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 Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.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
# TABLE OF CONTENTS

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
### TABLE OF CONTENTS

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
## TABLE OF CONTENTS

### 6 Description of Simulated & Experimental Systems  
6.1 Simulated Systems  
6.1.1 Overview  
6.1.2 Power Stage  
6.1.3 Modulators  
6.1.4 Controllers  
6.2 Experimental Prototype  
6.2.1 Overview  
6.2.2 Power Stage  
6.2.3 Controller Hardware  
6.2.4 Inter-GIIB Communication  
6.3 Summary  

### 7 Simulation & Experimental Results  
7.1 Overview  
7.2 Steady-state Operating Waveforms  
7.3 Open Loop Transients  
7.4 Closed Loop Transients  
7.4.1 Voltage Reference Step  
7.4.2 Load Change  
7.4.3 AC Load  
7.5 Summary  

### 8 Conclusions  
8.1 Contributions  
8.1.1 Harmonic Model  
8.1.2 Deadtime Modelling  
8.1.3 Maximised controller gains  
8.1.4 Adaptive controller gains
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  
2.2 A simple Flyback Converter  
2.3 Flyback Converter Operating Waveforms  
2.4 The Actively Clamped Bi-directional Flyback Converter  
2.5 A Current fed push-pull converter  
2.6 CFPP Operating Waveforms  
2.7 Bi-directional CFPP  
2.8 Phase Leg Topology & Operating Waveforms  
2.9 Bridge Converter Topologies  
2.10 Bi-directional half-bridge  
2.11 Dual Active Bridge Converters  
2.12 H-bridge and Modulator  
2.13 Pulse Width Modulation  
2.14 Block Modulation  
2.15 Switching Loss  
2.16 Ideal Soft-switching Waveforms  
2.17 Parallel Capacitance  
2.18 Active Clamp  
2.19 Series Resonance  
2.20 Buck Converter  
2.21 The DAB Converter & Operating Waveforms
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
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.18</td>
<td>Operating point dependence of the deadtime effect</td>
<td>68</td>
</tr>
<tr>
<td>3.19</td>
<td>Single Phase Leg with an Inductive Load</td>
<td>69</td>
</tr>
<tr>
<td>3.20</td>
<td>Deadtime Effect in a Phase Leg</td>
<td>70</td>
</tr>
<tr>
<td>3.21</td>
<td>Deadtime influence - HV bridge lags the LV bridge</td>
<td>72</td>
</tr>
<tr>
<td>3.22</td>
<td>Deadtime influence - HV bridge leads the LV bridge</td>
<td>73</td>
</tr>
<tr>
<td>3.23</td>
<td>Deadtime influence in the DAB converter</td>
<td>77</td>
</tr>
<tr>
<td>3.24</td>
<td>Block Diagram of Final Dynamic Model</td>
<td>77</td>
</tr>
<tr>
<td>3.25</td>
<td>Validating the Final Model $(N = 3)$</td>
<td>78</td>
</tr>
<tr>
<td>4.1</td>
<td>Basic closed loop block diagram of the DAB converter</td>
<td>80</td>
</tr>
<tr>
<td>4.2</td>
<td>Ideal Bode plot</td>
<td>82</td>
</tr>
<tr>
<td>4.3</td>
<td>Transient Responses of Ideal &amp; Digital implementations</td>
<td>83</td>
</tr>
<tr>
<td>4.4</td>
<td>Digital PSSW Modulator</td>
<td>84</td>
</tr>
<tr>
<td>4.5</td>
<td>Sample &amp; Hold</td>
<td>85</td>
</tr>
<tr>
<td>4.6</td>
<td>Controller Calculation &amp; Update</td>
<td>86</td>
</tr>
<tr>
<td>4.7</td>
<td>Closed loop block diagram - Including Transport Delay</td>
<td>86</td>
</tr>
<tr>
<td>4.8</td>
<td>Forward Path Bode Plot - Including Transport Delay</td>
<td>87</td>
</tr>
<tr>
<td>4.9</td>
<td>Linearised Transient Responses</td>
<td>87</td>
</tr>
<tr>
<td>4.10</td>
<td>$B_\delta$ term variation with operating phase shift</td>
<td>90</td>
</tr>
<tr>
<td>4.11</td>
<td>Closed Loop Block diagram of the DAB converter with an Adaptive PI controller</td>
<td>91</td>
</tr>
<tr>
<td>4.12</td>
<td>Closed loop Step Response Comparison</td>
<td>91</td>
</tr>
<tr>
<td>4.13</td>
<td>Comparison of Load &amp; Reference Transient Responses</td>
<td>92</td>
</tr>
<tr>
<td>4.14</td>
<td>DAB Block Diagram (with Disturbance)</td>
<td>94</td>
</tr>
<tr>
<td>4.15</td>
<td>Pole Zero map of Closed Loop Transfer Functions</td>
<td>94</td>
</tr>
<tr>
<td>4.16</td>
<td>DAB Closed Loop Block Diagram with Feed-forward</td>
<td>95</td>
</tr>
<tr>
<td>4.17</td>
<td>Final Closed Loop Block Diagram of the DAB Converter</td>
<td>96</td>
</tr>
<tr>
<td>4.18</td>
<td>Load Step Response - Without Feed-forward</td>
<td>97</td>
</tr>
<tr>
<td>4.19</td>
<td>Load Step Response - With Feed-forward</td>
<td>99</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

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 → 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 → 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
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.9</td>
<td>Experimental DAB Converter</td>
<td>127</td>
</tr>
<tr>
<td>6.10</td>
<td>Experimental High Frequency AC Inductor</td>
<td>128</td>
</tr>
<tr>
<td>6.11</td>
<td>Experimental High Frequency Transformer</td>
<td>128</td>
</tr>
<tr>
<td>6.12</td>
<td>Experimental Load Circuit Configuration</td>
<td>129</td>
</tr>
<tr>
<td>6.13</td>
<td>Experimental Load Elements</td>
<td>129</td>
</tr>
<tr>
<td>6.14</td>
<td>6 phase leg Circuit Diagram</td>
<td>130</td>
</tr>
<tr>
<td>6.15</td>
<td>PCB DC Bus Structure</td>
<td>130</td>
</tr>
<tr>
<td>6.16</td>
<td>Experimental 6 phase leg IGBT platform</td>
<td>131</td>
</tr>
<tr>
<td>6.17</td>
<td>DA2810 DSP Controller Board</td>
<td>133</td>
</tr>
<tr>
<td>6.18</td>
<td>Mini2810 Controller Board</td>
<td>134</td>
</tr>
<tr>
<td>6.19</td>
<td>GIIB Inverter Board</td>
<td>135</td>
</tr>
<tr>
<td>6.20</td>
<td>Linked GIIB Boards</td>
<td>136</td>
</tr>
<tr>
<td>6.21</td>
<td>Synchronisation</td>
<td>137</td>
</tr>
<tr>
<td>7.1</td>
<td>Experimental Setup</td>
<td>140</td>
</tr>
<tr>
<td>7.2</td>
<td>DAB Steady State Operating Waveforms</td>
<td>141</td>
</tr>
<tr>
<td>7.3</td>
<td>Deadtime Effect - HV bridge Lagging the LV bridge</td>
<td>142</td>
</tr>
<tr>
<td>7.4</td>
<td>Deadtime Effect - HV bridge Leading the LV bridge</td>
<td>143</td>
</tr>
<tr>
<td>7.5</td>
<td>Open Loop: Deadtime</td>
<td>144</td>
</tr>
<tr>
<td>7.6</td>
<td>Open Loop: No Deadtime</td>
<td>145</td>
</tr>
<tr>
<td>7.7</td>
<td>Fixed PI</td>
<td>147</td>
</tr>
<tr>
<td>7.8</td>
<td>Adaptive PI</td>
<td>148</td>
</tr>
<tr>
<td>7.9</td>
<td>Reducing Load</td>
<td>149</td>
</tr>
<tr>
<td>7.10</td>
<td>Increasing Load</td>
<td>150</td>
</tr>
<tr>
<td>7.11</td>
<td>Steady State AC Load</td>
<td>152</td>
</tr>
<tr>
<td>7.12</td>
<td>AC Step Change (4A ⇋ 6A)</td>
<td>153</td>
</tr>
<tr>
<td>7.13</td>
<td>AC Step Change (6A ⇋ 4A)</td>
<td>154</td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th><strong>RS-232</strong></th>
<th>Serial Communication Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SC</strong></td>
<td>Supercapacitor</td>
</tr>
<tr>
<td><strong>SMPS</strong></td>
<td>Switchmode Power Supplies</td>
</tr>
<tr>
<td><strong>SPI</strong></td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td><strong>TAB</strong></td>
<td>Triple Active Bridge</td>
</tr>
<tr>
<td><strong>THD</strong></td>
<td>Total Harmonic Distortion</td>
</tr>
<tr>
<td><strong>TTL</strong></td>
<td>Truth Table Logic</td>
</tr>
<tr>
<td><strong>UC</strong></td>
<td>Ultracapacitor</td>
</tr>
<tr>
<td><strong>UPS</strong></td>
<td>Uninterruptible Power Supply</td>
</tr>
<tr>
<td><strong>VSI</strong></td>
<td>Voltage Source Inverter</td>
</tr>
<tr>
<td><strong>ZCS</strong></td>
<td>Zero Current Switching</td>
</tr>
<tr>
<td><strong>ZIR</strong></td>
<td>Zero Impulse Response</td>
</tr>
<tr>
<td><strong>ZOH</strong></td>
<td>Zero Order Hold</td>
</tr>
<tr>
<td><strong>ZSR</strong></td>
<td>Zero State Response</td>
</tr>
<tr>
<td><strong>ZVS</strong></td>
<td>Zero Voltage Switching</td>
</tr>
</tbody>
</table>
List of Symbols

α  Relative phase angle
δ  Phase shift
δ₀  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
φ_z[n]  Impedance angle at nᵗʰ 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_{load₀}  Load current Q-point
K_p  Proportional Gain
m  Modulation Depth
n  Harmonic number
N  Number of significant harmonics considered
N_p, N_s  Transformer turns ratio
S_k  Phase leg switch state
S_k(t)  Time-domain representation of S_k
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_d$</td>
<td>Delay time</td>
</tr>
<tr>
<td>$T_p$</td>
<td>Plant time constant ((\frac{1}{A}))</td>
</tr>
<tr>
<td>$T_r$</td>
<td>Integrator reset time</td>
</tr>
<tr>
<td>$T_s$</td>
<td>Switching period</td>
</tr>
<tr>
<td>$V_{out}$</td>
<td>Output Voltage</td>
</tr>
<tr>
<td>$V_{out_0}$</td>
<td>Output Voltage Q-point</td>
</tr>
<tr>
<td>$V_{ref}$</td>
<td>Voltage reference command</td>
</tr>
<tr>
<td>$Z[n]$</td>
<td>Impedance at (n^{th}) harmonic</td>
</tr>
<tr>
<td>$</td>
<td>Z[n]</td>
</tr>
</tbody>
</table>
CHAPTER 1. INTRODUCTION

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


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](image)

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](image)

**Figure 2.2: A simple Flyback Converter**

![Figure 2.3: Flyback Converter Operating Waveforms](image)

**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.
CHAPTER 2. LITERATURE REVIEW

This circuit is called an \textit{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\textsuperscript{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.

\textsuperscript{1} Soft-switching concepts are discussed in Section 2.2.
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\footnote{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]](image)

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 \( V_{\text{out}} \) to switch between the upper and lower voltage rails \( +V_{\text{DC}} \) and \( V_{\text{DC}} \) \([10,12]\).

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 \([10--12]\).

![Phase Leg Structure](image1)

*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...
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 (\(C_1 \sim C_4\) 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].

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.

\[ L \]
\[ S1 \quad S2 \quad S4 \quad S3 \]
\[ S1 \quad S2 \quad S3 \quad S4 \]
\[ Vin \]
\[ Np \quad Ns \]
\[ Vout \]
\[ S1 \quad S2 \quad S3 \quad S4 \quad S5 \quad S6 \]
\[ Vout \]
\[ L \quad Three-phase \quad Tx \]

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 efficiency4, 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.

<table>
<thead>
<tr>
<th>Flyback Converter</th>
<th>Current-fed Push-pull Converter (CFPP)</th>
<th>Bridge Converters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voltage Rating</td>
<td>Low (&lt;100V)</td>
<td>Low (&lt;400V)</td>
</tr>
<tr>
<td>Power Rating</td>
<td>Low (≈500W)</td>
<td>Medium (&gt;2 kW)</td>
</tr>
</tbody>
</table>

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,
CHAPTER 2. LITERATURE REVIEW

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
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.
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
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.

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.

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 \(V_{\text{res}}\). 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 \(C_{\text{series}}\) in series with the transformer leakage inductance \(L_{\text{leak}}\). The switching processes of the bridge converter excites this resonance, resulting in an oscillatory output waveform \(V_{\text{out}}\). 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 \(C_{\text{series}}\) must withstand the rated voltage and current of the converter. As converter ratings rise, the size and cost of this capacitor becomes prohibitively large.
The literature in this area focuses on several different aspects of this converter, e.g. dynamic modelling and control\textsuperscript{5} \cite{23,90,94,95}, magnetics design \cite{76}, maximising efficiency and power density \cite{66,75,77,96,97}, as well as their applications, such as electric vehicles \cite{77,97} and telecoms applications \cite{66,75}.

Although the potential advantages of soft-switching are compelling, it has some significant drawbacks. \cite{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 \cite{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. \cite{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.

\textsuperscript{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\textsuperscript{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.

\textsuperscript{6} Differential equations and difference equations are used for continuous-time and discrete time systems, respectively.
CHAPTER 2. LITERATURE REVIEW

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 \( (i_L) \).

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

\(^7\) The load current input is included to model the effect of load variation.
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:

\[
\begin{aligned}
\dot{x}(t) &= A_n x(t) + B_n u(t) \\
y(t) &= C_n x(t)
\end{aligned}
\quad n = \text{ON or OFF (2.1)}
\]

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.:
\[ \dot{x}(t) = A'x(t) + B'u(t) \]
\[ y(t) = C'x(t) \]  

(2.2)

where

\[ A' = D(t)A_{ON} + \{1 - D(t)\}A_{OFF}, \]

\[ B' = D(t)B_{ON} + \{1 - D(t)\}B_{OFF}, \]

\[ C' = D(t)C_{ON} + \{1 - D(t)\}C_{OFF}. \]

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

\[ u'(t) = \begin{bmatrix} u(t) \\ D(t) \end{bmatrix} \]  

(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 \[ [99], \text{i.e.} : \]

\[ x(t) = x_0 + \hat{x} \]
\[ u'(t) = u'_0 + \hat{u}' \]
\[ y(t) = y_0 + \hat{y} \]  

(2.4)

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

\[ \dot{\hat{x}} = A'\hat{x}(t) + B'\hat{u}'(t) \]
\[ \hat{y} = C'\hat{x}(t) \]  

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

**Table 2.2: State Averaged Models**

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

![DAB Converter Structure](image1)

(a) DAB Converter Structure

![DAB Operating Waveforms](image2)

(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
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 \( \delta \) 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:

\[
V_1 = \frac{V_{1pk}}{\sqrt{2}} \sin(\omega t) \quad \therefore \quad V_1(t) = \frac{V_{1pk}(t)}{\sqrt{2}} \sin(\omega t) \tag{2.7a}
\]

\[
V_2 = \frac{V_{2pk}}{\sqrt{2}} \sin(\omega t) \quad \therefore \quad V_2(t) = \frac{V_{2pk}(t)}{\sqrt{2}} \sin(\omega t) \tag{2.7b}
\]

where \( V_{1pk} \) & \( V_{2pk} \) are the peak machine voltages and \( \omega \) is the fundamental frequency.

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

\[
P_{DC}(t) = \frac{V_1(t) V_2(t) \sin \delta(t)}{\omega L} \tag{2.8}
\]
CHAPTER 2. LITERATURE REVIEW

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

$$P_{\text{DC}}(t) = V_{\text{out}}(t) i_{\text{out}}(t) = \frac{V_1(t) V_2(t) \sin \delta(t)}{\omega L}$$

$$\therefore i_{\text{out}}(t) = \frac{\sqrt{2} V_1(t) \sin \delta(t)}{\omega L}$$

(2.9)

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

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

(2.10)

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

$$\frac{dV_{\text{out}}(t)}{dt} = \frac{1}{C}(i_{\text{out}}(t) - i_{\text{load}}(t))$$

(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 its 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
CHAPTER 2. LITERATURE REVIEW

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](image)

**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]:

\[
i_{\text{out}}(t) = \frac{V_{\text{in}}(t)}{2Lf_{\text{sw}}} \alpha(t)(1 - \alpha(t))
\] (2.12)

where \( f_{\text{sw}} \) 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.
\[
\frac{dV_{\text{out}}(t)}{dt} = \frac{1}{C} i_C = \frac{1}{C} (-i_{\text{load}}(t) + i_{\text{out}}(t)) \\
= -\frac{1}{RC} V_{\text{out}}(t) + \frac{1}{C} \left( \frac{V_{\text{in}}(t)}{2L_{f_{sw}}} \alpha(t) (1 - \alpha(t)) \right)
\] (2.13)

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

\[
V_{\text{aux}}(t) = V_{\text{in}}(t) \alpha(t) (1 - \alpha(t))
\] (2.14)

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

\[
\frac{dV_{\text{out}}(t)}{dt} = -\frac{1}{RC} V_{\text{out}}(t) + \frac{1}{2CL_{f_{sw}}} V_{\text{aux}}(t)
\] (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](image)

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](image)

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.
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.
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:

$$H_{PI}(s) = K_p \left(1 + \frac{1}{sT_r}\right)$$  \hspace{1cm} (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](image)

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.

<table>
<thead>
<tr>
<th>Circuit Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC Input Voltage ( (V_{in}) )</td>
<td>200 V</td>
</tr>
<tr>
<td>DC Output Voltage (Nominal) ( (V_{out}) )</td>
<td>200 V</td>
</tr>
<tr>
<td>DC Capacitance ( (C) )</td>
<td>20 ( \mu F )</td>
</tr>
<tr>
<td>AC Inductance ( (L) )</td>
<td>50 ( \mu H )</td>
</tr>
<tr>
<td>AC Resistance ( (R_L) )</td>
<td>0.1 ( \Omega )</td>
</tr>
<tr>
<td>Transformer Turns Ratio ( (N_{pri} : N_{sec}) )</td>
<td>10 : 15</td>
</tr>
<tr>
<td>Switching Frequency ( (f_s) )</td>
<td>20 kHz</td>
</tr>
<tr>
<td>Deadtime ( (t_{DT}) )</td>
<td>1.5 ( \mu s )</td>
</tr>
<tr>
<td>Nominal Output Power ( (P_{out}) )</td>
<td>3 kW</td>
</tr>
</tbody>
</table>

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](image)

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.

\[
\begin{array}{c|c}
S_1 & V_{out} \\
0 & -V_{DC} \\
1 & +V_{DC} \\
\end{array}
\]

\textbf{Figure 3.2: Phase Leg Structure}

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:

\[
V_{out} = 2V_{DC} \{S_1 - S_2\} \quad (3.1)
\]

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

47
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.

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:

\[ S_k(t) \in \{0, 1\} \; \text{where} \; k = 1, 2, \ldots \]  

\[ (3.2) \]
CHAPTER 3. CONVERTER MODELLING

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

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

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 \( V_{\text{Pri}} (t) \) & \( V_{\text{Sec}} (t) \) can be described in terms of the converter switching functions as:

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

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

Figure 3.6: DAB Operating Waveforms

The two bridge output voltages \( V_{\text{Pri}} (t) \) & \( V_{\text{Sec}} (t) \) are offset from each other by a phase difference \( \delta \). 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.
CHAPTER 3. CONVERTER MODELLING

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

\[
\frac{dV_{out}(t)}{dt} = \frac{i_C(t)}{C}
\]  \hspace{1cm} (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:

\[
i_C(t) = i_{DC}(t) - i_{load}(t)
\]  \hspace{1cm} (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.

<table>
<thead>
<tr>
<th>( S_3 )</th>
<th>( S_4 )</th>
<th>( i_{DC} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>(-i_L)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>(i_L)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**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:

\[
i_{DC}(t) = i_L(t) \{S_3(t) - S_4(t)\}
\]  \hspace{1cm} (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.
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:

\[ R_li_L(t) + L \frac{di_L}{dt}(t) = V_{in} \{S_1(t) - S_2(t)\} - \frac{N_p}{N_s}V_{out}(t) \{S_3(t) - S_4(t)\} \] (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]:
CHAPTER 3. CONVERTER MODELLING

\[ f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left\{ a_n \cos(nx) + b_n \sin(nx) \right\} \] (3.10)

where the harmonic coefficients \( a_n \) & \( b_n \) are defined as:

\[ a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(nx) \, dx, \quad n \geq 0 \] (3.11a)

\[ b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(nx) \, dx, \quad n \geq 1 \] (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]:

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

where \( \omega_s \) is the switching frequency of the square wave (in rad/s) and \( \alpha_k \) 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:

\[ S_k(t) = \frac{1}{2} + \frac{2}{\pi} \sum_{n=0}^{N} \frac{\sin \left( [2n+1] \{\omega_s t - \alpha_k\} \right)}{[2n+1]}, \quad N \geq 0, \quad k = 1, 2, 3 \ldots \] (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 \( \pi \), and the phase shift between the primary and secondary bridges is defined as \( \delta \).
This gives the following switching functions:

\[ S_1(t) = \frac{1}{2} + \frac{2}{\pi} \sum_{n=0}^{N} \frac{\sin((2n + 1) \{\omega_s t\})}{2n + 1} \]  
\[ S_2(t) = \frac{1}{2} + \frac{2}{\pi} \sum_{n=0}^{N} \frac{\sin((2n + 1) \{\omega_s t - \pi\})}{2n + 1} \]  
\[ S_3(t) = \frac{1}{2} + \frac{2}{\pi} \sum_{n=0}^{N} \frac{\sin((2n + 1) \{\omega_s t - \delta\})}{2n + 1} \]  
\[ S_4(t) = \frac{1}{2} + \frac{2}{\pi} \sum_{n=0}^{N} \frac{\sin((2n + 1) \{\omega_s t - \delta - \pi\})}{2n + 1} \]  

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.

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

(3.15)

where $\delta$ 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.

