG
152   long
153     grab_array[GRAB_LENGTH][GRAB_WIDTH];
154   #endif
155
156 #endif
157
158 /* =====================================================================
159 __Serial_input_variables()
160 ======================================================================== */
```

```
161 int mod_depth_serial =10000;        //In 2810 modulation depth go from 0 to 1000 (0-100%)
162 int step_mod_depth_serial = 100;
163 int mod_depth_max = 15000;
164
165 int16 f_switch_serial = 20000;          //Fundamental modulation frequency in Hz
166 double phase_serial=0.0;
167
168 int16 vref_serial = 10,
169       mosfet_count;
170
171 /* ======================================================================= */
172 /* Main */
173 /* ======================================================================= */
174 /* Idle time benchmark:
175 \li Ram based program with only bios interrupt and an empty main loop gives an
176 idle_diff of 4.69M (4,685,900)
177 \li 23/03/09 V1.02 1.23M with no modbus running
178 */
179 void main(void)
180 {
181   static int
182   i = 0;
183 //  initial=0;
184
185 // Disable CPU interrupts
186   DINT;
187 // Initialise DSP for PCB
188   lib_mini2810_init(150/*MHz*/,37500/*kHz*/,150000/*kHz*/,LIB_EVAENCLK
189     |LIB_EVBENCLK|LIB_ADCENCLK|LIB_SCIAENCLK|LIB_SCIBENCLK|LIB_MCBSPENCLK);
190
191   InitGpio();
192   spi_init(MODE_CPLD);
193 //  SpiaRegs.SPICCR.bit.SPILBK = 1;        //Set SPI on loop back for testing
194   cpld_reg_init();
195   giib_init();
196
197 // Initialize the PIE control registers to their default state.
198   InitPieCtrl();
199 // Disable CPU interrupts and clear all CPU interrupt flags:
200   IER = 0x0000;
201   IFR = 0x0000;
202 // Initialize the PIE vector table with pointers to the shell Interrupt
203 // Service Routines (ISR).
204 // This will populate the entire table, even if the interrupt
205 // is not used in this example.  This is useful for debug purposes.
206 // The shell ISR routines are found in DSP281x_DefaultIsr.c.
207 // This function is found in DSP281x_PieVect.c.
208   InitPieVectTable();
209
210 #ifndef BUILD_RAM
211 // Copy time critical code and Flash setup code to RAM
212 // The  RamfuncsLoadStart, RamfuncsLoadEnd, and RamfuncsRunStart
213 // symbols are created by the linker. Refer to the F2810.cmd file.
214   MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);
215
216 // Call Flash Initialization to setup flash waitstates
217 // This function must reside in RAM
218   InitFlash();
219 #endif
220
221   // Initialise COM port
222   bios_init_COM1(9600L);
223   InitAdc();
224   InitCpuTimers();
225
226 // Configure CPU-Timer 0 to interrupt every tenth of a second:
227 // 150MHz CPU Freq, 1ms Period (in uSeconds)
228   ConfigCpuTimer(&CpuTimer0, 150.0/*MHz*/, 1000.0/*us*/);
229   StartCpuTimer0();
230
231 // Interrupts that are used in this example are re-mapped to
232 // ISR functions found within this file.
233   EALLOW;  // This is needed to write to EALLOW protected register
234   PieVectTable.TINT0 = &isr_cpu_timer0;
235   EDIS;    // This is needed to disable write to EALLOW protected registers
236
237 // Enable TINT0 in the PIE: Group 1 interrupt 7
238   PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
239   IER |= M_INT1; // Enable CPU Interrupt 1
240   vsi_init();
```

```
241   EnableInterrupts();
242   //waste some time, so that the program can finish writing to the screen
243
244 #ifdef GRAB_INCLUDE
245   GrabInit();
246 #endif
247   spi_init(MODE_DAC);
248   spi_set_mode(MODE_DAC);
249   dac_init();
250   dac_set_ref(DAC_MODULE_D1,DAC_INT_REF);
251   dac_power_down(DAC_MODULE_D1,0x0F);
252   dac_write(DAC_MODULE_D1,DAC_WRn_UPDn,DAC_ADDR_ALL,2047);
253   spi_set_mode(MODE_CPLD);      //Use mode setting for CPLD for SPI to initialize SPI setting
254   DISABLE_CPLD();
255 /*
256 void main_loop(void)
257 */
258   while(quit == 0)
259   {
260
261     com_keyboard(); // process keypresses
262
263     if (time.msec != 0) // millisecond events
264     {
265       time.msec = 0;
266       vsi_state_machine();
267
268     }
269     else if (time.msec10 != 0) // ten millisecond events
270     {
271       time.msec10 = 0;
272     }
273       else if (time.sec0_1 != 0) // tenth of second events
274     {
275       time.sec0_1 = 0;
276       switch(initial)
277       {
278         /* case 0 never happens */
279         case 1: puts_COM1("\n GIIB-Based Bidirectional DC-DC Converter 2011");break;
280         case 2: puts_COM1("\n\te/d - start/end\n");break;
281
282         #ifdef OPEN_LOOP
283         case 3: puts_COM1("\tz/Z - Small/Large Phase Shift Increase\n"); break;
284         case 4: puts_COM1("\tx/X - Small/Large Phase Shift Decrease\n"); break;
285         #endif
286         #ifdef CLOSED_LOOP
287
288         //Vref
289         case 5: puts_COM1("\tm/M - Small/Large Vref Increase\n"); break;
290         case 6: puts_COM1("\tn/N - Small/Large Vref Decrease\n"); break;
291
292         //FF
293         case 7: puts_COM1("\tf/F - Feed Forward Disable/Enable\n"); break;
294         case 8: puts_COM1("\ta/A - AC/DC Feed Forward Selection\n"); break;
295
296         //DT Comp
297         case 9: puts_COM1("\tc/C - Deadtime Compensation Disable/Enable\n"); break;
298         #endif
299         case 10: puts_COM1("\tg/h - Start/Display Grab\n");break;
300         case 11: puts_COM1("\ts - Stop Grab\n");break;
301         case 12: puts_COM1("\tH  -  Display Help\n");break;
302         default: break;
303       }
304       if (initial<20) initial++;
305
306       if(GrabShowTrigger() && i < GRAB_LENGTH){
307         //GrabDisplay(0xFFFF);
308         GrabDisplay(i);
309         i++;
310         //GrabStop();
311       }
312       else if(GrabShowTrigger() && i == GRAB_LENGTH){
313         GrabStop();
314         i = 0;
315       }
316     }
317     else if (time.sec != 0) // update every 1sec
318     {
319 //      puts_COM1("\n counter:");
320 //      put_d(initial);
```

```
321       time.sec = 0;
322       idle_diff = idle_count - idle_count_old;
323       idle_count_old = idle_count;
324       if (initial>=15)  com_display();
325     }
326     else // low priority events
327     {
328       idle_count++;
329     }
330
331   } /* end while quit == 0 */
332
333 //  DISABLE_PWM();
334   EvaRegs.T1CON.bit.TENABLE = 0;
335   EvaRegs.ACTRA.all = 0x0000;
336   DINT;
337 } /* end main */
338
339
340 /* ====================================================================
341 __Local_Functions()
342 ===================================================================== */
343
344 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
345 /**
346 Display operating information out COM0.
347
348 \author A.McIver
349 \par History:
350 \li  22/06/05 AM - initial creation
351
352 \param[in] mode Select whether to start a new display option
353 */
354 void com_display(void)
355 {
356   Uint16
357     status;
358   puts_COM1("\n");
359   //If system is displaying grab data do nothing otherwise display normal status stuff
360   if(GrabShowTrigger()){
361   }
362   else
363     {
364       status = vsi_get_status();
365       if (status == VSI_FAULT)
366       {
367         putc_COM1('F');
368         putxx(vsi_get_faults());
369       }
370       else
371       {
372         if (status==0)
373         puts_COM1("  Init ");
374         else if (status==1)
375         puts_COM1("  Gate Charge ");
376         else if (status==2)
377         puts_COM1("  Ramp ");
378         else if (status==3)
379         puts_COM1("  Run ");
380         else if (status==4)
381         puts_COM1("  Settled ");
382         else if (status==5)
383         puts_COM1("  Idle ");
384         else if (status==6)
385         puts_COM1("  FAULT ");
386         else putxx(status);
387     }
388     #ifdef OPEN_LOOP
389     puts_COM1("\t Phase:");
390     putdbl(phase_serial,1);
391     #endif
392     #ifdef CLOSED_LOOP
393     puts_COM1("\t Vref:");
394     putu(vref_serial);
395     if (FF_ENABLE)  puts_COM1("FF ENABLED");
396     else            puts_COM1("FF DISABLED");
397     #endif
398     if(DT_COMP)
399     {
400       puts_COM1(" DT Comp:");
```

```
401       putl(phase_aug_DT_fixed);
402     }
403     puts_COM1(" I3_fixed:");
404     putl(I3_fixed);
405     puts_COM1(" I4_fixed:");
406     putl(I4_fixed);
407   }
408 } /* end com_display */
409
410 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
411 /* void com_keyboard
412 Parameters: none
413 Returns: nothing
414 Description: Process characters from COM0.
415 Notes:
416 History:
417   22/06/05 AM - initial creation
418 \li 27/11/07 PM - added in testing of the digital I/O
419 */
420 void com_keyboard(void)
421 {
422
423   char c;
424
425 //  puts_COM1("KEY");
426   if (kbhit_COM1())
427   {
428     c = getc_COM1();
429     switch (c)
430     {
431 //      case 'q': quit = 1;
432 //      break;
433       case 'e': vsi_enable();
434         puts_COM1("e");
435       break;
436       case 'd':
437         vsi_disable();
438       break;
439
440       //Open Loop phase shift variation
441       #ifdef OPEN_LOOP
442       case 'z'://lead secondary bridge phase shift (small)
443         if((phase_serial+PHASE_STEP_SMALL) < 90.0){
444           phase_serial +=PHASE_STEP_SMALL;
445         }
446         else{
447           phase_serial = 90.0;
448         }
449         vsi_set_phase(phase_serial);
450       break;
451       case 'x'://lag secondary bridge phase shift (small)
452         if((phase_serial-PHASE_STEP_SMALL) > -90.0){
453           phase_serial -=PHASE_STEP_SMALL;
454         }
455         else{
456           phase_serial = -90.0;
457         }
458         vsi_set_phase(phase_serial);
459       break;
460       case 'Z'://increase phase shift (small)
461         if((phase_serial+PHASE_STEP_LARGE) < 90.0){
462           phase_serial +=PHASE_STEP_LARGE;
463         }
464         else{
465           phase_serial = 90.0;
466         }
467         vsi_set_phase(phase_serial);
468       break;
469       case 'X'://decrease phase shift (small)
470         if((phase_serial-PHASE_STEP_LARGE) > -90.0){
471           phase_serial -=PHASE_STEP_LARGE;
472         }
473         else{
474           phase_serial = -90.0;
475         }
476         vsi_set_phase(phase_serial);
477       break;
478       #endif
479
480       //Set desired Voltage Reference
```

```
481       case 'm': if (vref_serial < VREF_MAX-VREF_STEP_S) vref_serial+=VREF_STEP_S;  vsi_set_vref(vref_serial);break;
482       case 'M': if (vref_serial < VREF_MAX-VREF_STEP_L) vref_serial+=VREF_STEP_L;  vsi_set_vref(vref_serial);break;
483       case 'n': if (vref_serial > VREF_MIN+VREF_STEP_S) vref_serial-=VREF_STEP_S;  vsi_set_vref(vref_serial);break;
484       case 'N': if (vref_serial > VREF_MIN+VREF_STEP_L) vref_serial-=VREF_STEP_L;  vsi_set_vref(vref_serial);break;
485
486       //Enable/Disable Feed Forward
487       case 'f': FF_ENABLE=0;break;
488       case 'F': FF_ENABLE=1;
489                 break;
490
491       // AC/DC Feed Forward Selection
492       case 'a': AC_FF=0;break;
493       case 'A': AC_FF=1;
494                 break;
495
496       //enable/disable Deadtime compensation
497       case 'c': DT_COMP=0;break;
498       case 'C': DT_COMP=1;break;
499
500       case 'H': // write help info
501         initial=0;
502       break;
503
504 #ifdef GRAB_INCLUDE
505       case 'g': /* grab interrupt data */
506         GrabClear();
507         GrabStart();
508         GrabRun();
509       break;
510       case 'h':
511         puts_COM1("\n\nGrab Display\nIndex\n");
512         GrabShow();
513       break;
514       case 's': /* stop grab display */
515         GrabClear();
516         GrabStop();
517       break;
518 #endif
519     }
520   }
521 } /* end com_keyboard */
522
523
524 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
525 /**
526 1 second CPU timer interrupt.
527
528 \author A.McIver
529 \par History:
530 \li  22/06/05 AM - initial creation (derived from k:startup.c)
531 */
532 #ifndef BUILD_RAM
533 #pragma CODE_SECTION(isr_cpu_timer0, "ramfuncs");
534 #endif
535 interrupt void isr_cpu_timer0(void)
536 {
537   static struct
538   {
539     Uint16
540       msec,
541       msec10,
542       msec100,
543       sec;
544   } i_count =
545   {
546     0, 0, 0
547   };
548
549   /*for (ii=0; ii<WD_TIMER_MAX; ii++)
550   {
551     if (wd_timer[ii] > 0)
552       wd_timer[ii]--;
553   }*/
554   i_count.msec++;
555   if (i_count.msec >= 10)
556   {
557     i_count.msec = 0;
558     i_count.msec10++;
559     if (i_count.msec10 >= 10)
560     {
```

```
561       i_count.msec10 = 0;
562       i_count.msec100++;
563       if (i_count.msec100 >= 10)
564       {
565         i_count.msec100 = 0;
566         time.sec = 1;
567       }
568       time.sec0_1 = 1;
569     }
570     time.msec10 = 1;
571   }
572   time.msec = 1;
573
574   // Acknowledge this interrupt to receive more interrupts from group 1
575   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
576 } /* end isr_cpu_timer0 */
577
578
579 /* ======================================================================
580 __Exported_Functions()
581 ======================================================================= */
582
583
584 /* ======================================================================
585 __Grab_Functions()
586 ======================================================================= */
587 #ifdef GRAB_INCLUDE
588
589 void GrabInit(void)
590 {
591   Uint16
592     i,j;
593
594   for (i=0; i<GRAB_LENGTH; i++)
595   {
596     for (j=0; j<GRAB_WIDTH; j++)
597     {
598       grab_array[i][j] = 0;
599     }
600   }
601   GrabClear();
602 }
603
604 /* call with index == 0xFFFF for title line
605 else index = 0..GRAB_LENGTH-1 for data */
606 void GrabDisplay(int16 index)
607 {
608   Uint16
609     i;
610
611   if (index == 0xFFFF)
612   {
613     puts_COM1("\nindex");
614     for (i=0; i<GRAB_WIDTH; i++)
615     {
616       puts_COM1("\tg");
617       put_d(i);
618     }
619   }
620   else
621   {
622     put_d(index);
623     for (i=0; i<GRAB_WIDTH; i++)
624     {
625       putc_COM1('\t');
626       #ifdef GRAB_LONG
627       putl(grab_array[index][i]);
628       #endif
629       #ifdef GRAB_DOUBLE
630       putdbl(grab_array[index][i],3);
631       #endif
632     }
633   }
634   puts_COM1("\n");
635 }
636
637 #endif
638 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

```
1 /**
2 \file
3 \brief VSI definitions
4
5 \par Developed By:
6   Creative Power Technologies, (C) Copyright 2009
7 \author A.McIver
8 \par History:
9 \li  23/04/09 AM - initial creation
10 \ Modified Dinesh Segaran
11 \   11/11/09 DS - Turning this into a GIIB-Based Bidirectional DC-DC Converter
12 \    26/08/10 DS - Fixed Point implementation of the Adaptive Controlled
13 \                 Bidirectional DC-DC Converter
14
15 */
16
17 /* ========================================================================
18 ___Includes()
19 ======================================================================== */
20
21
22 /* ========================================================================
23 ___Definitions()
24 ======================================================================== */
25
26 //this is to try and separate EVB stuff
27 #define EVB 1
28 // Address for modifying CPLD
29 #define ADD_EVB 0xCA //<write 0x01 to this Address to direct EVB to output. write 0x00 to disable
30
31 //For Fixed Point
32 #define FIXED_Q    11
33 #define FIXED_Q_SCALE 2048.0
34
35 /** @name VSI Status bit definitions */
36 //@{
37 #define VSI_INIT        0x0000
38 #define VSI_GATECHARGE  0x0001  ///< VSI is running
39 #define VSI_RAMP        0x0002  ///< VSI is running
40 #define VSI_RUNNING      0x0003  ///< VSI is running
41 #define VSI_SETTLED      0x0004  ///< set when target reached
42 #define VSI_STOP        0x0005  ///< VSI is running
43 #define VSI_FAULT        0x0006  ///< set when fault present in VSI system
44 //@}
45
46 /** @name Fault Codes */
47 //@{
48 #define FAULT_VSI_IAC_OL    0x0001
49 #define FAULT_VSI_IAC_OC    0x0002
50 #define FAULT_VSI_VDC_OV    0x0004
51 #define FAULT_VSI_VDC_UV    0x0008
52 #define FAULT_VSI_PDPINT    0x0010
53 #define FAULT_VSI_SPI        0x0020
54 //@}
55
56 #define  SW_FREQ_BIDC     ((int32)20000)
57 #define  PERIOD_2_BIDC    ((int32)HSPCLK/SW_FREQ_BIDC/2/2)        // Carrier timer half period in clock ticks
58 #define  PERIOD_BIDC      ((int32)HSPCLK/SW_FREQ_BIDC/2)
59
60 #define  SW_FREQ_VSI     ((int32)5000)
61 #define  PERIOD_2_VSI    ((int32)HSPCLK/SW_FREQ_VSI/2/2)        // Carrier timer half period in clock ticks
62 #define  PERIOD_VSI      ((int32)HSPCLK/SW_FREQ_VSI/2)
63
64 /******************
65 _CONTROLLER_FORM()
66 ******************/
67 //Closed or Open loop selection
68 #define CLOSED_LOOP 1
69 //#define OPEN_LOOP 1
70
71 #ifdef OPEN_LOOP
72 #undef CLOSED_LOOP
73 #endif
74
75
76 //Controller form
77 //#define PROP_CONTROL 1
78 //#define I_CONTROL 1
79 #define PI_CONTROL 1
80 #define ADAPTIVE 1
```

```
 81 #define FEED_FORWARD 1
 82
 83 /****************
 84 _ADC_Scaling()
 85 ****************/
 86 /// ADC calibration time
 87 #define ADC_CAL_TIME      1// seconds
 88 #define ADC_COUNT_CAL      (Uint16)(ADC_CAL_TIME * 20000 * 2.0)
 89
 90 /// DC averaging time
 91 #define ADC_DC_TIME        0.1 // seconds
 92 #define ADC_COUNT_DC      (Uint16)(ADC_DC_TIME * 20000 * 2.0)
 93
 94 #define  ADC_REAL_SC        1
 95
 96 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
 97 /// RMS scaling
 98 #define ADC_RMS_PS      4
 99
100 //DA2810 Scaling - 3V and 12 bits
101 //easier to multiply result by 3 and shift back by 12.
102 #define ADC_DA_SCALE_MULT    (long)3     //3.0/4096.0 - scaled by FIXED_Q+5 cos num is so small
103 #define ADC_DA_SCALE_SHIFT  12
104
105 #define ADC_DA_SHIFT        4
106
107 //GIIB Scaling Resistors
108 #define RFB_GIIB_VAC         (long)10000                        //10000.0  //feedback resistor on GIIB board
109 #define RIN_GIIB_VAC         (long)(150000+150000+150000)           //150000.0+150000.0+150000.0  //preloaded input resistor on GIIB board
110 #define RFB_GIIB_VDC         (long)10000                        //10000.0  //feedback resistor on GIIB board
111 #define RIN_GIIB_VDC         (long)(150000+180000+180000)  //50000.0+180000.0+180000.0  //preloaded input resistor on GIIB board
112
113 //AC Voltage Inputs
114 //GIIB Scaling
115 #define RIN_GIIB_ADD_VAC     (long)560000 //additional scaling resistor on GIIB board
116 #define RIN_GIIB_TOTAL_VAC   ((double)((double)RIN_GIIB_ADD_VAC*(double)RIN_GIIB_VAC)/(double)((double)RIN_GIIB_ADD_VAC+(double)RIN_GIIB_VAC))
117 #define VAC_GIIB_GAIN       (long)((-1.0*(double)RFB_GIIB_VAC*FIXED_Q_SCALE)/(double)RIN_GIIB_TOTAL_VAC)  //scaled by FIXED_Q
118 #define VAC_GIIB_GAIN_INV   (long)(((double)FIXED_Q_SCALE*(double)FIXED_Q_SCALE)/(double)VAC_GIIB_GAIN)  //scaled by FIXED_Q
119
120 //DC Voltage Inputs
121 //GIIB Scaling
122 #define RIN_GIIB_ADD_VDC     (long)470000       //additional scaling resistor on GIIB board
123 #define RIN_GIIB_TOTAL_VDC   ((double)((double)RIN_GIIB_ADD_VDC*(double)RIN_GIIB_VDC)/(double)((double)RIN_GIIB_ADD_VDC+(double)RIN_GIIB_VDC))
124 #define VDC_GIIB_GAIN       ((-1.0*(double)RFB_GIIB_VDC)/(double)RIN_GIIB_TOTAL_VDC)  //scaled by FIXED_Q
125 #define VDC_GIIB_GAIN_INV   (long)(FIXED_Q_SCALE/VDC_GIIB_GAIN)    //scaled by 2^9
126
127 //Mini2810 Scaling Resistors
128 #define RUP_MINI1            (long)6800
129 #define RUP_MINI2            (long)4700
130 #define RUP_MINI_TOTAL      (long)((RUP_MINI1*RUP_MINI2)/(RUP_MINI1+RUP_MINI2))
131 #define RDWN_MINI            (long)6800
132 #define RIN_MINI            (long)12000
133 #define RDOWN_MINI_TOTAL    (long)((RDWN_MINI*RIN_MINI)/(RDWN_MINI+RIN_MINI))
134
135 //Mini2810 ADC Scaling
136 #define ADC_MINI_GAIN        (((double)(RUP_MINI_TOTAL*RDOWN_MINI_TOTAL))/((double)((RUP_MINI_TOTAL+RDOWN_MINI_TOTAL)*RIN_MINI)))  //is a double
137 #define ADC_MINI_GAIN_INV    (long)(FIXED_Q_SCALE/ADC_MINI_GAIN)  //scaled by FIXED_Q
138
139 #define MINI_LEVEL_SHIFT     (long)(((double)RDOWN_MINI_TOTAL*2.5*FIXED_Q_SCALE)/((double)(RUP_MINI_TOTAL+RDOWN_MINI_TOTAL)))    //scaled by FIXED_Q
140 #define ADC_OFFSET          (long)(((MINI_LEVEL_SHIFT<<ADC_DA_SCALE_SHIFT)>>FIXED_Q)/ADC_DA_SCALE_MULT) //in counts
141
142 //Voltage Overall Gain
143 #define VDC_ANALOG_GAIN (long)((double)((double)VDC_GIIB_GAIN_INV*(double)ADC_MINI_GAIN_INV*(double)ADC_DA_SCALE_MULT)/(double)FIXED_Q_SCALE/(double)4096))  //40
144 #define VAC_ANALOG_GAIN     ((long)((double)((double)VAC_GIIB_GAIN_INV*(double)ADC_MINI_GAIN_INV*(double)ADC_DA_SCALE_MULT)/(double)FIXED_Q_SCALE/(double)4096))
145
146 //Vgen for DAC input
147 #define DAC_SCALE            ((2048.0)/10.0) //2048 counts gives 10V
148 #define VGEN_ANALOG_GAIN    ((long)((double)(DAC_SCALE*4.0*(double)ADC_MINI_GAIN_INV*(double)ADC_DA_SCALE_MULT)/(double)FIXED_Q_SCALE/(double)4096))
149 //4096 is the dac scale shift by 12. x4 is to scale to va.
150 #define VGEN_CAL            ((long)1200)    //this is done so that the two bridges voltage supplies don't fight.
151
152 //Current Inputs
153 //LEM Scaling
154 #define CT_RATIO                  4000.0        //For LA 100P SP13, it is 1000, for LA 100P - 2000
155 #define CT_TURNS                  2.0
156 #define BURDEN_R                 270.0          //Burden resistor - Load current
157 #define LEM_GAIN                 ((CT_TURNS*BURDEN_R)/CT_RATIO)
158 #define LEM_GAIN_INV             (1.0/LEM_GAIN)    //is a double
159
160 //GIIB Scaling
```

```
161 #define RIN1_GIIB_I              10000.0        //Input resistor to GIIB op amp stage
162 #define RIN2_GIIB_I              10000.0        //Input resistor to GIIB op amp stage
163 #define RIN_GIIB_TOTAL_I        ((RIN1_GIIB_I*RIN2_GIIB_I)/(RIN1_GIIB_I+RIN2_GIIB_I))        //Input resistor to GIIB op amp stage
164 #define RFB_GIIB_I              10000.0
165 #define I_GIIB_GAIN              (-1.0*RFB_GIIB_I/RIN_GIIB_TOTAL_I)      //Voltage gain of amplifier on GIIB for current (double)
166 #define I_GIIB_GAIN_INV          (1.0/I_GIIB_GAIN)                        //Voltage gain of amplifier on GIIB for current (double)
167
168 #define I_ANALOG_GAIN            ((long)(LEM_GAIN_INV*I_GIIB_GAIN_INV*(double)ADC_MINI_GAIN_INV*(double)ADC_DA_SCALE_MULT)>>ADC_DA_SCALE_SHIFT)
169 //load current scaling
170
171 /*End ADC Scaling*/
172
173 /* Topology parameters */
174 #define C                 11.9e-6  //used to be 30.8uF, now is 31.4
175 #define INV_CNEG         -1.0/C
176 #define L                 132e-6
177 #define R_L               0.01
178 #define R_L_2             R_L*R_L
179 #define OMEGA_BIDC_L     (OMEGA_BIDC*L)
180 #define OMEGA_BIDC_L_2   ((OMEGA_BIDC*L)*(OMEGA_BIDC*L))
181 #define NPRI             (10.0)
182 #define NSEC             (11.0)
183 #define NPRI_NSEC         (double)(NPRI/NSEC)
184 #define NPRI_NSEC_FIXED   ((int32)(NPRI_NSEC*FIXED_Q_SCALE))
185 #define VIN               (200.0)
186 #define _4VIN             (4.0*VIN)
187 #define VIN_FIXED         (long)(VIN*FIXED_Q_SCALE)
188 #define INV_NP_NS_VIN     (double)(NPRI/(NSEC*VIN))
189 #define INV_NP_NS_VIN_FIXED  (long)((NPRI*32768)/(NSEC*VIN)) // is shifted by FIXED_Q+4
190 #define VDCPRI           VIN/2.0
191 #define VDCPRI_FIXED     (long)((long)VIN/2)
192
193
194 /*  constants */
195 #define PI               3.14159265358979
196 #define _2PI             2*PI
197 #define PI_2             1.57079632679489
198 #define INV_PI           0.31830988618379
199 #define INV2_PI           0.636619772367581
200 #define INV2_PI_FIXED     (long)(INV2_PI*FIXED_Q_SCALE)
201
202 /* sine table definitions */
203 #define DEG_TO_COUNT     ((double)(3750.0/180.0));
204 #define COUNT_TO_RAD     PI/3750.0
205 #define COUNT_TO_RATIO   1.0/(2*3750.0)
206 #define RAD_TO_COUNT     3750.0/PI
207 #define DEG_TO_RAD       PI/180.0
208 #define RAD_TO_DEG       180.0/PI
209 #define COUNT_TO_SINTABLE (long)((4294967296.0/((double)PERIOD_BIDC*2.0)))
210
211
212 /* Controller definitions */
213 //BiDC parameters
214 #define TS_BIDC            ((double)(1.0/SW_FREQ_BIDC))
215 #define OMEGA_BIDC        (2.0*PI*(double)SW_FREQ_BIDC)
216 #define FSAMPLE_BIDC      (1.0*SW_FREQ_BIDC)
217 #define TSAMPLE_BIDC      (1.0/FSAMPLE_BIDC)
218 #define MAX_PHASE          (PERIOD_2_BIDC-1)  //maximum phase shift. above this, the nonlinearity is too great
219 #define T_DELAY_BIDC      (1.0*TSAMPLE_BIDC)
220 #define OMEGA_C_BIDC      (PI_2-(50*DEG_TO_RAD))/(T_DELAY_BIDC)    //60 deg phase margin
221 #define OMEGA_C_10_BIDC    (OMEGA_C_BIDC/10.0)    //60 deg phase margin
222 #define OMEGA_C_BIDC_FIXED  ((int32)(OMEGA_C_BIDC*FIXED_Q_SCALE*4.0))
223 #define PERIOD_SCALE_BIDC   ((int32)(PERIOD_2_BIDC*INV2_PI))
224 #define DAC_SCALE_VREF     2048.0/50.0
225 #define DAC_SCALE_PHASE     2048.0/100.0
226 #define COUNT_TO_DAC      (COUNT_TO_RAD*RAD_TO_DEG*DAC_SCALE_PHASE)
227
228 //Adaptive controller parameters
229 #define DELF_DELU_SCALE    (NPRI/NSEC)*16*VDCPRI/C/(PI*PI)
230 #define DELF_DELU_SCALE_FIXED  (long)(DELF_DELU_SCALE)
231 #define VDC_KP_INIT       0.001
232 #define VDC_KP_MAX       0.01
233 #define VDC_KP_MIN       0.001
234 #define VDC_KP_MAX_FIXED  (int32)(VDC_KP_MAX*FIXED_Q_SCALE*4.0)
235 #define VDC_KP_MIN_FIXED  (int32)(VDC_KP_MIN*FIXED_Q_SCALE*4.0)
236 #define VDC_KP_INIT_FIXED  (int32)(VDC_KP_INIT*FIXED_Q_SCALE*4.0)
237 #define DELF_DELU_CONST    (VDCPRI*NPRI_NSEC/(C*PI*PI))    //divide by 16.0 is for scaling purposes
238 #define DELF_DELX_CONST    ((-8.0*NPRI_NSEC*NPRI_NSEC)/(C*PI*PI))
239 #define BIDC_FF_CONST      ((16.0*VDCPRI*NPRI_NSEC/(PI*PI))/OMEGA_BIDC_L)
240 #define VDC_KI           (double)(OMEGA_C_10_BIDC/FSAMPLE_BIDC)
```

```
241 #define VDC_KI_FIXED      (int32)(VDC_KI*FIXED_Q_SCALE)
242
243 //deadtime compensation parameters
244 #define DEADBAND_BIDC        1.5e-6
245 #define DB_DEG_BIDC          (360.0*SW_FREQ_BIDC*DEADBAND_BIDC)
246 #define DB_RAD_BIDC          (DB_DEG_BIDC*DEG_TO_RAD)
247 #define DB_RATIO_BIDC        (DB_RAD_BIDC/_2PI)
248 #define DEADBAND_COUNT_BIDC  ((int16)(DEADBAND_BIDC*HSPCLK))
249
250 // Step size and max output voltage
251 #define VREF_MAX 201
252 #define VREF_MIN 10
253 #define VREF_STEP_S 1
254 #define VREF_STEP_L  10
255
256 /* ========================================================================
257 __Macros()
258 ========================================================================== */
259
260 /// Disable VSI switching
261 #define VSI_DISABLE()    {\
262                           EvaRegs.ACTRA.all = 0x0000;\
263                           EvbRegs.ACTRB.all = 0x0000;\
264                         }
265
266 /// Enable VSI switching
267 #ifdef EVB
268   #define VSI_ENABLE()    {\
269                             EvaRegs.ACTRA.all = 0x0066;\
270                             EvbRegs.ACTRB.all = 0x0066;\
271                             cpld_write(ADD_EVACOMCON,0x0001);\
272                           }  //single phase only
273     //  output pin 1 CMPR1 - active high
274     //  output pin 2 CMPR1 - active low
275     //  output pin 3 CMPR2 - active low
276     //  output pin 4 CMPR2 - active high
277     //  output pin 5 CMPR3 - active high
278     //  output pin 6 CMPR3 - active low    =>0000 0110 0110 0110
279 #endif
280 #ifndef EVB
281   #define VSI_ENABLE()    {\
282                             EvaRegs.ACTRA.all = 0x0096;\
283                             cpld_write(ADD_EVACOMCON,0x0001);\
284                           }  //single phase only
285     //  output pin 1 CMPR1 - active high
286     //  output pin 2 CMPR1 - active low
287     //  output pin 3 CMPR2 - active low
288     //  output pin 4 CMPR2 - active high
289     //  output pin 5 CMPR3 - active high
290     //  output pin 6 CMPR3 - active low    =>0000 0110 1001 0110
291 #endif
292 /// Turn low side devices on full for charge pump starting
293 #define VSI_GATE_CHARGE()  EvaRegs.ACTRA.all = 0x00CC
294
295 #define SIN_TABLE_READ(PHASE,SIN_VAL){\
296       SIN_VAL = sin_table[(PHASE>>22)|0x01];\
297         VAL_DIFF = (sin_table[((PHASE>>22)+1)|0x01]) - SIN_VAL;\
298         SIN_VAL += (int16)( ((PHASE&0x3FFFFF)*(int32)VAL_DIFF)>>22);}
299 // phase is a 32bit number, but the index is only 10 (513 values).
300 // shift right by 22 to know where to aim in the sine table. interpolate using the last 6 bits.
301
302
303
304 #define SIN_TABLE_READ_DINESH(_SIN_COUNT_, _VAL_){  \
305         _SIN_INDEX_ = _SIN_COUNT_*COUNT_TO_SINTABLE;\
306         _INDEX_ = (Uint16)(_SIN_INDEX_>>FIXED_Q);\
307         val_lo = sin_table[(_INDEX_>>6)|0x001];  \
308         val_diff = sin_table[((_INDEX_>>6)|0x001)+2] - val_lo;  \
309         _VAL_ = (val_lo + (int16)(((int32)(_INDEX_&0x007F)*(int32)val_diff)>>7));  }
310
311
312 /* ========================================================================
313 __Exported_Variables()
314 ========================================================================== */
315
316 typedef long long signed int  int64;
317
318 /* ========================================================================
319 __Control Loop Variables()
320 ========================================================================== */
```

```
321
322 /* =========================================================================
323 __Function_Prototypes()
324 ============================================================================ */
325
326 /// Core interrupt initialisation
327 void vsi_init(void);
328
329 /// Core interrupt VSI state machine for background processing
330 void vsi_state_machine(void);
331
332 /// Enables vsi switching (assuming no faults)
333 void vsi_enable(void);
334
335 /// Disable vsi switching
336 void vsi_disable(void);
337
338 // Set the target output phase shift
339 void vsi_set_phase(double phase_cont_signal);
340
341 // Set the desired output voltage
342 void vsi_set_vref(int16 vref);
343
344 /// Returns the status of the VSI
345 Uint16 vsi_get_status(void);
346
347 /// Report what faults are present in the VSI
348 Uint16 vsi_get_faults(void);
349
350 /// Clear some detected faults and re-check.
351 void vsi_clear_faults(void);
352
353 //  Print the current state of the state machine
354 void get_state(void);
355
356 // Calibrate ADCs online
357 void calibrate_adc(void);
358
359
360 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