![DAB Equivalent Circuit](a) DAB Equivalent Circuit

![Synchronous Machine 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:

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

(3.16)

This equation shows that the real power in the DAB converter is primarily determined by the phase shift ($\delta$) between the two voltage sources.
CHAPTER 3. CONVERTER MODELLING

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 \( V_{\text{Pri}}(t) \) & \( V_{\text{Sec}}(t) \) 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:

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

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

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

\[
V_{\text{Pri RMS}} = V_{\text{in}} \frac{4}{\pi} \frac{1}{\sqrt{2}} \frac{1}{[2n + 1]} \quad (3.18a)
\]

\[
V_{\text{out RMS}} = V_{\text{out}} \frac{4}{\pi} \frac{1}{\sqrt{2}} \frac{1}{[2n + 1]} \quad (3.18b)
\]

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:

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

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.
CHAPTER 3. CONVERTER MODELLING

<table>
<thead>
<tr>
<th>Significant Harmonics (N)</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Fundamental)</td>
<td>3.131 %</td>
</tr>
<tr>
<td>1</td>
<td>-0.573 %</td>
</tr>
<tr>
<td>2</td>
<td>0.178 %</td>
</tr>
<tr>
<td>3</td>
<td>-0.070 %</td>
</tr>
<tr>
<td>4</td>
<td>0.031 %</td>
</tr>
<tr>
<td>5</td>
<td>-0.014 %</td>
</tr>
<tr>
<td>6</td>
<td>0.006 %</td>
</tr>
</tbody>
</table>

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:

\[
R_L i_L(t) + L \frac{d i_L}{dt}(t) = V_{Pri}(t) - \frac{N_p}{N_s} V_{Sec}(t)
\]

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

(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:

\[
\begin{align*}
R_Li_L(t) + L\frac{di_L}{dt}(t) &= \{ R_L + j[2n + 1]\omega_s L \} I_{L[2n+1]} \\
V_{Pr[2n+1]} - \frac{N_p}{N_s}V_{Sec[2n+1]} &= \frac{4}{\pi} \frac{1}{2n + 1} \left\{ V_{in} - \frac{N_p}{N_s}V_{out} - [2n + 1] \delta \right\} 
\end{align*}
\]

(Eq. 3.21a)

(Eq. 3.21b)

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

\[
\{ R_L + j[2n + 1]\omega_s L \} I_{L[2n+1]} = \frac{4}{\pi} \frac{1}{2n + 1} \left\{ V_{in} - \frac{N_p}{N_s}V_{out} - [2n + 1] \delta \right\}
\]

\[
\therefore I_{L[2n+1]} = \frac{4}{\pi} \frac{1}{2n + 1} \left\{ V_{in} - \frac{N_p}{N_s}V_{out} - [2n + 1] \delta \right\} \left[ R_L + j[2n + 1]\omega_s L \right]^{-1}
\]

(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:

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

(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:

\[
i_C(t) = -i_{load}(t) + i_{DC}(t)
\]

\[
= -i_{load}(t) + \left\{ \frac{N_p}{N_s} i_L(t) \left( S_3(t) - S_4(t) \right) \right\}
\]

\[
= -i_{load}(t) + \left\{ \frac{N_p}{N_s} \sum_{m=0}^{N} \frac{1}{\pi} \left[ \begin{array}{c} \frac{V_{in}}{|Z[m]|} \sin \left( \frac{2m+1}{2} \omega_s t - \varphi_z[m] \right) - \\
\frac{V_{out}(t) N_p}{|Z[m]| N_s} \sin \left( \frac{2m+1}{2} \omega_s t - \delta - \varphi_z[m] \right) \end{array} \right] \right\} \times \left( \begin{array}{c} \frac{N_p}{N_s} \sum_{n=0}^{N} \frac{1}{\pi} \sin \left( \frac{2n+1}{2} \omega_s t - \delta \right) \end{array} \right)
\]

(3.24)

Expanding this equation gives:

\[
i_C(t) = -i_{load}(t) + \frac{8 N_p}{\pi^2 N_s} \sum_{n=0}^{N} \sum_{m=0}^{N} \frac{1}{[2n+1][2m+1]} \left[ \begin{array}{c} V_{in} \cos \left( \frac{2n+1}{2} \omega_s t - \delta \right) - [2m+1] \omega_s t + \varphi_z[m] \\
- \cos \left( \frac{2n+1}{2} \omega_s t - \delta \right) - [2m+1] \omega_s t - \varphi_z[m] \end{array} \right]
\]

\[
\times \left\{ \begin{array}{c} - N_p V_{out}(t) \cos \left( \frac{2n+1}{2} \omega_s t - \delta \right) - [2m+1] \omega_s t - \delta + \varphi_z[m] \\
- \cos \left( \frac{2n+1}{2} \omega_s t - \delta \right) - [2m+1] \omega_s t - \delta - \varphi_z[m] \end{array} \right\}
\]

(3.25)
CHAPTER 3. CONVERTER MODELLING

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 \((N = 3)\)](image)

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

\[
\frac{dV_{out}(t)}{dt} = \frac{i_C(t)}{C} = -i_{load}(t) + \frac{8}{C\pi^2} \frac{N_p}{N_s} \sum_{n=0}^{N} \sum_{m=0}^{N} \frac{1}{[2n+1][2m+1]} \left\{ \begin{array}{c}
\cos \left\{ [2n+1](\omega_s t - \delta) - [2m+1]\omega_s t + \varphi_z [m] \right\} \\
- \cos \left\{ [2n+1](\omega_s t - \delta) - [2m+1]\omega_s t - \varphi_z [m] \right\}
\end{array} \right\} \left\{ \begin{array}{c}
V_{in} \\
\frac{V_{out}(t)}{Z[m]} \\
- \frac{N_p}{N_s} \frac{V_{out}(t)}{Z[m]}
\end{array} \right\} \cos \left\{ [2n+1](\omega_s t - \delta) - [2m+1]\omega_s t + \varphi_z [m] \right\} - \cos \left\{ [2n+1](\omega_s t - \delta) - [2m+1]\omega_s t - \varphi_z [m] \right\}
\right\}
\]

(3.26)
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.
shift causes a relatively small change in the current magnitude, only 10 A for a peak-to-peak current of $\approx 40$ 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 $\delta$. 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 ($\delta$). 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
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:

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

(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.

\[ \frac{d(V_{out0} + \Delta V_{out0}(t))}{dt} \approx f(V_{out0}, \delta_0, i_{load0}) + \frac{\partial f}{\partial V_{out}} \bigg|_0 \Delta V_{out}(t) + \frac{\partial f}{\partial i_{load}} \bigg|_0 \Delta i_{load}(t) + \frac{\partial f}{\partial \delta} \bigg|_0 \Delta \delta \]  

(3.28)

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

\[ \frac{d\Delta V_{out}(t)}{dt} = A\Delta V_{out} + B_1\Delta \delta + B_i\Delta i_{load} \]

\[ = \left\{ \begin{array}{l} -\frac{8}{C\pi^2} \left( \frac{N_p}{N_s} \right)^2 \sum_{n=0}^{N} \left[ \cos(\varphi_z[n]) \right] \left[ \frac{2n+1}{|Z[n]|} \right] \Delta V_{out} \\ + \left\{ \begin{array}{l} -\Delta i_{load} \\ + \frac{8V_{in} N_p}{C\pi^2 N_s} \sum_{n=0}^{N} \left[ \sin(\varphi_z[n]) \right] \left[ \frac{2n+1}{|Z[n]|} \right] \Delta \delta \end{array} \right. \right\} \right\} \]

(3.29)
This small-signal linearised harmonic model is a first-order system, with two input variables ($\delta$ & $i_{load}$) and a single output variable ($V_{out}$). Two of the model coefficients are constants ($A$ & $B_I$), while $B_\delta$ 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 $B_\delta$ 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 $B_\delta$](image)

To test the validity of the linearised harmonic model, its response to a step change in $\delta$ 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 ($5^\circ$), 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 $10^\circ$ in Fig. 3.17b and $20^\circ$ 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 $10^\circ$ in operating condition. This corresponds to $\approx5\%$ of the entire $180^\circ$ dynamic range. However, for larger changes in phase angle, the model will need further adaptation.
Figure 3.17: Harmonic Model Verification: Output Voltage (Linearised Model) \((N = 3)\)
CHAPTER 3. CONVERTER MODELLING

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 $5^\circ$ at two different operating points – at a lower phase shift ($\delta = 20^\circ \rightarrow 25^\circ$), and at a higher phase shift ($\delta = 50^\circ \rightarrow 55^\circ$). 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 ($\approx 50^\circ$), 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 condition (Fig. 3.18b, $\approx 20^\circ$). 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.
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
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
CHAPTER 3. CONVERTER MODELLING

Figure 3.20: Deadtime Effect in a Phase Leg

(a) No Deadtime Effect

(b) Full Deadtime Effect

(c) Partial Deadtime Effect
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 & 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 ($\delta_{DT}$) in duration.

---

5 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 $\delta_{db}$ 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 $\delta_{db}$, 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 $\delta_{db}$ 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.

<table>
<thead>
<tr>
<th>Time Period</th>
<th>HV bridge lagging LV bridge (Fig. 3.21)</th>
<th>HV bridge leading LV bridge (Fig. 3.22)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_0) (+ve peak)</td>
<td>(i(t_0))</td>
<td>(i(t_0))</td>
</tr>
<tr>
<td>(t_0 \rightarrow t_1)</td>
<td>(i(t_1) = i(t_0) - \frac{V_H - V_L}{L} \left( \frac{\pi - \delta_s}{2\pi f_s} \right))</td>
<td>(i(t_1) = i(t_0) - \frac{V_H + V_L}{L} \left( \frac{\delta_s}{2\pi f_s} \right))</td>
</tr>
<tr>
<td>(t_1 \rightarrow t_2)</td>
<td>(i(t_2) = i(t_1) - \frac{V_H + V_L}{L} \left( \frac{\delta_s}{2\pi f_s} \right))</td>
<td>(i(t_2) = i(t_1) - \frac{V_H + V_L}{L} \left( \frac{\delta_s}{2\pi f_s} \right))</td>
</tr>
<tr>
<td>(t_2 \rightarrow t_3)</td>
<td>(i(t_3) = -\frac{V_H - V_L}{L} \left( \frac{\delta_s - \delta_dT}{2\pi f_s} \right))</td>
<td>(i(t_3) = -\frac{V_H - V_L}{L} \left( \frac{\delta_dT - \delta_s}{2\pi f_s} \right))</td>
</tr>
<tr>
<td>(t_3 \rightarrow t_4) (-ve peak)</td>
<td>(i(t_4) = i(t_3) - \frac{V_H + V_L}{L} \left( \frac{\pi - \delta_s - \delta_dT}{2\pi f_s} \right))</td>
<td>(i(t_4) = i(t_3) - \frac{V_H + V_L}{L} \left( \frac{\pi - \delta_s - \delta_dT}{2\pi f_s} \right))</td>
</tr>
</tbody>
</table>

**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 \(\delta_s\) defines (in radians) the time taken for the inductor current to slew to zero during the deadtime interval.

Since the phase shift error \((\delta_{db})\) is caused by the current that slews during the deadtime period, an expression that describes the slew time \(\delta_s\) 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


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

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

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

From the slew time equations (eq. 3.30), the phase shift error \( \delta_{db} \) 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 \( \delta_{db} \) to be determined based on \( \delta_s \). The relationship between \( \delta_s \) and \( \delta_{db} \) is therefore summarised in Table 3.5.

<table>
<thead>
<tr>
<th>Condition</th>
<th>( V_{in} &gt; \frac{N_p}{N_s} V_{out} )</th>
<th>( V_{in} &lt; \frac{N_p}{N_s} V_{out} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Bridge Leads Secondary</td>
<td>( \delta_s &gt; \delta_{DT} )</td>
<td>( \delta_{db} )</td>
</tr>
<tr>
<td>( \delta_s &lt; \delta_{DT} )</td>
<td>( \delta_{DT} - \delta_s )</td>
<td>( 0 &lt; \delta_s &lt; \delta_{DT} )</td>
</tr>
<tr>
<td>Primary Bridge Lags Secondary</td>
<td>( \delta_s &gt; 0 )</td>
<td>( \delta_{DT} )</td>
</tr>
<tr>
<td>( 0 &lt; \delta_s &lt; \delta_{DT} )</td>
<td>( \delta_s )</td>
<td>( 0 &lt; \delta_s &lt; \delta_{DT} )</td>
</tr>
</tbody>
</table>

| \( \delta_s < 0 \) | \( 0 \) | \( \delta_s > \delta_{DT} \) | \( 0 \) |

**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 auxiliary 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.
CHAPTER 3. CONVERTER MODELLING

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.

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
contribution 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.

\[
\Delta V_{out}(t) = A\Delta V_{out} + B_\delta \Delta \delta + B_I \Delta i_{load}
\]

where

\[
A = -\frac{8}{C\pi^2} \left( \frac{N_p}{N_s} \right)^2 \sum_{n=0}^{N} \left[ \cos (\varphi_z[n]) \left| Z[n]\right| \right]
\]

\[
B_\delta = \frac{8V_{in} N_p}{C\pi^2 N_s} \sum_{n=0}^{N} \left[ \sin (\varphi_z[n] - [2n + 1] \delta_0) \right] \left| Z[n]\right|\]

\[
B_I = -\frac{1}{C}
\]

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 \( V_{\text{out}} \), and only one controllable input – the phase shift \( \delta \). The load current input \( i_{\text{load}} \) 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 \( (\delta) \) such that the DC output voltage \( (V_{\text{out}}) \) tracks the reference \( (V_{\text{ref}}) \) \[99\].

![Figure 4.1: Basic closed loop block diagram of the DAB converter.](image)

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:
\[
\frac{d\Delta V_{\text{out}}(t)}{dt} = A\Delta V_{\text{out}} + B_\delta \Delta \delta + B_I \Delta i_{\text{load}}
\]

where
\[
A = \left\{ -\frac{8}{C\pi^2} \left( \frac{N_p}{N_s} \right)^2 \sum_{n=0}^{N} \left[ \frac{\cos(\varphi_z[n])}{[2n+1]^2|Z[n]|} \right] \right\}
\]
and
\[
B_\delta = \frac{8V_{\text{in}} N_p}{C\pi^2 N_s} \sum_{n=0}^{N} \left[ \frac{\sin(\varphi_z[n] - [2n+1]\delta_o)}{[2n+1]|Z[n]|} \right]
\]
and
\[
B_I = -\frac{1}{C}
\]

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

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]:
\[
H(s) = K_p \left( 1 + \frac{1}{sT_r} \right)
\]

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:
\[
F(s) = H(s)G(s) = K_p \left( 1 + \frac{1}{sT_r} \right) \frac{B_\delta T_p}{1 + sT_p}
\]
\[
= \frac{K_p B_\delta}{s} \frac{T_p}{T_r} \left( 1 + \frac{sT_r}{1 + sT_p} \right)
\]

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 ($\omega \to 0$, 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 $-90^\circ$, 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
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.
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.
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.

![Figure 4.5: Synchronous sampling](image)

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).
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 ($T_s/2$).

- **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 ($T_d/2$).

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

$$G(s) = \frac{V_{out}(s)}{V_{ref}(s)}$$

**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 ($\omega_c$). The transient performance achieved by the controller (in terms of rise time, settling time, overshoot, etc.) is governed by the available phase margin ($\varphi_m$) 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:

$$F(s) = H(s)G(s) = K_p \left( 1 + \frac{1}{sT_r} \right) e^{sT_d} \frac{B_p T_p}{1 + sT_p}$$

$$= \frac{K_p B_p T_p}{s} \left( 1 + \frac{T_r}{1 + sT_p} \right) e^{sT_d}$$

(4.4)

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

$$\angle F(j\omega_c) = \angle \left( \frac{1 + j\omega_c T_r}{j\omega_c T_r} \exp^{j\omega_c T_d} \frac{1}{1 + j\omega_c T_p} \right)$$

$$= -\pi + \varphi_m$$

(4.5)

which can be restated as:

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

(4.6)

This equation is further simplified by recognising that $\omega_c$ 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:

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

(4.7)

From this equation, it can be seen that the maximum value of $\omega_c$ 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 \( \omega_c \) \[129\], i.e.:

\[
T_r = \frac{10}{\omega_c} \tag{4.8}
\]

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

\[
\omega_c = \frac{\pi}{2} - \phi_m \frac{T_d}{T_r} \tag{4.9}
\]

The proportional gain \( K_p \) that gives the desired phase margin \( \phi_m \) 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, \( \omega_c \) \[17,18,122,129\]. This gives:

\[
1 = |G(j\omega_c)| = \left| \frac{K_pB_\delta T_p}{j\omega_c T_r} \left( \frac{1 + j\omega_c T_r}{1 + j\omega_c T_p} \right) \exp^{-j\omega_c T_d} \right|
\]

\[
= \frac{K_pB_\delta}{\omega_c} \tag{4.10}
\]

\[
\therefore K_p = \frac{\omega_c}{B_\delta}
\]

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

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

The \( \delta_o \) term in eq. 4.11 suggests that \( B_\delta \) 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) \[99\].
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.

<table>
<thead>
<tr>
<th>Circuit Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desired Phase Margin</td>
<td>$(\varphi_m)$</td>
</tr>
<tr>
<td>Transport Delay Time</td>
<td>$(T_d)$</td>
</tr>
<tr>
<td>Controller Bandwidth</td>
<td>$(\omega_c)$</td>
</tr>
<tr>
<td>Fixed PI Prop. Gain</td>
<td>$(K_p)$</td>
</tr>
<tr>
<td>Maximum Adaptive Prop. Gain</td>
<td>$(K_{PAdapt_{max}})$</td>
</tr>
<tr>
<td>Minimum Adaptive Prop. Gain</td>
<td>$(K_{PAdapt_{min}})$</td>
</tr>
<tr>
<td>Integrator Time Constant</td>
<td>$(T_r)$</td>
</tr>
</tbody>
</table>

Table 4.1: DAB Converter PI Controller Parameters
CHAPTER 4. CLOSED LOOP CONTROL

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

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.

![Graph](image)

(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:

\[
\frac{d\Delta V_{\text{out}}(t)}{dt} = A\Delta V_{\text{out}} + B_{\delta}\Delta \delta + B_{I}\Delta i_{\text{load}}
\]

where

\[
A = \frac{-8}{C\pi^2} \left(\frac{N_p}{N_s}\right)^2 \sum_{n=0}^{N} \left[ \cos(\varphi_n[n]) \right]^{2n+1} |Z_n[n]|,
\]

\[
B_{\delta} = \frac{8V_{\text{in}}N_p}{C\pi^2 N_s} \sum_{n=0}^{N} \left[ \sin(\varphi_n[n] - [2n+1]\delta_o) \right]^{2n+1} |Z_n[n]|
\]

and

\[
B_{I} = -\frac{1}{C}
\]

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:

\[
G_{\delta}(s) = \frac{T_pB_{\delta}}{1 + sT_p}
\]

\[
G_{I}(s) = \frac{T_pB_{I}}{1 + sT_p}
\]

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

\[
\frac{\Delta V_{\text{out}}}{\Delta V_{\text{ref}}} \bigg|_{\Delta i_{\text{load}}=0} = \frac{H(s)G_{\delta}(s)}{1 + H(s)G_{\delta}(s)} = \frac{K_p e^{sT_{\text{r}}}T_pB_{\delta}(1 + sT_{\text{r}})}{sT_{\text{r}}(1 + sT_p) + K_p e^{sT_{\text{d}}}T_pB_{\delta}(1 + sT_{\text{r}})}
\]

\[
\frac{\Delta V_{\text{out}}}{\Delta I_{\text{load}}} \bigg|_{\Delta V_{\text{ref}}=0} = \frac{G_{I}(s)}{1 + H(s)G_{\delta}(s)} = \frac{-sT_{\text{r}}T_p/C}{sT_{\text{r}}(1 + sT_p) + K_p e^{sT_{\text{d}}}T_pB_{\delta}(1 + sT_{\text{r}})}
\]
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 ($\approx -16$krad/s), because the low frequency pole is largely cancelled out by the nearby low frequency zero ($\approx -2$krad/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](image)

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 $\delta_{FF}$ 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](image)

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

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

(4.15)

This expression can then be solved for $I_{load}$ so the load current is known for any phase shift $\delta$. 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:
\[
\frac{I_{\text{load}}}{V_{\text{in}}} = \frac{8}{\pi^2} \frac{N_p}{N_s} \sum_{n=0}^{N} \left( \frac{1}{(2n+1)^3} \sin \left( \frac{1}{(2n+1) \delta} \right) \right)
\] (4.16)

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

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 \( \delta_{db} \) 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:

![Final Closed Loop Block Diagram of the DAB Converter](image)

**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
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.
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.

![Diagram of two-stage isolated bi-directional AC-DC converter](image)

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 cause 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:

\[
V_g\angle 0 = V_g \cos \{\omega_o t\} \quad (5.1a)
\]

\[
mV_{DC}\angle \varphi = mV_{DC} \cos \{\omega_o t + \varphi\} \quad (5.1b)
\]

\(^1\) 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.
CHAPTER 5. SYSTEM PERFORMANCE WITH AN AC LOAD

(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 $\omega_o$ is the fundamental frequency (50Hz in this case) expressed in rad/s and $\varphi$ is the relative angle between the inverter and the grid. Assuming that the power factor angle of the AC impedance is $\approx 90^\circ$, 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:

$$i(t) = \frac{m V_{DC} \angle \varphi - V_g \angle 0}{j \omega_o L}$$

$$= \frac{m V_{DC} \sin \{\omega_o t + \varphi\} - V_g \sin \{\omega_o t\}}{\omega_o L}$$

$$= \frac{V_g \sin \varphi}{\omega_o L} \cos \{\omega_o t + \varphi\}$$

$$+ \left[ \frac{m V_{DC} - V_g \cos \varphi}{\omega_o L} \right] \sin \{\omega_o t + \varphi\}$$

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, $i(t)$, i.e.:

$$P_{VSI}(t) = m V_{DC} \cos \{\omega_o t + \varphi\} i(t)$$

$$= m V_{DC} \cos \{\omega_o t + \varphi\} \left\{ \frac{V_g \sin \varphi}{\omega_o L} \cos \{\omega_o t + \varphi\} \right\}$$

$$+ \left[ \frac{m V_{DC} - V_g \cos \varphi}{\omega_o L} \right] \sin \{\omega_o t + \varphi\}$$

$$= \frac{m V_{DC} V_g \sin \varphi}{\omega_o L} \cos^2 \{\omega_o t + \varphi\}$$

$$+ \frac{m V_{DC} \left[m V_{DC} - V_g \cos \varphi\right]}{\omega_o L} \sin \{\omega_o t + \varphi\} \cos \{\omega_o t + \varphi\}$$

$$= \frac{m V_{DC} V_g \sin \varphi}{2 \omega_o L} + \frac{m V_{DC} \left[m V_{DC} - V_g \cos \varphi\right]}{2 \omega_o L} \cos \{2(\omega_o t + \varphi)\}$$

$$+ \frac{m V_{DC} \left[m V_{DC} - V_g \cos \varphi\right]}{2 \omega_o L} \sin \{2(\omega_o t + \varphi)\}$$

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.
CHAPTER 5. SYSTEM PERFORMANCE WITH AN AC LOAD

Figure 5.4: VSI fundamental average Voltage, Current & Power Flow

\[ V_{DC} = 200 \text{ V}, \quad V_g = 100 \text{ V}, \quad L = 1 \text{ mH}, \quad m = 0.8, \quad \phi = 30^\circ \]

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:

\[
P_{DAB} = \frac{8}{\pi^2} V_{in} V_{DC} \sum_{n=0}^{N} \left( \frac{1}{[2n + 1]^3} \sin ([2n + 1] \delta) \right)
\]  (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 \( \delta \) 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.
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 \(\phi_m\) as:

\[
\omega_{c_{VSI}} = \frac{\phi_m}{20 \times \pi}
\]

<table>
<thead>
<tr>
<th>Circuit Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC Input Voltage ((V_{in}))</td>
<td>200 V</td>
</tr>
<tr>
<td>DC Output Voltage ((V_{out}))</td>
<td>200 V</td>
</tr>
<tr>
<td>Peak AC Grid Voltage ((V_g))</td>
<td>100 V</td>
</tr>
<tr>
<td>Transformer Turns Ratio ((N_{Pri} : N_{Sec}))</td>
<td>10 : 15</td>
</tr>
<tr>
<td>VSI Switching Frequency (f_{VSI})</td>
<td>5 kHz</td>
</tr>
<tr>
<td>DAB Switching Frequency (f_{DAB})</td>
<td>20 kHz</td>
</tr>
<tr>
<td>DC Capacitance ((C))</td>
<td>20 (\mu)F</td>
</tr>
<tr>
<td>AC Inductor Inductance ((L))</td>
<td>50 (\mu)H</td>
</tr>
<tr>
<td>AC Inductor Resistance ((R_L))</td>
<td>0.1 (\Omega)</td>
</tr>
<tr>
<td>Output Inductance ((L_{out}))</td>
<td>5 mH</td>
</tr>
<tr>
<td>Output Load Resistance ((R_{out}))</td>
<td>0.5 (\Omega)</td>
</tr>
<tr>
<td>Nominal Output Power ((P_{out}))</td>
<td>3 kW</td>
</tr>
</tbody>
</table>

Table 5.1: DC-AC Converter Parameters

\(\) This assumption is valid as long as the modulator operates in the linear region.
\[ \omega_{c_{VSI}} = \frac{\left( \frac{\pi}{2} - \varphi_m \right)}{T_d} \]  \hspace{1cm} (5.5)

Controller gains can now be calculated as [129]:

\[ K_{pv_{VSI}} = \frac{\omega_{c_{VSI}} L_{out}}{V_{DC}} \]  \hspace{1cm} (5.6a)
\[ T_{rv_{VSI}} = \frac{10}{\omega_{c_{VSI}}} \]  \hspace{1cm} (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
CHAPTER 5. SYSTEM PERFORMANCE WITH AN AC LOAD

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.

<table>
<thead>
<tr>
<th>Circuit Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desired Phase Margin ($\varphi_{\text{m}_\text{VSI}}$)</td>
<td>40°</td>
</tr>
<tr>
<td>VSI Transport Delay Time ($T_{d_{\text{VSI}}}$)</td>
<td>150 µs</td>
</tr>
<tr>
<td>VSI Controller Bandwidth ($\omega_{\text{c}_{\text{VSI}}}$)</td>
<td>926 Hz</td>
</tr>
<tr>
<td>VSI Proportional Gain ($K_{p_{\text{VSI}}}$)</td>
<td>0.1454</td>
</tr>
<tr>
<td>VSI Integrator Time Constant ($T_{r_{\text{VSI}}}$)</td>
<td>10.8 ms</td>
</tr>
</tbody>
</table>

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.

![Closed-loop block diagram of the Voltage Regulated DAB](image)

**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

<table>
<thead>
<tr>
<th>Circuit Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desired Phase Margin ($\varphi_{m_{DAB}}$)</td>
<td>60°</td>
</tr>
<tr>
<td>DAB Transport Delay Time ($T_{DAB}$)</td>
<td>50 µs</td>
</tr>
<tr>
<td>DAB Controller Bandwidth ($\omega_{c_{DAB}}$)</td>
<td>1667 Hz</td>
</tr>
<tr>
<td>Maximum DAB Prop. Gain ($K_{PDAB_{max}}$)</td>
<td>0.0102</td>
</tr>
<tr>
<td>Minimum DAB Prop. Gain ($K_{PDAB_{min}}$)</td>
<td>0.2427</td>
</tr>
<tr>
<td>DAB Integrator Time Constant ($T_{I_{DAB}}$)</td>
<td>6 ms</td>
</tr>
</tbody>
</table>

**Table 5.3:** DAB Voltage Regulator Controller Parameters
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.
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
approximated by scaling the sampled peak current by the instantaneous modulation depth of the VSI, as:

\[ I_{load_{avg}} = mI_{load} \]  \hspace{1cm} (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
CHAPTER 5. SYSTEM PERFORMANCE WITH AN AC LOAD

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

![Graph showing output voltage]

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

![Graph showing harmonic spectrum]

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 \(40^{\circ}\) 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).

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.).

![PSIM Power Stage - DAB Converter](image)

**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 \(R_{\text{const}}\).

- **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 \(R_{\text{switch}}\).

- **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 \(L_{\text{VSI}} \& R_{\text{VSI}}\). This was used to explore the effects of the AC load (Chapter 5).

---

\(^1\) To match the experimental prototype.
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
CHAPTER 6. DESCRIPTION OF SIMULATED & EXPERIMENTAL SYSTEMS

(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.
## Table 6.1: DAB Voltage Regulator Controller Parameters

<table>
<thead>
<tr>
<th>Circuit Parameter</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>OL_CL_VSI</td>
<td>0</td>
<td>Open-loop Modulated VSI</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Closed-loop PI Current Regulated VSI</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Closed-loop PI Current Regulated VSI with Feed-forward Grid Disturbance Compensation</td>
</tr>
<tr>
<td>OL_CL_BiDC</td>
<td>0</td>
<td>Open-loop Modulated DAB</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Closed-loop Adaptive PI Voltage Regulated DAB</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Closed-loop Adaptive PI Voltage Regulated DAB with Feed-forward Load Current Disturbance Compensation (DC)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Closed-loop Adaptive PI Voltage Regulated DAB with Feed-forward Load Current Disturbance Compensation (AC)</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Closed-loop Fixed PI Voltage Regulated DAB</td>
</tr>
<tr>
<td>DT_COMP</td>
<td>0</td>
<td>Deadtime Compensation Inactive</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Deadtime Compensation Active</td>
</tr>
</tbody>
</table>
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.:

- **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.
## Table 6.2: DC-AC Experimental Converter Parameters

<table>
<thead>
<tr>
<th>Circuit Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC Input Voltage ((V_{in}))</td>
<td>200 V</td>
</tr>
<tr>
<td>DC Output Voltage ((V_{out}))</td>
<td>200 V</td>
</tr>
<tr>
<td>Transformer Turns Ratio ((N_p : N_s))</td>
<td>10 : 11</td>
</tr>
<tr>
<td>VSI Switching Frequency ((f_{VSI}))</td>
<td>5 kHz</td>
</tr>
<tr>
<td>DAB Switching Frequency ((f_{DAB}))</td>
<td>20 kHz</td>
</tr>
<tr>
<td>DC Capacitance ((C))</td>
<td>12 (\mu)F</td>
</tr>
<tr>
<td>AC Inductor Inductance ((L))</td>
<td>132 (\mu)H</td>
</tr>
<tr>
<td>AC Inductor Resistance ((R_L))</td>
<td>0.1 (\Omega)</td>
</tr>
<tr>
<td>Output Inductance ((L_{out}))</td>
<td>8 mH</td>
</tr>
<tr>
<td>Output Load Resistance ((R_{out}))</td>
<td>16.5 (\Omega)</td>
</tr>
<tr>
<td>Nominal Output Power ((P_{out}))</td>
<td>1 kW</td>
</tr>
</tbody>
</table>

**Figure 6.7: Laboratory Setup**
CHAPTER 6. DESCRIPTION OF SIMULATED & EXPERIMENTAL SYSTEMS

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 $\approx 20A$ 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
CHAPTER 6. DESCRIPTION OF SIMULATED & EXPERIMENTAL SYSTEMS

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.

![Circuit Diagram](image)

**Figure 6.12:** Experimental Load Circuit Configuration

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).

![Load Elements](image)

**Figure 6.13:** Experimental Load Elements

The final constructed IGBT platform is shown in Fig. 6.16. Only three of the six available phase legs are used – $S_1$ of the first phase leg provides the switch for the switched load ($S_{load}$ in Fig. 6.12), while the next two phase legs ($S_2, S_3$) 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].

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
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.
• 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.

![Linked GIIB Boards](image)

**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](image)

(x-axis: 2µs/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:

\[ I_{\text{load,avg}} = mI_{\text{load}} \]  

(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.

![Circuit Diagram of the Experimental Prototype](image)

**Figure 7.1:** Circuit Diagram of the Experimental Prototype

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

**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.
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 $\delta$. In each case a step change of $5^\circ$ 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.

![Figure 7.5: DAB Converter Open Loop Step Response – Affected by Deadtime](image-url)
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.

**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:

<table>
<thead>
<tr>
<th>Circuit Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desired Phase Margin ( (\phi_{m_{DAB}}) )</td>
<td>40°</td>
</tr>
<tr>
<td>DAB Transport Delay Time ( (T_{d_{DAB}}) )</td>
<td>50 µs</td>
</tr>
<tr>
<td>DAB Controller Bandwidth ( (\omega_{c_{DAB}}) )</td>
<td>2778 Hz</td>
</tr>
<tr>
<td>Maximum DAB Prop. Gain ( (K_{pDAB_{max}}) )</td>
<td>0.0236</td>
</tr>
<tr>
<td>Minimum DAB Prop. Gain ( (K_{pDAB_{min}}) )</td>
<td>0.2905</td>
</tr>
<tr>
<td>DAB Integrator Time Constant ( (T_{I_{DAB}}) )</td>
<td>3.6 ms</td>
</tr>
</tbody>
</table>

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.

![Graph showing transient response](image)

(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.

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 (38Ω ⇔ 32.9Ω).

Fig. 7.9 shows the transient responses caused by a load decrease (32.9Ω → 38.4Ω). 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.

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.

**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.

<table>
<thead>
<tr>
<th>Circuit Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desired Phase Margin ( \phi_{m_{VSI}} )</td>
<td>40°</td>
</tr>
<tr>
<td>VSI Transport Delay Time ( T_{d_{VSI}} )</td>
<td>150 µs</td>
</tr>
<tr>
<td>VSI Controller Bandwidth ( \omega_{c_{VSI}} )</td>
<td>926 Hz</td>
</tr>
<tr>
<td>VSI Proportional Gain ( K_{p_{VSI}} )</td>
<td>0.1454</td>
</tr>
<tr>
<td>VSI Integrator Time Constant ( T_{r_{VSI}} )</td>
<td>10.8 ms</td>
</tr>
</tbody>
</table>

**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.

**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 $\rightarrow$ 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.

**Figure 7.12:** Experimental Transient Waveforms - AC Load Step (4A $\rightarrow$ 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.

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

(a) Adaptive PI Control

(b) Adaptive PI + Feed-forward Control
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:

```c
/************************ Standard PSim DLL reads ******************************/
// This is a sample C program for Microsoft C/C++ 5.0 or 6.0.
// The generated DLL is to be linked to PSIM.

// To compile the program into DLL, you can open the workspace file "msvc_dll.dsw"
// as provided. Or you can create a new project by following the procedure below:

// - Create a directory called "C:\msvc_dll", and copy the file "msvc_dll.c"
// that comes with the PSIM software into the directory C:\msvc_dll.

// - Start Visual C++. From the "File" menu, choose "New". In the "Projects"
// page, select "Win32 Dynamic-Link Library", and set "Project name" as
// "ms_user0", and "Location" as "C:\msvc_dll". Make sure that
// "Create new workspace" is selected, and "Win32" is selected under
// "Platform",

// - [for Version 6.0] When asked "What kind of DLL would you like to create?",
// select "An empty DLL project."

// - From the "Project" menu, go to "Add to Project"/"Files...", and select
// "msvc_dll.c".

// - Add your own code as needed.

// - From the "Build" menu, go to "Set Active Configurations...", and select
// "Win32 Release". From the "Build" menu, choose "Rebuild All" to generate
```

160
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 //fixed point scaling
94 #define FIXED_Q 10 //1
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)
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

#define DEG_TO_RAD PI/180.0

//For DSP emulation
#define HSPCLK (150e6)

#define MIN_VSI_TIME 1e-6
#define MIN_VSI_COUNT (HSPCLK*MIN_VSI_TIME)
#define MAX_VSI_TIME (int16)(PERIOD_2_VSI-MIN_VSI_COUNT)
#define SIN_TABLE_SIZE 512

//vsi parameters
#define SW_FREQ_VSI (5000.0)
#define PERIOD_2_VSI ((Uint16)((HSPCLK/SW_FREQ_VSI)/4.0))
#define PERIOD_VSI ((Uint16)(2*PERIOD_2_VSI))
#define FSAMPLE_VSI (SW_FREQ_VSI*2.0)
#define TSAMPLE_VSI (1.0/FSAMPLE_VSI)
#define T_DELAY_VSI (1.5*TSAMPLE_VSI)
#define F_FUND 50.0
#define OMEGA_FUND (2*PI*F_FUND)
#define OMEGA_C_VSI (PI_2-(40*DEG_TO_RAD))/(T_DELAY_VSI)
#define KP_VSI (OMEGA_C_VSI*LVSI/VIN)
#define KI_VSI (OMEGA_C_VSI/10.0)

//BiDC parameters
#define SW_FREQ_BIDC (20000.0)
#define TS_BIDC ((double)(1.0/SW_FREQ_BIDC))
#define OMEGA_BIDC 2.0*PI*SW_FREQ_BIDC
#define PERIOD_2_BIDC ((Uint16)((HSPCLK/SW_FREQ_BIDC)/4.0))
#define PERIOD_BIDC (2*PERIOD_2_BIDC)
#define FSAMPLE_BIDC (1.0*SW_FREQ_BIDC)
#define TSAMPLE_BIDC (1.0/FSAMPLE_BIDC)
#define MAX_PHASE (PERIOD_2_BIDC-1)
#define T_DELAY_BIDC (1.0*TSAMPLE_BIDC)
#define OMEGA_C_BIDC (PI_2-(60*DEG_TO_RAD))/(T_DELAY_BIDC) //50 deg phase margin
#define OMEGA_C_BIDC_FIXED ((int32)(OMEGA_C_BIDC*SMALL_Q_SCALE)
#define PERIOD_SCALE_BIDC ((int32)(PERIOD_2_BIDC*INV2_PI)

//Topology parameters
#define C 20e-6
#define L 50e-6
#define R_L 0.1
#define R_L_2 R_L*R_L
#define LVSI (5e-3)
#define KP_CONST (int32)(LVSI*OMEGA_C_VSI*FIXED_Q_SCALE)
#define OMEGA_BIDC_L (OMEGA_BIDC*L)
#define OMEGA_BIDC_L_2 (OMEGA_BIDC_L*OMEGA_BIDC_L)
#define NPRI (10.0)
#define NSEC (15.0)
#define NPRI_NSEC ((double)(NPRI/NSEC))
#define NPRI_NSEC_FIXED ((int32)(NPRI_NSEC*FIXED_Q_SCALE)
#define VIN (200.0)
#define _4VIN (4.0*VIN)
#define VIN_FIXED ((int32)(VIN*FIXED_Q_SCALE)
#define VP_FIXED ((int32)(VIN*FIXED_Q_SCALE/2.0)
#define VDCPRI (VIN/2.0)
#define VBUS_NOM VIN
#define VBUS_NOM_FIXED ((int32)(VIN*FIXED_Q_SCALE)

//Adaptive controller parameters
#define VDC_KP_INIT 0.005
#define VDC_KP_MAX 0.04
#define VDC_KP_MIN 0.001
#define VDC_KP_MAX_FIXED (int32)(VDC_KP_MAX*SMALL_Q_SCALE)
#define VDC_KP_MIN_FIXED (int32)(VDC_KP_MIN*SMALL_Q_SCALE)
#define VDC_KP_INIT_FIXED (int32)(VDC_KP_INIT*SMALL_Q_SCALE)
#define DELF_DELU_CONST (VDCPRI+NPRI*VIN)/(C*PI*PI)) //divide by 16.0 is for scaling purposes
#define DELF_DELX_CONST ((-8.0*NPRI*VIN)/(C*PI*PI))
#define BIDC_FF_CONST ((16.0*VDCPRI*NPRI*VIN)/(PI*PI))/OMEGA_BIDC_L)

#define PHASE_STEP (Uint32)(4294967296.0*(double)F_FUND/(double)SW_FREQ_VSI/2.0)
#define PHASE_STEP (Uint16)(65536.0*(double)F_FUND/(double)SW_FREQ_VSI/2.0)
#define DEADBAND_BIDC 1e-6
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

#define DB_DEG_BIDC (360.0*SW_FREQ_BIDC*DEADBAND_BIDC)
#define DB_RAD_BIDC (DB_DEG_BIDC*DEG_TO_RAD)
#define DEADBAND_COUNT_BIDC ((int16)(DEADBAND_BIDC*HSPCLK))
#define DB_RATIO_BIDC ((double)DEADBAND_COUNT_BIDC/(PERIOD_BIDC*2))

// sine table hash definitions
#define COUNT_TO_SINTABLE (4294967296.0/(PERIOD_BIDC*2))
#define COUNT_TO_RAD PI/(PERIOD_BIDC)
#define COUNT_TO_RATIO 1.0/(2*3750.0)
#define RAD_TO_COUNT 3750.0/PI

/********
_MACROS()
********/
#define SIN_TABLE_READ(PHASE,SIN_VAL){
    SIN_VAL = sin_table[((Uint32)PHASE>>23)];
    VAL_DIFF = (sin_table[((Uint32)PHASE>>23)+1]) - SIN_VAL;
    SIN_VAL += (int16)( ((int32)((Uint32)PHASE&0x3FFFFF)*(int32)VAL_DIFF)>>23 );}

void __declspec(dllexport) simuser (double t, double delt, double *in, double *out)
{
    double ctrlclk = in[0];
    double VSI_ctrlclk = in[1];
    int16 OL_CL_VSI = (int16)in[2];
    double mod = in[3];
    double mag_Iref = in[4];
    double Iout = in[5];
    double emf_scaled = in[6];
    int32 phase_current = (int32)in[7];
    int16 OL_CL_BiDC = (int16)in[8];
    int32 DT_COMP = (int32)in[9];
    double phase_OL = in[10];
    double mag_VDCref = in[11];
    double Vout = in[12];
    double Iload = in[13];

    //Variable_Declarations
    /**************************
    _DSP_Emulation_Variables()
    **************************/
    static int16 sin_table[SIN_TABLE_SIZE+1],
    cos_table[SIN_TABLE_SIZE+1];
    static int32 int_table=0;
    static int16 UF,
    UF_VSI,
    int_count, //to tell which interrupt to run in.
    prev_ctrlclk,
    prev_VSI_ctrlclk;

    /*****************
    _Macro_Variables()
    *****************/
    static Uint32 PHASE;
    static int16 SIN_VAL,
    VAL_DIFF; // interpolation temp variable

    /*********************
    _DSP_Modulation_Variables()
    *********************/
    static Uint16 VSIa,
    VSIb,
    BiDC_Pri,
    BiDC_Sec,
    CMPR1,
    CMPR2,
    CMPR_Pri,
    CMPR_Sec,
    V_Asat = 0,
    V_Bsat = 0;

163
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

269 270 271 272 273 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 static int16 phase_shift=0; 289 290 291 292 293 typedef struct 294 { 295 double 296 Kp, 297 Ki, 298 ref, 299 error, 300 prop, 301 intnow, 302 intsum, 303 ctrl; 304 }type_pi_dbl; 305 306 307 #define fixed 308 309 static long Iref_mag_fixed, 310 I_VSI_fixed, 311 Iref_fixed, 312 VSIerror_fixed, 313 Kp_VSI_fixed, 314 Ki_VSI_fixed, 315 VSIprop_fixed, 316 VSI_intnow_fixed, 317 VSI_int_fixed, 318 VSI_ctrl_fixed, 319 emf_scaled_fixed; 320 321 322 323 324 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; 335 336 337 typedef struct 338 { 339 double 340 Kp, 341 Ki, 342 ref, 343 error, 344 prop, 345 intnow, 346 intsum, 347 ctrl; 348 }type_pi_dbl; 349 350 351 static long VDC_VSI_fixed; 352 353 354 static int16 prev_ZX, 355 test_point; 356 357 //current phase step variables
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

static int32 prev_phase_current = 0;

/********************
_DT_Comp_Variables()
********************/

// New version. Unified DT compensation
// floating point
static double VDCout_txscaled,
Vs_Vp_4Vs,
Vp_Vs_DB;

// fixed point
static int32 phase_rad_ratio_fixed,
VDCout_txscaled_fixed,
Vp_Vs_4Vp_fixed,
Vs_Vp_4Vp_fixed,
Vs_Vp_4Vs_fixed,
Vp_Vs_DB_fixed;

static int16 Tslew_count,
phase_aug_DT_fixed;

/********************
_Adaptive_Variables()
********************/
typedef struct {
    double delta0_aug,
delf_delu,
sin_val,
sin,
sum,
Kp;
}type_adapt;

static type_adapt Kp_float;
static double delf_delx,
delf_delx_temp,
delf_delx_scaled,
delf_delu_temp,
delf_delu,
delf_delu_scaled,
Kp_adapt,
// phi_z=PI/2.0,
Z_harm[7],
phi_z[7];

static int16 phase_shift_avrg,
phase_shift_record[4],
counter_avrg,
n_harm;

// in fixed point
static int16 phi_z_fixed[7],
harm[7]={1,3,5,7,9,11,13},
harm_sq[7]={1,9,25,49,81,121,169},
sin_val_adapt;

static double sin_val_adapt_double;

static int32 delta0_aug_fixed,
sin_count,
inv_Z_harm_fixed[7],
delf_delu_temp_fixed,
delf_delu_fixed,
delf_delu_fixed_scaled,
Kp_adapt_fixed;

// Kp_adapt_fixed_prev;

/************************
_BiDC_PI_Control_Variables()
*************************/
static type_pi_dbl VDC_PI_DBL;
// floating point version
static double VDCout_float,
VDC_Kp_float=0.02,
Vp_Vs_DB Fixed;

static double VDCout_fቡ mandates
VDC_Kp_fҏ mandates=0.02,
# APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 ******************
451 _BIDC_FF_Variables()
452 ******************/
453 static double Iload_FF_double;
454
455 static int32 Iload_fixed,
456 Iload_abs;
457
458 static int32 BIDC_FF,
459 Iload_FF_fixed[PERIOD_2_BIDC];
460
461 static int16 hi,
462 lo,
463 mid,
464 harm_3[7]={1,27,125,343,729,1331,2197};
465 
466 ******************
467 _BIDC_Resonant_Variables()
468 ******************/
469 // Canonical representation
470 static double Kemfint2,
471 int2,
472 Kemf,
473 Sum1,
474 int1,
475 int3,
476 PPlaceCan_out,
477 Komega1,
478 Komega2,
479 Komega3;
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 back EMF
506 04/03/2011 - Just a counter that makes it only work on 1 in 4 cycles. (int_count)
507 04/03/2011 - Determines the phase of the back emf - very miniature grid sync

166
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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  // 10/03/2011 - Synchronous Sampled Adaptive PI Controller (Floating Point)
512  // 07/03/2011 - Asynchronous Sampling - Why is it peak & trough now????
513  // 08/03/2011 - Closed loop control of the Bi-directional DC-DC Converter. Optimised Integrator
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  // 13/03/2011 - Resonant Controller - Floating Point - attempted
519  // 14/03/2011 - 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    phi_z[0] = atan2(OMEGA_BIDC_L,R_L);
566    phi_z[1] = atan2(OMEGA_BIDC_L*3.0,R_L);
567    phi_z[2] = atan2(OMEGA_BIDC_L*5.0,R_L);
568    phi_z[3] = atan2(OMEGA_BIDC_L*7.0,R_L);
569    phi_z[4] = atan2(OMEGA_BIDC_L*9.0,R_L);
570    phi_z[5] = atan2(OMEGA_BIDC_L*11.0,R_L);
571    phi_z[6] = atan2(OMEGA_BIDC_L*13.0,R_L);
572
573    inv_Z_harm_fixed[0]= (int32)(32768.0/(1.0*Z_harm[0]));
574    inv_Z_harm_fixed[1]= (int32)(32768.0/(3.0*Z_harm[1]));
575    inv_Z_harm_fixed[2]= (int32)(32768.0/(5.0*Z_harm[2]));
576    inv_Z_harm_fixed[3]= (int32)(32768.0/(7.0*Z_harm[3]));
577    inv_Z_harm_fixed[4]= (int32)(32768.0/(9.0*Z_harm[4]));
578    inv_Z_harm_fixed[5]= (int32)(32768.0/(11.0*Z_harm[5]));
579    inv_Z_harm_fixed[6]= (int32)(32768.0/(13.0*Z_harm[6]));
580
581    phi_z_fixed[0] = (int16)(phi_z[0]*RAD_TO_COUNT);
582    phi_z_fixed[1] = (int16)(phi_z[1]*RAD_TO_COUNT);
583    phi_z_fixed[2] = (int16)(phi_z[2]*RAD_TO_COUNT);
584    phi_z_fixed[3] = (int16)(phi_z[3]*RAD_TO_COUNT);
585    phi_z_fixed[4] = (int16)(phi_z[4]*RAD_TO_COUNT);
586    phi_z_fixed[5] = (int16)(phi_z[5]*RAD_TO_COUNT);
587    phi_z_fixed[6] = (int16)(phi_z[6]*RAD_TO_COUNT);
588
589    inv_Z_harm_fixed[0]= (int32)(32768.0/(1.0*Z_harm[0]));
590    inv_Z_harm_fixed[1]= (int32)(32768.0/(3.0*Z_harm[1]));
591    inv_Z_harm_fixed[2]= (int32)(32768.0/(5.0*Z_harm[2]));
592    inv_Z_harm_fixed[3]= (int32)(32768.0/(7.0*Z_harm[3]));
593    inv_Z_harm_fixed[4]= (int32)(32768.0/(9.0*Z_harm[4]));
594    inv_Z_harm_fixed[5]= (int32)(32768.0/(11.0*Z_harm[5]));
595    inv_Z_harm_fixed[6]= (int32)(32768.0/(13.0*Z_harm[6]));
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```
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  for (init_table=0;init_table<=PERIOD_2_BIDC;init_table++)
601  {
602    Iload_FF_double=0.0;
603    for (n_harm=0;n_harm<6;n_harm++)
604      Iload_FF_double += (1.0/harm_3[n_harm])*sin(harm[n_harm]*(init_table*COUNT_TO_RAD));
605  }
606  Iload_FF_fixed[init_table] = (int32)(BIDC_FF_CONST*Iload_FF_double*FIXED_Q_SCALE);
607  }
608  
609  //to help with initialisations
610  VDCout_fixed=0;
611  }
612  
613  /********************
614  _TIMER_INTERRUPT()
615  ********************/
616  if ((prev_VSI_ctrlclk <=0 && VSI_ctrlclk >=1)||(prev_VSI_ctrlclk >=1 && VSI_ctrlclk <=0)) //Sampling clock
617  {
618    if (UF_VSI==0) UF_VSI = 1;
619    else UF_VSI = 0;
620  
621  // Calculate current sin table value:
622  vsiphase+=PHASE_STEP;
623  
624  //check for step change in phase
625  if (phase_current!=prev_phase_current)
626    vsiphase = phase_current;
627  
628  SIN_TABLE_READ(vsiphase,sin_val);
629  
630  // We also want the ability to step change the flow of power.
631  // this is a step change in phase, with reference to the back emf.
632  //Run the ADCs
633  I_VSI_fixed = (int32)(Iout*FIXED_Q_SCALE);
634  emf_scaled_fixed = (int32)(emf_scaled*FIXED_Q_SCALE);
635  
636  //the whole point of closed loop control is to determine the required magnitude & phase
637  //that will give the desired output current.
638  
639  /*****************
640  _VSI_Current_Regulator()
641  *****************/
642  if ((prev_VSI_ctrlclk <=0 && VSI_ctrlclk >=1)||(prev_VSI_ctrlclk >=1 && VSI_ctrlclk <=0)) //Sampling clock
643  {
644    if (UF_VSI==0) UF_VSI = 1;
645    else UF_VSI = 0;
646  
647  //first, generate reference
648  Iref_fixed = (int32)(mag_Iref*FIXED_Q_SCALE);
649  
650  //scale KP by DC Bus
651  Kp_VSI_fixed=(long)((KP_CONST*2.0*FIXED_Q_SCALE)/VDCout_fixed); //scaled by dc
652  
653  //determine error
654  VSIerror_fixed = (Iref_fixed - I_VSI_fixed);
655  VSIprop_fixed = (VSIerror_fixed*Kp_VSI_fixed)>>FIXED_Q;
656  
657  //integrator
658  VSI_intnow_fixed = (VSIprop_fixed*VSI_int_fixed)>>FIXED_Q;
659  VSI_int_fixed += VSI_intnow_fixed;
660  VSI_ctrl_fixed = VSIprop_fixed + VSI_int_fixed;
```
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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

169
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 } *******/
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 else Iload_fixed = (int32)(I_VSI_fixed*((int32)va_temp<<2))/PERIOD_2_VSI); // scaled by mod depth*2
772 773 //first determine the Average operating phase_shift (moving average of the last 4 phaseshifts)
774 phase_shift_avrg=0; //in counts
775 counter_avrg=0;
776 phase_shift_record[int_count] = abs(phase_shift);
777 while(counter_avrg<=3)
778 {
779 phase_shift_avrg += phase_shift_record[counter_avrg]>>2;
780 counter_avrg++;
781 }
782 783 784 } *****************************/
785 __Deadtime_Compensation() */
786 787 phase_rad_ratio_fixed = (abs(phase_shift)<<FIXED_Q)/(PERIOD_BIDC<<1);
788 VDCout_txscaled_fixed = (VDCout_fixed*NPRI_NSEC_FIXED)>>FIXED_Q;
789 if (VDCout_txscaled_fixed==0) VDCout_txscaled_fixed=1;
790 VDCout_txscaled = (double)VDCout_txscaled_fixed/FIXED_Q_SCALE;
791 792 // First, calculate slew time
793 if (VIN_FIXED>VDCout_txscaled_fixed) //Vp>Vs
794 {
795 if (phase_shift_avrg<0) //leading
796 {
797 Tslew_count = (int16)((phase_rad_ratio_fixed - Vp_Vs_4Vp_fixed - Vs_Vp_DB_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
798 //then calculate phase augmentation
799 if (VIN_FIXED>VDCout_txscaled_fixed)>5<<FIXED_Q)
800 {
801 if (T slew_count<SLEW_DELAY_COUNT_BIDC) phase_aug_DT_fixed = 0;
802 else phase_aug_DT_fixed = SLEW_DELAY_COUNT_BIDC;
803 } *******/
804 else
805 {
806 if (T slew_count>SLEW_DELAY_COUNT_BIDC) phase_aug_DT_fixed = SLEW_DELAY_COUNT_BIDC;
807 798 799 800 801 802 803 804 805 806 807 808 else //lagging
809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828

APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```c
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>DEADBAND_COUNT_BIDC) phase_aug_DT_fixed = DEADBAND_COUNT_BIDC;  
842     else phase_aug_DT_fixed = 0;  
843   }  
844  }  
845  
846  }  
847  //VpVs
848  {  
849  if (phase_shift_avrg<0) // leading  
850   {  
851     Tslew_count = (int16)((Vs_Vp_4Vp_fixed - phase_rad_ratio_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);  
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     if ((VDCout_txscaled_fixed-VIN_FIXED)>(5<<FIXED_Q))  
869      {  
870         if (Tslew_count>DEADBAND_COUNT_BIDC) phase_aug_DT_fixed = 0;  
871         else if (Tslew_count<0) phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;  
872         else phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;  
873      }  
874      else  
875      {  
876         if (Tslew_count>DEADBAND_COUNT_BIDC) phase_aug_DT_fixed = 0;  
877         else phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;  
878      }  
879  }  
880  
881  }  
882 
883  }  
884  }  
885  // floating point first - make sure it works  
886  // Then, include the deadtime compensation  
887  if (DT_CMP)  
888   delta0_aug_fixed=abs(phase_shift_avrg-phase_aug_DT_fixed);  
889  else  
890   delta0_aug_fixed=abs(phase_shift_avrg);  
891  if (delta0_aug_fixed >= (MAX_PHASE-100))  
892   delta0_aug_fixed = MAX_PHASE-100;  
893  // floating point  
894  Kp_float.delta0_aug = (double)delta0_aug_fixed*COUNT_TO_RAD;  
895  n_harm=0;  
896  Kp_float.sum = 0.0;  
897  for (n_harm=0;n_harm<6;n_harm++)  
898   {  
899     Kp_float.sin_val = phi_z[n_harm] - harm[n_harm]*Kp_float.delta0_aug;  
900     Kp_float.sin = sin(Kp_float.sin_val);  
901   }  
902   171
```
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