```
1 /**
2 \file
3 \brief VSI Interrupt Service Routine
4
5 This file contains the code for the core interrupt routine for the CVT system.
6 This interrupt is the central system for the signal generation and
7 measurement. The carrier timer for the VSI generation also triggers the
8 internal ADC conversion at the peak of the carrier. The end of conversion then
9 triggers this interrupt. Its tasks are:
10
11 - Read internal ADC results
12 - Perform internal analog averaging and RMS calculations
13 - Update VSI phase and switching times
14
15 \par Developed By:
16   Creative Power Technologies, (C) Copyright 2009
17 \author A.McIver
18 \par History:
19 \li  23/04/09 AM - initial creation
20 \ Modified Dinesh Segaran
21 \   11/11/09 DS - Turning this into a GIIB-Based Bidirectional DC-DC Converter
22 \    26/08/10 DS - Fixed Point implementation of the Adaptive Controlled
23 \                  Bidirectional DC-DC Converter
24 */
25 /*****************************
26 _CODE_TASKS()
27 *****************************/
28 //  this code is ported over to rebuild the open-giib bidirectional dc-dc converter
29 //
30 //  13/4/2011  -  moved code to a flash project
31 //             -  replaced low voltage capacitors. New operating voltage - 200V at 1:1 transfer ratio
32 //             -  fixed state machine to actually display correctly.
33 //  14/4/2011  -  attempt to modulate an open-loop bidirectional dc-dc converter at 200V
34 //             -  scaling resistors - VAC inputs are used to measure DC. initially scaled to measure +/-450V,
35 //                                 now want to measure +/- 250V. 560kohm resistor needed
36 //                               - VDC inputs are initially scaled to measure +510V
37 //                                 now scaled to measure 250V, 470kohm used (245V)
38 //                               - current inputs used to measure the DC output current. +/-15A with 2 turns. 270 ohms used
39 //             -  Open loop modulation succesful
40 //             -  Testing ADCs - complete
41 //             - Test CL control - Adaptive Controller - no FF
42 //  21/4/2011  - Closed Loop H-bridge and a closed loop bidirectional DC-DC converter work.
43 //             - Need to implement feed-forward compensation. For this, need to synch switching and send mod depth info across.
44 //             - Stage 1:  -  Synchronise Carriers. use zaki's code.
45 //                         -  Synchronise the VSI to the BiDC because the BiDC uses a lot of DIGIO pins already.
46 //                         - Use the shielded ribbon cable for this. Build Loopback function and test.
47 //                         - Loopback cable - GPIOB0-4 (PWMB1-4) are routed back into DIGIN5-8. so Pins 1-4 are connected to 13-16.
48 //                         -  On the BiDC, send out a synch pulse at 5kHz (1 every 8 interrupts) on GPIOB4.
49 //                           This is DIGOUT5, pin 5 on the 20-pin header.
50 //                         - On the VSI, bring the synch pulse into CAP2. this is on DIGIN8, which is pin 16. ie connect pins 5 & 16.
51 //                         -  Also connect all the GNDs on the 20-pin header together. I.e, leave pins 18 & 20.
52 //                         - Disconnect VCC, i.e cut pins 17 & 19
53 //                         -  that lets you lift synch code from GridCon set, and  also the fault trigger when synch is lost.
54 //
55 //             - Stage 2:  -  Phase & Modulation depth information. Via SPI or via DAC?
56 // compiler standard include files
57 #include <math.h>
58
59 // processor standard include files
60 #include <DSP281x_Device.h>
61
62 #ifdef COM0_CONSOLE
63 #include <bios0.h>
64 #endif
65 #ifdef COM1_CONSOLE
66 #include <bios1.h>
67 #endif
68
69 // board standard include files
70 #include <lib_mini2810.h>
71 #include <dac_ad56.h>
72 #include <lib_cpld.h>
73 #include <lib_giib.h>
74
75 // local include files
76 #include "main.h"
77 #include "conio.h"
78 #include "vsi.h"
79
80 /* ======================================================================
```

```
81 __Definitions()
82 ========================================================================== */
83
84 /// Boot ROM sine table size for VSI and DFT
85 #define ROM_TABLE_SIZE    512
86 /// Boot ROM sine table peak magnitude for VSI and DFT
87 #define ROM_TABLE_PEAK    16384
88
89 #define GRAB_INCLUDE
90
91 /* =======================================================================
92 __Types()
93 ========================================================================== */
94
95 /// Internal ADC channel type
96 /** This structure hold variables relating to a single ADC channel. These
97 variables are used for filtering, averaging, and scaling of this analog
98 quantity. */
99 typedef struct
100 {
101   int16
102     raw, ///< raw ADC result from last sampling
103     filt; ///< decaying average fast filter of raw data
104   int32
105     rms_sum, ///< interrupt level sum of data
106     rms_sum_bak, ///< background copy of sum for averaging
107     dc_sum, ///< interrupt level sum
108     dc_sum_bak; ///< background copy of sum for processing
109   double
110     real; ///< background averaged and scaled measurement
111 } type_adc_ch;
112
113 /// Internal ADC storage type
114 /** This structure holds all the analog channels and some related variables
115 for the averaging and other processing of the analog inputs. There are also
116 virtual channels for quantities directly calculated from the analog inputs.
117 The vout and iout channels are for DC measurements of the VSI outputs when it
118 is producing a DC output. */
119 typedef struct
120 {
121   Uint16
122     count_cal, ///< counter for low speed calibration summation
123     count_rms, ///< counter for full fund. period for RMS calculations
124     count_rms_bak, ///< background copy of RMS counter
125     count_dc, ///< counter for DC averaging
126     count_dc_bak, ///< background copy of DC counter
127     flag_cal, ///< flag set to trigger background calibration averaging
128     flag_rms, ///< flag set to trigger background RMS averaging
129     flag_dc; ///< flag set to trigger background DC averaging
130   type_adc_ch
131     A0, ///< ADC channel A0
132     A1, ///< ADC channel A1
133     A2, ///< ADC channel A2
134     A3, ///< ADC channel A3
135     A4, ///< ADC channel A4
136     A5, ///< ADC channel A5
137     A6,
138     B0, ///< ADC channel B0
139     B1, ///< ADC channel B1
140     B2, ///< ADC channel B2
141     B3, ///< ADC channel B3
142     B4, ///< ADC channel B4
143     B5, ///< ADC channel B5
144     yHA, ///< bank A high reference
145     yLA, ///< bank A low reference
146     yHB, ///< bank B high reference
147     yLB; ///< bank B low reference
148 } type_adc_int;
149
150 /** @name Internal ADC Variables */
151 //@{
152 type_adc_int
153   adc_int =
154   {
155     0, // count_cal
156     0, // count_rms
157     0, // count_rms_bak
158     0, // count_dc
159     0, // count_dc_bak
160     0, // flag_cal
```

```
161     0, // flag_rms
162     0, // flag_dc
163     {  0, // raw
164        0, // filt
165        0L, // rms_sum
166        0L, // rms_sum_bak
167        0L, // dc_sum
168        0L, // dc_sum_bak
169        0.0 // real
170       }, // #A0
171     { 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // #B0
172     { 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // yHA
173     { 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // yLA
174     { 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // yHB
175     { 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // yLB
176   };
177
178 // ADC calibration variables
179 int16
180   cal_gainA = 1<<14,    // calibration gain factor for A channel
181   cal_gainB = 1<<14,    // calibration gain factor for B channel
182   cal_offsetA = 0,    // calibration offset for A channel
183   cal_offsetB = 0;    // calibration offset for B channel
184 double
185   cal_gain_A, cal_gain_B,
186   cal_offset_A, cal_offset_B;
187
188
189 double
190   yHA = 0.0,
191   yLA,
192   yHB,
193   yLB;
194
195 /* =====================================================================
196 __Variables()
197 ======================================================================== */
198 // state machine level variables
199 Uint16
200   vsi_status = 0,      /// Status of VSI system
201   is_switching = 0,    // flag set if PWM switching is active
202   vsi_counter = 0,    // counter for timing VSI regulation events
203   spi_fail_count;
204 // PWM Timer interrupt variables
205
206 // Boot ROM sine table starts at 0x003FF000 and has 641 entries of 32 bit sine
207 // values making up one and a quarter periods (plus one entry). For 16 bit
208 // values, use just the high word of the 32 bit entry. Peak value is 0x40000000 (2^30)
209 // therefore 1 period is 512 entries, 120 degrees offset is 170.67 entries.
210 // sin table actually starts with an offset of 2, odd numbers only
211 // so first value is in sin_table[3]
212 // max value of 16bit sign table is 2^14 =16384
213
214 int16
215   *sin_table = (int16 *)0x003FF000, // pointer to sine table in boot ROM
216   *cos_table = (int16 *)0x003FF100, // pointer to cos table in boot ROM
217   mod_targ = 0,      // target modulation depth
218   mod_ref = 0;
219
220 /// fault variables
221 Uint16
222   detected_faults = 0;  // bits set for faults detected (possibly cleared)
223
224 /***********************
225 _Modulation_variables()
226 ***********************/
227 int16 phase_scaled_fixed=0,
228       int_count=0,
229       phase_shift=0;
230
231 /****************
232 _ADC_VARIABLES()
233 ****************/
234 //ADC Variables
235 int32  VdcIN_fixed,
236       VdcOUT_fixed,
237       Iload_fixed,
238       IVSI_fixed;
239
240 int32  Vdc1_fixed,
```

```
241      Vdc2_fixed,
242      Vac1_fixed,
243      Vac2_fixed,
244      Vac3_fixed,
245      Vgen_fixed,
246      I1_fixed,
247      I2_fixed,
248      I3_fixed,
249      I4_fixed;
250
251 int32  Vdc1_cal = 0,
252      Vdc2_cal = 0,
253      Vac1_cal = 0,
254      Vac2_cal = 0,
255      Vac3_cal = 0,
256      I1_cal    = 0,
257      I2_cal   = 0,
258      I3_cal    = 0,
259      I4_cal   = 0;
260
261 /* ======================================================================
262 __Control_Loop_Variables()
263 ========================================================================= */
264 //Interface variables used to recieve controller loop parameters from background
265 //Controller loop turning parameters in real floating pointer number from background
266 int16   ref_volt=10;
267
268 //Uint16   PI_enable=1;
269
270 /*****************
271 _Macro_Variables()
272 *****************/
273 //sin table read variables
274 Uint32   PHASE;
275 int16    SIN_VAL,
276        VAL_DIFF;       // interpolation temp variable
277
278 /**************************
279 _BiDC_PI_Control_Variables()
280 **************************/
281 //fixed point version
282 int32  VDCref_fixed=(10<<FIXED_Q),
283      prev_VDCref_fixed,
284      VDCerror_fixed,
285      VDC_Kp_fixed,
286      VDC_prop_fixed,
287      VDC_intnow_fixed,
288      VDC_int_fixed=0,
289      VDC_cont_signal_fixed;
290
291 int16  saturated;
292 /*******************
293 _Adaptive_Variables()
294 *******************/
295 double   Z_harm[7],
296        phi_z[7];
297
298 int16   phase_shift_avrg,
299        phase_shift_record[5],
300        counter_avrg,
301        n_harm;
302
303 //in fixed point
304 int16    phi_z_fixed[7],
305        harm[7]={1,3,5,7,9,11,13},
306        sin_val_adapt;
307
308 int32  delta0_aug_fixed=0,
309      inv_Z_harm_fixed[7],
310      delf_delu_temp_fixed,
311      delf_delu_fixed,
312      delf_delu_fixed_scaled,
313      Kp_adapt_fixed;
314
315 Uint32  sin_count;
316
317 //end adaptive controller variables
318 /*******************
319 _DT_Comp_Variables()
320 *******************/
```

```
321 // New version. Unified DT compensation
322
323 int32    VdcOUT_fixed_avrg=0,
324         VdcOUT_fixed_record[5];
325
326 //fixed point
327 int32    phase_rad_ratio_fixed,
328         VDCout_txscaled_fixed,
329         Vp_Vs_4Vp_fixed,
330         Vs_Vp_4Vp_fixed,
331         Vs_Vp_4Vs_fixed,
332         Vs_Vp_DB_fixed,
333         Vp_Vs_DB_fixed;
334
335 int16    DT_COMP=0,
336         Tslew_count,
337         phase_aug_DT_fixed;
338
339 /*******************
340 _BIDC_FF_Variables()
341 *******************/
342 double Iload_FF_double;
343
344 int32  Iload_abs;
345
346 int32  BIDC_FF,
347       Iload_FF_fixed[PERIOD_2_BIDC];
348
349 int16  FF_ENABLE=0,
350       AC_FF=0,
351       hi,
352       lo,
353       mid,
354       va_VSI,
355       harm_3[7]={1,27,125,343,729,1331,2197},
356       init_table;  //initialises ff table
357
358 /* ======================================================================
359 __Local_Function_Prototypes()
360 ========================================================================= */
361
362 /* vsi state machine state functions */
363 void
364   st_vsi_init(void),    // initialises CFPP regulator
365   st_vsi_stop(void),    // waiting for start trigger
366   st_vsi_gate_charge(void), // delay to charge the high side gate drivers
367   st_vsi_ramp(void),    // ramping to target mod depth
368   st_vsi_run(void),    // maintaining target mod depth
369   st_vsi_fault(void);    // delay after faults are cleared
370
371 // ADC and VSI interrupt
372 interrupt void isr_adc(void);
373
374 // Gate fault (PDPINT) interrupt
375 interrupt void isr_gate_fault(void);
376
377 /* ====================================================================== */
378 /* State Machine Variable */
379 /* ====================================================================== */
380
381 type_state
382   vsi_state =
383   {
384     &st_vsi_init,
385     1
386   };
387
388
389 /* ======================================================================
390 __Exported_ADC_Functions()
391 ========================================================================= */
392
393 /**
394
395 This function initialises the ADC and VSI interrupt module. It sets the
396 internal ADC to sample the DA-2810 analog inputs and timer1 to generate a PWM
397 carrier and the event manager A to generate the VSI switching. It also
398 initialises all the relevant variables and sets up the interrupt service
399 routines.
400
```

```
401 This functions initialises the ADC unit to:
402 - Trigger a conversion sequence from timer 1 overflow
403 - Convert the appropriate ADC channels
404
405 Result registers as follows:
406 - ADCRESULT0 = ADCINA0
407 - ADCRESULT1 = ADCINB0
408 - ADCRESULT2 = ADCINA1
409 - ADCRESULT3 = ADCINB1
410 - ADCRESULT4 = ADCINA2
411 - ADCRESULT5 = ADCINB2
412 - ADCRESULT6 = ADCINA3
413 - ADCRESULT7 = ADCINB3
414 - ADCRESULT8 = ADCINA4
415 - ADCRESULT9 = ADCINB4
416 - ADCRESULT10 = ADCINA5
417 - ADCRESULT11 = ADCINB6
418 - ADCRESULT12 = ADCINA6 yHA
419 - ADCRESULT13 = ADCINB6  yHB
420 - ADCRESULT14 = ADCINA7 yLA
421 - ADCRESULT15 = ADCINB7 yLB
422
423 It initialises the Event Manager A unit to:
424 - drive PWM1-4 as PWM pins not GPIO
425 - a 0.48ns deadtime between the high and low side pins
426 - Timer 1 as an up/down counter for the PWM carrier
427
428 It initialises the PIE unit to:
429 - Take PDPINTA as a power stage interrupt
430 - Use the internal ADC completion interrupt to trigger the main ISR
431
432 \author A.McIver
433 \par History:
434 \li  12/10/07 AM - initial creation
435 \    26/08/10 DS - Fixed Point Bidirectional DC-DC Converter
436 */
437 void vsi_init(void)
438 {
439   //EVA
440   EvaRegs.ACTRA.all = 0x0000;
441   EvaRegs.GPTCONA.all = 0x0000;
442   EvaRegs.EVAIMRA.all = 0x0000;
443   EvaRegs.EVAIFRA.all = BIT0;
444   EvaRegs.COMCONA.all = 0x0000;
445
446   //EVB
447   #ifdef EVB
448   EvbRegs.ACTRB.all = 0x0000;
449   EvbRegs.GPTCONB.all = 0x0000;
450   EvbRegs.EVBIMRA.all = 0x0000;
451   EvbRegs.EVBIFRA.all = BIT0;
452   EvbRegs.COMCONB.all = 0x0000;
453   #endif
454 // Set up ISRs
455   EALLOW;
456   PieVectTable.ADCINT = &isr_adc;
457   PieVectTable.PDPINTA = &isr_gate_fault;
458   EDIS;
459
460   // Set up compare outputs
461   EALLOW;
462   GpioMuxRegs.GPDMUX.all = BIT0;
463   //EVA
464   GpioMuxRegs.GPAMUX.bit.PWM1_GPIOA0 = 1;  // enable PWM1 pin
465   GpioMuxRegs.GPAMUX.bit.PWM2_GPIOA1 = 1;  // enable PWM2 pin
466   GpioMuxRegs.GPAMUX.bit.PWM3_GPIOA2 = 1;  // enable PWM3 pin
467   GpioMuxRegs.GPAMUX.bit.PWM4_GPIOA3 = 1;  // enable PWM4 pin
468   GpioMuxRegs.GPAMUX.bit.PWM5_GPIOA4 = 0;  // enable GPIOA4
469   GpioMuxRegs.GPAMUX.bit.PWM6_GPIOA5 = 0;  // enable GPIOA5
470
471 //  //set up GPIOA12 to take the synch pulse from the Load GIIB
472 //  GpioMuxRegs.GPAMUX.bit.TCLKINA_GPIOA12 = 0;  //GPIOA12 is an IO
473 //  GpioMuxRegs.GPADIR.bit.GPIOA12 = 0;          //GPIOA12 is an Input
474
475   //EVB
476   #ifdef EVB
477   GpioMuxRegs.GPBMUX.bit.PWM7_GPIOB0 = 1;  // enable PWM7 pin
478   GpioMuxRegs.GPBMUX.bit.PWM8_GPIOB1 = 1;  // enable PWM8 pin
479   GpioMuxRegs.GPBMUX.bit.PWM9_GPIOB2 = 1;  // enable PWM9 pin
480   GpioMuxRegs.GPBMUX.bit.PWM10_GPIOB3 = 1;  // enable PWM10 pin
```

```
481
482   //for carrier synchronisim
483   GpioMuxRegs.GPBMUX.bit.PWM11_GPIOB4 = 0;  // enable GPIOB4  -  carrier synch
484   GpioMuxRegs.GPBDIR.bit.GPIOB4 = 1;        // GPIOB4  is an output
485
486   GpioMuxRegs.GPBMUX.bit.PWM12_GPIOB5 = 0;  // enable GPIOB5
487   #endif
488
489   GpioMuxRegs.GPDQUAL.bit.QUALPRD = 6; // 500ns qualification period
490
491   //set up GPIOB5 to send a synch pulse to the output VSI
492
493   EDIS;
494   //DEADBAND CONTROL
495   //EVA
496   EvaRegs.DBTCONA.bit.DBT = 8;      //1.5us deadtime
497   EvaRegs.DBTCONA.bit.EDBT1 = 1;
498   EvaRegs.DBTCONA.bit.EDBT2 = 1;
499   EvaRegs.DBTCONA.bit.EDBT3 = 1;
500   EvaRegs.DBTCONA.bit.DBTPS = 6;
501
502   #ifdef EVB
503   //EVB
504   EvbRegs.DBTCONB.bit.DBT = 8;      //1.5us deadtime
505   EvbRegs.DBTCONB.bit.EDBT1 = 1;
506   EvbRegs.DBTCONB.bit.EDBT2 = 1;
507   EvbRegs.DBTCONB.bit.EDBT3 = 1;
508   EvbRegs.DBTCONB.bit.DBTPS = 6;
509   #endif
510
511   //COMPARE REGISTERS
512   //EVA
513   EvaRegs.CMPR1 = PERIOD_2_BIDC;
514   EvaRegs.CMPR2 = PERIOD_2_BIDC;
515
516   #ifdef EVB
517   spi_set_mode(MODE_CPLD);
518   cpld_write(ADD_EVB,0x01);  //direct EVB to output
519   spi_set_mode(MODE_DAC);
520   //EVB
521   EvbRegs.CMPR4 = PERIOD_2_BIDC;
522   EvbRegs.CMPR5 = PERIOD_2_BIDC;
523   #endif
524
525   #ifndef EVB
526   spi_set_mode(MODE_CPLD);
527   cpld_write(ADD_EVB,0x00);
528   spi_set_mode(MODE_DAC);
529   #endif
530
531   // Setup and load COMCON
532   //EVA
533   EvaRegs.COMCONA.bit.ACTRLD = 1;    // reload ACTR on underflow or period match
534   EvaRegs.COMCONA.bit.SVENABLE = 0;  // disable space vector PWM
535   EvaRegs.COMCONA.bit.CLD = 1;    // reload on underflow & period match
536   EvaRegs.COMCONA.bit.FCOMPOE = 1;  // full compare enable
537   EvaRegs.COMCONA.bit.CENABLE = 1;  // enable compare operation
538
539   #ifdef EVB
540   //EVB
541   EvbRegs.COMCONB.bit.ACTRLD = 1;    // reload ACTR on underflow or period match
542   EvbRegs.COMCONB.bit.SVENABLE = 0;  // disable space vector PWM
543   EvbRegs.COMCONB.bit.CLD = 1;    // reload on underflow & period match
544   EvbRegs.COMCONB.bit.FCOMPOE = 1;  // full compare enable
545   EvbRegs.COMCONB.bit.CENABLE = 1;  // enable compare operation
546   #endif
547
548 // Set up Timer 1
549   EvaRegs.T1CON.all = 0x0000;
550   EvaRegs.T1PR = PERIOD_BIDC;
551   EvaRegs.T1CMPR = PERIOD_BIDC-1;      //modified for asynchronous sampling;
552   EvaRegs.T1CNT = 0x0000;
553
554 //Set up Timer 3
555   //EVB
556   #ifdef EVB
557   EvbRegs.T3CON.all = 0x0000;
558   EvbRegs.T3PR = PERIOD_BIDC;
559   EvbRegs.T3CMPR = 0;      //modified-unnecessary - DS
560   EvbRegs.T3CNT = 0x0000;
```

```
561   #endif
562   // Setup and load GPTCONA
563   EvaRegs.GPTCONA.bit.T1TOADC = 3;    //0: no event starts ADC 3: Compare match starts ADC  2: period int flag starts ADC
564   EvaRegs.GPTCONA.bit.TCMPOE = 1;
565 // Set up ADC
566
567   //these are being done in A/B pairs
568
569   AdcRegs.ADCMAXCONV.all = 0x0007;    // Setup 8 conv's on SEQ1                                              //To Oversample?
570   AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0x0;  // (A0/B0) - ADCRESULT0    - ADCINA0 - APOT1/I3   - SW_A - default I3    - DC Load Current
571                                           //                   1      ADCINB0 - VDC2                                 - Output DC Voltage
572   AdcRegs.ADCCHSELSEQ1.bit.CONV01 = 0x1;  // (A1/B1) - ADCRESULT2   - ADCINA1 - Vdc3/Vac3 - SW_A - default Vac3  -  Output DC Voltage
573                                           //                   3      ADCINB1 - I5                                   -
574   AdcRegs.ADCCHSELSEQ1.bit.CONV02 = 0x2;  // (A2/B2) - ADCRESULT4   - ADCINA2 - I1                                 - Output Current
575                                           //                   5      ADCINB2 - I4                                   -  DC Load Current
576   AdcRegs.ADCCHSELSEQ1.bit.CONV03 = 0x3;  // (A3/B3) - ADCRESULT6   - ADCINA3 - Vac1                               -  Output DC Voltage
577                                           //                   7      ADCINB3 - VDC1                                 - Input DC Voltage
578   AdcRegs.ADCCHSELSEQ2.bit.CONV04 = 0x4;  // (A4/B4) - ADCRESULT8   - ADCINA4 - I2                                 - Output Current
579                                           //                   9      ADCINB4 - APOT2/I6   - SW_B - default I6    -
580   AdcRegs.ADCCHSELSEQ2.bit.CONV05 = 0x5;  // (A5/B5) - ADCRESULT10  - ADCINA5 - Vac2                               - Output DC Voltage
581                                           //                   11    ADCINB5 - Vgen/Vdc4 - SW_B - default Vdc4  - DAC input
582   AdcRegs.ADCCHSELSEQ2.bit.CONV06 = 0x6;  // (A6/B6) - ADCRESULT12  -  ADCINA6 - 2.5V ref
583                                           //                   13    ADCINB6 - 2.5V ref
584   AdcRegs.ADCCHSELSEQ2.bit.CONV07 = 0x7;  // (A7/B7) - ADCRESULT14  - ADCINA7 - 1.25V ref
585                                           //                   15    ADCINB7 - 1.25V ref
586
587   AdcRegs.ADCTRL1.bit.ACQ_PS = 1;      // lengthen acq window size
588   AdcRegs.ADCTRL1.bit.SEQ_CASC = 1;    // cascaded sequencer mode
589   AdcRegs.ADCTRL2.bit.EVA_SOC_SEQ1 = 1;  // EVA manager start
590   AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = 1;  // enable interrupt
591   AdcRegs.ADCTRL2.bit.INT_MOD_SEQ1 = 0;  // int at end of every SEQ1
592   AdcRegs.ADCTRL3.bit.SMODE_SEL = 1;    // simultaneous sampling mode
593   AdcRegs.ADCTRL3.bit.ADCCLKPS = 0x04;  // ADCLK = HSPCLK/8 (9.375MHz)
594   SET_ADCB_NO();    //activates SW_B. ADCB4 = APOT2, ADCB5 = Vgen
595
596 // Enable interrupts
597   DINT;
598   EvaRegs.EVAIMRA.all = 0;        // disable all interrupts
599   // Enable PDPINTA: clear PDPINT flag, T1UFINT and T1PINT flag
600   EvaRegs.EVAIFRA.all = BIT0|BIT7;
601   EvaRegs.EVAIMRA.bit.PDPINTA = 1;
602
603   // Enable PDPINTA in PIE: Group 1 interrupt 1
604   PieCtrlRegs.PIEIER1.bit.INTx1 = 1;
605   // Enable ADC interrupt in PIE: Group 1 interrupt 6
606   PieCtrlRegs.PIEIER1.bit.INTx6 = 1;
607
608   IER |= M_INT1; // Enable CPU Interrupts 1
609   EINT;
610
611   AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1; // clear interrupt flag from ADC
612   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;   // Acknowledge interrupt to PIE Group 1 : PDPINT, ADC
613
614 /* Setup and load T1CON & T3CON to start operation */
615   EvaRegs.T1CON.bit.TMODE = 1;    // continous up/down count mode
616   EvaRegs.T1CON.bit.TPS = 0;      // input clock prescaler
617   EvaRegs.T1CON.bit.TCLD10 = 1;    // S.G. reload compare register on 0 or equals compare
618   EvaRegs.T1CON.bit.TECMPR = 1;    // enable time compare
619
620   #ifdef EVB
621   EvbRegs.T3CON.bit.TMODE = 1;    // continous up/down count mode
622   EvbRegs.T3CON.bit.TPS = 0;      // input clock prescaler
623   EvbRegs.T3CON.bit.TCLD10 = 1;    // S.G. reload compare register on 0 or equals compare
624   EvbRegs.T3CON.bit.TECMPR = 0;    // disable time compare
625   #endif
626
627 /*********************************
628 __initialise_adaptive_controller()
629 *********************************/
630   #ifdef ADAPTIVE
631     Z_harm[0]=  sqrt(R_L_2 + OMEGA_BIDC_L_2);
632     Z_harm[1]=   sqrt(R_L_2 + 3.0*3.0*OMEGA_BIDC_L_2);
633     Z_harm[2]=  sqrt(R_L_2 + 5.0*5.0*OMEGA_BIDC_L_2);
634     Z_harm[3]=  sqrt(R_L_2 + 7.0*7.0*OMEGA_BIDC_L_2);
635     Z_harm[4]=  sqrt(R_L_2 + 9.0*9.0*OMEGA_BIDC_L_2);
636     Z_harm[5]=  sqrt(R_L_2 + 11.0*11.0*OMEGA_BIDC_L_2);
637     Z_harm[6]=  sqrt(R_L_2 + 13.0*13.0*OMEGA_BIDC_L_2);
638
639     phi_z[0] =  atan2(OMEGA_BIDC_L,R_L);
640     phi_z[1] =  atan2(OMEGA_BIDC_L*3.0,R_L);
```

```
641     phi_z[2] =  atan2(OMEGA_BIDC_L*5.0,R_L);
642     phi_z[3] =  atan2(OMEGA_BIDC_L*7.0,R_L);
643     phi_z[4] =  atan2(OMEGA_BIDC_L*9.0,R_L);
644     phi_z[5] =  atan2(OMEGA_BIDC_L*11.0,R_L);
645     phi_z[6] =  atan2(OMEGA_BIDC_L*13.0,R_L);
646
647     phi_z_fixed[0] =  (int16)(phi_z[0]*RAD_TO_COUNT);
648     phi_z_fixed[1] =  (int16)(phi_z[1]*RAD_TO_COUNT);
649     phi_z_fixed[2] =  (int16)(phi_z[2]*RAD_TO_COUNT);
650     phi_z_fixed[3] =  (int16)(phi_z[3]*RAD_TO_COUNT);
651     phi_z_fixed[4] =  (int16)(phi_z[4]*RAD_TO_COUNT);
652     phi_z_fixed[5] =  (int16)(phi_z[5]*RAD_TO_COUNT);
653     phi_z_fixed[6] =  (int16)(phi_z[6]*RAD_TO_COUNT);
654
655     inv_Z_harm_fixed[0]=  (int32)(32768.0/(1.0*Z_harm[0]));
656     inv_Z_harm_fixed[1]=   (int32)(32768.0/(3.0*Z_harm[1]));
657     inv_Z_harm_fixed[2]=  (int32)(32768.0/(5.0*Z_harm[2]));
658     inv_Z_harm_fixed[3]=  (int32)(32768.0/(7.0*Z_harm[3]));
659     inv_Z_harm_fixed[4]=  (int32)(32768.0/(9.0*Z_harm[4]));
660     inv_Z_harm_fixed[5]=  (int32)(32768.0/(11.0*Z_harm[5]));
661     inv_Z_harm_fixed[6]=  (int32)(32768.0/(13.0*Z_harm[6]));
662     //scaled by 32768 = 2^15
663
664     //Feed forward initialisations
665     //Generate a lookup table of the steady state load current based on operating phase shift.
666     //I_load_FF = 16/pi^2 *Vp * Np/Ns * sum(1/(2n+1)^3 * sin((2n+1)delta)/(omega*L)
667     //done in floating point, converted to fixed point at the last step
668
669     for (init_table=0;init_table<=PERIOD_2_BIDC;init_table++)
670     {
671       Iload_FF_double=0.0;
672       for (n_harm=0;n_harm<6;n_harm++)
673       {
674         Iload_FF_double  +=   (1.0/harm_3[n_harm])*sin(harm[n_harm]*(init_table*COUNT_TO_RAD));
675       }
676       Iload_FF_fixed[init_table] = (int32)(BIDC_FF_CONST*Iload_FF_double*FIXED_Q_SCALE);
677     }
678   #endif
679
680   DINT;
681   EvaRegs.T1CON.bit.TENABLE = 1;    // enable timer1
682   EvbRegs.T3CON.bit.TENABLE = 1;    // enable timer3
683   #ifndef EVB
684   EvbRegs.T3CON.bit.TENABLE = 0;
685   #endif
686   EINT;
687   // Initialise state machine
688   vsi_state.first = 1;
689   vsi_state.f = &st_vsi_init;
690 } /* end vsi_init */
691
692
693 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
694 /**
695 This function is called from the main background loop once every millisecond.
696 It performs all low speed tasks associated with running the core interrupt
697 process, including:
698 - checking for faults
699 - calling the VSI state functions
700 - calling internal analog scaling functions
701
702 \author A.McIver
703 \par History:
704 \li  13/10/07 AM - derived from 25kVA:vsi:vsi.c
705 */
706 void vsi_state_machine(void)
707 {
708   SS_DO(vsi_state);
709   if (adc_int.flag_cal != 0)
710   {
711     adc_int.flag_cal = 0;
712     calibrate_adc();
713   }
714 } /* end vsi_state_machine */
715
716
717 /* ======================================================================
718 __Exported_VSI_Functions()
719 ========================================================================= */
720
```

```
721 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
722 /**
723 This function switches the VSI from the stopped state to a running state.
724
725 \author A.McIver
726 \par History:
727 \li  13/10/07 AM - derived from 25kVA:vsi:vsi.c
728 */
729 void vsi_enable(void)
730 {
731   if (detected_faults == 0)
732     {
733       is_switching = 1;
734     }
735 } /* end vsi_enable */
736
737
738 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
739 /**
740 This function switches the VSI from the running state to a stop state.
741
742 The ramp down process has the side effect of resetting the reference to zero.
743
744 \author A.McIver
745 \par History:
746 \li  13/10/07 AM - derived from 25kVA:vsi:vsi.c
747 */
748 void vsi_disable(void)
749 {
750   is_switching = 0;
751 } /* end vsi_disable */
752
753
754 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
755 /**
756 This function sets the target output phase shift.
757
758 The target is passed in ????.
759
760 \author A.McIver
761 \par History:
762 \li 24/04/09 AM - initial creation
763 \   24/04/09 DS - Changed from varying modulation depth to phase shift
764 \param[in] m Target output modulation depth
765 */
766 void vsi_set_phase(double phase_cont_signal)
767 {
768     phase_scaled_fixed = phase_cont_signal*DEG_TO_COUNT;  //scaled to +/- pi/2 (radians)
769     if (phase_scaled_fixed>MAX_PHASE)
770       {
771         phase_scaled_fixed=MAX_PHASE-1;
772         phase_cont_signal = 90.0;
773       }
774     else if (phase_scaled_fixed<-MAX_PHASE)
775       {
776         phase_scaled_fixed =1-MAX_PHASE;
777         phase_cont_signal = -90.0;
778       }
779 } /* end vsi_set_phase */
780
781 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
782 /**
783 This function sets the desired reference Voltage.
784
785 The target is passed in ????.
786
787 \author A.McIver
788 \par History:
789 \li 24/04/09 AM - initial creation
790 \   24/04/09 DS - Changed from varying modulation depth to phase shift
791 \param[in] m Target output modulation depth
792 */
793 void vsi_set_vref(int16 vref)
794 {
795   GrabClear();
796   GrabStart();
797   GrabRun();
798   set_vref=1;
799   ref_volt=vref;
800   VDCref_fixed = ((long)vref<<FIXED_Q);
```

```
801 } /* end vsi_set_phase */
802
803
804 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
805 /**
806 This function returns the status of the VSI output system. It returns
807 - stopped or running
808 - fault code
809 - ramping or settled
810
811 \author A.McIver
812 \par History:
813 \li  13/10/07 AM - derived from 25kVA:vsi:vsi.c
814
815 \retval VSI_RUNNING VSI system switching with output
816 \retval VSI_SETTLED Output has reached target
817 \retval VSI_FAULT VSI system has detected a fault
818 */
819 Uint16 vsi_get_status(void)
820 {
821   return vsi_status;
822 } /* end vsi_get_status */
823
824
825 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
826 /**
827 This function returns the fault word of the VSI module.
828
829 \author A.McIver
830 \par History:
831 \li  04/03/08 AM - initial creation
832
833 \returns The present fault word
834 */
835 /// Report what faults are present in the VSI
836 Uint16 vsi_get_faults(void)
837 {
838   return detected_faults;
839 } /* end vsi_get_faults */
840
841
842 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
843 /* void vsi_clear_faults(void)
844 Parameters: none
845 Returns: nothing
846 Description: Clear the detected faults.
847 Notes:
848 History:
849   13/10/05 AM - initial creation
850 \li 28/04/08 AM - added event reporting
851 */
852 void vsi_clear_faults(void)
853 {
854   Uint16
855     i;
856
857   if (detected_faults & FAULT_VSI_PDPINT)
858   {
859     for (i=0; i<100; i++)
860       i++; // delay for fault to clear
861
862     EvaRegs.COMCONA.all = 0;
863     EvaRegs.COMCONA.all = 0xAA00;
864   }
865   detected_faults = 0;
866 } /* end vsi_clear_faults */
867
868 /* ===================================================================== */
869 /* Interrupt Routines */
870 /* ===================================================================== */
871
872 /**
873 \fn interrupt void isr_time(void)
874 \brief Updates VSI and performs closed loop control
875
876 This interrupt is triggered by the ADC interrupts.
877 It then:
878 - takes the adc measurements (synch sample, throws away every alternate one)
879 - determines the gains for the adaptive controller
880 - performs closed loop control calculations
```

```
881 - updates phase angle & calculates switching times
882
883 \author A.McIver
884 \par History:
885 \li  12/10/07 AM - initial creation
886 */
887 #ifndef BUILD_RAM
888 #pragma CODE_SECTION(isr_adc, "ramfuncs");
889 #endif
890
891 interrupt void isr_adc(void) //closed loop interrupt structure
892 {
893   /*
894   the interrupt can be divided into two sections, before and after the ADC read.
895   The first half - before the ADC read.
896     During this time, the deadtime compensation calculations will be performed,
897     follwed by the adaptve controller calculations.
898   The second half - after the ADC read.
899     During this time, the closed loop & feed forward calculations will be performed
900
901     PORTED OVER TO OPEN GIIB STRUCTURE
902     - asynchronous interrupt.
903   */
904
905   static int   cal_count=0,
906               vsi_synch=0;
907
908   if (cal_count ==0)
909     {
910 /*************************************
911 __calibrate_ADC()
912 *************************************/
913
914     //Dinesh's Calibration
915     //take 1024 readings at 0V and find the average
916
917       //sum 1024 readings
918       while (cal_count<1024)
919       {
920         Vdc1_cal  =  Vdc1_cal+(AdcRegs.ADCRESULT7-(ADC_OFFSET<<4));
921         Vdc2_cal   = Vdc2_cal+(AdcRegs.ADCRESULT1-(ADC_OFFSET<<4));
922         Vac1_cal  =  Vac1_cal+(AdcRegs.ADCRESULT6-(ADC_OFFSET<<4));
923         Vac2_cal  =  Vac2_cal+(AdcRegs.ADCRESULT10-(ADC_OFFSET<<4));
924         Vac3_cal  =  Vac3_cal+(AdcRegs.ADCRESULT2-(ADC_OFFSET<<4));
925         I1_cal    =  I1_cal+(AdcRegs.ADCRESULT4-(ADC_OFFSET<<4));
926         I2_cal    =  I2_cal+(AdcRegs.ADCRESULT8-(ADC_OFFSET<<4));
927         I3_cal    =  I3_cal+(AdcRegs.ADCRESULT0-(ADC_OFFSET<<4));
928         I4_cal    =  I4_cal+(AdcRegs.ADCRESULT5-(ADC_OFFSET<<4));
929         cal_count++;
930       }
931       //take average - divide by 1024
932       if (cal_count==1024)
933       {
934         Vdc1_cal  =  Vdc1_cal>>10;
935         Vdc2_cal   = Vdc2_cal>>10;
936         Vac1_cal  =  Vac1_cal>>10;
937         Vac2_cal  =  Vac2_cal>>10;
938         Vac3_cal  =  Vac3_cal>>10;
939         I1_cal    = I1_cal>>10;
940         I2_cal    = I2_cal>>10;
941         I3_cal    = I3_cal>>10;
942         I4_cal    = I4_cal>>10;
943
944       }
945
946       // calibration from references
947       adc_int.yHA.dc_sum += (Uint32)(AdcRegs.ADCRESULT12>>4);
948       adc_int.yLA.dc_sum += (Uint32)(AdcRegs.ADCRESULT14>>4);
949       adc_int.yHB.dc_sum += (Uint32)(AdcRegs.ADCRESULT13>>4);
950       adc_int.yLB.dc_sum += (Uint32)(AdcRegs.ADCRESULT15>>4);
951       adc_int.count_cal++;
952
953       if (adc_int.count_cal > ADC_COUNT_CAL)
954       {
955         adc_int.count_cal       = 0;
956         adc_int.yHA.dc_sum_bak   = adc_int.yHA.dc_sum;
957         adc_int.yLA.dc_sum_bak   = adc_int.yLA.dc_sum;
958         adc_int.yHB.dc_sum_bak   = adc_int.yHB.dc_sum;
959         adc_int.yLB.dc_sum_bak   = adc_int.yLB.dc_sum;
960         adc_int.yHA.dc_sum        = 0;
```

```
961         adc_int.yLA.dc_sum        = 0;
962         adc_int.yHB.dc_sum       = 0;
963         adc_int.yLB.dc_sum       = 0;
964         adc_int.flag_cal         = 1;
965       }
966 
967       puts_COM1("\n\nCALIBRATION COMPLETE\n\n");
968 
969     }
970 
971   SET_TP11();  //timing bit
972 
973   //set a pin to trigger CRO on reference step
974   if (VDCref_fixed==prev_VDCref_fixed)
975     {
976       CLEAR_TP10();
977     }
978   else SET_TP10();
979   prev_VDCref_fixed = VDCref_fixed;
980 
981 /************
982 __ADC_CL()  *
983 *************/
984       //The bidirectional converter needs 2 analog inputs, ie DC bus voltage and load current.
985       //for feed-forward compensation, the load current needs to be scaled by the modulation depth.
986       //the modulation depth va is being passed using the DAC. DAC = va>>2.
987       //this is fed into the VGEN input as a 3rd ADC.
988 
989       Vdc1_fixed  =  (((AdcRegs.ADCRESULT7-Vdc1_cal)>>4)  -ADC_OFFSET)*VDC_ANALOG_GAIN;
990       Vdc2_fixed  =  (((AdcRegs.ADCRESULT1-Vdc2_cal)>>4)  -ADC_OFFSET)*VDC_ANALOG_GAIN;
991       Vac1_fixed  =  (((AdcRegs.ADCRESULT6-Vac1_cal)>>4)  -ADC_OFFSET)*VAC_ANALOG_GAIN;
992       Vac2_fixed  =  (((AdcRegs.ADCRESULT10-Vac2_cal)>>4)-ADC_OFFSET)*VAC_ANALOG_GAIN;
993       Vac3_fixed  =  (((AdcRegs.ADCRESULT2-Vac3_cal)>>4)  -ADC_OFFSET)*VAC_ANALOG_GAIN;
994       I1_fixed    = (((AdcRegs.ADCRESULT4-I1_cal)>>4)    -ADC_OFFSET)*I_ANALOG_GAIN;
995       I2_fixed    = (((AdcRegs.ADCRESULT8-I2_cal)>>4)    -ADC_OFFSET)*I_ANALOG_GAIN;
996       I3_fixed    = (((AdcRegs.ADCRESULT0-I3_cal)>>4)    -ADC_OFFSET)*I_ANALOG_GAIN;
997       I4_fixed    = (((AdcRegs.ADCRESULT5-I4_cal)>>4)    -ADC_OFFSET)*I_ANALOG_GAIN;
998       Vgen_fixed  =  (((AdcRegs.ADCRESULT11-VGEN_CAL)>>4)-ADC_OFFSET)*VGEN_ANALOG_GAIN;
999 
1000      VdcIN_fixed = Vdc1_fixed;
1001      va_VSI      =  Vgen_fixed;
1002      VdcOUT_fixed = (Vdc2_fixed+Vac1_fixed+Vac2_fixed+Vac3_fixed)>>2;
1003 
1004      if (vsi_synch==4)
1005      {
1006        IVSI_fixed = (int32)((((I1_fixed+I2_fixed)>>1)*(int32)(va_VSI))/(int32)PERIOD_2_VSI);     // scaled by mod depth
1007      }
1008 
1009      Iload_fixed = IVSI_fixed + ((I3_fixed+I4_fixed)>>1); // AC+DC components
1010 
1011  //first determine the Average operating phase_shift (moving average of the last 4 phaseshifts)
1012  phase_shift_avrg=0;  //in counts
1013  VdcOUT_fixed_avrg=0;
1014  counter_avrg=1;
1015  phase_shift_record[vsi_synch] = abs(phase_shift);
1016  VdcOUT_fixed_record[vsi_synch] = VdcOUT_fixed;
1017 
1018  while(counter_avrg<=4)
1019  {
1020    phase_shift_avrg += phase_shift_record[counter_avrg]>>2;
1021    VdcOUT_fixed_avrg += VdcOUT_fixed_record[counter_avrg]>>2;
1022    counter_avrg++;
1023  }
1024 
1025 /*******************
1026 _Adaptive_Gain_Calc()
1027 *******************/
1028 
1029    if(DT_COMP)
1030      delta0_aug_fixed=abs(phase_shift-phase_aug_DT_fixed);
1031    else
1032      delta0_aug_fixed=abs(phase_shift);
1033 
1034  if (delta0_aug_fixed>MAX_PHASE) delta0_aug_fixed=MAX_PHASE;  //IS IN counts
1035 
1036  //then determine the B value
1037  delf_delu_fixed = 0;
1038  n_harm=0;
1039 
1040  for (n_harm = 0;n_harm<6;n_harm++)
```

```
1041   {
1042     //fixed point
1043     sin_count             = (Uint32)((phi_z_fixed[n_harm]-harm[n_harm]*delta0_aug_fixed)*COUNT_TO_SINTABLE);
1044     SIN_TABLE_READ(sin_count,sin_val_adapt);
1045     //Determine B_delta value - for proportional term
1046     delf_delu_temp_fixed  =  (int32)(sin_val_adapt*inv_Z_harm_fixed[n_harm])>>(14+15-FIXED_Q);
1047     //shift right because Z-harm_fixed is scaled by 15 and 14 for the sine table, we want to leave it scaled to fixed_Q
1048     delf_delu_fixed       += delf_delu_temp_fixed;
1049   }
1050   //scale by constants
1051   delf_delu_fixed_scaled   = (int32)(delf_delu_fixed*(int32)DELF_DELU_CONST)>>(FIXED_Q-4);
1052                             //further shift by 4 is needed because delf_delu_const has been scaled by 4 earlier,
1053                             //and Kp is scaled by FIXED_Q+2 to give more room to operate
1054
1055   //scale the proportional gain
1056   Kp_adapt_fixed=((OMEGA_C_BIDC_FIXED)/delf_delu_fixed_scaled);
1057   if (Kp_adapt_fixed>=VDC_KP_MAX_FIXED) Kp_adapt_fixed = VDC_KP_MAX_FIXED;
1058   if (Kp_adapt_fixed<=VDC_KP_MIN_FIXED) Kp_adapt_fixed = VDC_KP_MIN_FIXED;
1059
1060 /*********
1061 _BIDC_FF()
1062 *********/
1063
1064       BIDC_FF=0;
1065       Iload_abs=abs(Iload_fixed);
1066       //Iload Feedforward - search algorithm
1067       lo=0;
1068       hi=PERIOD_2_BIDC-1;
1069       while (hi>lo)
1070       {
1071         mid = ((hi-lo)/2)+lo;
1072         if (Iload_abs<Iload_FF_fixed[mid])        hi=mid-1; //in the bottom half
1073         else if (Iload_abs>Iload_FF_fixed[mid])    lo=mid+1;
1074         else if (Iload_abs==Iload_FF_fixed[mid])
1075           {
1076             lo=mid;
1077             break;
1078           }
1079         else if ((hi-lo)<10) break;
1080       }
1081
1082       if (saturated==1) BIDC_FF=0;
1083       else
1084       {
1085         if  (Iload_fixed>0) BIDC_FF = lo;
1086         else BIDC_FF = -lo;
1087       }
1088
1089 /*************************
1090 _BIDC_DT_Compensation()
1091 *************************/
1092
1093     phase_rad_ratio_fixed  = ((int32)(abs((int32)phase_shift))<<FIXED_Q)/(PERIOD_BIDC<<1);
1094     VDCout_txscaled_fixed = (VdcOUT_fixed*NPRI_NSEC_FIXED)>>FIXED_Q;
1095     Vp_Vs_4Vp_fixed = ((VIN_FIXED-VDCout_txscaled_fixed)<<(FIXED_Q-2))/VIN_FIXED;
1096     Vs_Vp_4Vp_fixed = ((VDCout_txscaled_fixed-VIN_FIXED)<<(FIXED_Q-2))/VIN_FIXED;
1097     Vs_Vp_4Vs_fixed = ((VDCout_txscaled_fixed-VIN_FIXED)<<(FIXED_Q-2))/VDCout_txscaled_fixed;
1098     Vs_Vp_DB_fixed   = ((VDCout_txscaled_fixed/(PERIOD_BIDC<<1))*DEADBAND_COUNT_BIDC)/(int32)VIN;
1099     Vp_Vs_DB_fixed   = (((VIN_FIXED/(PERIOD_BIDC<<1))*DEADBAND_COUNT_BIDC)<<FIXED_Q)/VDCout_txscaled_fixed;
1100
1101
1102 // First, calculate slew time
1103
1104     if (VIN_FIXED>VDCout_txscaled_fixed)    //Vp>Vs
1105     {
1106       if (phase_shift_avrg<0)  //leading
1107       {
1108         Tslew_count = (int16)((phase_rad_ratio_fixed - Vp_Vs_4Vp_fixed  - Vs_Vp_DB_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
1109
1110         //then calculate phase augmentation
1111         if ((VIN_FIXED-VDCout_txscaled_fixed)>(20<<FIXED_Q))
1112         {
1113           if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = 0;
1114           else if (Tslew_count<0)                  phase_aug_DT_fixed = DEADBAND_COUNT_BIDC;
1115           else                                    phase_aug_DT_fixed = DEADBAND_COUNT_BIDC-Tslew_count;  //in counts
1116         }
1117         else
1118         {
1119           if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = 0;
1120           else                                    phase_aug_DT_fixed = DEADBAND_COUNT_BIDC;
```

```
1121          }
1122        }
1123        else  //lagging
1124        {
1125          Tslew_count = (int16)((Vp_Vs_4Vp_fixed - phase_rad_ratio_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
1126
1127          if ((VIN_FIXED - VDCout_txscaled_fixed)>(20<<FIXED_Q))
1128          {
1129            if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = DEADBAND_COUNT_BIDC;
1130            else if (Tslew_count<0)                  phase_aug_DT_fixed = 0;
1131            else                                    phase_aug_DT_fixed = Tslew_count;  //in counts
1132          }
1133          else
1134          {
1135            if (Tslew_count>0)                      phase_aug_DT_fixed = DEADBAND_COUNT_BIDC;
1136            else                                     phase_aug_DT_fixed = 0;
1137          }
1138        }
1139      }
1140      else  //Vp<Vs
1141      {
1142        if (phase_shift_avrg<0)  //leading
1143        {
1144          Tslew_count = (int16)((Vs_Vp_4Vp_fixed - phase_rad_ratio_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
1145
1146          if ((VDCout_txscaled_fixed-VIN_FIXED)>(20<<FIXED_Q))
1147          {
1148            if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
1149            else if (Tslew_count<0)                  phase_aug_DT_fixed = 0;
1150            else                                    phase_aug_DT_fixed = -Tslew_count;  //in counts
1151          }
1152          else
1153          {
1154            if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
1155            else                                     phase_aug_DT_fixed = 0;
1156          }
1157        }
1158        else  //lagging
1159        {
1160          Tslew_count = (int16)((phase_rad_ratio_fixed - Vs_Vp_4Vs_fixed  - Vp_Vs_DB_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
1161
1162          if ((VDCout_txscaled_fixed-VIN_FIXED)>(20<<FIXED_Q))
1163          {
1164            if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = 0;
1165            else if (Tslew_count<0)                  phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
1166            else                                    phase_aug_DT_fixed = -(DEADBAND_COUNT_BIDC-Tslew_count);  //in counts
1167          }
1168          else
1169          {
1170            if (Tslew_count>DEADBAND_COUNT_BIDC)    phase_aug_DT_fixed = 0;
1171            else                                     phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
1172          }
1173        }
1174      }
1175
1176 /*******************
1177 _BIDC_PI_Control_Loop()
1178 *******************/
1179   if(EvaRegs.GPTCONA.bit.T1STAT==1)   //so last int was an underflow
1180      {
1181        //only update once a cycle
1182        VDC_Kp_fixed  =  Kp_adapt_fixed;
1183
1184        //Now in fixed point
1185         VDCerror_fixed         =   VDCref_fixed-VdcOUT_fixed;
1186        VDC_prop_fixed         =   (VDCerror_fixed*VDC_Kp_fixed)>>(FIXED_Q+2);
1187        VDC_intnow_fixed       =   (VDC_prop_fixed*VDC_KI_FIXED)>>FIXED_Q;
1188        VDC_int_fixed          +=   VDC_intnow_fixed;
1189
1190        VDC_cont_signal_fixed =   VDC_prop_fixed + VDC_int_fixed;
1191      }
1192 /****************
1193 _BIDC_SET_PHASE()
1194 ****************/
1195      #ifdef OPEN_LOOP   //Open loop
1196        phase_shift = phase_scaled_fixed;
1197      #endif
1198      #ifdef CLOSED_LOOP
1199        phase_shift = (int16)((int32)(VDC_cont_signal_fixed*PERIOD_SCALE_BIDC)>>FIXED_Q);
1200        if(FF_ENABLE)  phase_shift +=  (int16)BIDC_FF;
```

```
1201       if(DT_COMP)    phase_shift -=   phase_aug_DT_fixed;
1202     #endif
1203
1204 /**************
1205 ___DESAT()    *
1206 ***************/
1207     if (abs(phase_shift)>=MAX_PHASE)
1208       {
1209         SET_TP13(); // desat bit
1210         saturated=1;
1211         VDC_int_fixed -= VDC_intnow_fixed;
1212         if (phase_shift>0)
1213           {
1214             phase_shift = MAX_PHASE;
1215           }
1216         if (phase_shift<0)
1217           {
1218             phase_shift = -MAX_PHASE;
1219           }
1220       }
1221     else
1222       {
1223         saturated=0;
1224         CLEAR_TP13();
1225       }
1226   //end control loop
1227 /*****************
1228 _BiDC_Modulator()
1229 *****************/
1230   if(EvaRegs.GPTCONA.bit.T1STAT==1)   //so last int was an underflow
1231     {
1232 /***************
1233 _SYNCH_PULSE()
1234 ***************/
1235       //synch pulse for VSI. sent out at 5kHz. this code is seen at 20kHz, so count to 4.
1236       if (vsi_synch >=4)
1237       {
1238         GpioDataRegs.GPBSET.bit.GPIOB4 = 1;  //Sets synch output high
1239         SET_TP12();
1240         vsi_synch=0;
1241       }
1242       vsi_synch++;
1243
1244       /***************************
1245       *  Update switching times    *
1246       ***************************/
1247       EvaRegs.T1CMPR = PERIOD_BIDC-1;        //set the next interrupt to be at the top
1248       /*  The Bidirectional DC-DC Converter is comprised of 2 single phase bridges.
1249           Primary Bridge is controlled by EVA
1250           Secondary Bridge is controlled by EVB
1251       */
1252       //phases C&D
1253       EvbRegs.CMPR4 = PERIOD_2_BIDC-phase_shift;
1254       EvbRegs.CMPR5 = PERIOD_2_BIDC-phase_shift;
1255     }
1256   else                  //if heading down, last interrupt was PERIOD MATCH
1257     {
1258       GpioDataRegs.GPBCLEAR.bit.GPIOB4 = 1;  //Sets synch output low
1259       CLEAR_TP12();
1260       /***************************
1261       *  Update switching times    *
1262       ***************************/
1263       EvaRegs.T1CMPR = 1;                //set the next interrupt to be at the bottom
1264
1265       //phases C&D
1266       EvbRegs.CMPR4 = PERIOD_2_BIDC+phase_shift;
1267       EvbRegs.CMPR5 = PERIOD_2_BIDC+phase_shift;
1268     }
1269
1270 /* =====================================================================
1271 isr_GrabCodeCL()
1272 ===================================================================== */
1273 #ifdef GRAB_INCLUDE
1274
1275   if (GrabRunning())
1276   {
1277     GrabStore(0,(I1_fixed+I2_fixed)>>1);
1278     GrabStore(1,va_VSI);
1279     GrabStore(2,IVSI_fixed);
1280     GrabStore(3,(I3_fixed+I4_fixed)>>1);
```

```
1281     GrabStore(4,Iload_fixed);  //weird
1282 //    GrabStore(5,(int32)(abs((int32)phase_shift))<<FIXED_Q);
1283 //    GrabStore(6,((int32)(abs((int32)phase_shift))<<FIXED_Q)/(PERIOD_BIDC<<1));  //weird
1284     grab_index++;
1285
1286     if (grab_index >= GRAB_LENGTH)
1287       grab_mode = GRAB_STOPPED;
1288   }
1289 #endif
1290
1291   // Reinitialize for next ADC interrupt
1292   EvaRegs.EVAIFRA.all = BIT7;        // clear T1PINT & T1UFINT interrupt flag
1293   AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;        // clear interrupt flag
1294   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;      // Acknowledge interrupt to PIE Group 2
1295   CLEAR_TP11();        // timing bit
1296 } /* end isr_timer_CL */
1297
1298 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1299 /**
1300 Handles the PDPINT interrupt caused by a gate fault.
1301
1302 \author A.McIver
1303 \par History:
1304 \li  02/05/07 AM - initial creation
1305 */
1306 #ifndef BUILD_RAM
1307 #pragma CODE_SECTION(isr_gate_fault, "ramfuncs");
1308 #endif
1309 interrupt void isr_gate_fault(void)
1310 {
1311   is_switching = 0;
1312   VSI_DISABLE();
1313   detected_faults |= FAULT_VSI_PDPINT;
1314   // Acknowledge this interrupt to receive more interrupts from group 1
1315   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
1316   EvaRegs.EVAIFRA.all = BIT0;
1317 } /* end isr_gate_fault */
1318
1319
1320 /* ======================================================================
1321 __VSI_State_Functions()
1322 ======================================================================= */
1323
1324
1325 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1326 /**
1327 This function initialises the VSI system. It resets the target modulation
1328 depth to zero.
1329
1330 It is followed by the stop state.
1331
1332 \author A.McIver
1333 \par History:
1334 \li  12/10/07 AM - initial creation
1335 */
1336 void st_vsi_init(void)
1337 {
1338   mod_ref = 0;
1339   mod_targ = 0;
1340   EvaRegs.ACTRA.all = 0x0000;
1341   VSI_DISABLE();
1342   vsi_status = VSI_INIT;
1343   SS_NEXT(vsi_state,st_vsi_stop);
1344 } /* end st_vsi_init */
1345
1346
1347 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1348 /**
1349 This is the state where the VSI is stopped. There is no switching. It waits
1350 for a start trigger.
1351
1352 \author A.McIver
1353 \par History:
1354 \li  12/10/07 AM - initial creation
1355 */
1356 void st_vsi_stop(void)
1357 {
1358   if (SS_IS_FIRST(vsi_state))
1359   {
1360     SS_DONE(vsi_state);
```