909 \( Kp\textunderscore{float}.sum \) += \( Kp\textunderscore{float}.\sin/(\text{harm}[\text{n\_harm}]*Z\textunderscore{harm}[\text{n\_harm}]) \);
910
911 }
912 \( Kp\textunderscore{float}.delf\_delu = (16.0*VDCPRI/(C*PI*PI))*(\text{NPRI}/\text{NSEC})*Kp\textunderscore{float}.sum; \)
913 if (\( Kp\textunderscore{float}.delf\_delu==0 \))
914 \( Kp\textunderscore{float}.delf\_delu=1e-4; \)
915 \( Kp\textunderscore{float}.Kp = \Omega C_B\_BIDC/Kp\textunderscore{float}.delf\_delu; \)
916
917 if (\( Kp\textunderscore{float}.Kp >VDC\_KP\_MAX \)) \( Kp\textunderscore{float}.Kp = VDC\_KP\_MAX; \)
918 if (\( Kp\textunderscore{float}.Kp <VDC\_KP\_MIN \)) \( Kp\textunderscore{float}.Kp = VDC\_KP\_MIN; \)
919
920 //fixed point
921 //then determine the B value
922 \( \text{delf\_delu\_fixed} = 0; \)
923 \( n\_harm=0; \)
924
925 for (\( n\_harm = 0; n\_harm<6; n\_harm++ \))
926 {
927 //fixed point
928 \( \text{sin\_count} = (int32)((\phi_z\_fixed[\text{n\_harm}]-\text{harm}[\text{n\_harm}]*\delta0\_aug\_fixed)*\text{COUNT\_TO\_SINTABLE}); \)
929 \( \text{sin\_val\_adapt} = \text{sin\_table}[(uint32)(\text{sin\_count}>>22)]; \)
930 \( \text{sin\_val\_adapt\_double} = (int16)(16384.0*\sin(\phi_z[\text{n\_harm}]-\text{harm}[\text{n\_harm}]*\delta0\_aug\_fixed+\text{COUNT\_TO\_RAD})); \)
931
932 //Determine B\_delta value - for proportional term
933 \( \text{delf\_delu\_temp\_fixed} = (int32)(\text{sin\_val\_adapt\_double}*\text{inv\_Z\_harm\_fixed}[\text{n\_harm}]>>\text{(14+15-SMALL\_Q)}); \)
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 \( \text{delf\_delu\_fixed} +\text{delf\_delu\_temp\_fixed}; \)
936
937 //scale by constants
938 \( \text{delf\_delu\_fixed\_scaled} = (int32)(\text{delf\_delu\_fixed\_adapted}*(\text{int32}DELF\_DELU\_CONST)>>\text{SMALL\_Q-4}); \)
939
940 \( \text{Kp\_adapt\_fixed} = ((\Omega C_B\_BIDC\_FIXED)/\text{delf\_delu\_fixed\_scaled}); \)
941 if (\( \text{Kp\_adapt\_fixed}\geq VDC\_KP\_MAX\_FIXED \)) \( \text{Kp\_adapt\_fixed} = VDC\_KP\_MAX\_FIXED; \)
942 if (\( \text{Kp\_adapt\_fixed}\leq VDC\_KP\_MIN\_FIXED \)) \( \text{Kp\_adapt\_fixed} = VDC\_KP\_MIN\_FIXED; \)
943
944 \( \text{Kp\_adapt\_fixed} = (int32)(\text{Kp\textunderscore{float}.Kp}\_\text{SMALL\_Q\_SCALE}); \)
945
946 //*********
947 _BIDC\_FF()
948 // if (UF)
949 // {
950 \( \text{Iload\_abs}=\text{abs(\text{Iload\_fixed})}; \)
951 //Iload Feedforward - search algorithm
952 lo=0;
953 hi=\text{PERIOD\_2\_BIDC}-1;
954 while (hi>lo)
955 {
956 mid = ((hi-lo)/2)+lo;
957 if (\( \text{Iload\_abs}<\text{Iload\_FF\_fixed}[\text{mid}] \)) hi=mid-1; //in the bottom half
958 else if (\( \text{Iload\_abs}>=\text{Iload\_FF\_fixed}[\text{mid}] \)) lo=mid+1;
959 else if (\( \text{Iload\_abs}==\text{Iload\_FF\_fixed}[\text{mid}] \))
960 {
961 lo=mid;
962 break;
963 }
964 else if (\( \text{hi-lo}<10 \)) break;
965 }
966 if (\( \text{saturated}==1 \)) \( \text{BIDC\_FF}=0; \)
967 else
968 {
969 if (\( \text{Iload\_fixed}>0 \)) \( \text{BIDC\_FF} = \text{lo}; \)
970 }
else BIDC_FF = -lo;
}
}
// }
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 //fixed point
1003 VDCref_fixed = (int32)(mag_VDCref*FIXED_Q_SCALE);
1004 VDCerror_fixed = VDCref_fixed-VDCout_fixed;
1005 VDC_prop_fixed = (VDCerror_fixed*VDC_Kp_fixed)>>SMALL_Q;
1006 VDC_intnow_fixed = (VDC_prop_fixed*VDC_Ki_fixed)>>FIXED_Q;
1007 VDC_cont_signal_fixed = (((VDC_prop_fixed + VDC_int_fixed)*PERIOD_SCALE_BIDC)>>FIXED_Q);
1008  
1009 if (saturated==0)
1010 {
1011 VDC_PI_DBL.intsum += VDC_PI_DBL.intnow;
1012 VDC_int_fixed += VDC_intnow_fixed;
1013 }
1014 }
1015  
1016 VDC_PI_DBL.ctrl = (VDC_PI_DBL.prop+VDC_PI_DBL.intsum)*(double)PERIOD_SCALE_BIDC;  
1017  
1018 /
1019 /***************************************************************************/
1020 _BIDC_SET_PHASE()
1021 /***************************************************************************/
1022 if (OL_CL_BiDC==0) phase_shift = (int16)((-phase_OL*PERIOD_BIDC)/180);  //Open Loop
1023 else
1024 {
1025 if ((OL_CL_BiDC==1)||(OL_CL_BiDC==4)) //no FF
1026 phase_shift = (int16)(-VDC_cont_signal_fixed);  //fixed point
1027 else
1028 phase_shift = (int16)(-(VDC_cont_signal_fixed + BIDC_FF));  //CL with Feedforward
1029 }
1030 else
1031 phase_shift = (int16)(-(VDC_cont_signal_fixed + BIDC_FF));  //CL with Feedforward
1032 if(DT_COMP) phase_shift = phase_shift+phase_aug_DT_fixed;
1033 
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  
1057 /***************************************************************************/
1058 _BiDC_Modulator()
1059 /***************************************************************************/
1060 CMPR_Pri = PERIOD_2_BIDC;
1061 if (UF)
1062 {
1063 CMPR_Sec = (Uint16)(PERIOD_2_BIDC+phase_shift);
1064 }
1065 else
1066 CMPR_Sec = (Uint16)(PERIOD_2_BIDC-phase_shift);
1067 
1068 /* --------- Finish Experimental Interrupt Code --------- */
/* End Interrupt Code */

To Do: Grid Synch

this interrupt will determine the phase & frequency of the backemf (grid) to allow synchronising.

OUTPUTS()

out[0] = CMPR1;
out[1] = CMPR2;
out[2] = CMPR_Pri;
out[3] = CMPR_Sec;
out[4] = Iref_fixed/FIXED_Q_SCALE;
out[5] = VDC_PI_DBL.ref;
out[6] = VDC_prop_fixed;
out[7] = VDC_int_fixed;
out[8] = VDC_cont_signal_fixed;
out[9] = BIDC_FF;
out[10] = phase_shift:// VSI_intnow_fixed/FIXED_Q_SCALE;
out[12] = VSI_ctrl_fixed/FIXED_Q_SCALE;
out[13] = va;
out[14] = (int16)((VSI_ctrl_fixed*PERIOD_2_VSI)>>FIXED_Q);
out[15] = UF_VSI;
prev_ctrlclk = (int16)ctrlclk;
prev_VSI_ctrlclk = (int16)VSI_ctrlclk;
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

```vhdl
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 comm.
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  port(
27    -- CPLD requirements:
28    clock : IN STD_LOGIC;
29    nRESET : IN STD_LOGIC;
30
31    -- SPI interface:
32    sclk : IN STD_LOGIC; -- clock from the 2810
33    nSYNC : IN STD_LOGIC; -- chip select from the 2810, not the boy band
34    mosi : IN STD_LOGIC; -- data from the 2810
35    miso : INOUT STD_LOGIC; -- data to the 2810
36
37    debug : OUT STD_LOGIC;
38
39    -- Minibus interface:
40    -- Minibus enable (active low)
41    nMINIBUS : OUT STD_LOGIC;
42
43    -- Minibus read (active low)
44    nRD : OUT STD_LOGIC;
45
46    -- Minibus write (active low)
47    nWR : OUT STD_LOGIC;
48
49    -- Minibus memory addresses (active low)
50    MA : OUT STD_LOGIC_VECTOR (2 downto 0);
51
52    -- Minibus bus (debugging)
53    minibus : INOUT STD_LOGIC_VECTOR (7 downto 0);
54
55    -- Communications
56    SCI_MODE : OUT STD_LOGIC;
57
58    -- CAPQEP
59    DIGIS : IN STD_LOGIC_VECTOR (3 downto 0);
60
61    INDEX : IN STD_LOGIC_VECTOR (1 downto 0);
```
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

54  CAP  : OUT STD_LOGIC_VECTOR (5 downto 0);
55  -- INTSEL:
56  XINT1A  : IN STD_LOGIC;
57  XINT1B  : IN STD_LOGIC;
58  XINT1  : OUT STD_LOGIC;
59  -- PWM EVA:
60  PWMIN  : IN STD_LOGIC_VECTOR (5 downto 0);
61  TxPWM  : IN STD_LOGIC_VECTOR (1 downto 0);
62  PWMOUT : OUT STD_LOGIC_VECTOR (7 downto 0);
63
64  -- PWM EVB:
65  PWMB7 : OUT STD_LOGIC;
66  PWMB8 : OUT STD_LOGIC;
67  T3PWM : IN STD_LOGIC;
68  T4PWM : IN STD_LOGIC;
69
70  -- Analog Switch:
71  ANIN  : OUT STD_LOGIC_VECTOR (6 downto 0); -- MSB [IN1 IN2 IN3 INA INB INC IND] LSB
72  -- GPIO Interface
73  GPIO : INOUT STD_LOGIC_VECTOR (1 downto 0); -- generic digital IO
74
76  -- Error Signals
77  PDPINTA : IN STD_LOGIC;
78
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  signal miso_reg : STD_LOGIC_VECTOR (7 downto 0); -- data sent to the 2810 SPI is loaded into here
93
94  signal spi_pos_count : STD_LOGIC_VECTOR (4 downto 0); -- overall position of the SPI registers
95
96  signal command : STD_LOGIC_VECTOR (7 downto 0); -- command fed from SPI
97  signal data_1 : STD_LOGIC_VECTOR (7 downto 0); -- data byte 1 fed from SPI
98
99  signal minibus_data : STD_LOGIC_VECTOR (7 downto 0); -- data read from minibus
100 signal mb_wait : STD_LOGIC_VECTOR (2 downto 0); -- to implement a delay for minibus I/O
101
102 signal command_ok : STD_LOGIC;
103 signal read_point : STD_LOGIC;
104 signal write_point : STD_LOGIC;
105
106 signal SCIBMODE_reg : STD_LOGIC;
107 signal CAPQEP_reg : STD_LOGIC;
108
109 signal INTSEL_reg : STD_LOGIC_VECTOR (7 downto 0); -- data read from minibus
110 signal INTSRC_reg : STD_LOGIC_VECTOR (7 downto 0);
111
112 signal XINT1A_prev : STD_LOGIC;
113 signal XINT1B_prev : STD_LOGIC;
114 signal XINT1A_int : STD_LOGIC;
115 signal XINT1A1_reg : STD_LOGIC;
116 signal XINT1A2_reg : STD_LOGIC;
117 signal XINT1A3_reg : STD_LOGIC;
118 signal XINT1A4_reg : STD_LOGIC;
119 signal XINT1B_int : STD_LOGIC;
120 signal XINT1B1_reg : STD_LOGIC;
121 signal XINT1B2_reg : STD_LOGIC;
122 signal XINT1B3_reg : STD_LOGIC;
123 signal XINT1B4_reg : STD_LOGIC;
124
125 signal INTSEL_reg : STD_LOGIC;
126 signal INTSRC_reg : STD_LOGIC;
127 signal XINT1A_prev : STD_LOGIC;
128 signal XINT1B_prev : STD_LOGIC;
129 signal XINT1A_int : STD_LOGIC;
130 signal XINT1B_int : STD_LOGIC;
131 signal XINT1A1_reg : STD_LOGIC;
132 signal XINT1A2_reg : STD_LOGIC;
133
176
signal PWMOUT_reg : STD_LOGIC_VECTOR (7 downto 0);
signal PWMprev : STD_LOGIC;
signal PWMreq : STD_LOGIC_VECTOR (1 downto 0);
signal PWMdb_state : STD_LOGIC;
signal EVB_ENABLE : STD_LOGIC;
signal PWMBprev : STD_LOGIC;
signal PWMBreq : STD_LOGIC_VECTOR (1 downto 0);
signal PWMBdb_state : STD_LOGIC;
signal slow_clockA_per : STD_LOGIC_VECTOR (4 downto 0); -- period register
signal slow_clockB_per : STD_LOGIC_VECTOR (4 downto 0);
signal slow_clockA : STD_LOGIC; -- the actual clock
signal slow_clockB : STD_LOGIC;
signal slow_clockA_prev : STD_LOGIC; -- previous value
signal slow_clockB_prev : STD_LOGIC;
signal EVACOMCON_reg : STD_LOGIC_VECTOR (7 downto 0) := "00000000";
signal EVACONDB_reg : STD_LOGIC_VECTOR (7 downto 0);
signal EVBCOMCON_reg : STD_LOGIC_VECTOR (7 downto 0);
signal EVBCONDB_reg : STD_LOGIC_VECTOR (7 downto 0);
signal CAP_reg : STD_LOGIC_VECTOR (5 downto 0);
signal ANLGSW_reg : STD_LOGIC_VECTOR (7 downto 0);
signal GPIO_reg : STD_LOGIC_VECTOR (7 downto 0);
signal debug_reg : STD_LOGIC_VECTOR (7 downto 0); -- used to read/write for debug purposes
begin
spi: process( sclk,nSYNC,mosi,command_ok,command,data_1_ok,data_2_ok,data_1,write_point,write_point,spi_pos_count,minibus,GPIO,read_point,
SCIBMODE_reg,CAPQEP_reg,XINT1A_int,XINT1B_int,DIGIN,INDEX,INTSEL_reg,EVACOMCON_reg,EVACONDB_reg,EVBCOMCON_reg,
EVBCONDB_reg,ANLGSW_reg,debug_reg,PDPINTA)
begin
-- spi, instruction decode and minibus comms:
--if(nRESET = '1') then
if(nSYNC = '0') then -- chip set active
-- the use of flags to signify data ready points is not optimal. It can be adjusted so that within the
-- main program looking at the spi_pos_count determines when something is ready
-- Read the SPI:
if(falling_edge(sclk)) then
mosi_reg(conv_integer(spi_pos_count)) <= mosi; -- read in the value, MSB first
spi_pos_count <= spi_pos_count - 1; -- decrement counter (starts at 23)
if((spi_pos_count >= "10000")) then -- (>16) if we’re not as yet at the point of having valid data
command_ok <= '0'; -- ensure we don’t go off doing a command
data_1_ok <= '0';
data_2_ok <= '0';
write_point <= '0';
end if;
if(spi_pos_count = "01100") then -- WRITE POINT definition (currently set at pos=5 (2 clocks after the data_1
write_point <= '1'; -- register has been read in)
end if;
end if;

if(spi_pos_count = "00000") then -- DEPENDING ON HOW FAST IT DOES THIS, THESE NUMBERS MAY NEED TO BE CHANGED
  -- data_2 is irrelevant and hence is not implemented
  data_2_ok <= '1';
end if;

end if; -- end SPI falling clock case

--------------------------------------------------------------------------------------------------
-- Write to SPI:
if(rising_edge(sclk)) then
  if(spi_pos_count <= "01000") then -- if we're down to 7, time to put data out
    miso <= miso_reg(conv_integer(spi_pos_count-1)); -- send out the value, MSB first
    -- the -1 with the offset of 8 needs to be there for
    -- it to work for some weird reason
  end if;
end if; -- end SPI rising clock case

--------------------------------------------------------------------------------------------------
-- Chip Select inactive:
else
  miso <= '2'; -- high impedance 09/09/09 S.G
  spi_pos_count <= "10111"; -- reset the counter (set to 23)
  command_ok <= '0'; -- ensure we don't go off doing a command
  data_1_ok <= '0';
  data_2_ok <= '0';
  read_point <= '0';
  write_point <= '0';
  nMINIBUS <= '1';
end if; -- end chip select case

--------------------------------------------------------------------------------------------------------------
-- Minibus communications:
if(command_ok = '1') then -- if we're allowed to operate (this flag is set when a new SPI command comes in)
  if((command(7)='0' or command(6)='0')) then -- check it's a minibus command
    case command (7 downto 6) is
    when "00" =>
      nCS <= "110";
    when "01" =>
      nCS <= "101";
    when "10" =>
      nCS <= "011";
    when others => -- will never happen
      nCS <= "111";
    end case;
    MA <= command (3 downto 1); -- map MA bits across
    if(command(0)='1') then -- if it's a write
      if(data_2_ok = '1') then -- (0) if the write command has completed
        nWR <= '1';
        nMINIBUS <= '1';
      elsif(write_point='1') then -- (5) wait 2 SPI clocks before saying OK to write (setup time)
        nWR <= '0';
      elsif(data_1_ok = '1') then -- (7) if the data is now ready
        minibus <= data_1; -- place the data on the bus
        miso_reg <= "00000000"; -- zero the read register
        nRD <= '1';
        nMINIBUS <= '0'; -- signal "go"
      else -- not doing anything yet
    end case;
    MA <= command (3 downto 1); -- map MA bits across
    if(command(0)='0') then -- if it's a write
      if(data_2_ok = '1') then -- (0) if the write command has completed
        nWR <= '1';
        nMINIBUS <= '1';
      elseif(write_point='1') then -- (5) wait 2 SPI clocks before saying OK to write (setup time)
        nWR <= '0';
      elseif(data_1_ok = '1') then -- (7) if the data is now ready
        minibus <= data_1; -- place the data on the bus
        miso_reg <= "00000000"; -- zero the read register
        nRD <= '1';
        nMINIBUS <= '0'; -- signal "go"
      else -- not doing anything yet
    end if;
    if(command(7)='0' or command(6)='0') then -- check it's a minibus command
      case command (7 downto 6) is
      when "00" =>
        nCS <= "110";
      when "01" =>
        nCS <= "101";
      when "10" =>
        nCS <= "011";
      when others => -- will never happen
        nCS <= "111";
      end case;
      MA <= command (3 downto 1); -- map MA bits across
      if(command(0)='1') then -- if it's a write
        if(data_2_ok = '1') then -- (0) if the write command has completed
          nWR <= '1';
          nMINIBUS <= '1';
        elsif(write_point='1') then -- (5) wait 2 SPI clocks before saying OK to write (setup time)
          nWR <= '0';
        end if;
        if(data_1_ok = '1') then -- (7) if the data is now ready
          minibus <= data_1; -- place the data on the bus
          miso_reg <= "00000000"; -- zero the read register
          nRD <= '1';
          nMINIBUS <= '0'; -- signal "go"
        else -- not doing anything yet
      end if;
minibus <= "ZZZZZZZZ";
zh <= '1';
zh <= '1';
--
zh <= "111";
-- reset chip selects high (PJM ADDED)
nMINIBUS <= '1';

end if; -- end delay waited case

else -- else it's a read

-- following needs to take precedence over the reading point
if(data_1_ok = '1') then -- (7) we've waited long enough to read it in
zh <= '1';
zh <= '0';
nMINIBUS <= '0'; -- signal "go"

elsif(read_point = '1') then -- (12) wait 3 SPI clocks before reading in the data (setup time)
miss_reg <= minibus; -- read the data in from the bus

elsif(command_ok = '1') then -- (15) kinda redundant but needed to clean up nicely
zh <= '1'; -- signal a read
zh <= '0';
nMINIBUS <= '1'; -- signal "go"

else -- not doing anything yet
minibus <= "ZZZZZZZZ";
zh <= '1';
zh <= '1';
nMINIBUS <= '1';

end if; -- end delay waited case

end if; -- end read/write case

else -- else, it's not to do with the minibus

-- leave the minibus comms in a known state
minibus <= "ZZZZZZZZ";
zh <= '1';
zh <= '1';
nMINIBUS <= '1';

end if; -- end read/write case

-- Peripheral device setup:

when "01101" => -- GPIO (0xDA)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
GPIO_reg <= data_1; -- place it on the output

end if; -- end data_1 valid if

else -- else it must be a read
if(read_point='1') then -- (13) wait 2 SPI clicks until sampling the DigIO
miss_reg <= "000000" & GPIO; -- place it in the register ready for output

end if; -- end waited enough to sample DigIO

end if; -- end read/write command if

when "00001" => -- SCIBMODE (0xC2)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
SCIBMODE_reg <= data_1; -- place the written command

end if; -- end data_1 valid case

else
if(read_point='1') then
miss_reg <= "000000" & SCIBMODE_reg;

end if;
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

374 end if; -- end read/write command if

375 when "00010" => -- CAPQEP (0xC4)
376 if(command(0)='0') then -- write command
377 if(data_1_ok = '1') then -- if our write data is valid
378 CAPQEP_reg <= data_1(0);
379 end if; -- end data_1 valid if
380 else -- else, read command
381 if(read_point='1') then
382 miso_reg <= "0000000" & CAPQEP_reg;
383 end if; -- end read point reached if
384 end if; -- end read/write command if

385 when "00011" => -- INTSEL (0xC6)
386 if(command(0)='0') then -- write command
387 if(data_1_ok = '1') then -- if our write data is valid
388 INTSEL_reg <= data_1;
389 end if; -- end data_1 valid if
390 else -- else, read command -- SPECIAL, corresponds to (0xC7)
391 if(read_point='1') then
392 miso_reg <= "000000" & XINT1B_int & XINT1A_int; -- pass the interrupt that occured up to the DSP
393 if(data_2_ok = '1') then -- wait until the SPI command is done, then reset interrupts
394 int_clear <= '1'; -- clear the interrupt registers (this is done in the int process below)
395 else
396 int_clear <= '0'; -- provide means to let the interrupt registers to be re-filled
397 end if;
398 end if; -- end read point reached if
399 end if; -- end read/write command if

400 when "00100" => -- DEBUG ONLY (0xC8)
401 if(command(0)='1') then -- read command
402 if(read_point='1') then -- if our read data is valid
403 miso_reg <= INTSEL_reg;
404 end if; -- end data_1 valid if
405 end if;

406 when "00101" => -- INTSEL (0xC6)
407 if(command(0)='0') then -- write command
408 if(data_1_ok = '1') then -- if our write data is valid
409 if (data_1(0) = '1') then -- this means that you activate EVB
410 EVB_enable <= '1';
411 else
412 EVB_enable <= '0';
413 end if;
414 end if;

415 when "01000" => -- EVACOMCON (0xD0)
416 if(command(0)='1') then -- read command
417 if(read_point='1') then -- if our read data is valid
418 miso_reg <= EVACOMCON_reg;
419 end if; -- end data_1 valid if
420 end if;

421 when "00101" => -- DEBUG ONLY (0xC8)
422 if(command(0)='0') then -- write command
423 if(data_1_ok = '1') then -- if our write data is valid
424 if (data_1(0) = '1') then -- this means that you activate EVB
425 EVB_enable <= '1';
426 else
427 EVB_enable <= '0';
428 end if;
429 end if;

430 when "01001" => -- ADD_EVB: 11 00101 0 0000 0000 0000 000(0/1)
431 if(command(0)='1') then -- read command
432 if(read_point='1') then -- if our read data is valid
433 if (data_1(0) = '1') then -- this means that you activate EVB
434 EVB_enable <= '1';
435 else
436 EVB_enable <= '0';
437 end if;
438 end if;

439 when "01010" => -- EVACOMCON (0xD0)
440 if(command(0)='1') then -- read command
441 if(read_point='1') then -- if our read data is valid
442 if (data_1(0) = '1') then -- this means that you activate EVB
443 EVB_enable <= '1';
444 else
445 EVB_enable <= '0';
446 end if;

447 when "01011" => -- EVACOMCON (0xD0)
448 if(command(0)='1') then -- read command
449 if(read_point='1') then -- if our read data is valid
450 if (data_1(0) = '1') then -- this means that you activate EVB
451 EVB_enable <= '1';
452 else
453 EVB_enable <= '0';
454 end if;

455 when "01000" => -- ADD_EVB: 11 00101 0 0000 0000 0000 000(0/1)
456 if(command(0)='1') then -- read command
457 if(read_point='1') then -- if our read data is valid
458 if (data_1(0) = '1') then -- this means that you activate EVB
459 EVB_enable <= '1';
460 else
461 EVB_enable <= '0';
462 end if;

463 -- Mod Dinesh 20th November 2009
464 -- This code will run when this command is written to the address
465 -- ADD_EVB: 11 00101 0 0000 0000 0000 000(0/1)
466 -- It will do the following things:
467 -- 1) activate EVA_enable, which will
468 -- route EVB signals (first 4 only, single phase) to the gate drivers
469
470 -- Additional case added 21/10/2009 by S.G
471 -- If there's no fault
472 -- EVACOMCON_reg <= data_1 & "11111110"; -- zero the LSB
473 end if;
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

end if; -- end data_1 valid if

else -- else, read command
if(read_point='1') then
  miso_reg <= EVACONCON_reg;
end if; -- end read point reached if

end if; -- end read/write command if

----------------------------------------------------------------------

when "01001" => -- EVACONDB (0xD2)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
  EVACONDB_reg <= data_1;
end if; -- end data_1 valid if

else -- else, read command
if(read_point='1') then
  miso_reg <= EVACONDB_reg;
end if; -- end read point reached if

end if; -- end read/write command if

----------------------------------------------------------------------

when "01010" => -- EVBCOMCON (0xD4)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
  EVBCOMCON_reg <= data_1;
end if; -- end data_1 valid if

else -- else, read command
if(read_point='1') then
  miso_reg <= EVBCOMCON_reg;
end if; -- end read point reached if

end if; -- end read/write command if

----------------------------------------------------------------------

when "01011" => -- EVACONDB (0xD6)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
  EVACONDB_reg <= data_1;
end if; -- end data_1 valid if

else -- else, read command
if(read_point='1') then
  miso_reg <= EVACONDB_reg;
end if; -- end read point reached if

end if; -- end read/write command if

----------------------------------------------------------------------

when "01100" => -- ANLGSW (0xD8)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
  ANLGSW_reg <= data_1;
end if; -- end data_1 valid if

else -- else, read command
if(read_point='1') then
  miso_reg <= ANLGSW_reg;
end if; -- end read point reached if

end if; -- end read/write command if

----------------------------------------------------------------------

when "01101" => -- EVACONDB (0xD9)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
  EVACONDB_reg <= data_1;
end if; -- end data_1 valid if

else -- else, read command
if(read_point='1') then
  miso_reg <= EVACONDB_reg;
end if; -- end read point reached if

end if; -- end read/write command if

----------------------------------------------------------------------

when "01110" => -- EVBCOMCON (0xD4)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
  EVBCOMCON_reg <= data_1;
end if; -- end data_1 valid if

else -- else, read command
if(read_point='1') then
  miso_reg <= EVBCOMCON_reg;
end if; -- end read point reached if

end if; -- end read/write command if

----------------------------------------------------------------------

when "01111" => -- EVACONDB (0xD6)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
  EVACONDB_reg <= data_1;
end if; -- end data_1 valid if

else -- else, read command
if(read_point='1') then
  miso_reg <= EVACONDB_reg;
end if; -- end read point reached if

end if; -- end read/write command if

----------------------------------------------------------------------

when "011100" => -- AHLGSR (0x6B)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
  AHLGSR_reg <= data_1;
end if; -- end data_1 valid if

else -- else, read command
if(read_point='1') then
  miso_reg <= AHLGSR_reg;
end if; -- end read point reached if

end if; -- end read/write command if

----------------------------------------------------------------------

when "011101" => -- AHLGSR (0x6B)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
  AHLGSR_reg <= data_1;
end if; -- end data_1 valid if

else -- else, read command
if(read_point='1') then
  miso_reg <= AHLGSR_reg;
end if; -- end read point reached if

end if; -- end read/write command if

----------------------------------------------------------------------

when "011110" => -- ANLGSW (0xD8)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
  ANLGSW_reg <= data_1;
end if; -- end data_1 valid if

else -- else, read command
if(read_point='1') then
  miso_reg <= ANLGSW_reg;
end if; -- end read point reached if

end if; -- end read/write command if

----------------------------------------------------------------------

when "011111" => -- EVACONDB (0xD9)

if(command(0)='0') then -- write command
if(data_1_ok = '1') then -- if our write data is valid
  EVACONDB_reg <= data_1;
end if; -- end data_1 valid if

else -- else, read command
if(read_point='1') then
  miso_reg <= EVACONDB_reg;
end if; -- end read point reached if

end if; -- end read/write command if

----------------------------------------------------------------------
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

when "11111" => -- DEBUG (temp register)
if(command(0)='0') then -- write command
  if(data_1_ok = '1') then -- if our write data is valid
    debug_reg <= data_1; -- place it on the output
  end if; -- end data_1 valid if
else -- else it must be a read
  if(read_point='1') then
    miso_reg <= debug_reg; -- place it in the register ready for output
  end if;
end if; -- end read/write command if

when others =>
  miso_reg <= "00000000";
end case;
end if; -- end minibus check case
else -- command is not okay yet
  miso <= "ZZZZZZZZ";
  nWR <= '1'; -- reset all the minibus paraphernalia
  nRD <= '1';
  nCS <= "111"; -- reset chip selects high (PJM ADDED)
  nMINIBUS <= '1';
end if; -- end command not ready case

--end if; -- end reset if
end process spi;

--------------------------------------------------------------------------------------------------------------
-- CAPQEP passthrough:
-- regrettably, needs to be synchronous
CAPQEP_proc: process(clock,DIGIN,INDEX)
begin
if(rising_edge(clock)) then
  if(CAPQEP_reg='1') then -- defined in page 31 of the manual
    CAP_reg(0) <= DIGIN(0);
    CAP_reg(1) <= DIGIN(1);
    CAP_reg(3) <= DIGIN(2);
    CAP_reg(4) <= DIGIN(3);
    CAP_reg(0) <= INDEX(0);
    CAP_reg(1) <= INDEX(1);
  else
    CAP_reg(3) <= DIGIN(0);
    CAP_reg(4) <= DIGIN(1);
    CAP_reg(0) <= DIGIN(2);
    CAP_reg(1) <= DIGIN(3);
    CAP_reg(5) <= INDEX(0);
    CAP_reg(2) <= INDEX(1);
  end if; -- end CAPQEP register selection
end if; -- end rising clock edge
end process CAPQEP_proc;

--------------------------------------------------------------------------------------------------------------
int_sel: process(XINT1A,XINT1B,INTSEL_reg,INTSRC_reg,XINT1A_int,int_clear,XINT1A1_reg,XINT1A2_reg,XINT1A3_reg,XINT1A4_reg,
XINT1B1_reg,XINT1B2_reg,XINT1B3_reg,XINT1B4_reg)
begin
if(rising_edge(clock)) then -- defined in page 31 of the manual
  CAP_reg(0) <= DIGIN(0);
  CAP_reg(1) <= DIGIN(1);
  CAP_reg(3) <= DIGIN(2);
  CAP_reg(4) <= DIGIN(3);
  CAP_reg(2) <= INDEX(0);
  CAP_reg(5) <= INDEX(1);
else
  CAP_reg(3) <= DIGIN(0);
  CAP_reg(4) <= DIGIN(1);
  CAP_reg(0) <= DIGIN(2);
  CAP_reg(1) <= DIGIN(3);
  CAP_reg(5) <= INDEX(0);
  CAP_reg(2) <= INDEX(1);
end if; -- end CAPQEP register selection
end if; -- end rising clock edge
end process CAPQEP_proc;

--- major assumption is that XINT1 is asserted when an interrupt occurs
if(INTSEL_reg(2 downto 0) = "011") then -- rising edge (and enabled)
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

614  --if(rising_edge(XINT1A)) then
615  -- XINT1A1_reg <= '1';
616  end if; -- end XINT1A rising edge
617  end if; -- end if
620
621  if(INTSEL_reg(2 downto 0) = "101") then  -- falling edge (and enabled)
622  if(falling_edge(XINT1A)) then
623    XINT1A2_reg <= '1';
624  end if; -- end XINT1A falling edge
625  end if; -- end if
628
630  if(INTSEL_reg(2 downto 0) = "001") then  -- active low (and enabled)
631  if(XINT1A = '0') then
632    XINT1A3_reg <= '1';
633  end if;
634  end if;
636
638  if(INTSEL_reg(2 downto 0) = "111") then  -- active high (and enabled)
639  if(XINT1A = '1') then
640    XINT1A4_reg <= '1';
641  end if;
643  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  end if;
655
658  if(INTSEL_reg(6 downto 4) = "011") then  -- rising edge (and enabled)
659  if(rising_edge(XINT1B)) then
660    XINT1B1_reg <= '1';
661  end if; -- end XINT1B rising edge
662  end if; -- end if
665
667  if(INTSEL_reg(6 downto 4) = "101") then  -- falling edge (and enabled)
668  if(falling_edge(XINT1B)) then
669    XINT1B2_reg <= '1';
670  end if; -- end XINT1B falling edge
671  end if; -- end if
674
676  if(INTSEL_reg(6 downto 4) = "001") then  -- active low (and enabled)
677  if(XINT1B = '0') then
678    XINT1B3_reg <= '1';
679  end if;
680  end if;
682
684  if(INTSEL_reg(6 downto 4) = "111") then  -- active high (and enabled)
685  if(XINT1B = '1') then
686    XINT1B4_reg <= '1';
687  end if;
688  end if;
690
692  if((int_clear = '1') or (INTSEL_reg(0) = '0')) then  -- activated if the user reads the INTSRC register
693  XINT1B1_reg <= '0'; -- or if it’s been disabled
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

XINT1B2_reg <= '0';
XINT1B3_reg <= '0';
XINT1B4_reg <= '0';
end if;

XINT1B_int <= XINT1B1_reg or XINT1B2_reg or XINT1B3_reg or XINT1B4_reg;
end process int_sel;

--------------------------------------------------------------------------------------------------------------

-- UNTESTED!

pwm_sys: process(clock,TxPWM,PWMIN,EVACONDB_reg,EVACONDB_reg,slow_clockA,slow_clockA_prev)

variable wait_reg : STD_LOGIC_VECTOR (3 downto 0);
variable period : STD_LOGIC_VECTOR (3 downto 0);

begin

-- ASSUMPTION: T1PWM and T2PWM are the pass through digital IO for PWM7/8
-- no hysteresis implemented as yet, pass through 6 PWM outputs
-- this is a messy implementation as we can't look for a rising AND falling edge on the
-- PWM input, which means we need to clock it and compare between clock cycles - but because
-- we're doing this, we can't clock off the clock that's been divided down (can't write to
-- the same variables in different clock edges) so yeah, that's why it has this structure.

if (EVB_ENABLE = '0') then
PWMOUT(5 downto 0) <= PWMIN;
period := EVACONDB_reg(6 downto 3); -- map period register across (add a zero to match reg sizes)

if(EVACONDB_reg(7) = '1') then -- setup complimentary deadtime legs sources from T1PWM (t1 is MBB)

if(rising_edge(clock)) then
if(PWMdb_state = '0') then -- the change detection state
if(TxPWM(1) /= PWMprev) then -- if a state change has occurred
PWMOUT(7 downto 6) <= "00"; -- turn off the legs
PWMprev <= TxPWM(1); -- record what our new value is

if(TxPWM(1) = '1') then -- if we've requested to go high
PWMreq <= "10"; -- make a note of what the final state should be
else -- else we've requested to go low
PWMreq <= "01"; -- make a note of what the final state should be
end if;
PWMdb_state <= '1'; -- jump to the other state to wait
wait_reg := "0000"; -- reset the waiting register

end if; --end state change if

else -- the waiting and implementation state
if(slow_clockA /= slow_clockA_prev) then -- if our slow clock has toggled
slow_clockA_prev <= slow_clockA; -- record what our new value is
wait_reg := wait_reg + 1; -- increment period counter

if(wait_reg = period) then -- if we've waited long enough
PWMOUT(6) <= PWMreq(1); -- implement the requested state
PWMOUT(7) <= PWMreq(0); -- go back to waiting for a change
wait_reg := "0000"; -- reset the waiting register
end if;

end if; -- end deadband state toggle

end if;

else -- else, pass through the TxPWM legs
PWMOUT(7 downto 6) <= TxPWM;
end if;

end if; -- end main clock rising edge

-----------------------------------------------
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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

-- TESTED OK (with exception of /1 clock which freezes the DSP)
clock_div: process(clock,EVACONDB_reg,EVBCONDB_reg)
variable slow_clock_regA : STD_LOGIC_VECTOR(7 downto 0);
variable slow_clock_regB : STD_LOGIC_VECTOR(7 downto 0);
begin
  -- for PWM set A:
  if (EVACONDB_reg(2 downto 0) = "000") then -- special /1 case
    --slow_clock <= clock; -- INCLUSION OF THIS LINE WILL HANG THE DSP! (soft clock output?)
  else
    if(rising_edge(clock)) then
      slow_clock_regA := slow_clock_regA + 1;
      case EVACONDB_reg(2 downto 0) is -- switch on clock scaling values
        when "001" => -- /2 clock
          if(slow_clock_regA = "00000001") then
            slow_clockA <= not slow_clockA;
            slow_clock_regA := "00000000";
          end if;
        when "010" => -- /4 clock
          if(slow_clock_regA = "00000010") then
            slow_clockA <= not slow_clockA;
            slow_clock_regA := "00000000";
          end if;
        when "011" => -- /8 clock
          if(slow_clock_regA = "00000100") then
            slow_clockA <= not slow_clockA;
            slow_clock_regA := "00000000";
          end if;
        when "100" => -- /16 clock
          if(slow_clock_regA = "00001000") then
            slow_clockA <= not slow_clockA;
            slow_clock_regA := "00000000";
          end if;
        when others => -- /32 clock
          if(slow_clock_regA = "00010000") then
            slow_clockA <= not slow_clockA;
            slow_clock_regA := "00000000";
          end if;
      end case;
    end if; -- end clock rising
    end if; -- end special /1 case
  end if; -- end special /1 case
  -- for PWM set B:
  if (EVBCONDB_reg(2 downto 0) = "000") then -- special /1 case
    --slow_clockB <= clock;
  else
    if(rising_edge(clock)) then
      slow_clock_regB := slow_clock_regB + 1;
      case EVBCONDB_reg(2 downto 0) is -- switch on clock scaling values
        when "001" => -- /2 clock
          if(slow_clock_regB = "00000001") then
            slow_clockB <= not slow_clockB;
            slow_clock_regB := "00000000";
          end if;
        when others => -- /32 clock
          if(slow_clock_regB = "00010000") then
            slow_clockB <= not slow_clockB;
            slow_clock_regB := "00000000";
          end if;
      end case;
    end if; -- slow_clock rising
    end if; -- end special /1 case
end if;

when "010" => -- /4 clock
if(slow_clock_regB = "00000010") then
  slow_clockB <= not slow_clockB;
  slow_clock_regB := "00000000";
end if;

when "011" => -- /8 clock
if(slow_clock_regB = "00000100") then
  slow_clockB <= not slow_clockB;
  slow_clock_regB := "00000000";
end if;

when "100" => -- /16 clock
if(slow_clock_regB = "00001000") then
  slow_clockB <= not slow_clockB;
  slow_clock_regB := "00000000";
end if;

when others => -- /32 clock
if(slow_clock_regB = "00010000") then
  slow_clockB <= not slow_clockB;
  slow_clock_regB := "00000000";
end if;

end case;
end if; -- end clock rising
end if; -- end special /1 case
end process clock_div;

--------------------------------------------------------------------------------------------------------------

-- process to ensure a known startup value and a known fault value. Added 20/10/2009 by S.G.
pwm_en: process(EVACOMCON_reg,PDPINTA)
begin
if((EVACOMCON_reg(0) = '0') or (PDPINTA = '0')) then -- if the control register is zeroed or a hardware fault exists
  PWMen <= '0'; -- drive the PWM outputs low
else
  PWMen <= '1'; -- else, enable the passthrough
end if;
end process pwm_en;

--------------------------------------------------------------------------------------------------------------

-- asynchronous declarations
GPIO <= GPIO_reg(1 downto 0);
SCIBMODE <= SCIBMODE_reg;
XINT1 <= (XINT1A_int or XINT1B_int);
CAP <= CAP_reg;
debug <= slow_clockA;

end behaviour;
A.2.2 DSP Code – Dual Active Bridge

```c
/*
** developed by:
** creative power technologies, (C) copyright 2009
** author: A. McIver
** history:
** 23/04/09 AM - initial creation
** modified Dinesh Segaran
** Bidirectional DC-DC converter
*/

/*==========================================================================
** __Definitions()                                                  
==========================================================================*/

#define __SQRT2 1.4142135624
#define __SQRT3 1.7320508075
#define __PI 3.1415926535
#define __PI_2 __PI/2.0
#define __INVPI 1/__PI
#define __INVPI_2 1/__PI_2

#define SYSCLK_OUT (150e6)
#define HSPCLK (SYSCLK_OUT)
#define LSPCLK (SYSCLK_OUT/4)

/*==========================================================================
** __State_Simple_Definitions()                                       
==========================================================================*/
typedef void (* funcPtr)(void);
typedef struct
{
  funcPtr f;
  unsigned int call_count;
  unsigned char first;
}s state;

#define SS_NEXT(_s_,_f_) { _s_.f = (funcPtr)_f_; 
  _s_.call_count = 0;
  _s_.first = 1; }
#define SS_IS_FIRST(_s_) (_s_.first == 1)
#define SS_DONE(_s_) { _s_.first = 0; }
#define SS_DO(_s_) { _s_.call_count++; 
  ((*(_s_.f))()); }
#define SS_IS_PRESENT(_s_,_f_) (_s_.f == (funcPtr)_f_)

/*==========================================================================
** __Grab_Code_Definitions()                                             
==========================================================================*/
#define GRAB_INCLUDE
#endif
// grab array size
#define GRAB_LENGTH 20
#define GRAB_WIDTH 5
# define GRAB_GO 0
#define GRAB_WAIT 1
#define GRAB_TRIGGER 2
#define GRAB_STOPPED 3
#define GRAB_SHOW 4
```

188
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 */
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 \li 26/08/10 DS - Fixed Point implementation of the Adaptive Controlled
12 \li 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
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 \par Variables()
74 */
75
76 /* =========================================================================
77 \__Variables()
78 ========================================================================= */
79
80 #define FREQ_STEP 100
81
82 /*
83 \par Timers()
84 */
85
86 // Time related flag type
87 /**< This structure holds flags used in background timing. */
88 typedef struct
89 {
90 Uint16
91 msec:1, //\< millisecond flag
92 msec10:1, //\< 10ms flag
93 sec0_1:1, //\< tenth of a second flag
94 sec:1; //\< second flag
95 } type_time_flag;
96
97 #ifdef BUILD_RAM
98 // These are defined by the linker (see F2812.cmd)
99 extern Uint16 RamfuncsLoadStart;
100 extern Uint16 RamfuncsLoadEnd;
101
102 #endif
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```c
extern Uint16 RamfuncRunStart;
#endif

// Background variables
Uint16 quit = 0; ///< exit flag

// timing variable
type_time_flag time =
{
  0, 0, 0, 0 // flags
};

Uint32 idle_count = 0, ///< count of idle time in the background
idle_count_old = 0, ///< previous count of idle time
idle_diff = 0; ///< change in idle time between low speed tasks

char str[40]; // string for displays

//to display correctly
int initial = 0;

/*****************************************************************************/
_External_Variables()
/*****************************************************************************/
extern int16 FF_ENABLE,
AC_FF,
DT_CHP,
phase_aug_DT_fixed;
extern int32 I3_fixed,
I4_fixed;
/* =========================================================================*/
__Local_Function_Prototypes()
/*============================================================================*/
/* 1 second interrupt for display */
interrupt void isr_cpu_timer0(void);
/* display operating info */
void com_display(void);
/* display help */
void display_help(void);
/* process keyboard input */
void com_keyboard(void);
/* =========================================================================*/
__Grab_Variables()
/*============================================================================*/
#ifdef GRAB_INCLUDE
#pragma DATA_SECTION(grab_array, "bss_grab")
int16 step = 0,
grab_mode = GRAB_STOPPED,
grab_index,
set_vref = 0;
long volt_req = 10,
wo = 314;
#ifdef GRAB_DOUBLE
double grab_array[GRAB_LENGTH][GRAB_WIDTH];
#endif
#ifdef GRAB_LONG
long grab_array[GRAB_LENGTH][GRAB_WIDTH];
#endif
 görmek
#endif
#endif
/* =========================================================================*/
__Serial_input_variables()
/*============================================================================*/
#endif
```
**APPENDIX A. SIMULATION & EXPERIMENTAL CODE**