```
1361     VSI_DISABLE();
1362     mod_targ = 0;
1363     vsi_status = VSI_STOP;
1364   }
1365
1366   if (detected_faults != 0)
1367   {
1368     SS_NEXT(vsi_state,st_vsi_fault);
1369     return;
1370   }
1371
1372   if (is_switching != 0) // start trigger
1373   {
1374     SS_NEXT(vsi_state,st_vsi_gate_charge);
1375   }
1376 } /* end st_vsi_stop */
1377
1378
1379 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1380 /**
1381 In this state the VSI gates are enabled and the low side gates held on to
1382 charge the high side gate drivers. The next state is either the ramp state.
1383
1384 \author A.McIver
1385 \par History:
1386 \li  12/10/07 AM - initial creation
1387 */
1388 void st_vsi_gate_charge(void)
1389 {
1390   if (SS_IS_FIRST(vsi_state))
1391   {
1392     SS_DONE(vsi_state);
1393     vsi_counter = 0;
1394 //    VSI_GATE_CHARGE();
1395 //    vsi_status |= VSI_RUNNING;
1396   }
1397   if (detected_faults != 0)
1398   {
1399     SS_NEXT(vsi_state,st_vsi_fault);
1400     return;
1401   }
1402   // check for stop signal
1403   if (is_switching == 0)
1404   {
1405     SS_NEXT(vsi_state,st_vsi_stop);
1406     return;
1407   }
1408   vsi_counter++;
1409   if (vsi_counter > 200)
1410   {
1411     SS_NEXT(vsi_state,st_vsi_ramp);
1412   }
1413 } /* end st_vsi_gate_charge */
1414
1415
1416 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1417 /**
1418 This state ramps up the target modulation depth to match the reference set by
1419 the background. It only changes the target every 100ms and synchronises the
1420 change with a zero crossing to avoid step changes in the output.
1421
1422 \author A.McIver
1423 \par History:
1424 \li  12/10/07 AM - initial creation
1425 \li 28/04/08 AM - added event reporting
1426 */
1427 void st_vsi_ramp(void)
1428 {
1429   if (SS_IS_FIRST(vsi_state))
1430   {
1431     SS_DONE(vsi_state);
1432     vsi_counter = 0;
1433     VSI_ENABLE();
1434     vsi_status = VSI_RAMP;
1435   }
1436   if (detected_faults != 0)
1437   {
1438     SS_NEXT(vsi_state,st_vsi_fault);
1439     return;
1440   }
```

```
1441   // check for stop signal
1442   if (is_switching == 0)
1443   {
1444     SS_NEXT(vsi_state,st_vsi_stop);
1445     return;
1446   }
1447   // check for target reached
1448   if (mod_targ == mod_ref)
1449   {
1450     SS_NEXT(vsi_state,st_vsi_run);
1451     return;
1452   }
1453   // ramp reference towards target
1454   if (mod_ref > mod_targ + 5)
1455   {
1456     mod_targ += 5;
1457   }
1458   else if (mod_ref < mod_targ - 5)
1459   {
1460     mod_targ -= 5;
1461   }
1462   else
1463   {
1464     mod_targ = mod_ref;
1465   }
1466 } /* end st_vsi_ramp */
1467
1468
1469 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1470 /**
1471 This state has the VSI running with the target voltage constant. The output is
1472 now ready for measurements to begin. If the reference is changed then the
1473 operation moves back to the ramp state.
1474
1475 \author A.McIver
1476 \par History:
1477 \li  12/10/07 AM - initial creation
1478 */
1479 void st_vsi_run(void)
1480 {
1481   if (SS_IS_FIRST(vsi_state))
1482   {
1483     SS_DONE(vsi_state);
1484     vsi_status = VSI_RUNNING;
1485   }
1486   if (detected_faults != 0)
1487   {
1488     SS_NEXT(vsi_state,st_vsi_fault);
1489     return;
1490   }
1491   // check for stop signal
1492   if (is_switching == 0)
1493   {
1494     SS_NEXT(vsi_state,st_vsi_stop);
1495   }
1496   // check for changes in reference
1497   if (mod_targ != mod_ref)
1498   {
1499     vsi_status &= ~VSI_SETTLED;
1500     SS_NEXT(vsi_state,st_vsi_ramp);
1501   }
1502 } /* end st_vsi_run */
1503
1504
1505 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1506 /* void st_vsi_fault(void)
1507 Parameters: none
1508 Returns: nothing
1509 Description: Delays for a while after faults are cleared.
1510 Notes:
1511 History:
1512   03/11/05 AM - initial creation
1513 \li 04/03/08 AM - set vsi_status with fault bit
1514 \li 28/04/08 AM - added event reporting
1515 */
1516 void st_vsi_fault(void)
1517 {
1518   if (SS_IS_FIRST(vsi_state))
1519   {
1520     SS_DONE(vsi_state);
```

```
1521     VSI_DISABLE();
1522     vsi_counter = 0;
1523     vsi_status = VSI_FAULT;
1524     putxx(detected_faults);
1525     puts_COM1("->VSI faults\n");
1526   }
1527   if (detected_faults == 0)
1528     vsi_counter++;
1529   else
1530     vsi_counter = 0;
1531   if (vsi_counter > 100)
1532   {
1533     SS_NEXT(vsi_state,st_vsi_stop);
1534   }
1535 } /* end st_vsi_fault */
1536
1537
1538 /* =====================================================================
1539 __Local_Functions()
1540 ======================================================================== */
1541
1542
1543 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1544 /**
1545 This function is called every fundamental period to perform the RMS
1546 calculations and scale the analog quantities to Volts and Amps for use in the
1547 background.
1548
1549 \author A.McIver
1550 \par History:
1551 \li  12/10/07 AM - derived from IR25kVA:vsi:adc_scale
1552 \li 21/08/08 AM - added VSI DC offset compensation
1553 \li 12/09/08 AM - added stop_count and moved to floating point data
1554 */
1555 //void scale_adc_rms(void)
1556 //{
1557 //  double
1558 //    val,
1559 //    temp;
1560 //
1561 //  // calculate A0 RMS quantity
1562 //  temp = (double)adc_int.A0.dc_sum_bak/(double)adc_int.count_rms_bak;
1563 //  val = (double)adc_int.A0.rms_sum_bak*(double)(1<<ADC_RMS_PS)
1564 //          / (double)adc_int.count_rms_bak - temp*temp;
1565 //  if (val < 0.0) val = 0.0;
1566 //  adc_int.A0.real = ADC_REAL_SC * sqrt(val);
1567 //} /* end scale_adc_rms */
1568
1569
1570 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1571 /**
1572 This function is called every ADC_DC_TIME to perform the DC calculations and
1573 scale the analog quantities to Volts and Amps for use in the background.
1574
1575 \author A.McIver
1576 \par History:
1577 \li  12/10/07 AM - derived from IR25kVA:vsi:adc_scale
1578 */
1579 //void scale_adc_dc(void)
1580 //{
1581 //  double
1582 //    val;
1583 //
1584 //  adc_int.A0.real = (double)adc_int.A0.dc_sum_bak/(double)ADC_COUNT_DC;
1585 //  adc_int.A2.real = (double)adc_int.A2.dc_sum_bak/(double)ADC_COUNT_DC;
1586 //  adc_int.A4.real = (double)adc_int.A4.dc_sum_bak/(double)ADC_COUNT_DC;
1587 //  adc_int.A6.real = (double)adc_int.A6.dc_sum_bak/(double)ADC_COUNT_DC;
1588 //
1589 //  // calculate B0 DC quantity
1590 //  val = (double)adc_int.B0.dc_sum_bak/(double)ADC_COUNT_DC;
1591 //  adc_int.B0.real = ADC_REAL_SC * val;
1592 //
1593 //} /* end scale_adc_dc */
1594
1595
1596 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1597 /**
1598 Calibrates the adc for gain and offset using the reference inputs.
1599
1600 See spra989a.pdf for calibration details
```

```
1601
1602 \author A.McIver
1603 \par History:
1604 \li  07/10/05 AM - initial creation
1605 */
1606 void calibrate_adc(void)
1607 {
1608 //  char
1609 //    str[60];
1610
1611   yHA = (double)adc_int.yHA.dc_sum_bak/(double)ADC_COUNT_CAL;
1612   yLA = (double)adc_int.yLA.dc_sum_bak/(double)ADC_COUNT_CAL;
1613   yHB = (double)adc_int.yHB.dc_sum_bak/(double)ADC_COUNT_CAL;
1614   yLB = (double)adc_int.yLB.dc_sum_bak/(double)ADC_COUNT_CAL;
1615
1616   cal_gain_A = (xH - xL)/(yHA - yLA);
1617   cal_offset_A = yLA * cal_gain_A - xL;
1618
1619   cal_gain_B = (xH - xL)/(yHB - yLB);
1620   cal_offset_B = yLB * cal_gain_B - xL;
1621
1622   // sanity check on gains
1623   if (   ( (cal_gain_A > 0.94) && (cal_gain_A < 1.05) )
1624     && ( (cal_gain_B > 0.94) && (cal_gain_B < 1.05) )
1625     && ( (cal_offset_A > -80.0) && (cal_offset_A < 80.0) )
1626     && ( (cal_offset_B > -80.0) && (cal_offset_B < 80.0) ) )
1627   {
1628     cal_gainA = (int16)(cal_gain_A*(double)(1<<14));
1629     cal_gainB = (int16)(cal_gain_B*(double)(1<<14));
1630     cal_offsetA = (int16)cal_offset_A;
1631     cal_offsetB = (int16)cal_offset_B;
1632   }
1633 //  sprintf(str,"cal:gA=%.3f,oA=%5.1f, gB=%.3f,oB=%5.1f\n",cal_gain_A,
1634 //          cal_offset_A,cal_gain_B,cal_offset_B);
1635 //  puts_COM1(str);
1636 } /* end calibrate_adc */
1637
1638 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1639
1640 void get_state(void){
1641   if(vsi_state.f == st_vsi_init){
1642     puts_COM1("INIT ");
1643   }
1644   else if(vsi_state.f == st_vsi_stop){
1645     puts_COM1("STOP ");
1646   }
1647   else if(vsi_state.f == st_vsi_gate_charge){
1648     puts_COM1("GATE ");
1649   }
1650   else if(vsi_state.f == st_vsi_ramp){
1651     puts_COM1("RAMP ");
1652   }
1653   else if(vsi_state.f == st_vsi_run){
1654     puts_COM1("RUN  ");
1655   }
1656   else if(vsi_state.f == st_vsi_fault){
1657     puts_COM1("FAU  ");
1658   }
1659 }
```

A.2.3 DSP Code – Voltage Source Inverter

```
1 /**
2 \file
3 \brief Main system definitions
4
5 \par Developed By:
6   Creative Power Technologies, (C) Copyright 2009
7 \author A.McIver
8 \par History:
9 \li  23/04/09 AM - initial creation
10 \ Modified Dinesh Segaran
11 \li 26/08/10
12 */
13
14
15 /* ======================================================================
16 __Definitions()
17 ========================================================================= */
18
19 #define __SQRT2          1.4142135624
20 #define __SQRT3          1.7320508075
21 #define __PI            3.1415926535
22 #define __PI_2          __PI/2.0
23 #define __INVPI         1/__PI
24 #define __INVPI_2       1/__PI_2
25
26 #define SYSCLK_OUT        (150e6)
27 #define  HSPCLK          (SYSCLK_OUT)
28 #define LSPCLK          (SYSCLK_OUT/4)
29
30 /* ======================================================================
31 __State_Simple_Definitions()
32 ========================================================================= */
33
34 /** Simple State Machine Type */
35  typedef void (* funcPtr)(void);
36 typedef struct
37 {
38   funcPtr f;
39   unsigned int call_count;
40   unsigned char first;
41 } type_state;
42
43
44 /* Simple State Handling Macros */
45 #define SS_NEXT(_s_,_f_)    { _s_.f = (funcPtr)_f_; \
46                   _s_.call_count = 0; \
47                   _s_.first = 1; }
48 #define SS_IS_FIRST(_s_)    (_s_.first == 1)
49 #define SS_DONE(_s_)      { _s_.first = 0; }
50 #define SS_DO(_s_)          { _s_.call_count++; \
51                   ((*(_s_.f))()); }
52 #define SS_IS_PRESENT(_s_,_f_)  (_s_.f == (funcPtr)_f_)
53
54
55 /* ======================================================================
56 __Grab_Code_Definitions()
57 ========================================================================= */
58 /**/
59 #define GRAB_INCLUDE
60
61 #ifdef GRAB_INCLUDE
62 // grab array size
63 #define GRAB_LENGTH        200
64 #define GRAB_WIDTH        3
65
66 // modes
67 #define GRAB_GO            0
68 #define GRAB_WAIT          1
69 #define GRAB_TRIGGER      2
70 #define GRAB_STOPPED      3
71 #define GRAB_SHOW          4
72
73 // macros
74 #define GrabStart()        grab_mode = GRAB_TRIGGER;
75 #define GrabStop()        grab_mode = GRAB_STOPPED;
76 #define GrabRun()          grab_mode = GRAB_GO;
77 #define GrabShow()        grab_mode = GRAB_SHOW;
```

```
78
79 #define GrabClear()        { grab_mode = GRAB_WAIT; \
80                             grab_index = 0; }
81
82 #define GrabTriggered()      (grab_mode == GRAB_TRIGGER)
83 #define GrabRunning()        (grab_mode == GRAB_GO)
84 #define GrabStopped()        (grab_mode == GRAB_STOPPED)
85 #define GrabAvail()          (grab_mode >= GRAB_STOPPED)
86 #define GrabShowTrigger()    (grab_mode == GRAB_SHOW)
87
88 #define GrabStore(_loc_,_data_)  grab_array[grab_index][_loc_] = _data_;
89
90 #define GrabStep()        { grab_index++; \
91                   if (grab_index >= GRAB_LENGTH) \
92                       grab_mode = GRAB_STOPPED; }
93
94 // variables
95 extern int16
96   step,
97   grab_mode,
98   grab_index,
99   set_vref;
100
101 extern long
102   volt_req,wo;
103 extern double
104   grab_array[GRAB_LENGTH][GRAB_WIDTH];
105
106 // functions
107 void GrabDisplay(int16 index);
108 void GrabInit(void);
109
110 #endif
111 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

```
1  /**
2 \file
3 \brief System software for the DA-2810 Demo code
4
5
6 \par Developed By:
7   Creative Power Technologies, (C) Copyright 2009
8 \author A.McIver
9 \par History:
10 \li  23/04/09 AM - initial creation
11 \ Modified Dinesh Segaran
12 \    26/08/10 DS - Fixed Point implementation of the Adaptive Controlled
13 \                  Bidirectional DC-DC Converter
14 \    02/11/10 DS  -  Load Step for Bidirectional DC-DC Converter
15 \    16/03/11 DS - Grid Connected H-Bridge, with DC links supplied by a
16 \                  Bidirectional DC-DC Converter
17 */
18
19 // compiler standard include files
20 #include <stdlib.h>
21 #include <stdio.h>
22 #include <math.h>
23
24 // processor standard include files
25 #include <DSP281x_Device.h>
26 #include <DSP281x_Examples.h>
27
28 #ifdef COM0_CONSOLE
29 #include <bios0.h>
30 #endif
31 #ifdef COM1_CONSOLE
32 #include <bios1.h>
33 #endif
34
35 // board standard include files
36 #include <lib_mini2810.h>
37 #include <dac_ad56.h>
38 #include <lib_cpld.h>
39 #include <lib_giib.h>
40
41 // common project include files
42
43 // local include files
44 #include "main.h"
45 #include "conio.h"
46 #include "vsi_InvLoad.h"
47
48
49 /* =====================================================================
50 _Hash_Definitions()
51 ======================================================================== */
52 // Serial step in frequency
53 #define VSI_FUNDSTEP 0.0001
54
55 //Serial step in phase
56 #define PHASE_STEP_LARGE 10
57 #define PHASE_STEP_SMALL 1
58
59 //serial step in modulation depth
60 #define VSI_MODSTEP 0.01
61
62 //Serial step in VSI current reference
63 #define VSI_CURRSTEP  2.0
64
65 //serial step in Phase Shift
66 #define DELTA_PHASE 1.0
67 /* =====================================================================
68 __Typedefs()
69 ======================================================================== */
70
71 /// Time related flag type
72 /** This structure holds flags used in background timing. */
73 typedef struct
74 {
75   Uint16
76     msec:1,    ///< millisecond flag
77     msec10:1,  ///< 10ms flag
78     sec0_1:1,  ///< tenth of a second flag
79     sec:1;    ///< second flag
80 } type_time_flag;
```

```
81
82
83 /* =====================================================================
84 __Variables()
85 ===================================================================== */
86
87 #ifndef BUILD_RAM
88 // These are defined by the linker (see F2812.cmd)
89 extern Uint16 RamfuncsLoadStart;
90 extern Uint16 RamfuncsLoadEnd;
91 extern Uint16 RamfuncsRunStart;
92 #endif
93
94 // state determination
95 extern int16 Vdc_fixed;
96 extern int16 load_enable;
97 extern double mag_serial;
98 // Background variables
99 Uint16
100   quit = 0;  ///< exit flag
101
102
103 /// timing variable
104 type_time_flag
105   time =
106   {
107     0,0,0,0 // flags
108   };
109
110 Uint32
111   idle_count = 0,        ///< count of idle time in the background
112   idle_count_old = 0,      ///< previous count of idle time
113   idle_diff = 0;        ///< change in idle time btwn low speed tasks
114
115 char
116   str[40];        // string for displays
117
118 //to display correctly
119 int initial=0;
120
121 /* =====================================================================
122 __Local_Function_Prototypes()
123 ===================================================================== */
124
125 /* 1 second interrupt for display */
126 interrupt void isr_cpu_timer0(void);
127
128 /// display operating info
129 void com_display(void);
130
131 /// display help
132 void display_help(void);
133
134 /* process keyboard input */
135 void com_keyboard(void);
136
137 /* =====================================================================
138 __Grab_Variables()
139 ===================================================================== */
140
141 #ifdef GRAB_INCLUDE
142 //#pragma DATA_SECTION(grab_array, "bss_grab")
143 int16
144   step=0,
145   grab_mode = GRAB_STOPPED,
146   grab_index,
147   set_vref=0;
148 long
149   volt_req=10,
150   wo=314;
151 double
152   grab_array[GRAB_LENGTH][GRAB_WIDTH];
153 #endif
154
155 /* =====================================================================
156 __Serial_input_variables()
157 ===================================================================== */
158
159 //VSI Modulation Depth Variation
160 double mod_serial=0.0;
```

```
161
162 //VSI Reference Current Variation
163 double Imag_serial=0.0;
164
165 //VSI fundamental frequency variation
166 double ffund_serial=50.05;
167
168 //BIDC Phase Shift Variation
169 double phase_serial=0.0;
170
171 //external debug variables. so they can be displayed
172
173 /* ====================================================================== */
174 /* Main */
175 /* ====================================================================== */
176 /* Idle time benchmark:
177 \li Ram based program with only bios interrupt and an empty main loop gives an
178 idle_diff of 4.69M (4,685,900)
179 \li 23/03/09 V1.02 1.23M with no modbus running
180 */
181 void main(void)
182 {
183   static int
184   i = 0;
185 //  initial=0;
186
187 // Disable CPU interrupts
188   DINT;
189 // Initialise DSP for PCB
190   lib_mini2810_init(150/*MHz*/,37500/*kHz*/,150000/*kHz*/,LIB_EVAENCLK
191     |LIB_EVBENCLK|LIB_ADCENCLK|LIB_SCIAENCLK|LIB_SCIBENCLK|LIB_MCBSPENCLK);
192
193   InitGpio();
194   spi_init(MODE_CPLD);
195 //  SpiaRegs.SPICCR.bit.SPILBK = 1;        //Set SPI on loop back for testing
196   cpld_reg_init();
197   giib_init();
198
199 // Initialize the PIE control registers to their default state.
200   InitPieCtrl();
201 // Disable CPU interrupts and clear all CPU interrupt flags:
202   IER = 0x0000;
203   IFR = 0x0000;
204 // Initialize the PIE vector table with pointers to the shell Interrupt
205 // Service Routines (ISR).
206 // This will populate the entire table, even if the interrupt
207 // is not used in this example.  This is useful for debug purposes.
208 // The shell ISR routines are found in DSP281x_DefaultIsr.c.
209 // This function is found in DSP281x_PieVect.c.
210   InitPieVectTable();
211
212 #ifndef BUILD_RAM
213 // Copy time critical code and Flash setup code to RAM
214 // The  RamfuncsLoadStart, RamfuncsLoadEnd, and RamfuncsRunStart
215 // symbols are created by the linker. Refer to the F2810.cmd file.
216   MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);
217
218 // Call Flash Initialization to setup flash waitstates
219 // This function must reside in RAM
220   InitFlash();
221 #endif
222
223   // Initialise COM port
224   bios_init_COM1(9600L);
225   InitAdc();
226   InitCpuTimers();
227
228 // Configure CPU-Timer 0 to interrupt every tenth of a second:
229 // 150MHz CPU Freq, 1ms Period (in uSeconds)
230   ConfigCpuTimer(&CpuTimer0, 150.0/*MHz*/, 1000.0/*us*/);
231   StartCpuTimer0();
232
233 // Interrupts that are used in this example are re-mapped to
234 // ISR functions found within this file.
235   EALLOW;  // This is needed to write to EALLOW protected register
236   PieVectTable.TINT0 = &isr_cpu_timer0;
237   EDIS;    // This is needed to disable write to EALLOW protected registers
238
239 // Enable TINT0 in the PIE: Group 1 interrupt 7
240   PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
```

```
241   IER |= M_INT1; // Enable CPU Interrupt 1
242   vsi_init();
243   EnableInterrupts();
244   //waste some time, so that the program can finish writing to the screen
245
246 #ifdef GRAB_INCLUDE
247   GrabInit();
248 #endif
249   spi_init(MODE_DAC);
250   spi_set_mode(MODE_DAC);
251   dac_init();
252   dac_set_ref(DAC_MODULE_D1,DAC_INT_REF);
253   dac_power_down(DAC_MODULE_D1,0x0F);
254   dac_write(DAC_MODULE_D1,DAC_WRn_UPDn,DAC_ADDR_ALL,2047);
255   spi_set_mode(MODE_CPLD);      //Use mode setting for CPLD for SPI to initialize SPI setting
256   DISABLE_CPLD();
257   spi_set_mode(MODE_DAC);
258 /*
259 void main_loop(void)
260 */
261   while(quit == 0)
262   {
263
264     com_keyboard(); // process keypresses
265
266     if (time.msec != 0) // millisecond events
267     {
268       time.msec = 0;
269       vsi_state_machine();
270
271     }
272     else if (time.msec10 != 0) // ten millisecond events
273     {
274       time.msec10 = 0;
275     }
276       else if (time.sec0_1 != 0) // tenth of second events
277     {
278       time.sec0_1 = 0;
279       switch(initial)
280       {
281         /* case 0 never happens */
282         case 1: puts_COM1("\n\t Single-phase Bidirectional DC-DC Converter");break;
283         case 2: puts_COM1("\n\t\t Supplying a Grid Connected VSI");break;
284         case 3: puts_COM1("\n\t\t Dinesh Segaran 2011");break;
285         #ifdef OL_VSI
286         case 4: puts_COM1("\n\t M/m - VSI modulation depth up/down");break;
287         #endif
288         #ifdef CL_VSI
289         case 5: puts_COM1("\n\t A/a - VSI Current Ref up/down");break;
290         #endif
291         case 6: puts_COM1("\n\t l/L - Load Switch OFF/ON");break;
292         case 8: puts_COM1("\n\te/d - VSI Enable/Disable\n");break;
293         case 10: puts_COM1("\tg/h - Start/Display Grab\n");break;
294         case 11: puts_COM1("\tH  -  Display Help\n");break;
295         default: break;
296       }
297       if (initial<20) initial++;
298
299
300       if(GrabShowTrigger() && i < GRAB_LENGTH){
301         //GrabDisplay(0xFFFF);
302         GrabDisplay(i);
303         i++;
304         //GrabStop();
305       }
306       else if(GrabShowTrigger() && i == GRAB_LENGTH){
307         GrabStop();
308         i = 0;
309       }
310     }
311     else if (time.sec != 0) // update every 1sec
312     {
313 //      puts_COM1("\n counter:");
314 //      put_d(initial);
315       time.sec = 0;
316       idle_diff = idle_count - idle_count_old;
317       idle_count_old = idle_count;
318       if (initial>=15)  com_display();
319     }
320     else // low priority events
```

```
321     {
322       idle_count++;
323     }
324
325   } /* end while quit == 0 */
326
327 //  DISABLE_PWM();
328   EvaRegs.T1CON.bit.TENABLE = 0;
329   EvaRegs.ACTRA.all = 0x0000;
330   DINT;
331 } /* end main */
332
333
334 /* ====================================================================
335 __Local_Functions()
336 ====================================================================== */
337
338 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
339 /**
340 Display operating information out COM0.
341
342 \author A.McIver
343 \par History:
344 \li  22/06/05 AM - initial creation
345
346 \param[in] mode Select whether to start a new display option
347 */
348 void com_display(void)
349 {
350   Uint16
351     status;
352   puts_COM1("\n");
353   //If system is displaying grab data do nothing otherwise display normal status stuff
354   if(GrabShowTrigger()){
355   }
356   else
357     {
358       status = vsi_get_status();
359       if (status == VSI_FAULT)
360       {
361         putc_COM1('F');
362         putxx(vsi_get_faults());
363       }
364       else
365       {
366         if (status==0)
367         puts_COM1("  Init ");
368         else if (status==1)
369         puts_COM1("  Gate Charge ");
370         else if (status==2)
371         puts_COM1("  Ramp ");
372         else if (status==3)
373         puts_COM1("  Run ");
374         else if (status==4)
375         puts_COM1("  Settled ");
376         else if (status==5)
377         puts_COM1("  Idle ");
378         else if (status==6)
379         puts_COM1("  FAULT ");
380         else putxx(status);
381       }
382     puts_COM1("VSI: ");
383     #ifdef OL_VSI
384     puts_COM1("OL ");
385     puts_COM1(" Mod Depth: ");
386     putdbl(mod_serial,2);
387     #endif
388     #ifdef CL_VSI
389     puts_COM1("CL ");
390     puts_COM1(" Iref Mag: ");
391     putdbl(Imag_serial,2);
392     #endif
393     if (load_enable==1)    puts_COM1(" Load ON ");
394     else                   puts_COM1(" Load OFF ");
395     }
396 } /* end com_display */
397
398 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
399 /* void com_keyboard
400 Parameters: none
```

```
401 Returns: nothing
402 Description: Process characters from COM0.
403 Notes:
404 History:
405   22/06/05 AM - initial creation
406 \li 27/11/07 PM - added in testing of the digital I/O
407 */
408 void com_keyboard(void)
409 {
410
411   char c;
412 //  int temp=0;
413
414 //  puts_COM1("KEY");
415   if (kbhit_COM1())
416   {
417     c = getc_COM1();
418     switch (c)
419     {
420
421       case 'e': vsi_enable();
422                 puts_COM1("e");
423                 break;
424       case 'd': vsi_disable();
425                 puts_COM1("d");
426                 break;
427
428       #ifdef OL_VSI
429       //step change in VSI Modulation depth
430       case 'M': if (mod_serial < (2.0-VSI_MODSTEP))  mod_serial+=VSI_MODSTEP;
431                 else                                mod_serial=2.0;
432                 vsi_set_mod(mod_serial);
433                 break;
434
435       case 'm': if (mod_serial > VSI_MODSTEP)  mod_serial-=VSI_MODSTEP;
436                 else                          mod_serial=0.0;
437                 vsi_set_mod(mod_serial);
438                 break;
439       #endif
440
441       #ifdef CL_VSI
442       //step change in VSI Current Reg Reference
443       case 'A': if (Imag_serial< (MAX_CURR-VSI_CURRSTEP))  Imag_serial+=VSI_CURRSTEP;
444                 else                                      Imag_serial=MAX_CURR;
445                 vsi_set_Iref_mag(Imag_serial);
446                 break;
447
448       case 'a': if (Imag_serial > VSI_CURRSTEP)  Imag_serial-=VSI_CURRSTEP;
449                 else                            Imag_serial=0.0;
450                 vsi_set_Iref_mag(Imag_serial);
451                 break;
452       #endif
453
454       //Load step
455       case 'l': load_enable=0;  //Load off
456                 break;
457
458       case 'L': load_enable=1;  //load on
459                 break;
460
461       case 'H': // write help info
462         initial=0;
463       break;
464
465 #ifdef GRAB_INCLUDE
466       case 'g': /* grab interrupt data */
467         GrabClear();
468         GrabStart();
469         GrabRun();
470       break;
471       case 'h':
472         puts_COM1("\n\nGrab Display\nIndex\n");
473         GrabShow();
474       break;
475       case 'c': /* stop grab display */
476         GrabClear();
477       break;
478 #endif
479       default: break;
480     }
```

```
481   }
482 } /* end com_keyboard */
483
484
485 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
486 /**
487 1 second CPU timer interrupt.
488
489 \author A.McIver
490 \par History:
491 \li  22/06/05 AM - initial creation (derived from k:startup.c)
492 */
493 #ifndef BUILD_RAM
494 #pragma CODE_SECTION(isr_cpu_timer0, "ramfuncs");
495 #endif
496 interrupt void isr_cpu_timer0(void)
497 {
498   static struct
499   {
500     Uint16
501       msec,
502       msec10,
503       msec100,
504       sec;
505   } i_count =
506   {
507     0, 0, 0
508   };
509
510   /*for (ii=0; ii<WD_TIMER_MAX; ii++)
511   {
512     if (wd_timer[ii] > 0)
513       wd_timer[ii]--;
514   }*/
515   i_count.msec++;
516   if (i_count.msec >= 10)
517   {
518     i_count.msec = 0;
519     i_count.msec10++;
520     if (i_count.msec10 >= 10)
521     {
522       i_count.msec10 = 0;
523       i_count.msec100++;
524       if (i_count.msec100 >= 10)
525       {
526         i_count.msec100 = 0;
527         time.sec = 1;
528       }
529       time.sec0_1 = 1;
530     }
531     time.msec10 = 1;
532   }
533   time.msec = 1;
534
535   // Acknowledge this interrupt to receive more interrupts from group 1
536   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
537 } /* end isr_cpu_timer0 */
538
539
540 /* ======================================================================
541 __Exported_Functions()
542 ========================================================================= */
543
544
545 /* ======================================================================
546 __Grab_Functions()
547 ========================================================================= */
548 #ifdef GRAB_INCLUDE
549
550 void GrabInit(void)
551 {
552   Uint16
553     i,j;
554
555   for (i=0; i<GRAB_LENGTH; i++)
556   {
557     for (j=0; j<GRAB_WIDTH; j++)
558     {
559       grab_array[i][j] = 0;
560     }
```

```
561   }
562   GrabClear();
563 }
564
565 /* call with index == 0xFFFF for title line
566 else index = 0..GRAB_LENGTH-1 for data */
567 void GrabDisplay(int16 index)
568 {
569   Uint16
570     i;
571
572   if (index == 0xFFFF)
573   {
574     puts_COM1("\nindex");
575     for (i=0; i<GRAB_WIDTH; i++)
576     {
577       puts_COM1("\tg");
578       put_d(i);
579     }
580   }
581   else
582   {
583     put_d(index);
584     for (i=0; i<GRAB_WIDTH; i++)
585     {
586       putc_COM1('\t');
587       putdbl(grab_array[index][i],3);
588     }
589   }
590   puts_COM1("\n");
591 }
592
593 #endif
594 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

```
1 /**
2 \file
3 \brief VSI definitions
4
5 \par Developed By:
6   Creative Power Technologies, (C) Copyright 2009
7 \author A.McIver
8 \par History:
9 \li  23/04/09 AM - initial creation
10 \ Modified Dinesh Segaran
11 \   11/11/09 DS - Turning this into a GIIB-Based Bidirectional DC-DC Converter
12 \    26/08/10 DS - Fixed Point implementation of the Adaptive Controlled
13 \                 Bidirectional DC-DC Converter
14 \    15/03/11 DS - Grid Connected H-Bridge, with DC links supplied by a
15 \                 Bidirectional DC-DC Converter
16 \    14/04/11 DS - Hbridge load for bidirectional dc-dc converter
17 */
18
19 /* ======================================================================
20 ___Includes()
21 ====================================================================== */
22
23
24 /* ======================================================================
25 __Macros()
26 ======================================================================== */
27
28 #define SW_ENABLE()      {\
29                           CPLD.EVACOMCON.bit.ENA = 1;\
30                           cpld_write(ADD_EVACOMCON,CPLD.EVACOMCON.all);\
31                         }
32
33 #define SW_DISABLE()    {\
34                           CPLD.EVACOMCON.bit.ENA = 0;\
35                           cpld_write(ADD_EVACOMCON,CPLD.EVACOMCON.all);\
36                         }
37
38 // Disable VSI switching
39 #define VSI_DISABLE()    {\
40                           EvaRegs.ACTRA.all = 0x0000;\
41                         }
42
43 // Enable VSI switching
44 #define VSI_ENABLE()    {\
45                           EvaRegs.ACTRA.all = 0x660;\
46                           CPLD.EVACOMCON.bit.ENA = 1;\
47                           cpld_write(ADD_EVACOMCON,CPLD.EVACOMCON.all);\
48                         }  //single phase only
49
50   //  output pin 1 CMPR1 - active high
51   //  output pin 2 CMPR1 - active low
52   //  output pin 3 CMPR2 - active low
53   //  output pin 4 CMPR2 - active high
54   //  output pin 5 CMPR3 - active high
55   //  output pin 6 CMPR3 - active low    =>0000 0110 1001 0110
56
57 /// Turn low side devices on full for charge pump starting
58 #define VSI_GATE_CHARGE()  EvaRegs.ACTRA.all = 0x0000
59
60 #define SIN_TABLE_READ(PHASE,SIN_VAL){\
61         SIN_VAL = (int16)sin_table[(((Uint16)PHASE>>6)|0x0001)];\
62         VAL_DIFF = (sin_table[(((Uint16)PHASE>>6)|0x0001)+2]) - SIN_VAL;\
63         SIN_VAL += (int16)( ((int32)((Uint16)PHASE&0x007F)*(int32)VAL_DIFF)>>6 );}
64 // phase is a 16bit number, but the index is only 10 (513 values). The whole sine wave is represented in 16bits (0->2^32),
65 // shift right by 6 to know where to aim in the sine table. interpolate using the last 7 bits.
66
67
68 /* ======================================================================
69 ___Hash_Definitions()
70 ====================================================================== */
71 //For Fixed Point
72 #define FIXED_Q    11
73 #define FIXED_Q_SCALE (long)2048
74
75 /** @name VSI Status bit definitions */
76 //@{
77 #define VSI_INIT        0x0000
78 #define VSI_GATECHARGE  0x0001  ///< VSI is running
79 #define VSI_RAMP        0x0002  ///< VSI is running
80 #define VSI_RUNNING      0x0003  ///< VSI is running
```

```
 81 #define VSI_SETTLED      0x0004  ///< set when target reached
 82 #define VSI_STOP        0x0005  ///< VSI is running
 83 #define VSI_FAULT        0x0006  ///< set when fault present in VSI system
 84 //@}
 85
 86 /** @name Fault Codes */
 87 //@{
 88 #define FAULT_VSI_IAC_OL    0x0001
 89 #define FAULT_VSI_IAC_OC    0x0002
 90 #define FAULT_VSI_VDC_OV    0x0004
 91 #define FAULT_VSI_VDC_UV    0x0008
 92 #define FAULT_VSI_PDPINT    0x0010
 93 #define FAULT_VSI_SPI        0x0020
 94 //@}
 95
 96 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
 97 /* Zero crossing states */
 98 #define ZX_LOST      0 ///< No idea of anything
 99 #define ZX_EST      1 ///< Initial fundamental frequency estimation
100 #define ZX_SYNC      2 ///< nudges the phase to stay synchronised
101 #define ZX_FREQ      3 ///< nudges the freq (phase_step) for persistent err
102 #define ZX_LOCK      4 ///< tests to see if system is locked into sync
103 #define ZX_MISC      5 ///< load levelling calculation state
104
105 /* Zero crossing constants */
106 /* Sync lost if no ZX in ~3.5 cycles */
107 #define ZX_MAX_COUNT  ((Uint16)(3.5*FSAMPLE_BIDC/F_FUND)) // 1050
108 #define ZX_CYCLE_AVG  64   /* Number of cycles for frequency estimate */
109 #define ZX_SYNC_LIMIT  10   /* Number of cycles in sync */
110 #define ZX_BIG_ERR    (400*65536)   /* ~2.2 degrees */
111 #define ZX_PHASE_ERR  (3600*65536) // ~20 degrees - maximum sync phase error
112 #define ZX_FREQ_ERR    (100*65536)   // Persistent phase error for freq change
113 #define ZX_FREQ_ERR_BIG  (200*65536)   // Persistent phase error for freq change
114 #define ZX_OFFSET_POS  (-4500*65536) // trim phase for +ve phase seq
115 #define ZX_OFFSET_NEG  (6700*65536) // trim phase for -ve phase seq
116
117 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
118
119 //Topology parameters
120 //{
121 #define C                  20e-6  //20uF
122 #define L                  132e-6
123 #define R_L                0.1
124 #define R_L_2              R_L*R_L
125 #define LVSI              (16e-3)
126 #define OMEGA_BIDC_L      (OMEGA_BIDC*L)
127 #define OMEGA_BIDC_L_2    (OMEGA_BIDC_L*OMEGA_BIDC_L)
128 #define NPRI              10.0
129 #define NSEC              11.0
130 #define NPRI_NSEC          ((double)(NPRI/NSEC))
131 #define VIN                200.0
132 #define VDCPRI            (VIN/2.0)
133 #define VDC_VSI            200.0
134 #define VBUS_NOM_FIXED    ((int32)(VDC_VSI*FIXED_Q_SCALE))
135 #define MAX_CURR          (12.0)
136 #define MAX_CURR_FIXED    ((long)(MAX_CURR*FIXED_Q_SCALE))
137 //}
138
139
140 //VSI Parameters
141 //{
142 #define  SW_FREQ_VSI       5000.0
143 #define  PERIOD_VSI         ((Uint16)(((double)HSPCLK/SW_FREQ_VSI)/2.0))
144 #define  PERIOD_2_VSI      (PERIOD_VSI>>1)        // Carrier timer half period in clock ticks
145 #define FSAMPLE_VSI        (SW_FREQ_VSI*2.0)
146 #define TSAMPLE_VSI        (1.0/FSAMPLE_VSI)
147 #define T_DELAY_VSI        (1.5*TSAMPLE_VSI)
148 #define  F_FUND_MAX        60.0
149 #define F_FUND            50.0
150 #define  F_FUND_MIN        40.0
151 #define OMEGA_FUND        (2*PI*F_FUND)
152 #define OMEGA_C_VSI        (PI_2-(40*DEG_TO_RAD))/(T_DELAY_VSI)    //40 deg phase margin
153 #define KP_CONST          ((int32)(LVSI*OMEGA_C_VSI*FIXED_Q_SCALE))
154 //the phase step is the difference in phase between two switching cycles.
155 //That is a 50Hz sin wave, switched at 5kHz, sampled at 10kHz. so the switching is 10kHz/50Hz faster.
156 //the switching is therefore 200x faster than the fundamental. so the phase step is 360 degrees/200.
157 //so in each switching cycle, the phase has advanced by 360*VSI_SW_FREQ/f_fund (in degrees)
158 /* the phase is scaled so that one fundamental is 2^32 counts. */
159 //#define PHASE_STEP          (Uint16)(65536.0*F_FUND/SW_FREQ_VSI/2.0)
160 #define PHASE_STEP          (Uint32)(4294967296.0*(double)F_FUND/(double)SW_FREQ_VSI/2.0)
```

```
161 #define KP_VSI              (OMEGA_C_VSI*LVSI/VDC_VSI)
162 #define KI_VSI              (OMEGA_C_VSI/10/FSAMPLE_VSI)
163 //}
164
165 /// Maximum VSI switching time in clock ticks
166 #define MIN_VSI_TIME        1e-6
167 #define MIN_VSI_COUNT        (HSPCLK*MIN_VSI_TIME)
168 #define MAX_VSI_TIME        (int16)(PERIOD_2_VSI-MIN_VSI_COUNT)
169
170 //constants
171 //{
172 #define SQRT3_ON2      FIXED_Q_SCALE*(0.866025403784439)               // 65536*sqrt(3)/2
173 #define INV_SQRT3      FIXED_Q_SCALE*(0.577350269189626)               // 65536/sqrt(3)
174 #define PI            3.14159265358979
175 #define _2PI          2*PI
176 #define PI_2          1.57079632679489
177 #define INV_PI        0.31830988618379
178 #define INV2_PI        0.636619772367581
179 #define  PI_FIXED      (long)(PI*FIXED_Q_SCALE)
180 #define DEG_TO_COUNT  ((double)(3750.0/180.0))
181 //}
182
183 //DAC hash defines
184 //{
185 #define DAC_SCALE_VREF    2048.0/50.0
186 #define DAC_SCALE_PHASE    2048.0/100.0
187 #define DAC_SCALE_IREF    ((long)((FIXED_Q_SCALE*2.0)/(MAX_CURR*GAIN_OFFSET_CURRENT)))  //scaled by FIXED_Q
188 #define DAC_SCALE_VA      ((long)(FIXED_Q_SCALE/(double)PERIOD_2_VSI))
189 //}
190
191 //sine table hash definitions
192 //{
193 #define COUNT_TO_SINTABLE (32768.0/PERIOD_BIDC)
194 #define COUNT_TO_RAD      PI/3750.0
195 #define RAD_TO_COUNT      3750.0/PI
196 #define DEG_TO_RAD        PI/180.0
197 //}
198
199 /*****************
200 _Controller_form()
201 *****************/
202 //#define OL_VSI
203 #define CL_VSI
204
205 #ifdef OL_VSI
206 #undef CL_VSI
207 #endif
208 #ifdef CL_VSI
209 #undef OL_VSI
210 #endif
211
212 /****************
213 _ADC_Scaling()
214 ****************/
215 /// ADC calibration time
216 #define ADC_CAL_TIME      1// seconds
217 #define ADC_COUNT_CAL      (Uint16)(ADC_CAL_TIME * 20000 * 2.0)
218
219 /// DC averaging time
220 #define ADC_DC_TIME        0.1 // seconds
221 #define ADC_COUNT_DC      (Uint16)(ADC_DC_TIME * 20000 * 2.0)
222 #define  ADC_REAL_SC        1
223
224 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
225 /// RMS scaling
226 #define ADC_RMS_PS      4
227
228 //DA2810 Scaling - 3V and 12 bits
229 //easier to multiply result by 3 and shift back by 12.
230 #define ADC_DA_SCALE_MULT    (long)3     //3.0/4096.0 - scaled by FIXED_Q+5 cos num is so small
231 #define ADC_DA_SCALE_SHIFT  12
232
233 #define ADC_DA_SHIFT        4
234
235 //GIIB Scaling Resistors
236 #define RFB_GIIB_VAC         (long)10000                        //10000.0  //feedback resistor on GIIB board
237 #define RIN_GIIB_VAC         (long)(150000+150000+150000)           //150000.0+150000.0+150000.0  //preloaded input resistor on GIIB board
238 #define RFB_GIIB_VDC         (long)10000                        //10000.0  //feedback resistor on GIIB board
239 #define RIN_GIIB_VDC         (long)(150000+180000+180000)  //50000.0+180000.0+180000.0  //preloaded input resistor on GIIB board
240
```

```
241 //AC Voltage Inputs
242 //GIIB Scaling
243 //#define RIN_GIIB_ADD_VAC   (long)0 //NO additional scaling resistor on GIIB board
244 #define RIN_GIIB_TOTAL_VAC   RIN_GIIB_VAC        //(RIN_GIIB_ADD_VAC*((RIN_GIIB_VAC<<FIXED_Q)/(RIN_GIIB_ADD_VAC+RIN_GIIB_VAC)))>>FIXED_Q
245 #define VAC_GIIB_GAIN       (long)((-1.0*(double)RFB_GIIB_VAC*FIXED_Q_SCALE)/(double)RIN_GIIB_TOTAL_VAC)  //scaled by FIXED_Q
246 #define VAC_GIIB_GAIN_INV   (long)(((double)FIXED_Q_SCALE*(double)FIXED_Q_SCALE)/(double)VAC_GIIB_GAIN)  //scaled by FIXED_Q
247
248 //DC Voltage Inputs
249 //GIIB Scaling
250 //#define RIN_GIIB_ADD_VDC     (long)150000       //NO additional scaling resistor on GIIB board
251 #define RIN_GIIB_TOTAL_VDC   RIN_GIIB_VDC
252 #define VDC_GIIB_GAIN       ((-1.0*(double)RFB_GIIB_VDC)/(double)RIN_GIIB_TOTAL_VDC)  //scaled by FIXED_Q
253 #define VDC_GIIB_GAIN_INV   (long)(FIXED_Q_SCALE/VDC_GIIB_GAIN)    //scaled by 2^9
254
255 //Mini2810 Scaling Resistors
256 #define RUP_MINI1            (long)6800
257 #define RUP_MINI2            (long)4700
258 #define RUP_MINI_TOTAL      (long)((RUP_MINI1*RUP_MINI2)/(RUP_MINI1+RUP_MINI2))
259 #define RDWN_MINI            (long)6800
260 #define RIN_MINI            (long)12000
261 #define RDOWN_MINI_TOTAL    (long)((RDWN_MINI*RIN_MINI)/(RDWN_MINI+RIN_MINI))
262
263 //Mini2810 ADC Scaling
264 #define ADC_MINI_GAIN        (((double)(RUP_MINI_TOTAL*RDOWN_MINI_TOTAL))/((double)((RUP_MINI_TOTAL+RDOWN_MINI_TOTAL)*RIN_MINI))) //is a double
265 #define ADC_MINI_GAIN_INV    (long)(FIXED_Q_SCALE/ADC_MINI_GAIN)  //scaled by FIXED_Q
266
267 #define MINI_LEVEL_SHIFT     (long)(((double)RDOWN_MINI_TOTAL*2.5*FIXED_Q_SCALE)/((double)(RUP_MINI_TOTAL+RDOWN_MINI_TOTAL)))    //scaled by FIXED_Q
268 #define ADC_OFFSET          (long)(((MINI_LEVEL_SHIFT<<ADC_DA_SCALE_SHIFT)>>FIXED_Q)/ADC_DA_SCALE_MULT) //in counts
269
270 //Voltage Overall Gain
271 #define VDC_ANALOG_GAIN     ((long)((double)((double)VDC_GIIB_GAIN_INV*(double)ADC_MINI_GAIN_INV*(double)ADC_DA_SCALE_MULT)/(double)FIXED_Q_SCALE/(double)4096))
272 #define VAC_ANALOG_GAIN     ((long)((double)((double)VAC_GIIB_GAIN_INV*(double)ADC_MINI_GAIN_INV*(double)ADC_DA_SCALE_MULT)/(double)FIXED_Q_SCALE/(double)4096))
273
274 //Current Inputs
275 //LEM Scaling
276 #define CT_RATIO                  2000.0        //For LA 100P SP13, it is 1000, for LA 100P - 2000
277 #define CT_TURNS                  2.0
278 #define BURDEN_R                 380.0         //Burden resistor
279 #define LEM_GAIN                 ((CT_TURNS*BURDEN_R)/CT_RATIO)
280 #define LEM_GAIN_INV             (1.0/LEM_GAIN)    //is a double
281
282 //GIIB Scaling
283 #define RIN1_GIIB_I              10000.0        //Input resistor to GIIB op amp stage
284 #define RIN2_GIIB_I              10000.0        //Input resistor to GIIB op amp stage
285 #define RIN_GIIB_TOTAL_I        ((RIN1_GIIB_I*RIN2_GIIB_I)/(RIN1_GIIB_I+RIN2_GIIB_I))        //Input resistor to GIIB op amp stage
286 #define RFB_GIIB_I              10000.0
287 #define I_GIIB_GAIN              (-1.0*RFB_GIIB_I/RIN_GIIB_TOTAL_I)      //Voltage gain of amplifier on GIIB for current (double)
288 #define I_GIIB_GAIN_INV          (1.0/I_GIIB_GAIN)                       //Voltage gain of amplifier on GIIB for current (double)
289
290 #define I_ANALOG_GAIN            ((long)(LEM_GAIN_INV*I_GIIB_GAIN_INV*(double)ADC_MINI_GAIN_INV*(double)ADC_DA_SCALE_MULT)>>ADC_DA_SCALE_SHIFT)
291 //load current scaling
292
293 #define GAIN_OFFSET_CURRENT      1.0//1.08  //this is to account for the differences in scaling resistors
294
295 //Scaling for reading VGEN - takes a +/-10V signal
296
297
298 /*End ADC Scaling*/
299
300 /* Step size of modulation depth */
301 #define MAG_SMALL_STEP  0.01
302 #define MAG_LARGE_STEP  0.1
303
304 // Step size and max output voltage
305 #define VREF_MAX 205//101
306 #define VREF_MIN 10
307 #define VREF_STEP_S 1
308 #define VREF_STEP_L  10
309
310 /* ======================================================================
311 __Exported_Variables()
312 ====================================================================== */
313
314 typedef long long signed int  int64;
315
316 /* ======================================================================
317 __Global Variables()
318 ====================================================================== */
319
320 /* ======================================================================
```