```c
// In 2810 modulation depth goes from 0 to 1000 (0-100%)
int mod_depth_serial = 10000;
// Fundamental modulation frequency in Hz
double phase_serial = 0.0;
int vref_serial = 10;
```

```c
int mod_depth_max = 15000;
int16 f_switch_serial = 20000;
int mosfet_count;
```

```c
void main(void)
{
    static int i = 0;
    // Disable CPU interrupts
    DINT;
    // Initialize DSP for PCB
    lib_mini2810_init(150/*MHz*/, 37500/*kHz*/, 150000/*kHz*/, LIB_EVAENCLK | LIB_EVBENCLK | LIB_ADCENCLK | LIB_SCIAENCLK | LIB_SCIBENCLK | LIB_MCBSPENCLK);
    InitGpio();
    spi_init(MODE_CPLD);
    cpld_reg_init();
    giib_init();
    // Initialize the PIE control registers to their default state.
    InitPieCtrl();
    // Disable CPU interrupts and clear all CPU interrupt flags:
    IER = 0x0000;
    IFR = 0x0000;
    // Initialize the PIE vector table with pointers to the shell Interrupt Service Routines (ISR).
    // This will populate the entire table, even if the interrupt is not used in this example. This is useful for debug purposes.
    // The shell ISR routines are found in DSP281x_DefaultIsr.c.
    // This function is found in DSP281x_PieVect.c.
    InitPieVectTable();
    #ifndef BUILD_RAM
    // Copy time critical code and Flash setup code to RAM
    // The RamfuncsLoadStart, RamfuncsLoadEnd, and RamfuncsRunStart symbols are created by the linker. Refer to the F2810.cmd file.
    MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);
    // Call Flash Initialization to setup flash waitstates
    // This function must reside in RAM
    InitFlash();
    #endif
    // Initialize the COM port
    bios_init_COM1(9600L);
    InitAdc();
    InitCpuTimers();
    // Configure CPU-Timer 0 to interrupt every tenth of a second:
    // 150MHz CPU Freq, 1ms Period (in uSeconds)
    ConfigCpuTimer(&CpuTimer0, 150.0/*MHz*/, 1000.0/*us*/);
    StartCpuTimer0();
    EALLOW; // This is needed to write to EALLOW protected register
    PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
    // Set SPI on loop back for testing
    spid_reg_init();
    gsk_init();
    // Initialize the PIE control registers to their default state.
    InitPieCtrl();
    // Disable CPU interrupts and clear all CPU interrupt flags:
    IER = 0x0000;
    IFR = 0x0000;
    // Initialize the PIE vector table with pointers to the shell Interrupt Service Routines (ISR).
    // This will populate the entire table, even if the interrupt is not used in this example. This is useful for debug purposes.
    // The shell ISR routines are found in DSP281x_DefaultIsr.c.
    // This function is found in DSP281x_PieVect.c.
    InitPieVectTable();
    #ifndef BUILD_RAM
    // Copy time critical code and Flash setup code to RAM
    // The RamfuncsLoadStart, RamfuncsLoadEnd, and RamfuncsRunStart symbols are created by the linker. Refer to the F2810.cmd file.
    MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);
    // Call Flash Initialization to setup flash waitstates
    // This function must reside in RAM
    InitFlash();
    #endif
    // Initialize the COM port
    bios_init_COM1(9600L);
    InitAdc();
    InitCpuTimers();
    // Configure CPU-Timer 0 to interrupt every tenth of a second:
    // 150MHz CPU Freq, 1ms Period (in uSeconds)
    ConfigCpuTimer(&CpuTimer0, 150.0/*MHz*/, 1000.0/*us*/);
    StartCpuTimer0();
    EALLOW; // This is needed to write to EALLOW protected register
    PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
    // This is needed to disable write to EALLOW protected registers
    EDIS;
    // Enable TINT0 in the PIE: Group 1 interrupt 7
    PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
    IER |= M_INT1; // Enable CPU interrupt 1
    vxi_init();
}
```
EnableInterrupts();

//waste some time, so that the program can finish writing to the screen

#ifdef GRAB_INCLUDE
GrabInit();
#endif
spi_init(MODE_DAC);
spi_set_mode(MODE_DAC);
dac_init();
dac_set_ref(DAC_MODULE_D1,DAC_INT_REF);
dac_power_down(DAC_MODULE_D1,0x0F);
dac_write(DAC_MODULE_D1,DAC_WRn_UPn,DAC_ADDR_ALL,2047);
spi_set_mode(MODE_CPLD); //Use mode setting for CPLD for SPI to initialize SPI setting
DISABLE_CPLD();

while(quit == 0)
{
    com_keyboard(); // process keypresses
    if (time.msec != 0) // millisecond events
    {
        time.msec = 0;
        vsi_state_machine();
    }
    else if (time.msec10 != 0) // ten millisecond events
    {
        time.msec10 = 0;
    }
    else if (time.sec0_1 != 0) // tenth of second events
    {
        time.sec0_1 = 0;
        switch(initial)
        {
            /* case 0 never happens */
            case 1: puts_COM1("\n GIIB-Based Bidirectional DC-DC Converter 2011");break;
            case 2: puts_COM1("\ntt/d - start/end\n");break;
            #ifdef OPEN_LOOP
            case 3: puts_COM1("\tz/Z - Small/Large Phase Shift Increase\n"); break;
            case 4: puts_COM1("\tx/X - Small/Large Phase Shift Decrease\n"); break;
            #endif
            #ifdef CLOSED_LOOP
            //Vref
            case 5: puts_COM1("\tm/M - Small/Large Vref Increase\n"); break;
            case 6: puts_COM1("\tn/N - Small/Large Vref Decrease\n"); break;
            //FF
            case 7: puts_COM1("\tf/F - Feed Forward Disable/Enable\n"); break;
            case 8: puts_COM1("\ta/A - AC/DC Feed Forward Selection\n"); break;
            //DT Comp
            case 9: puts_COM1("\tc/C - Deadtime Compensation Disable/Enable\n"); break;
            #endif
            case 10: puts_COM1("\tg/h - Start/Display Grab\n");break;
            case 11: puts_COM1("\ts - Stop Grab\n");break;
            case 12: puts_COM1("\td - Display Help\n");break;
            default: break;
        }
        if (initial<20) initial++;  
    }
    if(GrabShowTrigger() && i < GRAB_LENGTH){
        //GrabDisplay(0xFFFF);
        GrabDisplay(i);
        i++;
        //GrabStop();
    } else if(GrabShowTrigger() && i == GRAB_LENGTH){
        GrabStop();
        i = 0;
    }
    else if (time.sec != 0) // update every 1sec
    {
        // puts_COM1("\n counter:");
        // put_d(initial);
        if (initial<20) initial++;
    }
    if(GrabShowTrigger() && i < GRAB_LENGTH){
        //GrabDisplay(0xFFFF);
        GrabDisplay(i);
        i++;
        //GrabStop();
    } else if(GrabShowTrigger() && i == GRAB_LENGTH){
        GrabStop();
        i = 0;
    } else if (time.sec != 0) // update every 1sec
    {
        // puts_COM1("\n counter:");
        // put_d(initial);
time.sec = 0;
idle_count = idle_count_old;
if (initial>=15) com_display();
else // low priority events

 idle_count++;
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```c
401     putl(phase_aug_DT_fixed);
402     }
403     putl(I3_fixed);
404     puts_COM1(" I3_fixed:");
405     putl(I4_fixed);
406     puts_COM1(" I4_fixed:");
407     }
408     /* end com_display */
409     */
410     /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
411     void com_keyboard()
412     {  // puts_COM1("KEY");
413         if (kbhit_COM1())
414             switch (c)
415             {
416                 case 'q': quit = 1;
417                 break;
418                 case 'e': vsi_enable();
419                     puts_COM1("e");
420                     break;
421                 case 'd':
422                     vsi_disable();
423                     break;
424                 case 'z':// lead secondary bridge phase shift (small)
425                     if((phase_serial+PHASE_STEP_SMALL) < 90.0)
426                         phase_serial +=PHASE_STEP_SMALL;
427                     else{
428                         phase_serial = 90.0;
429                     }
430                     vsi_set_phase(phase_serial);
431                     break;  //Open Loop phase shift variation
432                     #ifdef OPEN_LOOP
433                 case 'x':// lag secondary bridge phase shift (small)
434                     if((phase_serial-PHASE_STEP_SMALL) > -90.0)
435                         phase_serial -=PHASE_STEP_SMALL;
436                     else{
437                         phase_serial = -90.0;
438                     }
439                     vsi_set_phase(phase_serial);
440                     break;
441                 case 'Z':// increase phase shift (small)
442                     if((phase_serial+PHASE_STEP_LARGE) < 90.0)
443                         phase_serial +=PHASE_STEP_LARGE;
444                     else{
445                         phase_serial = 90.0;
446                     }
447                     vsi_set_phase(phase_serial);
448                     break;
449                 case 'X'://decrease phase shift (small)
450                     if((phase_serial-PHASE_STEP_LARGE) > -90.0)
451                         phase_serial -=PHASE_STEP_LARGE;
452                     else{
453                         phase_serial = -90.0;
454                     }
455                     vsi_set_phase(phase_serial);
456                     break;
457                 }  //Set desired Voltage Reference
```
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 /*Enable/Disable Feed Forward*/
486 case 'f': FF_ENABLE=0;break;
487 case 'F': FF_ENABLE=1;break;
488 /* AC/DC Feed Forward Selection*/
489 case 'a': AC_FF=0;break;
490 case 'A': AC_FF=1;break;
491 /*enable/disable Deadtime compensation*/
492 case 'c': DT_COMP=0;break;
493 case 'C': DT_COMP=1;break;
494 case 'H': // write help info
495 initial=0;break;
496 #ifdef GRAB_INCLUDE
497 case 'g': /* grab interrupt data */
498 GrabClear();
499 GrabStart();
500 GrabRun();
501 break;
502 case 'h':
503 puts_COM1("Grab Display
Index");
504 GrabShow();
505 break;
506 case 's': /* stop grab display */
507 GrabClear();
508 GrabStop();
509 break;
510 #endif
511 */
512 /* 1 second CPU timer interrupt.*/
513 
514 static struct
515 {
516 Uinit16
517 msec,
518 msec10,
519 msec100,
520 sec;
521 } i_count =
522 {
523 0, 0, 0
524 };
525 for (ii=0; ii<WD_TIMER_MAX; ii++)
526 {  
527 if (wd_timer[ii] > 0)
528 wd_timer[ii]--;  
529 }  
530 };  
531 if (i_count.msec >= 10)
532 {  
533 i_count.msec = 0  
534 i_count.msec10++;
535 }  
536 if (i_count.msec10 >= 10)
537 {  
538 i_count.msec = 0  
539 i_count.msec10 = 0;  
540 i_count.msec100++;  
541 if (i_count.msec100 >= 10)
542 {  
543 i_count.msec100 = 0  
544 i_count.msec100 = 0;}  
545 i_count.msec100++;
546 }  
547 i_count.msec100 = 0;  
548 i_count.msec100 = 0;  
549 i_count.msec100++;
550 i_count.msec100 = 0;  
551 if (i_count.msec100 >= 10)
552 {  
553 i_count.msec100 = 0  
554 i_count.msec100 = 0;}  
555 i_count.msec100 = 0;  
556 i_count.msec100 = 0;  
557 if (i_count.msec100 >= 10)
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 // Acknowledge this interrupt to receive more interrupts from group 1
574 PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
575 /* end isr_cpu_timer0 */
576 */
577
578 /*----------------------------------------------------------------------*/
579 _Exported_Functions()
580 /*----------------------------------------------------------------------*/
581 /*----------------------------------------------------------------------*/
582 _Grab_Functions()
583 /*----------------------------------------------------------------------*/
584 #ifdef GRAB_INCLUDE
585 
586 void GrabInit(void)
587 {
588 
589 
590 Uint16 i,j;
591 for (i=0; i<GRAB_LENGTH; i++)
592 {
593 for (j=0; j<GRAB_WIDTH; j++)
594 
595 
596 grab_array[i][j] = 0;
597 
598 GrabClear();
599 }
600 
601 if (index == 0xFFFF)
602 
603 
604 puts_COM1("index");
605 for (i=0; i<GRAB_WIDTH; i++)
606 
607 puts_COM1("g");
608 put_d(i);
609 
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 puts_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 
639 */
640 
641 */
642 
643 
644 
645 
646 
647 
648 
649 
650 
651 
652 
653 
654 
655 
656 
657 
658 
659 
660 
661 
662 
663 
664 
665 
666 
667 
668 
669 
670 
671 
672 
673 
674 
675 
676 
677 
678 
679 
680 
681 
682 
683 
684 
685 
686 
687 
688 
689 
690 
691 
692 
693 
694 
695 
696 
697 
698 
699 
700 
701 
702 
703 
704 
705 
706 
707 
708 
709 
710 
711 
712 
713 
714 
715 
716 
717 
718 
719 
720 
721 
722 
723 
724 
725 
726 
727 
728 
729 
730 
731 
732 
733 
734 
735 
736 
737 
738 
197
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

1//**
2* file
3*/brief VSI definitions
4
5/par Developed By:
6Creative Power Technologies, (C) Copyright 2009
7A. McIver
8/par History:
9\li 23/04/09 AM - initial creation
10\li Modified Dinesh Segaran
11\li 11/11/09 BS - Turning this into a GIIB-Based Bidirectional DC-DC Converter
12\li 26/08/10 BS - Fixed Point implementation of the Adaptive Controlled
13\li Bidirectional DC-DC Converter
1415*/
16
17/===============================
18#include()
19===============================
20
21
22/===============================
23#include()
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/**************************************************************************
36* @name VSI Status bit definitions
37* @}*/
38#define VSI_INIT 0x0000
39#define VSI_GATECHARGE 0x0001 // VSI is running
40#define VSI_RAMP 0x0002 // VSI is running
41#define VSI_RUNNING 0x0003 // VSI is running
42#define VSI_SETTLED 0x0004 // set when target reached
43#define VSI_STOP 0x0005 // VSI is running
44#define VSI_FAULT 0x0006 // set when fault present in VSI system
45
46/**************************************************************************
47* @name Fault Codes
48* @}*/
49#define FAULT_VSI_IAC_OL 0x0001
50#define FAULT_VSI_IAC_OC 0x0002
51#define FAULT_VSI_VDC_OV 0x0004
52#define FAULT_VSI_VDC_UV 0x0008
53#define FAULT_VSI_PDPINT 0x0010
54#define FAULT_VSI_SPI 0x0020
55
56/**************************************************************************
57* _CONTROLLER_FORM()
58* @}*/
59#define SW_FREQ_BIDC ((int32)20000)
60#define PERIOD_2_BIDC ((int32)HSPCLK/SW_FREQ_BIDC/2/2) // Carrier timer half period in clock ticks
61#define PERIOD_BIDC ((int32)HSPCLK/SW_FREQ_BIDC/2)
62
63#define SW_FREQ_VSI ((int32)5000)
64#define PERIOD_2_VSI ((int32)HSPCLK/SW_FREQ_VSI/2/2) // Carrier timer half period in clock ticks
65#define PERIOD_VSI ((int32)HSPCLK/SW_FREQ_VSI/2)
66
67/**************************************************************************
68* Closed or Open Loop selection
69* @}*/
70#define CLOSED_LOOP 1
71#ifndef OPEN_LOOP 1
72#define OPEN_LOOP 1
73#endif
74
75#ifndef OPEN_LOOP
76#ifndef CLOSED_LOOP 1
77#endif
78#endif
79#endif
80#define ADAPTIVE 1
#define FEED_FORWARD 1

/********************
_FUNCTIONS()
********************/

/// ADC calibration time
#define ADC_CAL_TIME 1 // seconds

#define ADC_COUNT_CAL (Uint16)(ADC_CAL_TIME * 20000 * 2.0)

/// DC averaging time
#define ADC_DC_TIME 0.1 // seconds

#define ADC_COUNT_DC (Uint16)(ADC_DC_TIME * 20000 * 2.0)

#define ADC_REAL_SC 1

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/// RMS scaling
#define ADC_RMS_PS 4

//DA2810 Scaling - 3V and 12 bits
easier to multiply result by 3 and shift back by 12.
#define ADC_DA_SCALE_MULT (long)3 //3.0/4096.0 - scaled by FIXED_Q+5 cos num is so small
#define ADC_DA_SCALE_SHIFT 12

#define ADC_DA_SHIFT 4

//GIIB Scaling Resistors
#define RFB_GIIB_VAC (long)10000 //10000.0 //feedback resistor on GIIB board
#define RIN_GIIB_VAC (long)(150000+150000) //150000.0+150000.0 //preloaded input resistor on GIIB board
#define RFB_GIIB_VDC (long)(10000) //feedback resistor on GIIB board
#define RIN_GIIB_VDC (long)(150000+150000+150000) //50000.0+150000.0+150000.0 //preloaded input resistor on GIIB board

#define RIN_GIIB_ADD_VAC (long)560000 //additional scaling resistor on GIIB board
#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))
#define VAC_GIIB_GAIN ((-1.0*(double)RFB_GIIB_VAC*FIXED_Q_SCALE)/(double)RIN_GIIB_TOTAL_VAC) //scaled by FIXED_Q
#define VAC_GIIB_GAIN_INV (long)(((double)FIXED_Q_SCALE*(double)FIXED_Q_SCALE)/(double)VAC_GIIB_GAIN) //scaled by 2^9

#define RIN_GIIB_ADD_VDC (long)470000 //additional scaling resistor on GIIB board
#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))
#define VDC_GIIB_GAIN ((-1.0*(double)RFB_GIIB_VDC)/(double)RIN_GIIB_TOTAL_VDC) //scaled by FIXED_Q
#define VDC_GIIB_GAIN_INV (long)(FIXED_Q_SCALE/VDC_GIIB_GAIN) //scaled by 2^9

#define RUP_MINI1 (long)6800
#define RUP_MINI2 (long)4700
#define RDOWN_MINI_TOTAL (long)((RUP_MINI1*RUP_MINI2)/(RUP_MINI1+RUP_MINI2))
#define RDWN_MINI (long)6800
#define RIN_MINI (long)12000
#define RDOWN_MINI_TOTAL (long)((RDWN_MINI*RIN_MINI)/(RDWN_MINI+RIN_MINI))

#define ADC_MINI_GAIN (((double)(RUP_MINI_TOTAL*RDOWN_MINI_TOTAL))/((double)((RUP_MINI_TOTAL+RDOWN_MINI_TOTAL)*RIN_MINI))) //is a double
#define ADC_MINI_GAIN_INV (long)(FIXED_Q_SCALE/ADC_MINI_GAIN) //scaled by FIXED_Q

#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
#define ADC_OFFSET (long)(((MINI_LEVEL_SHIFT<<ADC_DA_SCALE_SHIFT)>>FIXED_Q)/ADC_DA_SCALE_MULT) //in counts

#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)) //scaled by 4096 is the dac scale shift by 12. x4 is to scale to va.
#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(ADC_DA_SCALE_MULT)) //scaled by 4096 is the dac scale shift by 12. x4 is to scale to va.
#define DAC_SCALE ((2048.0)/10.0) //2048 counts gives 10V
#define VGEN_CAL ((long)1200) //this is done so that the two bridges voltage supplies don't fight.

#define VGEN_ANALOG_GAIN ((long)((double)(DAC_SCALE*4.0*(double)ADC_MINI_GAIN_INV*(double)ADC_DA_SCALE_MULT)/(double)FIXED_Q.Scale(ADC_DA_SCALE_MULT)) //scaled by 4096 is the dac scale shift by 12. x4 is to scale to va.
#define ADC_OFFSET (long)((double)((double)VDC_GAIN_INV*(double)ADC_MINI_GAIN_INV*(double)ADC_DA_SCALE_MULT)/(double)FIXED_Q.Scale(ADC_DA_SCALE_MULT)) //scaled by 4096 is the dac scale shift by 12. x4 is to scale to va.
#define ADC_OFFSET (long)((double)(ADC_CAL_TIME*20000*2.0)) //scaled by 4096 is the dac scale shift by 12. x4 is to scale to va.
#define CT_RATIO 4000.0 //For LA 100P SP13, it is 1000, for LA 100P - 2000
#define CT_TURNS 2.0
#define BURDEN_R 270.0 //Burden resistor - Load current
#define CT_RATIO 4000.0 //For LA 100P SP13, it is 1000, for LA 100P - 2000
#define CT_TURNS 2.0
#define BURDEN_R 270.0 //Burden resistor - Load current

#define CT_RATIO 4000.0 //For LA 100P SP13, it is 1000, for LA 100P - 2000
#define CT_TURNS 2.0
#define BURDEN_R 270.0 //Burden resistor - Load current
#define RIN1_GIIB_I 10000.0 //Input resistor to GIIB op amp stage
#define RIN2_GIIB_I 10000.0 //Input resistor to GIIB op amp stage
#define RIN_GIIB_TOTAL_I ((RIN1_GIIB_I*RIN2_GIIB_I)/(RIN1_GIIB_I+RIN2_GIIB_I)) //Input resistor to GIIB op amp stage
#define RFB_GIIB_I 10000.0
#define I_GIIB_GAIN (-1.0*RFB_GIIB_I/RIN_GIIB_TOTAL_I) //Voltage gain of amplifier on GIIB for current (double)
#define I_GIIB_GAIN_INV (1.0/I_GIIB_GAIN) //Voltage gain of amplifier on GIIB for current (double)
#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)

/*End ADC Scaling*/

/* Topology parameters */
#define C 11.9e-6 //used to be 30.8uF, now is 31.4
#define INV_CNEG -1.0/C
#define L 132e-6
#define R_L 0.01
#define R_L_2 R_L*R_L
#define OMEGA_BIDC_L (OMEGA_BIDC*L)
#define OMEGA_BIDC_L_2 ((OMEGA_BIDC*L)*(OMEGA_BIDC*L))
#define NPRI (10.0)
#define NSEC (11.0)
#define NPRI_NSEC (double)(NPRI/NSEC)
#define NPRI_NSEC_FIXED ((int32)(NPRI_NSEC*FIXED_Q_SCALE))
#define VIN (200.0)
#define _4VIN (4.0*VIN)
#define VIN_FIXED (long)(VIN*FIXED_Q_SCALE)
#define INV_NP_NS_VIN (double)(NPRI/(NSEC*VIN))
#define INV_NP_NS_VIN_FIXED (long)((NPRI*32768)/(NSEC*VIN)) // is shifted by FIXED_Q+4
#define VDCPRI VIN/2.0
#define VDCPRI_FIXED (long)((long)VIN/2)

/* constants */
#define PI 3.14159265358979
#define _2PI 2*PI
#define PI_2 1.57079632679489
#define INV_PI 0.31830988618379
#define INV2_PI 0.636619772367581
#define INV2_PI_FIXED (long)(INV2_PI*FIXED_Q_SCALE)

/* sine table definitions */
#define DEG_TO_COUNT ((double)(3750.0/180.0));
#define COUNT_TO_RAD PI/3750.0
#define COUNT_TO_RATIO 1.0/(2*3750.0)
#define RAD_TO_COUNT 3750.0/PI
#define DEG_TO_RAD PI/180.0
#define RAD_TO_DEG 180.0/PI
#define COUNT_TO_SINTABLE (long)((4294967296.0/((double)PERIOD_BIDC*2.0))

/* Controller definitions */
#define TS_BIDC ((double)(1.0/SW_FREQ_BIDC))
#define OMEGA_BIDC (2.0*PI*(double)SW_FREQ_BIDC)
#define FSAMPLE_BIDC (1.0*SW_FREQ_BIDC)
#define TSAMPLE_BIDC (1.0/FSAMPLE_BIDC)
#define MAX_PHASE (PERIOD_2_BIDC-1) //maximum phase shift. above this, the nonlinearity is too great
#define T_DELAY_BIDC (1.0*TSAMPLE_BIDC) //60 deg phase margin
#define OMEGA_C_BIDC_FIXED ((int32)(OMEGA_C_BIDC*FIXED_Q_SCALE*4.0))
#define PERIOD_SCALE_BIDC ((int32)(PERIOD_2_BIDC*INV2_PI))
#define DAC_SCALE_VREF 2048.0/50.0
#define DAC_SCALE_PHASE 2048.0/100.0
#define COUNT_TO_DAC (COUNT_TO_RAD*RAD_TO_DEG*DAC_SCALE_PHASE)

/*Adaptive controller parameters */
#define DELF_DELU_SCALE (NPRI/NSEC)*16*VDCPRI/C/(PI*PI) //divide by 16.0 is for scaling purposes
#define DELF_DELU_CONST (VDCPRI*NPRI_NSEC/(C*PI*PI)) //divide by 16.0 is for scaling purposes
#define BIDC_FF_CONST ((16.0*VDCPRI*NPRI_NSEC/(PI*PI))/OMEGA_BIDC_L)
#define VDC_KI (double)(OMEGA_C_10_BIDC/FSAMPLE_BIDC)
#define VDC_KI_FIXED (int32)(VDC_KI*FIXED_Q_SCALE)

//deadtime compensation parameters
#define DEADBAND_BIDC 1.5e-6
#define DB_DEG_BIDC (360.0*SW_FREQ_BIDC*DEADBAND_BIDC)
#define DB_RAD_BIDC (DB_DEG_BIDC*DEG_TO_RAD)
#define DB_RATIO_BIDC (DB_RAD_BIDC/_2PI)
#define DEADBAND_COUNT_BIDC ((int16)(DEADBAND_BIDC*HSPCLK))

// Step size and max output voltage
#define VREF_MAX 201
#define VREF_MIN 10
#define VREF_STEP_S 1
#define VREF_STEP_L 10

/* =========================================================================
__Macros()
============================================================================ */

#define VSI_DISABLE() {
EvaRegs.ACTRA.all = 0x0000;
EvbRegs.ACTRB.all = 0x0000;
}

#define SIN_TABLE_READ(PHASE,SIN_VAL){
SIN_VAL = sin_table[(PHASE>>22)|0x01];
VAL_DIFF = (sin_table[((PHASE>>22)+1)|0x01]) - SIN_VAL;
SIN_VAL += (int16)((int32)(VAL_DIFF)*((int32)(PHASE&0x3FFFFF))>>22);
}

// phase is a 32bit number, but the index is only 10 (513 values).
// shift right by 22 to know where to aim in the sine table. interpolate using the last 6 bits.