```
321 __Function_Prototypes()
322 ========================================================================= */
323
324 /// Core interrupt initialisation
325 void vsi_init(void);
326
327 /// Core interrupt VSI state machine for background processing
328 void vsi_state_machine(void);
329
330 /// Enables vsi switching (assuming no faults)
331 void vsi_enable(void);
332
333
334 /// Disable vsi switching
335 void vsi_disable(void);
336
337 ///// Set the target output frequency in Hz
338 Uint16 vsi_set_fund(double f);
339
340 // Set the target output modulation depth
341 void vsi_set_mod(double mod_serial);
342
343 //Set the target output current magnitude
344 void vsi_set_Iref_mag(double mag_serial);
345
346 // Set the desired output voltage
347 void vsi_set_vref(int16 vref);
348
349 /// Returns the status of the VSI
350 Uint16 vsi_get_status(void);
351
352 /// Report what faults are present in the VSI
353 Uint16 vsi_get_faults(void);
354
355 /// Clear some detected faults and re-check.
356 void vsi_clear_faults(void);
357
358 //  Print the current state of the state machine
359 void get_state(void);
360
361 // Calibrate ADCs online
362 void calibrate_adc(void);
363
364 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

```
 1 /**
 2 \file
 3 \brief VSI Interrupt Service Routine
 4
 5 This file contains the code for the core interrupt routine for the CVT system.
 6 This interrupt is the central system for the signal generation and
 7 measurement. The carrier timer for the VSI generation also triggers the
 8 internal ADC conversion at the peak of the carrier. The end of conversion then
 9 triggers this interrupt. Its tasks are:
10
11 - Read internal ADC results
12 - Perform internal analog averaging and RMS calculations
13 - Update VSI phase and switching times
14
15 \par Developed By:
16   Creative Power Technologies, (C) Copyright 2009
17 \author A.McIver
18 \par History:
19 \li  23/04/09 AM - initial creation
20 \ Modified Dinesh Segaran
21 \   11/11/09 DS - Turning this into a GIIB-Based Bidirectional DC-DC Converter
22 \    26/08/10 DS - Fixed Point implementation of the Adaptive Controlled
23 \                  Bidirectional DC-DC Converter
24 \    27/10/10 DS - Load Step Function for the Bidirectional DC-DC Converter
25 \    15/03/11 DS - Grid Connected H-Bridge, with DC links supplied by a
26 \                  Bidirectional DC-DC Converter
27 */
28
29 /************
30 CODE_TASKS()
31 ************/
32 /*
33   15/03/2011   -   Trying to implement a single phase VSI on the
34                   E-10 Gate Driver Board (GDB), whose DC link is
35                   supplied by a Single Phase Bi-directional DC-DC
36                   Converter.
37                 ->  PWM to be generated on EVB,i.e CMPR4&5, and routed out
38                     through CPLD (Needs to be coded), and to the GDB.
39               -   First, run a 10kHz interrupt. open loop VSI
40   20/3/2011    -    Gate Drive Resistors - 27 ohms.
41   21/3/2011    -    Scaling Resistors:
42                 ->  AC Voltage - Standard scaling to read +/- 450VAC
43                 ->  DC Voltage - Scaled to read 510Vdc
44                                - Scaled for 420Vdc trip -
45                 ->  AC Current - Scaled for +/- 15A - 2 turns - Burden Resistor - 270ohm
46                 ->  DC Current - Scaled for +/- 15A - 2 turns - Burden Resistor - 270ohm
47
48   6/4/2011    - ADCs fully tested
49               - Grid Synch code included (not yet working)
50               - phase now a 32-bit number
51               - Change in strategy:
52                 -> PWM for H-bridge comes from PWMA
53                 -> PSSW for Sec Bridge comes from PWMB
54                 -> 2 lines sent to Master Bridge: 1)  Synch Pulse
55                                                   2)  Fault trigger
56   8/4/2011    - Unable to use CPLD to route gate signals, because it will use up capture port
57               - Instead, use hysteresis inputs to route gate drives for Sec Side
58               - To synch, still use two lines:   1) Synch Pulse
59                                                 2) Enable/Disable
60               - Rising Edge of Enable = start switching
61               - Falling Edge of Enable= stop switching (used for emergency stop as well
62               - Synchronisation - Cap2 in use. Input leaves the Secondary on GPIOB4 (DIGOUT5), comes through DIGIN8.
63                 On the DIGIO header, leaves Sec on pin 5, and comes in on pin 16.
64               - Next, Enable. Cap1 in use. Input leaves secondary on GPIOB5 (DIGOUT6), comes through DIGIN7.
65                 On the DIGIO header, leaves Sec on pin 6, and comes in on pin 15.
66
67 PORTED OVER TO OPEN GIIB!!!!!
68   For this experiment I want a current regulated H bridge running from a bidirectional that supplies 200V
69   13/4/2011    -  Open Loop H-bridge running at 200V operational
70               - DC voltage measurements set at +510V - no scaling resistors
71               - AC current measurements set at 95 ohms - 2 turns - so +/- 20A
72     21/4/2011  - Closed Loop H-bridge and a closed loop bidirectional DC-DC converter work.
73               - Need to implement feed-forward compensation. For this, need to synch switching and send mod depth info across.
74               - Stage 1:  -  Synchronise Carriers. use zaki's code.
75                           -  Synchronise the VSI to the BiDC because the BiDC uses a lot of DIGIO pins already.
76                           - Use the shielded ribbon cable for this. Build Loopback function and test.
77                           - Loopback cable - GPIOB0-4 (PWMB1-4) are routed back into DIGIN5-8. so Pins 1-4 are connected to 13-16.
78                           -  On the BiDC, send out a synch pulse at 5kHz (1 every 8 interrupts) on GPIOB4.
79                             This is DIGOUT5, pin 5 on the 20-pin header.
80                           - On the VSI, bring the synch pulse into CAP2. this is on DIGIN8, which is pin 16. ie connect pins 5 & 16.
```

```
81                          -  Also connect all the GNDs on the 20-pin header together. I.e, leave pins 18 & 20.
82                          - Disconnect VCC, i.e cut pins 17 & 19
83                          -  that lets you lift synch code from GridCon set, and  also the fault trigger when synch is lost.
84
85              - Stage 2:  -  Phase & Modulation depth information. Via SPI or via DAC?
86     22/4/2011  -  Bridges Synchronised. NEED TO ADD IN EMERGENCY STOP CODE
87              - able to send va over DAC. will be read in by Vgen. needs a mascon header.
88              -  uses the shielded ribbon, cable tied to the synch pulse cable. Connect the AGND as well. seems to work fine for now.
89 */
90 // compiler standard include files
91 #include <math.h>
92
93 // processor standard include files
94 #include <DSP281x_Device.h>
95
96 #ifdef COM0_CONSOLE
97 #include <bios0.h>
98 #endif
99 #ifdef COM1_CONSOLE
100 #include <bios1.h>
101 #endif
102
103
104 // board standard include files
105 #include <lib_mini2810.h>
106 #include <dac_ad56.h>
107 #include <lib_cpld.h>
108 #include <lib_giib.h>
109
110 // local include files
111 #include "main.h"
112 #include "conio.h"
113 #include "vsi_InvLoad.h"
114
115 /* ======================================================================
116 __Definitions()
117 ========================================================================= */
118
119 /// Boot ROM sine table size for VSI and DFT
120 #define ROM_TABLE_SIZE    512
121 /// Boot ROM sine table peak magnitude for VSI and DFT
122 #define ROM_TABLE_PEAK    16384
123 #define GRAB_INCLUDE
124
125
126 //
127 /* ======================================================================
128 __Types()
129 ========================================================================= */
130
131 /// Internal ADC channel type
132 /** This structure hold variables relating to a single ADC channel. These
133 variables are used for filtering, averaging, and scaling of this analog
134 quantity. */
135 typedef struct
136 {
137   int16
138     raw, ///< raw ADC result from last sampling
139     filt; ///< decaying average fast filter of raw data
140   int32
141     rms_sum, ///< interrupt level sum of data
142     rms_sum_bak, ///< background copy of sum for averaging
143     dc_sum, ///< interrupt level sum
144     dc_sum_bak; ///< background copy of sum for processing
145   double
146     real; ///< background averaged and scaled measurement
147 } type_adc_ch;
148
149 /// Internal ADC storage type
150 /** This structure holds all the analog channels and some related variables
151 for the averaging and other processing of the analog inputs. There are also
152 virtual channels for quantities directly calculated from the analog inputs.
153 The vout and iout channels are for DC measurements of the VSI outputs when it
154 is producing a DC output. */
155 typedef struct
156 {
157   Uint16
158     count_cal, ///< counter for low speed calibration summation
159     count_rms, ///< counter for full fund. period for RMS calculations
160     count_rms_bak, ///< background copy of RMS counter
```

```
161     count_dc, ///< counter for DC averaging
162     count_dc_bak, ///< background copy of DC counter
163     flag_cal, ///< flag set to trigger background calibration averaging
164     flag_rms, ///< flag set to trigger background RMS averaging
165     flag_dc; ///< flag set to trigger background DC averaging
166   type_adc_ch
167     A0, ///< ADC channel A0
168     A1, ///< ADC channel A1
169     A2, ///< ADC channel A2
170     A3, ///< ADC channel A3
171     A4, ///< ADC channel A4
172     A5, ///< ADC channel A5
173     A6,
174     B0, ///< ADC channel B0
175     B1, ///< ADC channel B1
176     B2, ///< ADC channel B2
177     B3, ///< ADC channel B3
178     B4, ///< ADC channel B4
179     B5, ///< ADC channel B5
180     yHA, ///< bank A high reference
181     yLA, ///< bank A low reference
182     yHB, ///< bank B high reference
183     yLB; ///< bank B low reference
184 } type_adc_int;
185
186 /** @name Internal ADC Variables */
187 //@{
188 type_adc_int
189   adc_int =
190   {
191     0, // count_cal
192     0, // count_rms
193     0, // count_rms_bak
194     0, // count_dc
195     0, // count_dc_bak
196     0, // flag_cal
197     0, // flag_rms
198     0, // flag_dc
199     {  0, // raw
200        0, // filt
201        0L, // rms_sum
202        0L, // rms_sum_bak
203        0L, // dc_sum
204        0L, // dc_sum_bak
205        0.0 // real
206       }, // #A0
207     { 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // #B0
208     { 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // yHA
209     { 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // yLA
210     { 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // yHB
211     { 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // yLB
212   };
213
214 // ADC calibration variables
215 int16
216   cal_gainA = 1<<14,    // calibration gain factor for A channel
217   cal_gainB = 1<<14,    // calibration gain factor for B channel
218   cal_offsetA = 0,    // calibration offset for A channel
219   cal_offsetB = 0;    // calibration offset for B channel
220 double
221   cal_gain_A, cal_gain_B,
222   cal_offset_A, cal_offset_B;
223
224
225 double
226   yHA = 0.0,
227   yLA,
228   yHB,
229   yLB;
230
231 /* =====================================================================
232 __Variables()
233 ======================================================================== */
234 // state machine level variables
235 Uint16
236   vsi_status = 0,      /// Status of VSI system
237   is_switching = 0,    // flag set if PWM switching is active
238 //  vsi_is_switching=0,
239 //  bidc_is_switching=0,
240   vsi_counter = 0,    // counter for timing VSI regulation events
```

```
241   dac_vref=0,
242   spi_fail_count,
243   dac_phaseref=0;  //FOR DAC
244
245 // Boot ROM sine table starts at 0x003FF000 and has 641 entries of 32 bit sine
246 // values making up one and a quarter periods (plus one entry). For 16 bit
247 // values, use just the high word of the 32 bit entry. Peak value is 0x40000000 (2^30)
248 // therefore 1 period is 512 entries, 120 degrees offset is 170.67 entries.
249 // sin table actually starts with an offset of 2, odd numbers only
250 // so first value is in sin_table[3]
251 // max value of 16bit sign table is 2^14 =16384
252
253 int16
254   *sin_table = (int16 *)0x003FF000, // pointer to sine table in boot ROM
255   *cos_table = (int16 *)0x003FF100, // pointer to cos table in boot ROM
256   mod_targ = 0,      // target modulation depth
257   mod_ref = 0,
258   init_table=0;
259
260 int32
261   cont_signal_scaled;
262
263 /// fault variables
264 Uint16
265   detected_faults = 0,
266   timer_synch_count = 100,
267   first=0;  // bits set for faults detected (possibly cleared)
268
269 //DAC Variable
270 Uint16  data_out;
271 int      i_spi;
272
273 /*****************
274 _Macro_Variables()
275 *****************/
276 //sin table read variables
277 Uint16   PHASE;
278 int16    SIN_VAL,
279         VAL_DIFF;        // interpolation temp variable
280
281 /*************************
282 _DSP_Emulation_Variables()
283 *************************/
284 int16      UF_VSI,
285           int_vsi_count,//to tell which interrupt to run in.
286           int_count=0;
287
288 /***************************
289 _VSI_Modulation_Variables()
290 ***************************/
291 int16   va,
292         max_time,
293         t_A,
294         t_B,
295         sin_val,
296         cos_val,
297         val_diff;
298
299 Uint32  vsiphase = 0,
300         prev_sin_val = 0,
301         ZX_vsiphase=0,
302         phase_step;
303
304 Uint16  V_Asat = 0,
305         V_Bsat = 0;
306
307 double    mod=0.0;
308
309 /*********************
310 _Grid_Synch_Variables()
311 *********************/
312 /** @name Zero Crossing Synch Variables */
313 //@{
314 Uint16
315   ZX_seen = 0, ///< flag set when a zx event is detected
316   in_sync = 0,  ///< Flag to indicate that sync is achieved
317   ZX_in_sync = 0,  ///< > ZX_SYNC_LIMIT means that sync has been achieved
318   ZX_state = ZX_LOST,  ///< State of the zero crossing synch process
319   ZX_count = 0,  ///< The number of switching cycles between ZX interrupts
320   ZX_count_grab, // for grab code only
```

```
321   ZX_cycles = 0,  ///< Count of number of ZXs during averaging
322   ZX_sum = 0;    ///< Running sum for average
323 Uint32
324   ZX_phase_step = PHASE_STEP;///< Change in phase angle in half a switching cycle
325
326 int16
327   ZX_time = 0;  ///< Time of captured ZX in timer units
328
329 int32
330   ZX_time_phase = 0L, ///< Time of captured ZX in phase units
331   zx_offset = ZX_OFFSET_POS, ///< variable offset for tuning
332   ZX_phase_scale = 0L, ///< Scale factor between timer and phase units
333   ZX_phase_err = 0L, ///< Difference in phase units (2^16 == 360deg)
334   ZX_err_sum = 0L;  ///< Integral for frequency control
335 //@}
336
337 /* =====================================================================
338 __Control_Loop_Variables()
339 ======================================================================== */
340 //Interface variables used to recieve controller loop parameters from background
341 //Controller loop turning parameters in real floating pointer number from background
342
343 /*********************
344 _VSI_Curreg_Variables()
345 *********************/
346 long  Iref_mag_fixed=0,
347       Iref_mag_fixed_timed=0,
348       Iref_fixed,
349       VSIerror_fixed,
350       Kp_VSI_fixed,
351       Ki_VSI_fixed,
352       VSIprop_fixed,
353       VSI_intnow_fixed,
354       VSI_int_fixed,
355       VSI_ctrl_fixed,
356       emf_scaled_fixed;
357
358 int16 vref_temp=0,
359       ref_volt=12;
360
361 /***************************
362 _ADC_Calibration_Variables()
363 ***************************/
364 int16  cal_count=0;
365
366 int32 I1_cal=0,
367       I2_cal=0,
368       I3_cal=0,
369       I4_cal=0,
370       I5_cal=0,
371       I6_cal=0,
372       Vdc1_cal=0,
373       Vdc2_cal=0,
374       Vdc3_cal=0,
375       Vdc4_cal=0,
376       Vac1_cal=0,
377       Vac2_cal=0,
378       Vac3_cal=0,
379       Vdc2_fixed,
380       Vdc1_fixed,
381       Vac1_fixed,
382       Vac2_fixed,
383       I3_fixed,
384       I4_fixed,
385       I1_fixed,
386       I2_fixed;
387
388 /*************
389 _ADC_Results()
390 *************/
391 int32  Vdc_fixed,
392       Vac_fixed,
393       IDC_fixed,
394       IACout_fixed;
395
396 /***************
397 _DAC_Variables()
398 ***************/
399 Uint16 dac_va=0;
400
```

```
401 /********************
402 _LoadStep_Variables()
403 *********************/
404 int16   load_enable=0,
405         prev_load_enable;
406
407 /* ====================================================================
408 __Local_Function_Prototypes()
409 ======================================================================= */
410
411 /* vsi state machine state functions */
412 void
413   st_vsi_init(void),    // initialises CFPP regulator
414   st_vsi_stop(void),    // waiting for start trigger
415   st_vsi_gate_charge(void), // delay to charge the high side gate drivers
416   st_vsi_ramp(void),    // ramping to target mod depth
417   st_vsi_run(void),    // maintaining target mod depth
418   st_vsi_fault(void);    // delay after faults are cleared
419
420 // ADC and VSI interrupt
421 interrupt void isr_adc(void);
422
423 //capture port interrupt
424 interrupt void isr_cap2(void);
425
426 // Gate fault (PDPINT) interrupt
427 interrupt void isr_gate_fault(void);
428
429 /* ==================================================================== */
430 /* State Machine Variable */
431 /* ==================================================================== */
432
433 type_state
434   vsi_state =
435   {
436     &st_vsi_init,
437     1
438   };
439
440 /* ====================================================================
441 __Exported_ADC_Functions()
442 ======================================================================= */
443
444 /**
445 This function initialises the ADC and VSI interrupt module. It sets the
446 internal ADC to sample the DA-2810 analog inputs and timer1 to generate a PWM
447 carrier and the event manager A to generate the VSI switching. It also
448 initialises all the relevant variables and sets up the interrupt service
449 routines.
450
451 This functions initialises the ADC unit to:
452 - Trigger a conversion sequence from timer 1 overflow
453 - Convert the appropriate ADC channels
454
455 Result registers as follows:
456 - ADCRESULT0 = ADCINA0
457 - ADCRESULT1 = ADCINB0
458 - ADCRESULT2 = ADCINA1
459 - ADCRESULT3 = ADCINB1
460 - ADCRESULT4 = ADCINA2
461 - ADCRESULT5 = ADCINB2
462 - ADCRESULT6 = ADCINA3
463 - ADCRESULT7 = ADCINB3
464 - ADCRESULT8 = ADCINA4
465 - ADCRESULT9 = ADCINB4
466 - ADCRESULT10 = ADCINA5
467 - ADCRESULT11 = ADCINB6
468 - ADCRESULT12 = ADCINA6 yHA
469 - ADCRESULT13 = ADCINB6  yHB
470 - ADCRESULT14 = ADCINA7 yLA
471 - ADCRESULT15 = ADCINB7 yLB
472
473 It initialises the Event Manager A unit to:
474 - drive PWM1-4 as PWM pins not GPIO
475 - a 0.48ns deadtime between the high and low side pins
476 - Timer 1 as an up/down counter for the PWM carrier
477
478 It initialises the PIE unit to:
479 - Take PDPINTA as a power stage interrupt
480 - Use the internal ADC completion interrupt to trigger the main ISR
```

```
481
482 \author A.McIver
483 \par History:
484 \li  12/10/07 AM - initial creation
485 \    26/08/10 DS - Fixed Point Bidirectional DC-DC Converter
486 */
487 void vsi_init(void)
488 {
489   //EVA
490   EvaRegs.ACTRA.all = 0x0000;
491   EvaRegs.GPTCONA.all = 0x0000;
492   EvaRegs.EVAIMRA.all = 0x0000;
493   EvaRegs.EVAIFRA.all = BIT0;
494   EvaRegs.COMCONA.all = 0x0000;
495
496 // Set up ISRs
497   EALLOW;
498   PieVectTable.ADCINT = &isr_adc;
499   PieVectTable.CAPINT2 = &isr_cap2;
500   PieVectTable.PDPINTA = &isr_gate_fault;
501   EDIS;
502
503   // Set up compare outputs
504   EALLOW;
505   GpioMuxRegs.GPDMUX.all = BIT0;
506   //EVA
507   GpioMuxRegs.GPAMUX.bit.PWM1_GPIOA0 = 1;  // enable PWM3 pin
508   GpioMuxRegs.GPAMUX.bit.PWM3_GPIOA2 = 1;  // enable PWM3 pin
509   GpioMuxRegs.GPAMUX.bit.PWM4_GPIOA3 = 1;  // enable PWM4 pin
510   GpioMuxRegs.GPAMUX.bit.PWM5_GPIOA4 = 1;  // enable PWM3 pin
511   GpioMuxRegs.GPAMUX.bit.PWM6_GPIOA5 = 1;  // enable PWM4 pin
512   EDIS;
513
514   //DEADBAND CONTROL
515   //EVA
516   EvaRegs.DBTCONA.bit.DBT = 5;      //1.0us deadtime
517   EvaRegs.DBTCONA.bit.EDBT1 = 1;
518   EvaRegs.DBTCONA.bit.EDBT2 = 1;
519   EvaRegs.DBTCONA.bit.EDBT3 = 1;
520   EvaRegs.DBTCONA.bit.DBTPS = 6;
521
522   //COMPARE REGISTERS
523   //EVA - Current Reg H-bridge
524   EvaRegs.CMPR2 = PERIOD_2_VSI;
525   EvaRegs.CMPR3 = PERIOD_2_VSI;
526
527   // Setup and load COMCON
528   //EVA
529   EvaRegs.COMCONA.bit.ACTRLD = 1;    // reload ACTR on underflow or period match
530   EvaRegs.COMCONA.bit.SVENABLE = 0;  // disable space vector PWM
531   EvaRegs.COMCONA.bit.CLD = 1;    // reload on underflow & period match
532   EvaRegs.COMCONA.bit.FCOMPOE = 1;  // full compare enable
533   EvaRegs.COMCONA.bit.CENABLE = 1;  // enable compare operation
534
535 // Set up Timer 1
536   EvaRegs.T1CON.all = 0x0000;
537   EvaRegs.T1PR = PERIOD_VSI;
538   EvaRegs.T1CMPR = 1;
539   EvaRegs.T1CNT = 0x0000;
540   EvaRegs.T1CON.bit.TMODE = 1;    // continous up/down count mode
541   EvaRegs.T1CON.bit.TPS = 0;      // input clock prescaler
542   EvaRegs.T1CON.bit.TCLD10 = 1;    // S.G. reload compare register on 0 or equals compare
543   EvaRegs.T1CON.bit.TECMPR = 1;    // enable time compare
544
545 // Set up Timer 2
546   EvaRegs.T2CON.all = 0x0000;
547   EvaRegs.T2PR = PERIOD_VSI<<1;
548   EvaRegs.T2CMPR = 1;
549   EvaRegs.T2CNT = 0x0000;
550   EvaRegs.T2CON.bit.TMODE = 2;    // continous up mode
551   EvaRegs.T2CON.bit.TPS = 0;      // input clock prescaler
552   EvaRegs.T2CON.bit.TCLD10 = 1;    //
553   EvaRegs.T2CON.bit.TECMPR = 0;    // disable time compare
554   EvaRegs.T2CON.bit.T2SWT1 = 1;  // Use TENABLE bit of GP Timer 1
555   EvaRegs.T2CON.bit.SET1PR = 0;    //use own period register
556
557 //Set up capture port 2
558   //Set up capture port
559   EvaRegs.CAPCONA.all = 0x0000;
560   EvaRegs.CAPFIFOA.all = 0x0000;
```

```
561
562   // Capture 2 gets Timer 1 on rising edge
563   EvaRegs.CAPCONA.bit.CAPRES = 1; // Release from reset
564   EvaRegs.CAPCONA.bit.CAP12TSEL = 0; //Select Timer 2
565   EvaRegs.CAPCONA.bit.CAP12EN = 1; // Enable captures 1 and 2
566   EvaRegs.CAPCONA.bit.CAP2EDGE = 1; // detects rising edge on Capture 2
567   //Keep an initial value in Capfifo register so that the first edge is indeed captured
568   EvaRegs.CAPFIFOA.bit.CAP2FIFO = 1;
569   GpioMuxRegs.GPAMUX.bit.CAP1Q1_GPIOA8 = 0;  //select GPIO
570   GpioMuxRegs.GPAMUX.bit.CAP2Q2_GPIOA9 = 1;  //select capture port 2  -  synch
571   GpioMuxRegs.GPAMUX.bit.CAP3QI1_GPIOA10 = 0;  //select GPIO
572   GpioMuxRegs.GPBMUX.bit.CAP4Q1_GPIOB8 = 0;  //select GPIO
573   GpioMuxRegs.GPBMUX.bit.CAP5Q2_GPIOB9 = 0;  //select GPIO
574   GpioMuxRegs.GPBMUX.bit.CAP6QI2_GPIOB10 = 0;  //select GPIO
575
576 // Set up ADC
577
578   // Setup and load GPTCONA
579   EvaRegs.GPTCONA.bit.T1TOADC = 3;    //0: no event starts ADC 1: UF starts ADC 2: period int flag starts ADC 3: Compare match starts ADC
580   //these are being done in A/B pairs
581
582   AdcRegs.ADCMAXCONV.all = 0x0007;    // Setup 8 conv's on SEQ1                                                    //To Oversample?
583   AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0x0;  // (A0/B0) - ADCRESULT0   -   ADCINA0 - APOT1/I3  - SW_A - default I3    -
584                                           //                    1        ADCINB0 - VDC2                            - VDC
585   AdcRegs.ADCCHSELSEQ1.bit.CONV01 = 0x1;  // (A1/B1) - ADCRESULT2   -   ADCINA1 - Vdc3/Vac3 - SW_A - default Vac3  -
586                                           //                    3        ADCINB1 - I5                              -
587   AdcRegs.ADCCHSELSEQ1.bit.CONV02 = 0x2;  // (A2/B2) - ADCRESULT4   -   ADCINA2 - I1                             - IAC OUT
588                                           //                    5        ADCINB2 - I4                              -
589   AdcRegs.ADCCHSELSEQ1.bit.CONV03 = 0x3;  // (A3/B3) - ADCRESULT6    -   ADCINA3 - Vac1                           -  ZX
590                                           //                    7        ADCINB3 - VDC1                            - VDC
591   AdcRegs.ADCCHSELSEQ2.bit.CONV04 = 0x4;  // (A4/B4) - ADCRESULT8   -   ADCINA4 - I2                             -
592                                           //                    9        ADCINB4 - APOT2/I6  - SW_B - default I6    -
593   AdcRegs.ADCCHSELSEQ2.bit.CONV05 = 0x5;  // (A5/B5) - ADCRESULT10  -   ADCINA5 - Vac2                           -
594                                           //                    11      ADCINB5 - Vgen/Vdc4 - SW_B - default Vdc4  -
595   AdcRegs.ADCCHSELSEQ2.bit.CONV06 = 0x6;  // (A6/B6) - ADCRESULT12  -   ADCINA6 - 2.5V ref
596                                           //                    13      ADCINB6 - 2.5V ref
597   AdcRegs.ADCCHSELSEQ2.bit.CONV07 = 0x7;  // (A7/B7) - ADCRESULT14  -   ADCINA7 - 1.25V ref
598                                           //                    15      ADCINB7 - 1.25V ref
599
600   AdcRegs.ADCTRL1.bit.ACQ_PS = 1;      // lengthen acq window size
601   AdcRegs.ADCTRL1.bit.SEQ_CASC = 1;    // cascaded sequencer mode
602   AdcRegs.ADCTRL2.bit.EVA_SOC_SEQ1 = 1;  // EVA manager start - enabled
603   AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = 1;  // interrupt enable
604   AdcRegs.ADCTRL2.bit.INT_MOD_SEQ1 = 0;  // int at end of every SEQ1
605   AdcRegs.ADCTRL3.bit.SMODE_SEL = 1;    // simultaneous sampling mode
606   AdcRegs.ADCTRL3.bit.ADCCLKPS = 0x04;  // ADCLK = HSPCLK/8 (9.375MHz)
607
608 // Enable interrupts
609   DINT;
610   EvaRegs.EVAIMRA.all = 0;        // disable all interrupts
611   // Enable PDPINTA: clear PDPINT flag,
612   EvaRegs.EVAIFRA.all = BIT0;
613   EvaRegs.EVAIMRA.bit.PDPINTA = 1;
614
615   //Capture port interrupts
616   EvaRegs.EVAIMRC.all = 0;        //Disable all capture port interrupt
617   EvaRegs.EVAIFRC.all = 0;        //Clearing interrupt flag for capture port
618   EvaRegs.EVAIMRC.bit.CAP2INT = 1;    //Enabling capture port 2 interrupt
619
620   // Enable PDPINTA in PIE: Group 1 interrupt 1
621   PieCtrlRegs.PIEIER1.bit.INTx1 = 1;
622   // Enable ADC interrupt in PIE: Group 1 interrupt 6
623   PieCtrlRegs.PIEIER1.bit.INTx6 = 1;
624   //Enable CAP2INT in PIE: Group 3 Int6
625   PieCtrlRegs.PIEIER3.bit.INTx6 = 1;
626
627   IER |= M_INT1; // Enable CPU Interrupts 1
628   IER |= M_INT3; // Enable CPU Interrupts 3
629   EINT;
630
631   AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1; // clear interrupt flag from ADC
632   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;   // Acknowledge interrupt to PIE Group 1 : PDPINT, ADC
633   PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;   // Acknowledge interrupt to PIE Group 2 : CAPINT2
634 /******************
635 _CONTROLLER_INITIALISATIONS()
636 ******************/
637   //VSI initialisations
638   max_time = MAX_VSI_TIME;
639   phase_step = PHASE_STEP;
640   //determine gains
```

```
641   Kp_VSI_fixed=(int32)(KP_VSI*FIXED_Q_SCALE);
642   Ki_VSI_fixed=(int32)(KI_VSI*FIXED_Q_SCALE);
643
644   DINT;
645   EvaRegs.T1CON.bit.TENABLE = 1;    // enable timer1 &2
646 //  EvaRegs.T2CON.bit.TENABLE = 1;    // enable timer2
647
648   EINT;
649   // Initialise state machine
650   vsi_state.first = 1;
651   vsi_state.f = &st_vsi_init;
652 } /* end vsi_init */
653
654
655 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
656 /**
657 This function is called from the main background loop once every millisecond.
658 It performs all low speed tasks associated with running the core interrupt
659 process, including:
660 - checking for faults
661 - calling the VSI state functions
662 - calling internal analog scaling functions
663
664 \author A.McIver
665 \par History:
666 \li  13/10/07 AM - derived from 25kVA:vsi:vsi.c
667 */
668 void vsi_state_machine(void)
669 {
670   SS_DO(vsi_state);
671   if (adc_int.flag_cal != 0)
672   {
673     adc_int.flag_cal = 0;
674     calibrate_adc();
675   }
676 } /* end vsi_state_machine */
677
678
679 /* ======================================================================
680 __Exported_VSI_Functions()
681 ========================================================================= */
682
683 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
684 /**
685 This function switches the VSI from the stopped state to a running state.
686
687 \author A.McIver
688 \par History:
689 \li  13/10/07 AM - derived from 25kVA:vsi:vsi.c
690 */
691 void vsi_enable(void)
692 {
693   if (detected_faults == 0)
694     {
695       is_switching = 1;
696       VSI_ENABLE();
697       SW_ENABLE();
698       VSI_int_fixed=0;
699     }
700 } /* end vsi_enable */
701
702 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
703 /**
704 This function switches the VSI from the running state to a stop state.
705
706 The ramp down process has the side effect of resetting the reference to zero.
707
708 \author A.McIver
709 \par History:
710 \li  13/10/07 AM - derived from 25kVA:vsi:vsi.c
711 */
712 void vsi_disable(void)
713 {
714   is_switching=0;
715   VSI_DISABLE();
716   SW_DISABLE();
717 } /* end vsi_disable */
718
719 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
720 /**
```

```
721 This function sets the VSI target modulation depth.
722
723 The target is passed in ????.
724
725 \author A.McIver
726 \par History:
727 \li 24/04/09 AM - initial creation
728 \   16/03/11 DS - Changed to set modulation depth in decimal
729 \param[in] m Target output modulation depth
730 */
731 void vsi_set_mod(double mod_serial)
732 {
733     mod=mod_serial;
734     if (mod>=2.0)
735       {
736         mod = 2.0;
737       }
738     else if (mod<=0)
739       {
740         mod = 0;
741       }
742 } /* end vsi_set_mod */
743
744 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
745 /**
746 This function sets the VSI target modulation depth.
747
748 The target is passed in ????.
749
750 \author A.McIver
751 \par History:
752 \li 24/04/09 AM - initial creation
753 \   16/03/11 DS - Changed to set modulation depth in decimal
754 \param[in] m Target output modulation depth
755 */
756 void vsi_set_Iref_mag(double Imag_serial)
757 {
758     Iref_mag_fixed=(long)(Imag_serial*GAIN_OFFSET_CURRENT*(double)FIXED_Q_SCALE);
759     if (Iref_mag_fixed>=MAX_CURR_FIXED)
760       {
761         Iref_mag_fixed = MAX_CURR_FIXED;
762       }
763     else if (Iref_mag_fixed<=0)
764       {
765         Iref_mag_fixed = 0;
766       }
767 } /* end vsi_set_Iref_mag */
768
769 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
770 /**
771 Set the target Fundamental frequency in Hz.
772
773 \author A.McIver
774 \par History:
775 \li  12/10/07 AM - initial creation
776 \li 04/03/08 AM - added return of new frequency
777 \    17/03/11 DS - modified to work with my code.
778
779 \returns The new frequency in Hz
780
781 \param[in] f Target fundamental frequency in Hz
782 */
783 Uint16 vsi_set_fund(double f)
784 {
785   phase_step  = (Uint16)(65536.0*f/SW_FREQ_VSI/2.0);
786   return phase_step;
787 } /* end vsi_set_freq */
788 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
789 /**
790 This function sets the desired reference Voltage.
791
792 The target is passed in ????.
793
794 \author A.McIver
795 \par History:
796 \li 24/04/09 AM - initial creation
797 \   24/04/09 DS - Changed from varying modulation depth to phase shift
798 \param[in] m Target output modulation depth
799 */
800 void vsi_set_vref(int16 vref)
```

```
801 {
802   GrabClear();
803   GrabStart();
804   GrabRun();
805   set_vref=1;
806   dac_vref = vref*DAC_SCALE_VREF+2048;
807 } /* end vsi_set_phase */
808
809 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
810 /**
811 This function returns the status of the VSI output system. It returns
812 - stopped or running
813 - fault code
814 - ramping or settled
815
816 \author A.McIver
817 \par History:
818 \li  13/10/07 AM - derived from 25kVA:vsi:vsi.c
819
820 \retval VSI_RUNNING VSI system switching with output
821 \retval VSI_SETTLED Output has reached target
822 \retval VSI_FAULT VSI system has detected a fault
823 */
824 Uint16 vsi_get_status(void)
825 {
826   return vsi_status;
827 } /* end vsi_get_status */
828
829
830 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
831 /**
832 This function returns the fault word of the VSI module.
833
834 \author A.McIver
835 \par History:
836 \li  04/03/08 AM - initial creation
837
838 \returns The present fault word
839 */
840 /// Report what faults are present in the VSI
841 Uint16 vsi_get_faults(void)
842 {
843   return detected_faults;
844 } /* end vsi_get_faults */
845
846
847 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
848 /* void vsi_clear_faults(void)
849 Parameters: none
850 Returns: nothing
851 Description: Clear the detected faults.
852 Notes:
853 History:
854   13/10/05 AM - initial creation
855 \li 28/04/08 AM - added event reporting
856 */
857 void vsi_clear_faults(void)
858 {
859   Uint16
860     i;
861
862   if (detected_faults & FAULT_VSI_PDPINT)
863   {
864     for (i=0; i<100; i++)
865       i++; // delay for fault to clear
866
867     EvaRegs.COMCONA.all = 0;
868     EvaRegs.COMCONA.all = 0xAA00;
869   }
870   detected_faults = 0;
871 } /* end vsi_clear_faults */
872
873 /* ====================================================================== */
874 /* Interrupt Routines */
875 /* ====================================================================== */
876
877 #ifndef BUILD_RAM
878 #pragma CODE_SECTION(isr_cap2, "ramfuncs");
879 #endif
880
```

```
881 interrupt void isr_cap2(void) //closed loop interrupt structure
882 {
883   int temp;
884
885   SET_TP13();
886   temp=0;
887   while (temp<5)
888   {
889 //      SET_TP13();
890       temp++;
891   }
892   if (first==0) first++;
893   timer_synch_count++;
894   CLEAR_TP13();
895   // Reinitialize for next interrupt
896   EvaRegs.EVAIFRC.bit.CAP2INT = 1;        // clear T1PINT & T1UFINT interrupt flag
897   PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;      // Acknowledge interrupt to PIE Group 2
898 }
899
900 /**
901 \fn interrupt void isr_time(void)
902 \brief Updates VSI and performs closed loop control
903
904 This interrupt is triggered by the ADC interrupts.
905 It then:
906 - takes the adc measurements (synch sample, throws away every alternate one)
907 - determines the gains for the adaptive controller
908 - performs closed loop control calculations
909 - updates phase angle & calculates switching times
910
911 \author A.McIver
912 \par History:
913 \li  12/10/07 AM - initial creation
914 */
915 #ifndef BUILD_RAM
916 #pragma CODE_SECTION(isr_adc, "ramfuncs");
917 #endif
918
919 interrupt void isr_adc(void) //closed loop interrupt structure
920 {
921   //synch variables
922   static Uint16 CAP2_read;
923   static int16   period_2_vsi=PERIOD_2_VSI,period_vsi=PERIOD_VSI;
924   static int16 carrier, carrier_adjust, adjust_time;
925
926   SET_TP10();  //timing bit
927
928   if (cal_count ==0)
929     {
930 /*************************************
931 __calibrate_ADC()
932 *************************************/
933
934     //Dinesh's Calibration
935     //Calibrate the zero offset of the ADCs by taking 1024 readings at 0V and 0A and finding the average
936
937       //sum 1024 readings
938       while (cal_count<1024)
939       {
940         Vdc1_cal  = Vdc1_cal+(AdcRegs.ADCRESULT7-(ADC_OFFSET<<4));
941         Vdc2_cal   = Vdc2_cal+(AdcRegs.ADCRESULT1-(ADC_OFFSET<<4));
942         Vdc3_cal  = Vdc3_cal+(AdcRegs.ADCRESULT2-(ADC_OFFSET<<4)); //useless - is directed to Vac3
943         Vdc4_cal  =  Vdc4_cal+(AdcRegs.ADCRESULT11-(ADC_OFFSET<<4));
944         Vac1_cal  =  Vac1_cal+(AdcRegs.ADCRESULT6-(ADC_OFFSET<<4));
945         Vac2_cal  =  Vac2_cal+(AdcRegs.ADCRESULT10-(ADC_OFFSET<<4));
946         Vac3_cal  = Vac3_cal+(AdcRegs.ADCRESULT2-(ADC_OFFSET<<4));
947         I1_cal    =  I1_cal+(AdcRegs.ADCRESULT4-(ADC_OFFSET<<4));
948         I2_cal    =  I2_cal+(AdcRegs.ADCRESULT8-(ADC_OFFSET<<4));
949         I3_cal    =  I3_cal+(AdcRegs.ADCRESULT0-(ADC_OFFSET<<4));
950         I4_cal    =  I4_cal+(AdcRegs.ADCRESULT5-(ADC_OFFSET<<4));
951         I5_cal    =  I5_cal+(AdcRegs.ADCRESULT3-(ADC_OFFSET<<4));
952         I6_cal    =  I6_cal+(AdcRegs.ADCRESULT9-(ADC_OFFSET<<4));
953         cal_count++;
954       }
955       //take average - divide by 1024
956       if (cal_count==1024)
957         {
958           Vdc1_cal  =  Vdc1_cal>>10;
959           Vdc2_cal   = Vdc2_cal>>10;
960           Vdc3_cal  =  Vdc3_cal>>10;
```

```
961          Vdc4_cal   = Vdc4_cal>>10;
962          Vac1_cal  =  Vac1_cal>>10;
963          Vac2_cal  =  Vac2_cal>>10;
964          Vac3_cal  =  Vac3_cal>>10;
965          I1_cal    = I1_cal>>10;
966          I2_cal    = I2_cal>>10;
967          I3_cal    = I3_cal>>10;
968          I4_cal    = I4_cal>>10;
969          I5_cal    = I5_cal>>10;
970          I6_cal    = I6_cal>>10;
971          puts_COM1("\n CALIBRATION COMPLETE \n");
972        }
973    }
974  else
975    {
976      //Use this when running 40kHz interrupt - resets the compare
977      if (EvaRegs.GPTCONA.bit.T1STAT==1) //last interrupt was an underflow
978      {
979        UF_VSI=1;
980        EvaRegs.T1CMPR = period_vsi-1;
981      }
982      else                              //last interrupt was a compare match
983      {
984        UF_VSI=0;
985        EvaRegs.T1CMPR = 1;
986
987 /*************
988 _SYNCH_CODE()
989 **************/
990        CAP2_read = EvaRegs.CAP2FIFO;
991        if (CAP2_read > period_vsi)
992          carrier = CAP2_read - period_vsi;
993        else
994          carrier  = CAP2_read;
995
996        if (first!=0)  timer_synch_count--;
997
998        if(carrier < 320 )
999        {
1000          // We are lagging the master
1001          // Reduce the period to catch up
1002          carrier_adjust = -1;
1003        }
1004        else if (carrier > 325 )
1005        {
1006          // We are leading the master
1007          // Increase the period to catch up
1008          carrier_adjust = 1;
1009        }
1010        else
1011          carrier_adjust = 0;
1012
1013        // We want it to wobble around the original FSW
1014        adjust_time++;
1015        if (adjust_time>=0)
1016        {
1017          period_vsi = PERIOD_VSI + carrier_adjust;
1018          period_2_vsi = period_vsi>>1;
1019          EvaRegs.T1PR = period_vsi;
1020          EvaRegs.T2PR = (period_vsi<<1)-1;
1021          adjust_time=0;
1022        }
1023      }
1024
1025        //EMERGENCY STOP - if synchronism lost
1026      if (timer_synch_count<5)
1027      {
1028        detected_faults=1;
1029      }
1030
1031 /************
1032 _LOAD_STEP()
1033 *************/
1034      if (load_enable!=prev_load_enable)
1035      {
1036        SET_TP11();
1037        if(load_enable!=0)  EvaRegs.ACTRA.bit.CMP1ACT=3;  //turn on switch
1038        else                EvaRegs.ACTRA.bit.CMP1ACT=0;  //turn off switch
1039
1040        if ((detected_faults==0)&(CPLD.EVACOMCON.bit.ENA == 0))  SW_ENABLE();
```

```
1041       }
1042       else CLEAR_TP11();
1043
1044       prev_load_enable=load_enable;
1045 /***********
1046 _VSI_INT()
1047 ***********/
1048     //This section of code looks after the H-bridge
1049 /**********
1050 _ADC_VSI()
1051 **********/
1052
1053         //For the current-regulated VSI, three ADC inputs are needed
1054         //  -  DC bus voltage for compensation
1055         //  -  Output AC current
1056         //  - BackEMF voltage
1057         Vdc2_fixed  =  (((AdcRegs.ADCRESULT1-Vdc2_cal)>>4)-ADC_OFFSET)*VDC_ANALOG_GAIN;
1058         Vdc1_fixed  =  (((AdcRegs.ADCRESULT7-Vdc1_cal)>>4)-ADC_OFFSET)*VDC_ANALOG_GAIN;
1059         Vac1_fixed  =  (((AdcRegs.ADCRESULT6-Vac1_cal)>>4)-ADC_OFFSET)*VAC_ANALOG_GAIN;
1060         I1_fixed    = (((AdcRegs.ADCRESULT4-I1_cal)>>4)-ADC_OFFSET)*I_ANALOG_GAIN;
1061         I2_fixed    = (((AdcRegs.ADCRESULT8-I2_cal)>>4)-ADC_OFFSET)*I_ANALOG_GAIN;
1062
1063         Vdc_fixed    = (Vdc2_fixed+Vdc1_fixed))>>1;
1064         Vac_fixed    = Vac1_fixed;
1065         IACout_fixed= (I1_fixed+I2_fixed))>>1;
1066
1067 /****************
1068 _VSI_MODULATOR()
1069 ****************/
1070       //this piece of code tells you when you are at the peak of the sine wave
1071         if ((16384-sin_val)<=10)
1072         {
1073           if (Iref_mag_fixed_timed != Iref_mag_fixed)
1074           {
1075             Iref_mag_fixed_timed = Iref_mag_fixed;
1076             SET_TP11();
1077           }
1078
1079         }
1080         else
1081         {
1082           CLEAR_TP11();
1083         }
1084
1085         vsiphase+=PHASE_STEP;
1086         sin_val = sin_table[(vsiphase>>22)|0x00000001];
1087         prev_sin_val = sin_val;
1088 //       SIN_TABLE_READ((Uint16)(vsiphase>>16),sin_val);
1089
1090 /****************
1091 _OPEN_LOOP_VSI()
1092 ****************/
1093         #ifdef OL_VSI
1094         //Need to scale the modulator reference between 0 to period_2
1095         //mod_target*sin_table*period_2
1096         //mod target = 1->2^14
1097         //sin_val = -16384 -> 16384 (uses SIN_TABLE_READ_DINESH(phase,val) 2^14
1098         //PERIOD_2 = 30000
1099         va = (int16)(((int32)(mod*sin_val*period_2_vsi))>>14);
1100         #endif
1101
1102 /****************
1103 _CURR_REG_VSI()
1104 ****************/
1105       //in fixed point - scaled by FIXED_Q
1106       //first, generate reference
1107         Iref_fixed = (long)(Iref_mag_fixed_timed*(long)sin_val)>>(4+FIXED_Q);//scaled by 2^15 from sin table
1108 //         Iref_fixed = (long)((1l<<FIXED_Q)*(long)sin_val)>>(4+FIXED_Q);//scaled by 2^15 from sin table
1109         //write Iref to DAC
1110         //FAST DAC WRITE
1111 //         CLEAR_OC_SPI_EN();
1112 //         SET_SPI_MASTER();
1113 //         ENABLE_DAC1();
1114 //         spi_fail_count = 65535;
1115
1116         //VERY FAST SPI
1117 //         dac_iref = ((Iref_fixed*DAC_SCALE_IREF>>FIXED_Q)+2048);
1118
1119 //         SpiaRegs.SPITXBUF = (DAC_WRn_UPDA|DAC_ADDR_A)<<8;
1120 //         SpiaRegs.SPITXBUF = ((((dac_iref << DAC_SHIFT)>>8)&0x00FF)<<8);
```

```
1121 //        SpiaRegs.SPITXBUF = ((dac_iref << DAC_SHIFT)&0x00FF)<<8;
1122
1123         //scale KP by DC Bus
1124         Kp_VSI_fixed=(long)((KP_CONST*FIXED_Q_SCALE)/Vdc_fixed);
1125         //determine error
1126          VSIerror_fixed = (Iref_fixed - IACout_fixed);
1127         //proportional control
1128         VSIprop_fixed = (VSIerror_fixed*Kp_VSI_fixed)>>FIXED_Q;
1129         //integrator
1130         VSI_intnow_fixed = (VSIprop_fixed*Ki_VSI_fixed)>>FIXED_Q;
1131         VSI_int_fixed += VSI_intnow_fixed;
1132         //control signal
1133         VSI_ctrl_fixed = VSIprop_fixed + VSI_int_fixed;
1134
1135         #ifdef CL_VSI
1136         va = (VSI_ctrl_fixed*period_2_vsi)>>FIXED_Q;
1137         #endif
1138
1139 /********************
1140 _VSI_SWITCHING_TIMES()
1141 ********************/
1142         /* Switching duty cycles */
1143         t_A = va;
1144         t_B = -t_A;
1145
1146 /**************
1147 _VSI_DESAT()
1148 **************/
1149 /* clamp switch times for pulse deletion and saturation */
1150
1151 // A phase
1152         // A phase
1153         if (t_A > max_time)
1154         {
1155                   EvaRegs.CMPR2 = 1;
1156                   dac_va = max_time>>2;
1157         }
1158         else if (t_A < (-max_time))
1159         {
1160           if (!V_Asat && UF_VSI)  EvaRegs.CMPR2 = period_vsi - 1;
1161           else                    EvaRegs.CMPR2 = period_vsi - 1;
1162           V_Asat = 1;
1163           dac_va = -max_time>>2;
1164         }
1165         else
1166         {
1167           if (V_Asat && UF_VSI)    EvaRegs.CMPR2 = period_vsi - 1;
1168           else                    EvaRegs.CMPR2 = (Uint16)(period_2_vsi - t_A);
1169           V_Asat = 0;
1170           dac_va = va>>2;
1171         }
1172
1173         // B phase
1174         if (t_B > max_time)        EvaRegs.CMPR3 = 1;
1175         else if (t_B < (-max_time))
1176         {
1177           if (!V_Bsat && UF_VSI)  EvaRegs.CMPR3 = period_vsi - 1;
1178           else                    EvaRegs.CMPR3 = period_vsi - 1;
1179           V_Bsat = 1;
1180         }
1181         else
1182         {
1183           if (V_Bsat && UF_VSI)    EvaRegs.CMPR3 = period_vsi - 1;
1184           else                    EvaRegs.CMPR3 = (Uint16)(period_2_vsi - t_B);
1185           V_Bsat = 0;
1186         }
1187
1188         //freeze integrator
1189         if((V_Asat==1)||(V_Bsat==1))
1190         {
1191           VSI_int_fixed -= VSI_intnow_fixed;
1192         }
1193
1194 //now write va to the DAC to be picked up on the other side for FF purposes
1195         //FAST DAC WRITE
1196         CLEAR_OC_SPI_EN();
1197         SET_SPI_MASTER();
1198         ENABLE_DAC1();
1199         spi_fail_count = 65535;
1200
```

```
1201         //VERY FAST SPI
1202         SpiaRegs.SPITXBUF = (DAC_WRn_UPDn|DAC_ADDR_B)<<8;
1203         SpiaRegs.SPITXBUF = (((((dac_va+2048) << DAC_SHIFT)>>8)&0x00FF)<<8);
1204         SpiaRegs.SPITXBUF = (((dac_va+2048) << DAC_SHIFT)&0x00FF)<<8;
1205
1206
1207 ///*****************
1208 //_GRID_CONNECTION()
1209 //*****************/
1210 //
1211 //        if (EvaRegs.CAPFIFOA.bit.CAP1FIFO != 0)
1212 //        {
1213 //          ZX_time = PERIOD_VSI - EvaRegs.CAP1FIFO;
1214 //          ZX_seen = 1;
1215 //      //    SET_TP11();
1216 //      //    temp=0;
1217 //      //    while(temp<100) temp++;
1218 //      //    CLEAR_TP11();
1219 //          EvaRegs.CAPFIFOA.all = 0x0000; // dump any other captured values
1220 //        }
1221 //
1222 //        ZX_count++;
1223 //        if (ZX_count > ZX_MAX_COUNT)  /* Zero crossing signal lost           */
1224 //        {
1225 //    //      VSI_DISABLE();    /* Halt modulation                     */
1226 //          in_sync = 0;
1227 //          ZX_state = ZX_LOST;    /* Restart searching for sync          */
1228 //          ZX_in_sync = 0;
1229 //          ZX_count = 0;
1230 //        }
1231 //
1232 //        if (ZX_state == ZX_LOST)    /* No idea of anything: start freq est.*/
1233 //        {
1234 //          in_sync = 0;
1235 //          if (ZX_seen != 0)
1236 //          {
1237 //            ZX_seen = 0;
1238 //            ZX_cycles = 0;
1239 //            ZX_sum = 0;
1240 //            ZX_count = 0;
1241 //            ZX_state = ZX_EST;
1242 //          }
1243 //        }
1244 //
1245 //        else if (ZX_state == ZX_EST)  /* Roughly measure period and average  */
1246 //        {
1247 //          if (ZX_seen != 0)
1248 //          {
1249 //            ZX_seen = 0;
1250 //            ZX_cycles++;
1251 //            ZX_sum += ZX_count;
1252 //            ZX_count = 0;        /* Reset counter                   */
1253 //          }
1254 //          if (ZX_cycles >= ZX_CYCLE_AVG)
1255 //          {
1256 //            ZX_sum = ZX_sum/ZX_CYCLE_AVG;
1257 //            ZX_phase_step = ((Uint32)(0xFFFF/ZX_sum))<<16; // Approximate frequency
1258 //            ZX_sum -= ZX_sum/8;      /* Also use for glitch filter      */
1259 //            ZX_vsiphase = ZX_phase_step + zx_offset;  /* Within phase_step */
1260 //            ZX_state = ZX_MISC; /* Calculate ZX_phase_scale first  */
1261 //          }
1262 //        }
1263 //
1264 //        else if (ZX_state == ZX_SYNC) /* Accurately measure phase error  */
1265 //        {
1266 //          if (ZX_seen != 0)
1267 //          {
1268 //            ZX_seen = 0;
1269 //            if (ZX_count > ZX_sum)    /* Ignore glitches */
1270 //            {
1271 //              ZX_count_grab = ZX_count;
1272 //              ZX_count = 0;
1273 //              /* Rescale to phase units */
1274 //              ZX_time_phase = zx_offset + (((int32)ZX_time*ZX_phase_scale)>>5);
1275 //              /* Calculate phase error captured time */
1276 //              ZX_phase_err = ZX_vsiphase - ZX_time_phase;
1277 //              /* Limit size of phase change */
1278 //              if (ZX_phase_err > ZX_BIG_ERR)
1279 //              {
1280 //                ZX_vsiphase -= ZX_BIG_ERR;
```

```
1281 //              // Integrate phase errors
1282 //              ZX_err_sum = (ZX_err_sum+ZX_BIG_ERR)>>1;
1283 //            }
1284 //            else if (ZX_phase_err < -ZX_BIG_ERR)
1285 //            {
1286 //              ZX_vsiphase += ZX_BIG_ERR;
1287 //              ZX_err_sum = (ZX_err_sum-ZX_BIG_ERR)>>1;
1288 //            }
1289 //            else
1290 //            {
1291 //              ZX_vsiphase -= ZX_phase_err;
1292 //              ZX_err_sum = (ZX_err_sum+ZX_phase_err)>>1;
1293 //            }
1294 //     //        vsiphase = ZX_vsiphase;
1295 //            ZX_state = ZX_FREQ;
1296 //          }
1297 //        }
1298 //      }
1299 //
1300 //      else if (ZX_state == ZX_FREQ)  /* Nudge frequency if needed */
1301 //      {
1302 //        /* If too large, nudge freq (phase_step) */
1303 //        if (ZX_err_sum > ZX_FREQ_ERR)
1304 //        {
1305 //          ZX_phase_step -= 100L;
1306 //          if (ZX_err_sum > ZX_FREQ_ERR_BIG)
1307 //          {
1308 //            ZX_phase_step -= 1000L;
1309 //          }
1310 //        }
1311 //        else if (ZX_err_sum < -ZX_FREQ_ERR)
1312 //        {
1313 //          ZX_phase_step += 100L;
1314 //          if (ZX_err_sum < -ZX_FREQ_ERR_BIG)
1315 //          {
1316 //            ZX_phase_step += 1000L;
1317 //          }
1318 //        }
1319 //        ZX_state = ZX_LOCK;
1320 //      }
1321 //
1322 //      else if (ZX_state == ZX_LOCK)  /* Test to see if still in sync */
1323 //      {
1324 //        if (ZX_in_sync >= ZX_SYNC_LIMIT)
1325 //        {
1326 //          if ((ZX_phase_err>ZX_PHASE_ERR)||(ZX_phase_err<-ZX_PHASE_ERR))
1327 //          {            /* Gone out of sync */
1328 //   //          VSI_DISABLE();
1329 //            ZX_in_sync = 0;
1330 //            in_sync = 0;
1331 //          }
1332 //          else
1333 //          {
1334 //            in_sync = 1;
1335 //          }
1336 //        }
1337 //        else if ((ZX_phase_err<ZX_PHASE_ERR)&&(ZX_phase_err>-ZX_PHASE_ERR))
1338 //        {              /* In sync this cycle                 */
1339 //          ZX_in_sync++;
1340 //        }
1341 //        else
1342 //        {
1343 //          ZX_in_sync = 0;
1344 //        }
1345 //        ZX_state = ZX_MISC;
1346 //      }
1347 //      else if (ZX_state == ZX_MISC)
1348 //      {
1349 //        ZX_phase_scale = (phase_step<<5)/PERIOD_VSI;
1350 //        ZX_state = ZX_SYNC;
1351 //      }
1352 //      //end grid connection
1353
1354         //Finish the DAC write for the Ref Step before starting the next one.
1355         //this is almost the last thing  done, to give as much as time as possible
1356         //for the DAC write to complete (because SPI is slow)
1357         while ((SpiaRegs.SPIFFRX.bit.RXFFST < 3)&&(spi_fail_count>0) )
1358         {
1359           spi_fail_count--; // counter to avoid hang up if SPI fails
1360         } // wait for tx to finish
```

```
1361           i_spi=SpiaRegs.SPIRXBUF;
1362           i_spi=SpiaRegs.SPIRXBUF;
1363           i_spi=SpiaRegs.SPIRXBUF;
1364           DISABLE_DAC1();
1365
1366     //end vsi
1367
1368     }
1369
1370 /* =====================================================================
1371 isr_GrabCode()
1372 ======================================================================= */
1373 #ifdef GRAB_INCLUDE
1374
1375   if (GrabRunning())
1376   {
1377     GrabStore(0,sin_val);
1378     GrabStore(1,0);
1379     GrabStore(2,va);
1380 //    GrabStore(3,EvaRegs.CMPR3);
1381 //    GrabStore(4,dac_iref);
1382 //    GrabStore(5,carrier);
1383 //    GrabStore(6,EvaRegs.CMPR2);
1384 //    GrabStore(7,EvaRegs.CMPR3);
1385
1386     grab_index++;
1387
1388     if (grab_index >= GRAB_LENGTH)
1389       grab_mode = GRAB_STOPPED;
1390   }
1391 #endif
1392
1393
1394
1395   // Reinitialize for next interrupt
1396   AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;        // clear interrupt flag
1397   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;      // Acknowledge interrupt to PIE Group 2
1398   CLEAR_TP10();        // timing bit
1399 } /* end isr_timer_CL */
1400
1401 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1402 /**
1403 Handles the PDPINT interrupt caused by a gate fault.
1404
1405 \author A.McIver
1406 \par History:
1407 \li  02/05/07 AM - initial creation
1408 */
1409 #ifndef BUILD_RAM
1410 #pragma CODE_SECTION(isr_gate_fault, "ramfuncs");
1411 #endif
1412 interrupt void isr_gate_fault(void)
1413 {
1414   is_switching = 0;
1415   VSI_DISABLE();
1416   SW_DISABLE();
1417 //  SET_TP12();
1418   mod_targ = 0;
1419   detected_faults = FAULT_VSI_PDPINT;
1420   GrabClear();
1421   GrabStart();
1422   GrabRun();
1423
1424   // Acknowledge this interrupt to receive more interrupts from group 1
1425   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
1426   EvaRegs.EVAIFRA.all = BIT0;
1427 } /* end isr_gate_fault */
1428
1429
1430 /* =====================================================================
1431 __VSI_State_Functions()
1432 ======================================================================= */
1433
1434
1435 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1436 /**
1437 This function initialises the VSI system. It resets the target modulation
1438 depth to zero.
1439
1440 It is followed by the stop state.
```

```
1441
1442 \author A.McIver
1443 \par History:
1444 \li  12/10/07 AM - initial creation
1445 */
1446 void st_vsi_init(void)
1447 {
1448   mod_ref = 0;
1449   mod_targ = 0;
1450   EvaRegs.ACTRA.all = 0x0000;
1451   vsi_status = VSI_INIT;
1452   VSI_DISABLE();
1453   SW_DISABLE();
1454   SS_NEXT(vsi_state,st_vsi_stop);
1455 } /* end st_vsi_init */
1456
1457
1458 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1459 /**
1460 This is the state where the VSI is stopped. There is no switching. It waits
1461 for a start trigger.
1462
1463 \author A.McIver
1464 \par History:
1465 \li  12/10/07 AM - initial creation
1466 */
1467 void st_vsi_stop(void)
1468 {
1469   if (SS_IS_FIRST(vsi_state))
1470   {
1471     SS_DONE(vsi_state);
1472     VSI_DISABLE();
1473     SW_DISABLE();
1474     mod_targ = 0;
1475     vsi_status = VSI_STOP;
1476 //    vsi_status &= ~(VSI_RUNNING|VSI_SETTLED);
1477   }
1478
1479   if (detected_faults != 0)
1480   {
1481     SS_NEXT(vsi_state,st_vsi_fault);
1482     return;
1483   }
1484
1485   if (is_switching != 0) // start trigger
1486   {
1487     SS_NEXT(vsi_state,st_vsi_gate_charge);
1488   }
1489 } /* end st_vsi_stop */
1490
1491
1492 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1493 /**
1494 In this state the VSI gates are enabled and the low side gates held on to
1495 charge the high side gate drivers. The next state is either the ramp state.
1496
1497 \author A.McIver
1498 \par History:
1499 \li  12/10/07 AM - initial creation
1500 */
1501 void st_vsi_gate_charge(void)
1502 {
1503   if (SS_IS_FIRST(vsi_state))
1504   {
1505     SS_DONE(vsi_state);
1506     vsi_counter = 0;
1507 //    VSI_GATE_CHARGE();
1508 //    vsi_status = VSI_GATECHARGE;
1509 //    vsi_status |= VSI_RUNNING;
1510   }
1511   if (detected_faults != 0)
1512   {
1513     SS_NEXT(vsi_state,st_vsi_fault);
1514     return;
1515   }
1516   // check for stop signal
1517   if (is_switching == 0)
1518   {
1519     SS_NEXT(vsi_state,st_vsi_stop);
1520     return;
```

```
1521   }
1522   vsi_counter++;
1523   if (vsi_counter > 100)
1524   {
1525     SS_NEXT(vsi_state,st_vsi_ramp);
1526   }
1527 } /* end st_vsi_gate_charge */
1528
1529
1530 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1531 /**
1532 This state ramps up the target modulation depth to match the reference set by
1533 the background. It only changes the target every 100ms and synchronises the
1534 change with a zero crossing to avoid step changes in the output.
1535
1536 \author A.McIver
1537 \par History:
1538 \li  12/10/07 AM - initial creation
1539 \li 28/04/08 AM - added event reporting
1540 */
1541 void st_vsi_ramp(void)
1542 {
1543   if (SS_IS_FIRST(vsi_state))
1544   {
1545     SS_DONE(vsi_state);
1546     VSI_ENABLE();
1547     SW_ENABLE();
1548     vsi_counter = 0;
1549     vsi_status = VSI_RAMP;
1550   }
1551   if (detected_faults != 0)
1552   {
1553     SS_NEXT(vsi_state,st_vsi_fault);
1554     return;
1555   }
1556   // check for stop signal
1557   if (is_switching == 0)
1558   {
1559     SS_NEXT(vsi_state,st_vsi_stop);
1560     return;
1561   }
1562   else
1563   {
1564     SS_NEXT(vsi_state,st_vsi_run);
1565     return;
1566   }
1567
1568 } /* end st_vsi_ramp */
1569
1570 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1571 /**
1572 This state has the VSI running with the target voltage constant. The output is
1573 now ready for measurements to begin. If the reference is changed then the
1574 operation moves back to the ramp state.
1575
1576 \author A.McIver
1577 \par History:
1578 \li  12/10/07 AM - initial creation
1579 */
1580 void st_vsi_run(void)
1581 {
1582   if (SS_IS_FIRST(vsi_state))
1583   {
1584     SS_DONE(vsi_state);
1585     vsi_status = VSI_RUNNING;
1586   }
1587   if (detected_faults != 0)
1588   {
1589     SS_NEXT(vsi_state,st_vsi_fault);
1590     return;
1591   }
1592   // check for stop signal
1593   if (is_switching == 0)
1594   {
1595     SS_NEXT(vsi_state,st_vsi_stop);
1596   }
1597   // check for changes in reference
1598   if (mod_targ != mod_ref)
1599   {
1600     vsi_status &= ~VSI_SETTLED;
```

```
1601     SS_NEXT(vsi_state,st_vsi_ramp);
1602   }
1603 } /* end st_vsi_run */
1604
1605
1606 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1607 /* void st_vsi_fault(void)
1608 Parameters: none
1609 Returns: nothing
1610 Description: Delays for a while after faults are cleared.
1611 Notes:
1612 History:
1613   03/11/05 AM - initial creation
1614 \li 04/03/08 AM - set vsi_status with fault bit
1615 \li 28/04/08 AM - added event reporting
1616 */
1617 void st_vsi_fault(void)
1618 {
1619   if (SS_IS_FIRST(vsi_state))
1620   {
1621     SS_DONE(vsi_state);
1622     VSI_DISABLE();
1623     SW_DISABLE();
1624     vsi_counter = 0;
1625     vsi_status = VSI_FAULT;
1626 //    vsi_status &= ~(VSI_RUNNING|VSI_SETTLED);
1627     putxx(detected_faults);
1628     puts_COM1("->VSI faults\n");
1629   }
1630   if (detected_faults == 0)
1631     vsi_counter++;
1632   else
1633     vsi_counter = 0;
1634   if (vsi_counter > 100)
1635   {
1636 //    vsi_status &= ~VSI_FAULT;
1637     SS_NEXT(vsi_state,st_vsi_stop);
1638   }
1639 } /* end st_vsi_fault */
1640
1641
1642 /* ======================================================================
1643 __Local_Functions()
1644 ========================================================================= */
1645
1646
1647 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1648 /**
1649 This function is called every fundamental period to perform the RMS
1650 calculations and scale the analog quantities to Volts and Amps for use in the
1651 background.
1652
1653 \author A.McIver
1654 \par History:
1655 \li  12/10/07 AM - derived from IR25kVA:vsi:adc_scale
1656 \li 21/08/08 AM - added VSI DC offset compensation
1657 \li 12/09/08 AM - added stop_count and moved to floating point data
1658 */
1659 //void scale_adc_rms(void)
1660 //{
1661 //  double
1662 //    val,
1663 //    temp;
1664 //
1665 //  // calculate A0 RMS quantity
1666 //  temp = (double)adc_int.A0.dc_sum_bak/(double)adc_int.count_rms_bak;
1667 //  val = (double)adc_int.A0.rms_sum_bak*(double)(1<<ADC_RMS_PS)
1668 //          / (double)adc_int.count_rms_bak - temp*temp;
1669 //  if (val < 0.0) val = 0.0;
1670 //  adc_int.A0.real = ADC_REAL_SC * sqrt(val);
1671 //} /* end scale_adc_rms */
1672
1673
1674 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1675 /**
1676 This function is called every ADC_DC_TIME to perform the DC calculations and
1677 scale the analog quantities to Volts and Amps for use in the background.
1678
1679 \author A.McIver
1680 \par History:
```

```
1681 \li  12/10/07 AM - derived from IR25kVA:vsi:adc_scale
1682 */
1683 //void scale_adc_dc(void)
1684 //{
1685 //  double
1686 //    val;
1687 //
1688 //  adc_int.A0.real = (double)adc_int.A0.dc_sum_bak/(double)ADC_COUNT_DC;
1689 //  adc_int.A2.real = (double)adc_int.A2.dc_sum_bak/(double)ADC_COUNT_DC;
1690 //  adc_int.A4.real = (double)adc_int.A4.dc_sum_bak/(double)ADC_COUNT_DC;
1691 //  adc_int.A6.real = (double)adc_int.A6.dc_sum_bak/(double)ADC_COUNT_DC;
1692 //
1693 //  // calculate B0 DC quantity
1694 //  val = (double)adc_int.B0.dc_sum_bak/(double)ADC_COUNT_DC;
1695 //  adc_int.B0.real = ADC_REAL_SC * val;
1696 //
1697 //} /* end scale_adc_dc */
1698
1699
1700 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1701 /**
1702 Calibrates the adc for gain and offset using the reference inputs.
1703
1704 See spra989a.pdf for calibration details
1705
1706 \author A.McIver
1707 \par History:
1708 \li  07/10/05 AM - initial creation
1709 */
1710 void calibrate_adc(void)
1711 {
1712 //  char
1713 //    str[60];
1714
1715   yHA = (double)adc_int.yHA.dc_sum_bak/(double)ADC_COUNT_CAL;
1716   yLA = (double)adc_int.yLA.dc_sum_bak/(double)ADC_COUNT_CAL;
1717   yHB = (double)adc_int.yHB.dc_sum_bak/(double)ADC_COUNT_CAL;
1718   yLB = (double)adc_int.yLB.dc_sum_bak/(double)ADC_COUNT_CAL;
1719
1720   cal_gain_A = (xH - xL)/(yHA - yLA);
1721   cal_offset_A = yLA * cal_gain_A - xL;
1722
1723   cal_gain_B = (xH - xL)/(yHB - yLB);
1724   cal_offset_B = yLB * cal_gain_B - xL;
1725
1726   // sanity check on gains
1727   if (   ( (cal_gain_A > 0.94) && (cal_gain_A < 1.05) )
1728     && ( (cal_gain_B > 0.94) && (cal_gain_B < 1.05) )
1729     && ( (cal_offset_A > -80.0) && (cal_offset_A < 80.0) )
1730     && ( (cal_offset_B > -80.0) && (cal_offset_B < 80.0) ) )
1731   {
1732     cal_gainA = (int16)(cal_gain_A*(double)(1<<14));
1733     cal_gainB = (int16)(cal_gain_B*(double)(1<<14));
1734     cal_offsetA = (int16)cal_offset_A;
1735     cal_offsetB = (int16)cal_offset_B;
1736   }
1737 //  sprintf(str,"cal:gA=%.3f,oA=%5.1f, gB=%.3f,oB=%5.1f\n",cal_gain_A,
1738 //          cal_offset_A,cal_gain_B,cal_offset_B);
1739 //  puts_COM1(str);
1740 } /* end calibrate_adc */
1741
1742 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1743
1744 void get_state(void){
1745   if(vsi_state.f == st_vsi_init){
1746     puts_COM1("INIT ");
1747   }
1748   else if(vsi_state.f == st_vsi_stop){
1749     puts_COM1("STOP ");
1750   }
1751   else if(vsi_state.f == st_vsi_gate_charge){
1752     puts_COM1("GATE ");
1753   }
1754   else if(vsi_state.f == st_vsi_ramp){
1755     puts_COM1("RAMP ");
1756   }
1757   else if(vsi_state.f == st_vsi_run){
1758     puts_COM1("RUN  ");
1759   }
1760   else if(vsi_state.f == st_vsi_fault){
```

```
1761     puts_COM1("FAU  ");
1762   }
1763
1764 }
```

References

[1] C. Gang, L. Yim-Shu, S. Y. R. Hui, D. Xu, and Y. Wang., "Actively clamped bidirectional flyback converter," *IEEE Trans. Ind. Electron.*, vol. 47, no. 4, pp. 770--779, 2000.

[2] T. Haimin, J. L. Duarte, and M. A. M. Hendrix, "Three-port triple-half-bridge bidirectional converter with zero-voltage switching," *IEEE Trans. Power Electron.*, vol. 23, no. 2, pp. 782--792, 2008.

[3] A. Ipakchi and F. Albuyeh, "Grid of the future," *IEEE Power Energy Mag.*, vol. 7, no. 2, pp. 52--62, 2009.

[4] K. Moslehi and R. Kumar, "A reliability perspective of the smart grid," *IEEE Trans. Smart Grid*, vol. 1, no. 1, pp. 57--64, 2010.

[5] M. Galus and G. Andersson, "Demand management of grid connected plug-in hybrid electric vehicles (phev)," in *Energy 2030 Conference, 2008. ENERGY 2008. IEEE*, Nov 2008, pp. 1 --8.

[6] N. Rotering and M. Ilic, "Optimal charge control of plug-in hybrid electric vehicles in deregulated electricity markets," *IEEE Trans. Power Syst.*, vol. 26, no. 3, pp. 1021--1029, 2011.

[7] P. Kulshrestha, L. Wang, M.-Y. Chow, and S. Lukic, "Intelligent energy management system simulator for phevs at municipal parking deck in a smart grid environment," in *Power Energy Society General Meeting, 2009. PES '09. IEEE*, july 2009, pp. 1 --6.

[8] B. Gemmell, J. Dorn, D. Retzmann, and D. Soerangr, "Prospects of multilevel vsc technologies for power transmission," in *Proc. IEEE/PES Transmission and Distribution Conference and Exposition*, April 2008, pp. 1--16.

[9] J. W., A. Huang, W. Sung, Y. Liu, and B. Baliga, "Smart grid technologies," *Industrial Electronics Magazine, IEEE*, vol. 3, no. 2, pp. 16 --23, June 2009.

[10] N. Mohan, T. Undeland, and W. Robbins, *Power electronics: Converters, applications, and design.* Hoboken, NJ: John Wiley, 2003.

[11] A. Pressman, K. Billings, and T. Morey, *Switching Power Supply Design*, 3rd ed. New York, NY: McGraw-Hill, 2009.

[12] D. G. Holmes and T. A. Lipo, *Pulse Width Modulation For Power Converters - Principles And Practise*, ser. IEEE press series on power engineering. Hoboken, NJ: John Wiley, 2003.

[13] K. Mets, T. Verschueren, W. Haerick, C. Develder, and F. De Turck, ‘‘Optimizing smart energy control strategies for plug-in hybrid electric vehicle charging,’’ in *Proc. IEEE/IFIP Network Operations and Management Symposium Workshop*, April 2010, pp. 293--299.

[14] H. Rongjun and S. K. Mazumder, ‘‘A soft-switching scheme for an isolated dc/dc converter with pulsating dc output for a three-phase high-frequency-link pwm converter,’’ *IEEE Trans. Power Electron.*, vol. 24, no. 10, pp. 2276--2288, 2009.

[15] B. P. McGrath and D. G. Holmes, ‘‘Analytical modelling of voltage balance dynamics for a flying capacitor multilevel converter,’’ *IEEE Trans. Power Electron.*, vol. 23, no. 2, pp. 543--550, 2008, 0885-8993.

[16] D. Segaran, D. G. Holmes, and B. P. McGrath, ‘‘Enhanced load step response for a bi-directional dc-dc converter,’’ in *Proc. IEEE Energy Conversion Congress and Exposition (ECCE)*, 2011, pp. 3649--3656.