#define SIN_TABLE_READ_DINESH(_SIN_COUNT_, _VAL_){
_INDEX_ = _SIN_INDEX_ = _SIN_COUNT_>>FIXED_Q;
_INDEX_ = ((_INDEX_>>6)|0x001);
_INDEX_ = (int16)(_INDEX_>>FIXED_Q);
不断向下滑动
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

 /* =========================================================================
  __Function_Prototypes() 
============================================================================ */

/// Core interrupt initialisation
void vsi_init(void);

/// Core interrupt VSI state machine for background processing
void vsi_state_machine(void);

/// Enables vsi switching (assuming no faults)
void vsi_enable(void);

/// Disable vsi switching
void vsi_disable(void);

// Set the target output phase shift
void vsi_set_phase(double phase_cont_signal);

// Set the desired output voltage
void vsi_set_vref(int16 vref);

/// Returns the status of the VSI
Uint16 vsi_get_status(void);

/// Report what faults are present in the VSI
Uint16 vsi_get_faults(void);

/// Clear some detected faults and re-check.
void vsi_clear_faults(void);

// Print the current state of the state machine
void get_state(void);

// Calibrate ADCs online
void calibrate_adc(void);

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
This file contains the code for the core interrupt routine for the CVT system. This interrupt is the central system for the signal generation and 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 triggers this interrupt. Its tasks are:

- Read internal ADC results
- Perform internal analog averaging and RMS calculations
- Update VSI phase and switching times

Developed By:
Creative Power Technologies, (C) Copyright 2009

Author: A. McIver

History:
- 23/04/09 AM - initial creation
- Modified Dinesh Segaran
- 11/11/09 DS - Turning this into a Giib-Based Bidirectional DC-DC Converter
- 26/08/10 DS - Fixed Point implementation of the Adaptive Controlled Bidirectional DC-DC Converter

Developed over to rebuild the open-giib bidirectional DC-DC converter

This code is ported over to rebuild the open-giib bidirectional DC-DC converter

13/4/2011 - moved code to a flash project
- replaced low voltage capacitors. New operating voltage - 200V at 1:1 transfer ratio
- fixed state machine to actually display correctly.

14/4/2011 - attempt to modulate an open-loop bidirectional DC-DC converter at 200V
- scaling resistors - VAC inputs are used to measure DC. Initially scaled to measure +/-450V,
  now want to measure +/-250V. 560kohms resistor needed

26/08/10 DS - Fixed Point implementation of the Adaptive Controlled Bidirectional DC-DC Converter

Compilers standard include files
#include <math.h>

Processor standard include files
#include <DSP281x_Device.h>

#include <bios0.h>
#include <bios1.h>

Board standard include files
#include <lib_mini2810.h>
#include <dac_ad56.h>
#include <lib_cpld.h>
#include <lib_giib.h>

Local include files
#include "main.h"
#include "conio.h"
#include "vsi.h"

Stage 1:
- Synchronize Carriers. use zaki's code.
- Synchronize the VSI to the BiDC because the BiDC uses a lot of DIGIO pins already.
- Use the shielded ribbon cable for this. Build Loopback function and test.
- Loopback cable - GP1OB0-4 (PWMB1-4) are routed back into DIGINS-8. so Pins 1-4 are connected to 13-16.
- On the BiDC, send out a sync pulse at 5kHz (1 every 8 interrupts) on GP1OB4.
- This is DIGOUT5, pin 5 on the 20-pin header.
- On the VSI, bring the sync pulse into CAP2. this is on DIGINS, which is pin 16. 15 connect pins 5 & 16.
- Also connect all the GNDs on the 20-pin header together. i.e, leave pins 16 & 20.
- Disconnect VCC, i.e cut pins 17 & 19
- That lets you lift sync code from GridCon set, and also the fault trigger when sync is lost.

Stage 2:
- Phase & Modulation depth information. Via SPI or via DAC?
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```
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  * Definitions
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 raw, ///< raw ADC result from last sampling
102     filt; ///< decaying average fast filter of raw data
103     int32 raw_sum, ///< interrupt level sum of data
104     raw_sum_bak, ///< background copy of sum for averaging
105     dc_sum, ///< interrupt level sum
106     dc_sum_bak; ///< background copy of sum for processing
107     double real; ///< background averaged and scaled measurement
108 } type_adc_ch;
109 
110  // Internal ADC storage type
111  /** This structure holds all the analog channels and some related variables
112  * for the averaging and other processing of the analog inputs. There are also
113  * virtual channels for quantities directly calculated from the analog inputs.
114  * The vout and iout channels are for DC measurements of the VSI outputs when it
115  * is producing a DC output. */
116  typedef struct
117  {
118     Uint16 count_cal, ///< counter for low speed calibration summation
119     count_rms, ///< counter for full fund. period for RMS calculations
120     count_rms_bak, ///< background copy of RMS counter
121     count_dc, ///< counter for DC averaging
122     count_dc_bak, ///< background copy of DC counter
123     flag_cal, ///< flag set to trigger background calibration averaging
124     flag_rms, ///< flag set to trigger background RMS averaging
125     flag_dc; ///< flag set to trigger background DC averaging
126 } type_adc_int;
127 
128  @} 
129 
130  // Internal ADC Variables
131  typedef struct
132  {
133     type_adc_int adc_int =
134     {
135         0, // count_cal
136         0, // count_rms
137         0, // count_rms_bak
138         0, // count_dc
139         0, // count_dc_bak
140     };
141 
142  @} 
143 
```
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```c
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

0, // flag_rms
0, // flag_dc
( 0, // raw
0, // filt
0L, // rms_sum
0L, // rms_sum_bak
0L, // dc_sum
0L, // dc_sum_bak
0.0, // real
}, // #A0
{ 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // #B0
{ 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // yHA
{ 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // yLA
{ 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // yHB
{ 0, 0, 0L, 0L, 0L, 0L, 0.0 }, // yLB
};

// ADC calibration variables
int16 cal_gainA = 1<<14, // calibration gain factor for A channel
cal_gainB = 1<<14, // calibration gain factor for B channel
cal_offsetA = 0, // calibration offset for A channel
cal_offsetB = 0;
double cal_gain_A, cal_gain_B,
cal_offset_A, cal_offset_B;

double yHA = 0.0,
yLA,
yHB,
yLB;

/* =========================================================================
__Variables()
============================================================================ */
// state machine level variables
Uint16 vsi_status = 0, /// Status of VSI system
is_switching = 0, // flag set if PWM switching is active
vsi_counter = 0, // counter for timing VSI regulation events
spi_fail_count;

// PWM Timer interrupt variables
int16 *sin_table = (int16 *)0x003FF000, // pointer to sine table in boot ROM
*cos_table = (int16 *)0x003FF100, // pointer to cos table in boot ROM
mod_targ = 0, // target modulation depth
mod_ref = 0;

/// fault variables
Uint16 detected_faults = 0; // bits set for faults detected (possibly cleared)

/********************
_Modulation_variables()
********************/
int16 phase_scaled_fixed=0,
int_count=0,
phase_shift=0;

/********************
_ADC_VARIABLES()
********************/
//ADC Variables
int32 VdcIN_fixed,
VdcOUT_fixed,
Iload_fixed,
IVSI_fixed;

int32 Vdc1_fixed,

// Boot ROM sine table starts at 0x003FF000 and has 641 entries of 32 bit sine
// values making up one and a quarter periods (plus one entry). For 16 bit
// values, use just the high word of the 32 bit entry. Peak value is 0x40000000 (2^30)
// therefore 1 period is 512 entries, 120 degrees offset is 170.67 entries.
// sin table actually starts with an offset of 2, odd numbers only
// so first value is in sin_table[3]
// max value of 16bit sign table is 2^14 =16384

int16

/****************************
_Vars()
******************************/
int16

/********************
_fault_variables()
********************/
Uint16

// detected_faults = 0; // bits set for faults detected (possibly cleared)

/********************
_VARIABLES()
********************/
VdcIN_fixed,
VdcOUT_fixed,
Iload_fixed,
IVSI_fixed,

/****************************
_Vars()
******************************/
Vdc1_fixed,

/****************************
_VARIABLES()
******************************/
205
```
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 receive 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_prop_fixed,
286 VDC_intnow_fixed,
287 VDC_int_fixed=0,
288 VDC_cont_signal_fixed;
289
290 int16 saturated;
291
292 /*****************
293 _Adaptive_Variables()
294 *****************/
295 double Z_harm[7],
296 phi_z[7];
297
298 int16 phase_shift_avg,
299 phase_shift_record[5],
300 counter_avg,
301 a_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 delta_0_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
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 */
341 _BIDC_FF_Variables()
342 /**************************************************************************/
343
344 double Iload_FF_double;
345
346 int32 Iload_abs;
347
348 int32 BIDC_FF,
349 Iload_FF_fixed[PERIOD_2_BIDC];
350
351 int16 FF_ENABLE=0,
352 AC_FF=0,
353 hi,
354 lo,
355 mid,
356 va_VSI,
357 init_table; //initialises ff table
358
359 /**************************************************************************
360 */
361 __Exported_ADC_Functions()
362 /**************************************************************************/
363
364 type_state
365 vsi_state =
366 {
367 &st_vsi_init,
368 1
369};
370
371 // ADC and VSI interrupt
372 interrupt void isr_adc(void);
373
374 // Gate fault (PDIINT) interrupt
375 interrupt void isr_gate_fault(void);
376
377 /**************************************************************************
378 */
379 __Local_Function_Prototypes()
380 /**************************************************************************/
381
382 /**************************************************************************
383 */
384 This function initialises the ADC and VSI interrupt module. It sets the
385 internal ADC to sample the DA-2810 analog inputs and timer1 to generate a PWM
386 carrier and the event manager A to generate the VSI switching. It also
387 initializes all the relevant variables and sets up the interrupt service
388 routines.
389
390}
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

This function initializes the ADC unit to:

- Trigger a conversion sequence from timer 1 overflow
- Convert the appropriate ADC channels

Result registers as follows:

- ADCRESULT0 = ADCINA0
- ADCRESULT1 = ADCINB0
- ADCRESULT2 = ADCINA1
- ADCRESULT3 = ADCINB1
- ADCRESULT4 = ADCINA2
- ADCRESULT5 = ADCINB2
- ADCRESULT6 = ADCINA3
- ADCRESULT7 = ADCINB3
- ADCRESULT8 = ADCINA4
- ADCRESULT9 = ADCINB4
- ADCRESULT10 = ADCINA5
- ADCRESULT11 = ADCINB6
- ADCRESULT12 = ADCINA6 yHA
- ADCRESULT13 = ADCINB6 yHB
- ADCRESULT14 = ADCINA7 yLA
- ADCRESULT15 = ADCINB7 yLB

It initializes the Event Manager A unit to:

- Drive PWM1-4 as PWM pins not GPIO
- A 0.48ns deadtime between the high and low side pins
- Timer 1 as an up/down counter for the PWM carrier

It initializes the PIE unit to:

- Take PDPINTA as a power stage interrupt
- Use the internal ADC completion interrupt to trigger the main ISR

#define EVB

// Set up ISRs
EALLOW;
PieVectTable.ADCINT = &isr_adc;
PieVectTable.PDPINTA = &isr_gate_fault;
EDIS;

// Set up compare outputs
EALLOW;
OpinMuxRegs.GPMUX.bit.PWM1_GPIOA0 = 1; // enable PWM1 pin
OpinMuxRegs.GPMUX.bit.PWM2_GPIOA1 = 1; // enable PWM2 pin
OpinMuxRegs.GPMUX.bit.PWM3_GPIOA2 = 1; // enable PWM3 pin
OpinMuxRegs.GPMUX.bit.PWM4_GPIOA3 = 1; // enable PWM4 pin
OpinMuxRegs.GPMUX.bit.PWM5_GPIOA4 = 0; // enable GPIOA5
OpinMuxRegs.GPMUX.bit.PWM6_GPIOA5 = 0; // enable GPIOA6

// Set up GPIOA12 to take the sync pulse from the Load GIIB
OpinMuxRegs.GPMUX.bit.TCLKINA_GPIOA12 = 0; //GPIOA12 is an IO

// Set up compare outputs
EALLOW;
OpinMuxRegs.GPMUX.bit.PWM1_GPIOA0 = 1; // enable PWM1 pin
OpinMuxRegs.GPMUX.bit.PWM2_GPIOA1 = 1; // enable PWM2 pin
OpinMuxRegs.GPMUX.bit.PWM3_GPIOA2 = 1; // enable PWM3 pin
OpinMuxRegs.GPMUX.bit.PWM4_GPIOA3 = 1; // enable PWM4 pin
OpinMuxRegs.GPMUX.bit.PWM5_GPIOA4 = 0; // enable GPIOA5
OpinMuxRegs.GPMUX.bit.PWM6_GPIOA5 = 0; // enable GPIOA6

// Set up GPIOD12 to take the sync pulse from the Load GIIB
OpinMuxRegs.GPMUX.bit.TCLKINA_GPIOA12 = 0; //GPIOA12 is an IO

// Set up GPIOD12 to take the sync pulse from the Load GIIB
OpinMuxRegs.GPMUX.bit.TCLKINA_GPIOA12 = 0; //GPIOA12 is an IO

OpinMuxRegs.GPMUX.bit.PWM7_GPIOB0 = 1; // enable PWM7 pin
OpinMuxRegs.GPMUX.bit.PWM8_GPIOB1 = 1; // enable PWM8 pin
OpinMuxRegs.GPMUX.bit.PWM9_GPIOB2 = 1; // enable PWM9 pin
OpinMuxRegs.GPMUX.bit.PWM10_GPIOB3 = 1; // enable PWM10 pin
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

// for carrier synchronisation
GpioMuxRegs.GPBMUX.bit.GPIOB4 = 0; // enable GPIOB4 - carrier synch
GpioMuxRegs.GPBDIR.bit.GPIOB4 = 1; // GPIOB4 is an output

// set up GPIOB5 to send a synch pulse to the output VSI
EDIS;

// DEADBAND CONTROL
// EVA
EvaRegs.DBTCONA.bit.DBT = 8; // 1.5us deadtime
EvaRegs.DBTCONA.bit.EDBT1 = 1;
EvaRegs.DBTCONA.bit.EDBT2 = 1;
EvaRegs.DBTCONA.bit.EDBT3 = 1;
EvaRegs.DBTCONA.bit.DBTPS = 6;

// EVB
spi_set_mode(MODE_CPLD);
cpld_write(ADD_EVB,0x01); // direct EVB to output
spi_set_mode(MODE_DAC);

// EVA
EvaRegs.COMCONA.bit.ACTRLD = 1; // reload ACTR on underflow or period match
EvaRegs.COMCONA.bit.SVENABLE = 0; // disable space vector PWM
EvaRegs.COMCONA.bit.CLD = 1; // reload on underflow & period match
EvaRegs.COMCONA.bit.FCOMPOE = 1; // full compare enable
EvaRegs.COMCONA.bit.CENABLE = 1; // enable compare operation

// EVB
spi_set_mode(MODE_CPLD);
cpld_write(ADD_EVB,0x00);
spi_set_mode(MODE_DAC);

// EVA
EvaRegs.COMCONB.bit.ACTRLD = 1; // reload ACTR on underflow or period match
EvaRegs.COMCONB.bit.SVENABLE = 0; // disable space vector PWM
EvaRegs.COMCONB.bit.CLD = 1; // reload on underflow & period match
EvaRegs.COMCONB.bit.FCOMPOE = 1; // full compare enable
EvaRegs.COMCONB.bit.CENABLE = 1; // enable compare operation

// EVB
spi_set_mode(MODE_CPLD);
cpld_write(ADD_EVB,0x00);
spi_set_mode(MODE_DAC);

// Setup and load COMCON
// EVA
EvaRegs.COMCONA.bit.ACTRLD = 1; // reload ACTR on underflow or period match
EvaRegs.COMCONA.bit.SVENABLE = 0; // disable space vector PWM
EvaRegs.COMCONA.bit.CLDR = 1; // reload on underflow & period match
EvaRegs.COMCONA.bit.FCMPUE = 1; // full compare enable
EvaRegs.COMCONA.bit.CENABLE = 1; // enable compare operation

// EVB
spi_set_mode(MODE_CPLD);
cpld_write(ADD_EVB,0x00);
spi_set_mode(MODE_DAC);

// Setup and load COMCON
// EVB
spi_set_mode(MODE_CPLD);
cpld_write(ADD_EVB,0x00);
spi_set_mode(MODE_DAC);

// Set up Timer 1
EvaRegs.T1CON.all = 0x0000;
EvaRegs.T1PR = PERIOD_BIDC;
EvaRegs.T1CMPR = PERIOD_BIDC-1; // modified for asynchronous sampling;
EvaRegs.T1CNT = 0x0000;

// Set up Timer 3
EvaRegs.T3CON.all = 0x0000;
EvaRegs.T3PR = PERIOD_BIDC;
EvaRegs.T3CMPR = 0; // modified - unnecessary - DS
EvaRegs.T3CNT = 0x0000;
# ifndef
// Setup and load GPTCONA
EvaRegs.GPTCONA.bit.T1TOADC = 3; //0: no event starts ADC 3; Compare match starts ADC 2: period int flag starts ADC
EvaRegs.GPTCONA.bit.TCMPOE = 1;
// Setup ADC
//these are being done in A/B pairs
AdcRegs.ADCMAXCONV.all = 0x0007; // Setup 8 conv's on SEQ1 //To Oversample?
AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0x0; // (A0/B0) - ADCRESULT0 - ADCINA0 - APOT1/I3 - SW_A - default I3 - DC Load Current
AdcRegs.ADCCHSELSEQ1.bit.CONV01 = 0x1; // (A1/B1) - ADCRESULT2 - ADCINA1 - Vdc3/Vac3 - SW_A - default Vac3 - Output DC Voltage
AdcRegs.ADCCHSELSEQ1.bit.CONV02 = 0x2; // (A2/B2) - ADCRESULT4 - ADCINA2 - I1 - Output Current
AdcRegs.ADCCHSELSEQ1.bit.CONV03 = 0x3; // (A3/B3) - ADCRESULT6 - ADCINA3 - Vac1 - Output DC Voltage
AdcRegs.ADCCHSELSEQ2.bit.CONV04 = 0x4; // (A4/B4) - ADCRESULT8 - ADCINA4 - I2 - Output Current
AdcRegs.ADCCHSELSEQ2.bit.CONV05 = 0x5; // (A5/B5) - ADCRESULT10 - ADCINA5 - Vac2 - Output DC Voltage
AdcRegs.ADCCHSELSEQ2.bit.CONV06 = 0x6; // (A6/B6) - ADCRESULT12 - ADCINA6 - 2.5V ref
AdcRegs.ADCCHSELSEQ2.bit.CONV07 = 0x7; // (A7/B7) - ADCRESULT14 - ADCINA7 - 1.25V ref
AdcRegs.ADCTRL1.bit.ACQ_PS = 1; // lengthen acq window size
AdcRegs.ADCTRL1.bit.SEQ_CASC = 1; // cascaded sequencer mode
AdcRegs.ADCTRL2.bit.EVA_SOC_SEQ1 = 1; // EVA manager start
AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = 1; // enable interrupt
AdcRegs.ADCTRL2.bit.INT_MOD_SEQ1 = 0; // int at end of every SEQ1
AdcRegs.ADCTRL3.bit.SMODE_SEL = 1; // simultaneous sampling mode
AdcRegs.ADCTRL3.bit.ADCCLKPS = 0x04; // ADCLK = HSPCLK/8 (9.375MHz)
SET_ADCB_NO(); //activates SW_B. ADCB4 = APOT2, ADCB5 = Vgen
AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1; // clear interrupt flag from ADC
PieCtrlRegs.PIEACK.all = PIEACK_GROUP1; // Acknowledge interrupt to PIE Group 1 : PDPINT, ADC
/* Setup and load T1CON & T3CON to start operation */
EvaRegs.T1CON.bit.TMODE = 1; // continous up/down count mode
EvaRegs.T1CON.bit.TPS = 0; // input clock prescaler
EvbRegs.T3CON.bit.TMODE = 1; // continous up/down count mode
EvbRegs.T3CON.bit.TPS = 0; // input clock prescaler
Z_harm[0] = sqrt(R_L_2 + OMEGA_BIDC_L_2);
Z_harm[1] = sqrt(R_L_2 + 3.0*3.0*OMEGA_BIDC_L_2);
Z_harm[2] = sqrt(R_L_2 + 5.0*5.0*OMEGA_BIDC_L_2);
Z_harm[3] = sqrt(R_L_2 + 7.0*7.0*OMEGA_BIDC_L_2);
Z_harm[4] = sqrt(R_L_2 + 9.0*9.0*OMEGA_BIDC_L_2);
Z_harm[5] = sqrt(R_L_2 + 11.0*11.0*OMEGA_BIDC_L_2);
Z_harm[6] = sqrt(R_L_2 + 13.0*13.0*OMEGA_BIDC_L_2);
phi_z[0] = atan2(OMEGA_BIDC_L,R_L);
phi_z[1] = atan2(OMEGA_BIDC_L*3.0,R_L);
phi_z[2] = atan2(OMEGA_BIDC_L*5.0,R_L);
phi_z[3] = atan2(OMEGA_BIDC_L*7.0,R_L);
phi_z[4] = atan2(OMEGA_BIDC_L*9.0,R_L);
phi_z[5] = atan2(OMEGA_BIDC_L*11.0,R_L);
phi_z[6] = atan2(OMEGA_BIDC_L*13.0,R_L);
210