[17] D. Segaran, D. G. Holmes, and B. P. McGrath, ‘‘High-performance bi-directional ac-dc converters for phev with minimised dc bus capacitance,’’ in *Proc. 37th IEEE Annual Conference on Industrial Electronics (IECON)*, 2011, pp. 3620 -- 3625.

[18] D. Segaran, B. P. McGrath, and D. G. Holmes, ‘‘Adaptive dynamic control of a bi-directional dc-dc converter,’’ in *Proc. IEEE Energy Conversion Congress and Exposition (ECCE)*, 2010, pp. 1442--1449.

[19] D. Segaran, D. G. Holmes, and B. P. McGrath, ‘‘Comparative analysis of single and three-phase dual active bridge bidirectional dc-dc converters,’’ in *Proc. Australasian Universities Power Engineering Conference (AUPEC)*, 2008, pp. 1--6.

[20] D. Segaran, D. Holmes, and B. McGrath, ‘‘Comparative analysis of single- and three-phase dual active bridge bidirectional dc-dc converters,’’ *Aust. J. Electr. Electron. Eng.*, vol. 6, no. 3, pp. 1--12, 2009.

[21] D. Segaran, D. Holmes, and B. McGrath, ‘‘Enhanced load step response for a bi-directional dc-dc converter,’’ *IEEE Trans. Power Electron.*, vol. 28, no. 1, pp. 371--379, 2013.

[22] W. Kunrong, F. C. Lee, and J. Lai, ‘‘Operation principles of bi-directional full-bridge dc/dc converter with unified soft-switching scheme and soft-starting capability,’’ in *Proc. 15th IEEE Annual Applied Power Electronics Conference and Exposition (APEC)*, vol. 1, 2000, pp. 111--118.

[23] S. Liping, X. Dehong, and C. Min, ‘‘Dynamic modeling of a pwm plus phase-shift (pps) controlled active clamping boost to full bridge bi-directional dc/dc converter,’’ in *Proc. 37th IEEE Power Electronics Specialists Conference (PESC)*, 2006, pp. 1--6.

[24] F. Haifeng and X. Dehong, ‘‘A family of pwm plus phase-shift bidirectional dc-dc converters,’’ in *Proc. 35th IEEE Power Electronics Specialists Conference (PESC)*, vol. 2, 2004, pp. 1668--1674.

[25] K. Wang, C. Y. Lin, L. Zhu, D. Qu, F. C. Lee, and J. S. Lai, ‘‘Bi-directional dc to dc converters for fuel cell systems,’’ in *Proc. IEEE Power Electronics in Transportation Conference (PET)*, 1998, pp. 47--51.

[26] J. Su-Jin, L. Tae-Won, L. Won-Chul, and W. Chung-Yuen, ‘‘Bi-directional dc-dc converter for fuel cell generation system,’’ in *Proc. 35th IEEE Power Electronics Specialists Conference (PESC)*, vol. 6, 2004, pp. 4722--4728.

[27] J. S. Lai and D. J. Nelson, ‘‘Energy management power converters in hybrid electric and fuel cell vehicles,’’ *Proc. IEEE*, vol. 95, no. 4, pp. 766--777, 2007, 0018-9219.

[28] W. Kunrong, F. C. Lee, and W. Dong, ‘‘A new soft-switched quasi-single-stage (qss) bi-directional inverter/charger,’’ in *Proc. 34th IEEE Industry Applications Society Annual Meeting (IAS)*, vol. 3, 1999, pp. 2031--2038.

[29] T. Haimin, J. L. Duarte, and M. A. M. Hendrix, ‘‘Multiport converters for hybrid power sources,’’ in *Proc. 39th IEEE Power Electronics Specialists Conference (PESC)*, 2008, pp. 3412--3418.

[30] H. R. Karshenas, H. Daneshpajooh, A. Safaee, A. Bakhshai, and P. Jain, ‘‘Basic families of medium-power soft-switched isolated bidirectional dc-dc converters,’’

in *Proc. 2nd Power Electronics, Drive Systems and Technologies Conference (PEDSTC)*, 2011, pp. 92--97.

[31] K. Vangen, T. Melaa, S. Bergsmark, and R. Nilsen, ‘‘Efficient high-frequency soft-switched power converter with signal processor control,’’ in *Proc. 13th International Telecommunications Energy Conference (INTELEC)*, 1991, pp. 631--639.

[32] K. Vangen, T. Melaa, and A. K. Adnanes, ‘‘Soft-switched high-frequency, high power dc/ac converter with igbt,’’ in *Proc. 23rd IEEE Power Electronics Specialists Conference (PESC)*, vol. 1, 1992, pp. 26--33.

[33] H. J. Cha and P. N. Enjeti, ‘‘A three-phase ac-ac high-frequency link matrix converter for vscf applications,’’ in *Proc. 34th IEEE Power Electronics Specialists Conference (PESC)*, vol. 4, 2003, pp. 1971--1976.

[34] D. C. and L. L., ‘‘Bi-polarity phase-shifted controlled voltage mode ac/ac converters with high frequency ac link,’’ in *Proc. 34th IEEE Power Electronics Specialists Conference (PESC)*, vol. 2, 2003, pp. 677 -- 682.

[35] M. Carpita, M. Marchesoni, M. Pellerin, and D. Moser, ‘‘Multilevel converter for traction applications: Small-scale prototype tests results,’’ *IEEE Trans. Ind. Electron.*, vol. 55, no. 5, pp. 2203--2212, 2008, 0278-0046.

[36] C. Zimmermann, A. Rufer, and C. Chabert, ‘‘Non-linear properties and efficiency improvements of a bi-directional isolated dc-ac converter with soft commutation,’’ in *Proc. 40th IEEE Industry Applications Society Annual Meeting (IAS)*, vol. 3, 2005, pp. 1985--1991.

[37] Q. Hengsi and J. W. Kimball, ‘‘Ac-ac dual active bridge converter for solid state transformer,’’ in *Proc. IEEE Energy Conversion Congress and Exposition (ECCE)*, 2009, pp. 3039--3044.

[38] Q. Hengsi and J. W. Kimball, ‘‘Generalized average modeling of dual active bridge dc-dc converter,’’ *IEEE Trans. Power Electron.*, vol. 27, no. 4, pp. 2078--2084, 2012.

[39] G. Waltrich, J. Duarte, and M. Hendrix, ‘‘Multiport converters for fast chargers of electrical vehicles - focus on high-frequency coaxial transformers,’’ in *Proc. 2010 IEEE International Power Electronics Conference (IPEC)*, 2010, pp. 3151 -- 3157.

[40] H. Tao, A. Kotsopoulos, J. Duarte, and M. Hendrix, ‘‘Family of multiport bidirectional dc-dc converters,’’ *IEE Electric Power Applications*, vol. 153, no. 3, pp. 451--458, 2006.

[41] M. Cacciato, F. Caricchi, F. Giuhlii, and E. Santini, ‘‘A critical evaluation and design of bi-directional dc/dc converters for super-capacitors interfacing in fuel cell applications,’’ in *Proc. 39th IEEE Industry Applications Society Annual Meeting (IAS)*, vol. 2, 2004, pp. 1127--1133.

[42] K. Z., Z. C., Y. S., and C. S., ‘‘Study of bidirectional dc-dc converter for power management in electric bus with supercapacitors,’’ in *Proc. IEEE Vehicle Power and Propulsion Conference (VPPC)*, 2006, pp. 1 --5.

[43] Z. Fanghua, X. Lan, and Y. Yangguang, ‘‘Bi-directional forward-flyback dc-dc converter,’’ in *Proc. 35th IEEE Power Electronics Specialists Conference (PESC)*, vol. 5, 2004, pp. 4058--4061.

[44] C. Gang, X. Dehong, and L. Yim-Shu, ‘‘A novel fully zero-voltage-switching phase-shift bidirectional dc-dc converter,’’ in *Proc. 16th IEEE Annual Applied Power Electronics Conference and Exposition (APEC)*, vol. 2, 2001, pp. 974--979.

[45] D. G. Holmes, P. Atmur, C. C. Beckett, M. P. Bull, W. Y. Kong, W. J. Luo, D. K. C. Ng, N. Sachchithananthan, P. W. Su, D. P. Ware, and P. Wrzos, ‘‘An innovative, efficient current-fed push-pull grid connectable inverter for distributed generation systems,’’ in *Proc. 37th IEEE Power Electronics Specialists Conference (PESC)*, 2006, pp. 1--7.

[46] C. Gang, X. Dehong, W. Yousheng, and L. Yirn-Shu, ‘‘A new family of soft-switching phase-shift bidirectional dc-dc converters,’’ in *Proc. 32nd IEEE Power Electronics Specialists Conference (PESC)*, vol. 2, 2001, pp. 859--865.

[47] M. Jain, M. Daniele, and P. K. Jain, ‘‘A bidirectional dc-dc converter topology for low power application,’’ *IEEE Trans. Power Electron.*, vol. 15, no. 4, pp. 595--606, 2000, 0885-8993.

[48] R. Garcia-Gil, J. M. Espi, E. J. Dede, and E. Sanchis-Kilders, ‘‘A bidirectional and isolated three-phase rectifier with soft-switching operation,’’ *IEEE Trans. Ind. Electron.*, vol. 52, no. 3, pp. 765--773, 2005, 0278-0046.

[49] A. D. Swingler and W. G. Dunford, ‘‘Development of a bi-directional dc/dc converter for inverter/charger applications with consideration paid to large signal operation and quasi-linear digital control,’’ in *Proc. 33rd IEEE Power Electronics Specialists Conference (PESC)*, vol. 2, 2002, pp. 961--966.

[50] K. Yamamoto, E. Hiraki, T. Tanaka, M. Nakaoka, and T. Mishima, ''Bidirectional dc-dc converter with full-bridge / push-pull circuit for automobile electric power systems,'' in *Proc. 37th IEEE Power Electronics Specialists Conference (PESC)*, 2006, pp. 1--5.

[51] E. Hiraki, K. Yamamoto, and T. Mishima, ''An isolated bidirectional dc-dc soft switching converter for super capacitor based energy storage systems,'' in *Proc. 38th IEEE Power Electronics Specialists Conference (PESC)*, 2007, pp. 390--395.

[52] T. Mishima and E. Hiraki, ''Zvs-sr bidirectional dc-dc converter for supercapacitor-applied automotive electric energy storage systems,'' in *Proc. IEEE Vehicle Power and Propulsion Conference (VPPC)*, 2005, p. 6 pp.

[53] R. W. A. A. De Doncker, D. M. Divan, and M. H. Kheraluwala, ''A three-phase soft-switched high-power-density dc-dc converter for high-power applications,'' *IEEE Trans. Ind. Appl.*, vol. 27, no. 1, pp. 63--73, 1991, 0093-9994.

[54] F. Krismer and J. W. Kolar, ''Efficiency-optimized high-current dual active bridge converter for automotive applications,'' *IEEE Trans. Ind. Electron.*, vol. 59, no. 7, pp. 2745--2760, 2012.

[55] S. Gui-Jia, F. Z. Peng, and D. J. Adams, ''Experimental evaluation of a soft-switching dc/dc converter for fuel cell vehicle applications,'' in *Proc. IEEE Power Electronics in Transportation Conference (PET)*, 2002, pp. 39--44.

[56] S. Gui-Jia and T. Lixin, ''A multiphase, modular, bidirectional, triple-voltage dc-dc converter for hybrid and fuel cell vehicle power systems,'' *IEEE Trans. Power Electron.*, vol. 23, no. 6, pp. 3035--3046, 2008.

[57] T. Lixin and S. Gui-Jia, ''An interleaved, reduced component count, multi-voltage bus dc/dc converter for fuel cell powered electric vehicle applications,'' in *Proc. 42nd IEEE Industry Applications Society Annual Meeting (IAS)*, 2007, pp. 616--621.

[58] X. Xinyu, A. M. Khambadkone, and R. Oruganti, ''A soft-switched back-to-back bi-directional dc/dc converter with a fpga based digital control for automotive applications,'' in *Proc. 33rd IEEE Annual Conference on Industrial Electronics (IECON)*, 2007, pp. 262--267.

[59] K. H. Edelmoser and F. A. Himmelstoss, ''Bidirectional dc-to-dc converter for solar battery backup applications,'' in *Proc. 35th IEEE Power Electronics Specialists Conference (PESC)*, vol. 3, 2004, pp. 2070--2074.

[60] J. Shepherd, A. Morton, and L. Spence, *Higher Electrical Engineering*, 2nd ed. Essex: Longman Scientific and Technical, 1977.

[61] T. Wildi, *Electrical Machines, Drives, and Power Systems*, 6th ed. New Jersey: Prentice Hall, 2006.

[62] J. W.H. Hayt and J. A. Buck, *Engineering Electromagnetics*, 6th ed., ser. Electrical Engineering Series. McGraw-Hill, 2001.

[63] X. Jing, F. Wang, D. Boroyevich, and S. Zhiyu, ''Single-phase vs. three-phase high density power transformers,'' in *Proc. IEEE Energy Conversion Congress and Exposition (ECCE)*, 2010, pp. 4368--4375.

[64] M. Pavlovsky, S. W. H. de Haan, and J. A. Ferreira, ''Concept of 50 kw dc/dc converter based on zvs, quasi-zcs topology and integrated thermal and electromagnetic design,'' in *Proc. IEEE European Conference on Power Electronics and Applications (EPE)*, 2005, p. 9 pp.

[65] M. N. Kheraluwala, R. W. Gascoigne, D. M. Divan, and E. D. Baumann, ''Performance characterization of a high-power dual active bridge,'' *IEEE Trans. Ind. Appl.*, vol. 28, no. 6, pp. 1294--1301, 1992, 0093-9994.

[66] J. Biela, U. Badstuebner, and J. Kolar, ''Design of a 5-kw, 1-u, 10-kw/dm3 resonant dcdc converter for telecom applications,'' *IEEE Trans. Power Electron.*, vol. 24, no. 7, pp. 1701--1710, 2009.

[67] A. Alonso, J. Sebastian, D. Lamar, M. Hernando, and A. Vazquez, ''An overall study of a dual active bridge for bidirectional dc/dc conversion,'' in *Proc. IEEE Energy Conversion Congress and Exposition (ECCE)*, 2010, pp. 1129--1135.

[68] X. Yanhui, J. Sun, and J. S. Freudenberg, ''Power flow characterization of a bidirectional galvanically isolated high-power dc-dc converter over a wide operating range,'' *IEEE Trans. Power Electron.*, vol. 25, no. 1, pp. 54--66, 2010.

[69] F. Krismer and J. W. Kolar, ''Accurate small-signal model for the digital control of an automotive bidirectional dual active bridge,'' *IEEE Trans. Power Electron.*, vol. 24, no. 12, pp. 2756--2768, 2009.

[70] G. Demetriades and H. P. Nee, ''Dynamic modeling of the dual-active bridge topology for high-power applications,'' in *Proc. 39th IEEE Power Electronics Specialists Conference (PESC)*, 2008, pp. 457--464.

[71] W. Zhan and L. Hui, ''A soft switching three-phase current-fed bidirectional dc-dc converter with high efficiency over a wide input voltage range,'' *IEEE Trans. Power Electron.*, vol. 27, no. 2, pp. 669--684, 2012.

[72] D. De and V. Ramanarayanan, ''High frequency link topology based double conversion ups system,'' in *Proc. Joint International Conference on Power Electronics, Drives and Energy Systems (PEDES)*, 2010, pp. 1--6.

[73] M. H. Kheraluwala, D. W. Novotny, and D. M. Divan, ''Design considerations for high power high frequency transformers,'' in *Proc. 21th IEEE Power Electronics Specialists Conference (PESC)*, 1990, pp. 734--742.

[74] T. Haimin, J. L. Duarte, and M. A. M. Hendrix, ''High-power three-port three-phase bidirectional dc-dc converter,'' in *Proc. 42nd IEEE Industry Applications Society Annual Meeting (IAS)*, 2007, pp. 2022--2029.

[75] U. Badstuebner, J. Biela, and J. Kolar, ''Power density and efficiency optimization of resonant and phase-shift telecom dc-dc converters,'' in *Proc. 23rd IEEE Annual Applied Power Electronics Conference and Exposition (APEC)*, 2008, pp. 311 --317.

[76] L. Heinemann, ''An actively cooled high power, high frequency transformer with high insulation capability,'' in *Proc. 17th IEEE Annual Applied Power Electronics Conference and Exposition (APEC)*, vol. 1, 2002, pp. 352--357.

[77] J. Jacobs, A. Averberg, and R. De Doncker, ''A novel three-phase dc/dc converter for high-power applications,'' in *Proc. 35th IEEE Power Electronics Specialists Conference (PESC)*, vol. 3, 2004, pp. 1861--1867.

[78] D. M. Divan and G. Skibinski, ''Zero-switching-loss inverters for high-power applications,'' *IEEE Trans. Ind. Appl.*, vol. 25, no. 4, pp. 634--643, 1989.

[79] J. A. Sabate, V. Vlatkovic, R. B. Ridley, F. C. Lee, and B. H. Cho, ''Design considerations for high-voltage high-power full-bridge zero-voltage-switched pwm converter,'' in *Proc. 5th IEEE Annual Applied Power Electronics Conference and Exposition (APEC)*, V. Vlatkovic, Ed., 1990, pp. 275--284.

[80] R. W. De Doncker, D. M. Divan, and M. H. Kheraluwala, ''A three-phase soft-switched high power density dc/dc converter for high power applications,'' in *Proc. 23rd IEEE Industry Applications Society Annual Meeting (IAS)*, vol. 1, 1988, pp. 796--805.

[81] T. Haimin, A. Kotsopoulos, J. L. Duarte, and M. A. M. Hendrix, ‘‘Transformer-coupled multiport zvs bidirectional dc-dc converter with wide input range,’’ *IEEE Trans. Power Electron.*, vol. 23, no. 2, pp. 771--781, 2008, 0885-8993.

[82] B. P. McGrath, D. G. Holmes, P. J. McGoldrick, and A. D. McIver, ‘‘Design of a soft-switched 6-kw battery charger for traction applications,’’ *IEEE Trans. Power Electron.*, vol. 22, no. 4, pp. 1136--1144, 2007, 0885-8993.

[83] J. M. Zhang, D. M. Xu, and Q. Zhaoming, ‘‘An improved dual active bridge dc/dc converter,’’ in *Proc. 32nd IEEE Power Electronics Specialists Conference (PESC)*, vol. 1, 2001, pp. 232--236.

[84] P. Imbertson and N. Mohan, ‘‘Asymmetrical duty cycle permits zero switching loss in pwm circuits with no conduction loss penalty,’’ *IEEE Trans. Ind. Appl.*, vol. 29, no. 1, pp. 121--125, 1993, 0093-9994.

[85] G. G. Oggier, G. O. Garcia, and A. R. Oliva, ‘‘Switching control strategy to minimize dual active bridge converter losses,’’ *IEEE Trans. Power Electron.*, vol. 24, no. 7, pp. 1826--1838, 2009.

[86] H. Tao, J. Duarte, and M. Hendrix, ‘‘Novel zero-voltage switching control methods for a multiple-input converter interfacing a fuel cell and supercapacitor,’’ in *Proc. 32nd IEEE Annual Conference on Industrial Electronics (IECON)*, 2006, pp. 2341 --2346.

[87] L. Hui, Z. P. Fang, and J. S. Lawler, ‘‘A natural zvs medium-power bidirectional dc-dc converter with minimum number of devices,’’ *IEEE Trans. Ind. Appl.*, vol. 39, no. 2, pp. 525--535, 2003, 0093-9994.

[88] F. Z. Peng, L. Hui, S. Gui-Jia, and J. S. Lawler, ‘‘A new zvs bidirectional dc-dc converter for fuel cell and battery application,’’ *IEEE Trans. Power Electron.*, vol. 19, no. 1, pp. 54--65, 2004, 0885-8993.

[89] L. Zhu, ‘‘A novel soft-commutating isolated boost full-bridge zvs-pwm dc-dc converter for bidirectional high power applications,’’ *IEEE Trans. Power Electron.*, vol. 21, no. 2, pp. 422--429, 2006, 0885-8993.

[90] X. Dehong, Z. Chuanhong, and F. Haifeng, ‘‘A pwm plus phase-shift control bidirectional dc-dc converter,’’ *IEEE Trans. Power Electron.*, vol. 19, no. 3, pp. 666--675, 2004, 0885-8993.

[91] C. Huang-Jen and L. Li-Wei, ‘‘A bidirectional dc-dc converter for fuel cell electric vehicle driving system,’’ *IEEE Trans. Power Electron.*, vol. 21, no. 4, pp. 950--958, 2006.

[92] L. Rongyuan, A. Pottharst, N. Frohleke, and J. Bocker, ‘‘Analysis and design of improved isolated full-bridge bidirectional dc-dc converter,’’ in *Proc. 35th IEEE Power Electronics Specialists Conference (PESC)*, vol. 1, 2004, pp. 521--526.

[93] Q. Zhao, Y. Xu, X. Jin, W. Wu, and L. Cao, ‘‘Dsp-based closed-loop control of bi-directional voltage mode high frequency link inverter with active clamp,’’ in *Proc. 40th IEEE Industry Applications Society Annual Meeting (IAS)*, vol. 2, 2005, pp. 928--933.

[94] F. Cavalcante and J. Kolar, ‘‘Small-signal model of a 5kw high-output voltage capacitive-loaded series-parallel resonant dc-dc converter,’’ in *Proc. 36th IEEE Power Electronics Specialists Conference (PESC)*, 2005, pp. 1271 --1277.

[95] W. Xinke, Z. Chen, Z. Junming, and Q. Zhaoming, ‘‘A novel phase shift controlled zvzcs full bridge dc-dc converter: analysis and design considerations,’’ in *Proc. 39th IEEE Industry Applications Society Annual Meeting (IAS)*, vol. 3, 2004, pp. 1790--1796.

[96] F. Krismer, J. Biela, and J. W. Kolar, ‘‘A comparative evaluation of isolated bi-directional dc/dc converters with wide input and output voltage range,’’ in *Proc. 40th IEEE Industry Applications Society Annual Meeting (IAS)*, vol. 1, 2005, pp. 599--606.

[97] J. Walter and R. W. De Doncker, ‘‘High-power galvanically isolated dc/dc converter topology for future automobiles,’’ in *Proc. 34th IEEE Power Electronics Specialists Conference (PESC)*, vol. 1, 2003, pp. 27--32.

[98] F. Krismer, S. Round, and J. W. Kolar, ‘‘Performance optimization of a high current dual active bridge with a wide operating voltage range,’’ in *Proc. 37th IEEE Power Electronics Specialists Conference (PESC)*, 2006, pp. 1--7.

[99] G. Goodwin, S. Graebe, and M. Salgado, *Control System Design*. Prentice Hall, 2001.

[100] G. Franklin, J. Powell, and W. M.L., *Digital Control of Dynamic Systems*, 3rd ed. California: Addison-Wesley Longman, 1998.

[101] G. Love, ‘‘Small signal modelling of power electronic converters, for the study of time-domain waveforms, harmonic domain spectra, and control interactions,’’ Ph.D. dissertation, University of Cantebury, 2007.

[102] A. Sedra and S. K.C., *Microelectronic Circuits*, 6th ed. New York: Oxford University Press, 2011.

[103] B. Johansson, ‘‘Dc-dc converters - dynamic model design and experimental verification,’’ Ph.D. dissertation, Lund University, 2004.

[104] S. Inoue and H. Akagi, ‘‘A bidirectional dc-dc converter for an energy storage system with galvanic isolation,’’ *IEEE Trans. Power Electron.*, vol. 22, no. 6, pp. 2299--2306, 2007, 0885-8993.

[105] F. Krismer and J. Kolar, ‘‘Closed form solution for minimum conduction loss modulation of dab converters,’’ *IEEE Trans. Power Electron.*, vol. 27, no. 1, pp. 174--188, 2011.

[106] S. Liping, X. Dehong, C. Min, and Z. Xuancai, ‘‘Dynamic model of pwm plus phase-shift (pps) control bidirectional dc-dc converters,’’ in *Proc. 40th IEEE Industry Applications Society Annual Meeting (IAS)*, vol. 1, 2005, pp. 614--619.

[107] H. Bai, C. Mi, C. Wang, and S. Gargies, ‘‘The dynamic model and hybrid phase-shift control of a dual-active-bridge converter,’’ in *Proc. 34th IEEE Annual Conference on Industrial Electronics (IECON)*, 2008, pp. 2840--2845.

[108] D. D. M. Cardozo, J. C. Balda, D. Trowler, and H. A. Mantooth, ‘‘Novel nonlinear control of dual active bridge using simplified converter model,’’ in *Proc. 25th IEEE Annual Applied Power Electronics Conference and Exposition (APEC)*, 2010, p. 7 pp.

[109] Z. Chuanhong, S. D. Round, and J. W. Kolar, ‘‘An isolated three-port bidirectional dc-dc converter with decoupled power flow management,’’ *IEEE Trans. Power Electron.*, vol. 23, no. 5, pp. 2443--2453, 2008.

[110] H. Akagi and R. Kitada, ‘‘Control and design of a modular multilevel cascade btb system using bidirectional isolated dc-dc converters,’’ *IEEE Trans. Power Electron.*, vol. 26, no. 9, pp. 2457 -- 2464, 2011.

[111] B. Hua, C. C. Mi, and S. Gargies, ‘‘The short-time-scale transient processes in high-voltage and high-power isolated bidirectional dc-dc converters,’’ *IEEE Trans. Power Electron.*, vol. 23, no. 6, pp. 2648--2656, 2008.

[112] H. Khalil, *Nonlinear Systems.* Prentice Hall, 1996.

[113] H. Z., T. D., S. T.S., and A. Khambadkone, ‘‘Interleaved bi-directional dual active bridge dc-dc converter for interfacing ultracapacitor in micro-grid application,’’ in *Proc. 2010 IEEE International Symposium on Industrial Electronics (ISIE)*, 2010, pp. 2229--2234.

[114] M. Fliess, J. Levine, P. Martin, and P. Rouchon, ‘‘Flatness and defect of non-linear systems: Introductory theory and examples,’’ *Int. Journal of Control*, vol. 61, pp. 1327--1361, 1995.

[115] M. van Nieuwstadt, M. Rathinam, and R. Murray, ‘‘Differential flatness and absolute equivalence,’’ in *Proc. 33rd IEEE Conf. on Decision and Control*, vol. 1, 1994, pp. 326 --332.

[116] M. Phattanasak, R. Gavagsaz-Ghoachani, J. P. Martin, S. Pierfederici, and B. Davat, ‘‘Flatness based control of an isolated three-port bidirectional dc-dc converter for a fuel cell hybrid source,’’ in *Proc. IEEE Energy Conversion Congress and Exposition (ECCE)*, 2011, pp. 977--984.

[117] M. Phattanasak, R. Gavagsaz-Ghoachani, J. Martin, B. Nahid-Mobarakeh, S. Pierfederici, and B. Davat, ‘‘Comparison of two nonlinear control strategies for a hybrid source system using an isolated three-port bidirectional dc-dc converter,’’ in *Proc. IEEE Vehicle Power and Propulsion Conference (VPPC)*, 2011, pp. 1--6.

[118] W. Kunrong, Z. Lizhi, Q. Dayu, H. Odendaal, J. Lai, and F. C. Lee, ‘‘Design, implementation, and experimental results of bi-directional full-bridge dc/dc converter with unified soft-switching scheme and soft-starting capability,’’ in *Proc. 31st IEEE Power Electronics Specialists Conference (PESC)*, vol. 2, 2000, pp. 1058--1063.

[119] S. Lei, S. Liping, X. Dehong, and C. Min, ‘‘Optimal design and control of 5kw pwm plus phase-shift (pps) control bidirectional dc-dc converter,’’ in *Proc. 21st IEEE Annual Applied Power Electronics Conference and Exposition (APEC)*, 2006, p. 5 pp.

[120] M. Rosekeit and R. W. De Doncker, ‘‘Smoothing power ripple in single phase chargers at minimized dc-link capacitance,’’ in *Proc. IEEE Energy Conversion Congress and Exposition Asia (ECCE Asia)*, 2011, pp. 2699--2703.

[121] A. Watson, P. Wheeler, and J. Clare, ‘‘Field programmable gate array based control of dual active bridge dc/dc converter for the uniflex-pm project,’’ in *Proc. IEEE European Conference on Power Electronics and Applications (EPE)*, 2011, pp. 1 --9.

[122] S. Buso and P. Mattavelli, *Digital Control in Power Electronics*, 1st ed. Morgan and Claypool, 2006.

[123] T. A. Meynard, M. Fadel, and N. Aouda, ‘‘Modeling of multilevel converters,’’ *IEEE Trans. Ind. Electron.*, vol. 44, no. 3, pp. 356--364, 1997, 0278-0046.

[124] G. James, D. Burley, D. Clements, P. Dyke, and J. Searl, *Modern Engineering Mathematics*. Pearson Prentice Hall, 2007.

[125] H. Akagi and R. Kitada, ‘‘Control of a modular multilevel cascade btb system using bidirectional isolated dc/dc converters,’’ in *Proc. IEEE Energy Conversion Congress and Exposition (ECCE)*, 2010, pp. 3549--3555.

[126] C. Mi, H. Bai, C. Wang, and S. Gargies, ‘‘Operation, design and control of dual h-bridge-based isolated bidirectional dc-dc converter,’’ *IET Power Electron.*, vol. 1, no. 4, pp. 507--517, 2008.

[127] B. Hua, N. Ziling, and C. C. Mi, ‘‘Experimental comparison of traditional phase-shift, dual-phase-shift, and model-based control of isolated bidirectional dc-dc converters,’’ *IEEE Trans. Power Electron.*, vol. 25, no. 6, pp. 1444--1449, 2010.

[128] F. Krismer and J. W. Kolar, ‘‘Accurate small-signal model for an automotive bidirectional dual active bridge converter,’’ in *Proc. 11th IEEE Workshop on Control and Modeling for Power Electronics (COMPEL)*, 2008, pp. 1--10.

[129] D. G. Holmes, T. A. Lipo, B. P. McGrath, and W. Y. Kong, ‘‘Optimized design of stationary frame three phase ac current regulators,’’ *IEEE Trans. Power Electron.*, vol. 24, no. 11, pp. 2417--2426, 2009.

[130] K. Lee, T. M. Jahns, G. Venkataramanan, and W. E. Berkopec, ‘‘Dc-bus electrolytic capacitor stress in adjustable-speed drives under input voltage unbalance and sag conditions,’’ *IEEE Trans. Ind. Appl.*, vol. 43, no. 2, pp. 495--504, 2007.

[131] G. Buiatti, S. Cruz, and A. Cardoso, ‘‘Lifetime of film capacitors in single-phase regenerative induction motor drives,’’ in *IEEE International Symposium on Diagnostics for Electric Machines, Power Electronics and Drives (SDEMPED)*, 2007, pp. 356--362.

[132] P. Inc., *PSIM User Manual*. PowerSim Inc., 2001.

[133] C. P/L, *Modu-T1 Moduconverter Push-Pull Transformer*. CPT P/L, 2006.

[134] C. P/L, *CPT-DA2810 Card TMS320F2810 DSP Controller Card Technical Manual*. CPT P/L, 2009.

[135] C. P/L, *CPT-Mini2810 Card Technical Manual*. CPT P/L, 2010.

[136] C. P/L, *CS-GIIB Technical Manual*. CPT P/L, 2008.

[137] T. Instruments, *C28x IQmath Library*. Texas Instruments, 2009.