APPENDIX A. SIMULATION & EXPERIMENTAL CODE
APPENDIX A. SIMULATION \\& EXPERIMENTAL CODE

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 phi_z_fixed[0] = (int16)(phi_z[0]*RAD_TO_COUNT);
647 phi_z_fixed[1] = (int16)(phi_z[1]*RAD_TO_COUNT);
648 phi_z_fixed[2] = (int16)(phi_z[2]*RAD_TO_COUNT);
649 phi_z_fixed[3] = (int16)(phi_z[3]*RAD_TO_COUNT);
650 phi_z_fixed[4] = (int16)(phi_z[4]*RAD_TO_COUNT);
651 phi_z_fixed[5] = (int16)(phi_z[5]*RAD_TO_COUNT);
652 phi_z_fixed[6] = (int16)(phi_z[6]*RAD_TO_COUNT);
653 inv_Z_harm_fixed[0]= (int32)(32768.0/(1.0*Z_harm[0]));
654 inv_Z_harm_fixed[1]= (int32)(32768.0/(3.0*Z_harm[1]));
655 inv_Z_harm_fixed[2]= (int32)(32768.0/(5.0*Z_harm[2]));
656 inv_Z_harm_fixed[3]= (int32)(32768.0/(7.0*Z_harm[3]));
657 inv_Z_harm_fixed[4]= (int32)(32768.0/(9.0*Z_harm[4]));
658 inv_Z_harm_fixed[5]= (int32)(32768.0/(11.0*Z_harm[5]));
659 inv_Z_harm_fixed[6]= (int32)(32768.0/(13.0*Z_harm[6]));
660 //scaled by 32768 = 2^15
661 //Feed forward initialisations
662 //Generate a lookup table of the steady state load current based on operating phase shift.
663 //I_load_FF = 16/pi^2 *Vp * Np/Ns * sum(1/(2n+1)^3 * sin((2n+1)delta)/(omega*L)
664 //done in floating point, converted to fixed point at the last step
665 for (init_table=0;init_table<=PERIOD_2_BIDC;init_table++)
666 {
667 Iload_FF_double=0.0;
668 for (n_harm=0;n_harm<6;n_harm++)
669 {
670 Iload_FF_double += (1.0/harm_3[n_harm])*sin(harm[n_harm]*(init_table*COUNT_TO_RAD));
671 }
672 Iload_FF_fixed[init_table] = (int32)(BIDC_FF_CONST*Iload_FF_double*FIXED_Q_SCALE);
673 }
674 #endif
675 DINT;
676 EvaRegs.T1CON.bit.TENABLE = 1; // enable timer1
677 EvbRegs.T3CON.bit.TENABLE = 1; // enable timer3
678 #ifndef EVB
679 EvbRegs.T3CON.bit.TENABLE = 0;
680 #endif
681 EINT;
682 // Initialise state machine
683 vsi_state.first = 1;
684 vsi_state.f = &st_vsi_init;
685 #endif
686 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
687 /**
688 This function is called from the main background loop once every millisecond.
689 It performs all low speed tasks associated with running the core interrupt
690 process, including:
691 - checking for faults
692 - calling the VSI state functions
693 - calling internal analog scaling functions
694 */
695 void vsi_state_machine(void)
696 {
697 SS_DO(vsi_state);
698 vsi_state.first = 1;
699 vsi_state.f = &st_vsi_init;
700 } /* end vsi_init */
701 /* =========================================================================
702 __Exported_VSI_Functions()
703 ========================================================================= */
704 
705 This function is called from the main background loop once every millisecond.
706 It performs all low speed tasks associated with running the core interrupt
707 process, including:
708 - checking for faults
709 - calling the VSI state functions
710 - calling internal analog scaling functions
711 /* authored A.McIver
712 History:
713 11 13/10/07 AM - derived from 25kVA:vsi:vsi.c
714 */
715 void vsi_state_machine(void)
716 {
717 if (adc_int.flag_cal != 0)
718 {
719 } /* end vsi_state_machine */
720 /* =========================================================================
721 __Exported_VSI_Functions()
722 ========================================================================= */
723 

APPENDIX A. SIMULATION & EXPERIMENTAL CODE

721 /* ***********************************************************************/
722 /**
723 * This function switches the VSI from the stopped state to a running state.
724 */
725 void vsi_enable(void)
726 {
727     if (detected_faults == 0)
728         is_switching = 1;
729 }
730 /* end vsi_enable */
731
732 /* ***********************************************************************/
733 /**
734 * This function switches the VSI from the running state to a stop state.
735 * The ramp down process has the side effect of resetting the reference to zero.
736 */
737 void vsi_disable(void)
738 {
739     is_switching = 0;
740 }
741 /* end vsi_disable */
742
743 /* ***********************************************************************/
744 /**
745 * This function sets the target output phase shift.
746 */
747 void vsi_set_phase(double phase_cont_signal)
748 {
749     phase_scaled_fixed = phase_cont_signal*DEG_TO_COUNT; //scaled to +/- pi/2 (radians)
750     if (phase_scaled_fixed>MAX_PHASE)
751         phase_scaled_fixed=MAX_PHASE-1;
752         phase_cont_signal = 90.0;
753     else if (phase_scaled_fixed<-MAX_PHASE)
754         phase_scaled_fixed =1-MAX_PHASE;
755         phase_cont_signal = -90.0;
756 }
757 /* end vsi_set_phase */
758
759 /* ***********************************************************************/
760 /**
761 * This function sets the desired reference Voltage.
762 */
763 void vsi_set_vref(int16 vref)
764 {
765     GrabClear();
766     GrabStart();
767     set_vref=dip;  
768     ref_volt=vref;
769     VDCref_fixed = ((long)vref<<FIXED_Q);
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 This function returns the fault word of the VSI module.
827 */
828 \author{A.McIver}
829 \par History:
830 \li 04/03/08 AM - initial creation
831
832 \returns{The present fault word}
833} */
834 \fn{Uint16 vsi_get_faults(void)}
835 \par Parameters: none
836 \par Returns: nothing
837 \par Description: Clear the detected faults.
838 \par Notes:
839 \par History:
840 \li 13/10/05 AM - initial creation
841 \li 28/04/08 AM - added event reporting
842 */
843 \fn{void vsi_clear_faults(void)}
844 \par Parameters: none
845 \par Returns: nothing
846 \par Description: Clear the detected faults.
847 \par Notes:
848 \par History:
849 \li 13/10/05 AM - initial creation
850 \li 28/04/08 AM - added event reporting
851 */
852 \fn{void vsi_clear_faults(void)}
853 {
854 \fn{Uint16}
855 \fn{i};
856
857 if (detected_faults & FAULT_VSI_PDPINT)
858 {
859 for (i=0; i<100; i++)
860 \{ // delay for fault to clear
861 EvaRegs.COMCONA.all = 0;
862 EvaRegs.COMCONA.all = 0xAA00;
863 \}
864 \}
865 \}
866} /* end vsi_clear_faults */
867
868} /* Interrupt Routines */
869
870} /* Interrupt Routines */
871
872\fn{void isr_time(void)}
873 \fn{brief Updates VSI and performs closed loop control}
874
875 \fn{This interrupt is triggered by the ADC interrupts.}
876 \fn{It then:
877 - takes the adc measurements (synch sample, throws away every alternate one)
878 - determines the gains for the adaptive controller
879 - performs closed loop control calculations}
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

881 - updates phase angle & calculates switching times  
882 883 \author A.McIver  
884 \par History:  
885 12/10/07 AM - initial creation  
886 */  
887 #ifndef BUILD_RAM  
888 #ifdef 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 followed by the adaptive 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 GIIIB 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.ADCRESULT12-(ADC_OFFSET<<4));  
928 I4_cal = I4_cal+(AdcRegs.ADCRESULT14-(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.ADCRESULT2>>44);  
948 adc_int.yLa_dc_sum = (Uint32)(AdcRegs.ADCRESULT4>>44);  
949 adc_int.yHb_dc_sum = (Uint32)(AdcRegs.ADCRESULT10>>44);  
950 adc_int.yLb_dc_sum = (Uint32)(AdcRegs.ADCRESULT12>>44);  
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 */
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```c
adc_int.yLA.dc_sum = 0;
adc_int.yLB.dc_sum = 0;
adc_int.yLB.dc_sum = 0;
adc_int.flag_cal = 1;
}
}
puts_COM1("CALIBRATION COMPLETE\n\n");
}
SET_TP11(); //timing bit
if (VDCref_fixed==prev_VDCref_fixed)
{ 
CLEAR_TP10();
}
else SET_TP10();
prev_VDCref_fixed = VDCref_fixed;
}  
************
__ADC_CL() *
************
//The bidirectional converter needs 2 analog inputs, ie DC bus voltage and load current.
//for feed-forward compensation, the load current needs to be scaled by the modulation depth.
//the modulation depth va is being passed using the DAC. DAC = va>>2.
//this is fed into the VGEN input as a 3rd ADC.
Vdc1_fixed = (((AdcRegs.ADCRESULT7-Vdc1_cal)>>4) -ADC_OFFSET)*VDC_ANALOG_GAIN;
Vdc2_fixed = (((AdcRegs.ADCRESULT1-Vdc2_cal)>>4) -ADC_OFFSET)*VDC_ANALOG_GAIN;
Vac1_fixed = (((AdcRegs.ADCRESULT6-Vac1_cal)>>4) -ADC_OFFSET)*VAC_ANALOG_GAIN;
Vac2_fixed = (((AdcRegs.ADCRESULT10-Vac2_cal)>>4)-ADC_OFFSET)*VAC_ANALOG_GAIN;
Vac3_fixed = (((AdcRegs.ADCRESULT2-Vac3_cal)>>4) -ADC_OFFSET)*VAC_ANALOG_GAIN;
I1_fixed = (((AdcRegs.ADCRESULT4-I1_cal)>>4) -ADC_OFFSET)*I_ANALOG_GAIN;
I2_fixed = (((AdcRegs.ADCRESULT8-I2_cal)>>4) -ADC_OFFSET)*I_ANALOG_GAIN;
I3_fixed = (((AdcRegs.ADCRESULT0-I3_cal)>>4) -ADC_OFFSET)*I_ANALOG_GAIN;
I4_fixed = (((AdcRegs.ADCRESULT5-I4_cal)>>4) -ADC_OFFSET)*I_ANALOG_GAIN;
Vgen_fixed = (((AdcRegs.ADCRESULT11-VGEN_CAL)>>4)-ADC_OFFSET)*VGEN_ANALOG_GAIN;
VdcIN_fixed = Vdc1_fixed;
va_VSI = Vgen_fixed;
VdcOUT_fixed = (Vdc2_fixed+Vac1_fixed+Vac2_fixed+Vac3_fixed)>>2;
if (vsi_synch==4)
{
IVSI_fixed = (int32)((((I1_fixed+I2_fixed)>>1)*(int32)(va_VSI))/(int32)PERIOD_2_VSI); // scaled by mod depth
}
Iload_fixed = IVSI_fixed + ((I3_fixed+I4_fixed)>>1); // AC+DC components
while(counter_avrg<=4)
{
phase_shift_avrg += phase_shift_record[counter_avrg]>>2;
VdcOUT_fixed_avrg += VdcOUT_fixed_record[counter_avrg]>>2;
counter_avrg++;
}
if (delta0_aug_fixed>MAX_PHASE) delta0_aug_fixed=MAX_PHASE; //IS IN counts
//then determine the B value
delf_delu_fixed = 0;
n_harm=0;
for (n_harm = 0;n_harm<6;n_harm++)
{

}
if (DT_COMP)
delta0_aug_fixed=abs(phase_shift-phase_aug_DT_fixed);
else
delta0_aug_fixed=abs(phase_shift);
if (delta0_aug_fixed>MAG_PHASE) delta0_aug_fixed=MAG_PHASE; //15 IN counts
for (n_harm = 0;n_harm<6;n_harm++)

total
215
```
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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*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 BIDC_FF=0;
1064 Iload_abs=abs(Iload_fixed);
1065 //Iload Feedforward - search algorithm
1066 lo=0;
1067 hi=PERIOD_2_BIDC-1;
1068 while (hi>lo)
1069 {
1070   mid = ((hi-lo)/2)+lo;
1071   if (Iload_abs<Iload_FF_fixed[mid]) hi=mid-1; //in the bottom half
1072   else if (Iload_abs>Iload_FF_fixed[mid]) lo=mid+1;
1073   else if (Iload_abs==Iload_FF_fixed[mid])
1074   {
1075     lo=mid;
1076     break;
1077   }
1078   else if ((hi-lo)<10) break;
1079 }
1080
1081 if (saturated==1) BIDC_FF=0;
1082 else
1083 {
1084   if (Iload_fixed>0) BIDC_FF = lo;
1085   else BIDC_FF = -lo;
1086 }
1087
1088 /*************************
1089 _BIDC_DT_Compensation()
1090 *************************/
1091 phase_rad_ratio_fixed = ((int32)(abs((int32)phase_shift))<<FIXED_Q)/(PERIOD_BIDC<<1);
1092 VDCout_txd_scaled_fixed = (VdcOUT_fixed*NPRI_NSEC_FIXED)>>FIXED_Q;
1093 Vp_Vs_4Vp_fixed = ((VIN_FIXED-VDCout_txd_scaled_fixed)<<(FIXED_Q-2))/VIN_FIXED;
1094 Vs_Vp_4Vs_fixed = ((VDCout_txd_scaled_fixed-VIN_FIXED)<<(FIXED_Q-2))/VIN_FIXED;
1095 Vs_Vp_DB_fixed = ((VDCout_txd_scaled_fixed/(PERIOD_BIDC<<1))*DEADBAND_COUNT_BIDC)/(int32)VIN;
1096 Vp_Vs_DB_fixed = (((VIN_FIXED/(PERIOD_BIDC<<1))*DEADBAND_COUNT_BIDC)<<FIXED_Q)/VDCout_txd_scaled_fixed;
1097
1098 // First, calculate slew time
1099 if (VIN_FIXED>VDCout_txd_scaled_fixed) //Vp>Vs
1100 {
1101   if (phase_shift_avg<0) //leading
1102   {
1103     Telw_count = (int16)((phase_rad_ratio_fixed - Vp_Vs_4Vp_fixed - Vs_Vp_DB_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
1104   }
1105   //then calculate phase augmentation
1106   if ((VIN_FIXED-VDCout_txd_scaled_fixed)>20<<FIXED_Q)
1107   {
1108     phase_aug_DT_fixed = 0;
1109     else if (Telw_count<20) phase_aug_DT_fixed = DEADBAND_COUNT_BIDC;
1110     else phase_aug_DT_fixed = DEADBAND_COUNT_BIDC-Telw_count; //in counts
1111   }
1112   else
1113   {
1114     phase_aug_DT_fixed = 0;
1115   }
1116   }
1117   else
1118   {
1119     phase_aug_DT_fixed = 0;
1120   }
1121 }
1122 216
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

1121 }
1122 }
1123 } //lagging
1124 }
1125 Telew_count = (int16)((Vp_Vs_4Vp_fixed - phase_rad_ratio_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
1126 if ((VIN_FIXED - VDCout_txscaled_fixed)>(20<<FIXED_Q))
1127 {
1128 if (Telew_count<0) phase_aug_DT_fixed = TElew_count; //in counts
1129 else if (Telew_count>DEADBAND_COUNT_BIDC) phase_aug_DT_fixed = TElew_count; //in counts
1130 else
1131
1132 else //Vp<Vs
1133 {
1134 if (phase_shift_avrg<0) //leading
1135 {
1136 Tslew_count = (int16)((Vs_Vp_4Vs_fixed - phase_rad_ratio_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
1137 if ((VDCout_txscaled_fixed-VIN_FIXED)>(20<<FIXED_Q))
1138 {
1139 if (Tslew_count>DEADBAND_COUNT_BIDC) phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
1140 else if (Tslew_count<0) phase_aug_DT_fixed = 0;
1141 else phase_aug_DT_fixed = -Tslew_count; //in counts
1142 }
1143 else //lagging
1144 {
1145 Telew_count = (int16)((phase_rad_ratio_fixed - Vs_Vp_4Vs_fixed - Vp_Vs_DB_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
1146 if ((VDCout_txscaled_fixed-VIN_FIXED)>(20<<FIXED_Q))
1147 {
1148 if (Telew_count<0) phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
1149 else if (Telew_count>DEADBAND_COUNT_BIDC) phase_aug_DT_fixed = TElew_count; //in counts
1150 }
1151 else
1152 else
1153
1154 if (Telew_count>DEADBAND_COUNT_BIDC) phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
1155 else
1156 }
1157 } //Vp>Vs
1158 } //lagging
1159 }
1160 }
1161 Telew_count = (int16)((phase_rad_ratio_fixed - Vs_Vp_4Vs_fixed - Vp_Vs_DB_fixed)*(PERIOD_BIDC<<1)>>FIXED_Q);
1162 if ((VDCout_txscaled_fixed-VIN_FIXED)>(20<<FIXED_Q))
1163 {
1164 if (Telew_count<0) phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
1165 else if (Telew_count>DEADBAND_COUNT_BIDC) phase_aug_DT_fixed = TElew_count; //in counts
1166 }
1167 else
1168 else
1169 {
1170 if (Telew_count>DEADBAND_COUNT_BIDC) phase_aug_DT_fixed = -DEADBAND_COUNT_BIDC;
1171 else
1172 }
1173 }
1174 }
1175 }
1176 } //BIDC_PI_Control_Loop()
1177 } //BIDC_PI_Control_Loop()
1178 } //BIDC_PI_Control_Loop()
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 VDCprop_fixed = (VDCerror_fixed*VDC_KP_fixed)>>FIXED_Q;
1187 VDC_intnow_fixed = (VDCprop_fixed*VDC_KI_FIXED)>>FIXED_Q;
1188 VDC_int_fixed += VDC_intnow_fixed;
1189 VDC_cont_signal_fixed = VDC_prop_fixed + VDC_int_fixed;
1190 }
1191 } //BIDC_SET_PHASE()
if (DT_COMP) phase_shift -= phase_aug_DT_fixed;
#endif

if (abs(phase_shift)>=MAX_PHASE)
{
  SET_TP13(); // desat bit
  saturated=1;
  VDC_int_fixed -= VDC_intnow_fixed;
  if (phase_shift>0)
  {
    phase_shift = MAX_PHASE;
  }
  if (phase_shift<0)
  {
    phase_shift = -MAX_PHASE;
  }
  saturated=1;
  CLEAR_TP13();
}
else
{
  saturated=0;
  CLEAR_TP13();
}
  
  //end control loop
  
  if(EvaRegs.GPTCONA.bit.T1STAT==1) //so last int was an underflow
  {
    //synch pulse for VSI. sent out at 5kHz. this code is seen at 20kHz, so count to 4.
    if (vsi_synch >=4)
    {
      GpioDataRegs.GPBSET.bit.GPIOB4 = 1; //Sets synch output high
      SET_TP12();
      vsi_synch=0;
    }
    vsi_synch++;
    
    /****************************
    * Update switching times *
    ****************************/
    EvaRegs.T1CMPR = PERIOD_BIDC-1; //set the next interrupt to be at the top
    
    /* The Bidirectional DC-DC Converter is comprised of 2 single phase bridges. */
    Primary Bridge is controlled by EVA
    Secondary Bridge is controlled by EVB
    
    //phases C&D
    EvbRegs.CMPR4 = PERIOD_2_BIDC-phase_shift;
    EvbRegs.CMPR5 = PERIOD_2_BIDC-phase_shift;
  }
  else //if heading down, last interrupt was PERIOD MATCH
  {
    
    QpioDataRegs.GPCLEAR.bit.GPIOB4 = 1; //Sets synch output low
    CLEAR_TP12();
    
    /****************************
    * Update switching times *
    ****************************/
    EvaRegs.T1CMPR = 1; //set the next interrupt to be at the bottom
  }
  
  /************************************************************
  *isr_GrabCodeCL()*/
  
#ifdef GRAB_INCLUDE
  
#endif
  
if (GrabRunning())
  {
    GrabStore(0,(I1_fixed+I2_fixed)>>1);
    GrabStore(1,va_VSI);
    GrabStore(2,IVSI_fixed);
    GrabStore(3,(I3_fixed+I4_fixed)>>1);
    
    /*****************************/
  
    #include "GRAB_INCLUDE"
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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.ADCCST.bit.INT_SEQ1_CLR = 1; // clear interrupt flag
1294  PieCtrlRegs.PIEACK.all = PIEACK_GROUP1; // Acknowledge interrupt to PIE Group 2
1295  CLEAR_TP1(); // timing bit
1296  } /* end isr_timer_CL */
1297
1298  /* ********************************************************** */
1299  /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1300  /*******************************************/
1301  /* This function initialises the VSI system. It resets the target modulation */
1302  /* depth to zero. */
1303  /* It is followed by the stop state. */
1304  /*******************************************/
1305  /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1306  void st_vsi_init(void)
1307  {
1308      mod_ref = 0;
1309      mod_targ = 0;
1310      EvaRegs.ACTRA.all = 0x0000;
1311      VSI_DISABLE();
1312      vsi_status = VSI_INIT;
1313      SS_NEXT(vsi_state, st_vsi_stop);
1314  } /* end st_vsi_init */
1315
1316  /* ********************************************************** */
1317  /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1318  /*******************************************/
1319  /* This is the state where the VSI is stopped. There is no switching. It waits */
1320  /* for a start trigger. */
1321  /*******************************************/
1322  void st_vsi_stop(void)
1323  {
1324      if (SS_IS_FIRST(vsi_state))
1325          SS_DONE(vsi_state);
1326  }
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```c
VSI_DISABLE();
mod_targ = 0;
vsi_status = VSI_STOP;
}
if (detected_faults != 0)
{
    SS_NEXT(vsi_state, st_vsi_fault);
    return;
}
if (is_switching != 0) // start trigger
{
    SS_NEXT(vsi_state, st_vsi_gate_charge);
}
/* end st_vsi_stop */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/**
In this state the VSI gates are enabled and the low side gates held on to
charge the high side gate drivers. The next state is either the ramp state.

\author A.McIver
\par History:
\li 12/10/07 AM - initial creation
\li 28/04/08 AM - added event reporting
*/
void st_vsi_gate_charge(void)
{
if (SS_IS_FIRST(vsi_state))
{
    VSI_GATE_CHARGE();
    vsi_status |= VSI_RUNNING;
}
if (detected_faults != 0)
{
    SS_NEXT(vsi_state, st_vsi_fault);
    return;
}
// check for stop signal
if (is_switching == 0)
{
    SS_NEXT(vsi_state, st_vsi_stop);
    return;
}
vsi_counter++;
if (vsi_counter > 200)
{
    SS_NEXT(vsi_state, st_vsi_ramp);
}
/* end st_vsi_gate_charge */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/**
This state ramps up the target modulation depth to match the reference set by
the background. It only changes the target every 100ms and synchronises the
change with a zero crossing to avoid step changes in the output.

\author A.McIver
\par History:
\li 12/10/07 AM - initial creation
\li 28/04/08 AM - added event reporting
*/
void st_vsi_ramp(void)
{
if (SS_IS_FIRST(vsi_state))
{
    VSI_ENABLE();
    vsi_status = VSI_RAMP;
}
if (detected_faults != 0)
{
    SS_NEXT(vsi_state, st_vsi_fault);
}
/* end st_vsi_gate_charge */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

220
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```c
// check for stop signal
if (is_switching == 0)
{
    SS_NEXT(vsi_state, st_vsi_stop);
    return;
}

// check for target reached
if (mod_targ == mod_ref)
{
    SS_NEXT(vsi_state, st_vsi_run);
    return;
}

// ramp reference towards target
if (mod_ref > mod_targ + 5)
{
    mod_targ += 5;
}
else if (mod_ref < mod_targ - 5)
{
    mod_targ -= 5;
}
else
{
    mod_targ = mod_ref;
}

/* end st_vsi_ramp */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/**
 * 
 * This state has the VSI running with the target voltage constant. The output is now ready for measurements to begin. If the reference is changed then the 1473 operation moves back to the ramp state.
 */

void st_vsi_run(void)
{
    if (SS_IS_FIRST(vsi_state))
    {
        SS_DONE(vsi_state);
        vsi_status = VSI_RUNNING;
    }

    if (detected_faults != 0)
    {
        SS_NEXT(vsi_state, st_vsi_fault);
        return;
    }

    // check for stop signal
    if (is_switching == 0)
    {
        SS_NEXT(vsi_state, st_vsi_stop);
    }

    // check for changes in reference
    if (mod_targ != mod_ref)
    {
        vsi_status &= ~VSI_SETTLED;
        SS_NEXT(vsi_state, st_vsi_ramp);
    }

    /* end st_vsi_run */

    /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

    /* void st_vsi_fault(void)
    Parameters: none
    Returns: nothing
    Description: Delays for a while after faults are cleared.
    Notes:
    History:
    03/11/05 AM - initial creation
    04/03/08 AM - set vsi_status with fault bit
    28/04/08 AM - added event reporting
    */

    void st_vsi_fault(void)
    {
        if (SS_IS_FIRST(vsi_state))
        {
            SS_DONE(vsi_state);
        }
```
VSI_DISABLE();
vs_status = VSI_FAULT;
puts(COM1("VSI faults\n");

if (detected_faults == 0)
  vsi_counter++;
else
  vsi_counter = 0;

if (vsi_counter > 100)
  SS_NEXT(vsi_state,st_vsi_stop);

/* =========================================================================
 * Local_Functions()
============================================================================ */

void scale_adc_rms(void)
{
  double
  temp;

  // calculate A0 RMS quantity
  temp = (double)adc_int.A0.dc_sum_bak/(double)adc_int.count_rms_bak;
  val = (double)adc_int.A0.rms_sum_bak*(double)(1<<ADC_RMS_PS)
       / (double)adc_int.count_rms_bak - temp*temp;
  if (val < 0.0) val = 0.0;
  adc_int.A0.real = ADC_REAL_SC * sqrt(val);
}

void scale_adc_dc(void)
{
  double
    val;

  adc_int.A0.real = (double)adc_int.A0.dc_sum_bak/(double)ADC_COUNT_DC;
  adc_int.A2.real = (double)adc_int.A2.dc_sum_bak/(double)ADC_COUNT_DC;
  adc_int.A4.real = (double)adc_int.A4.dc_sum_bak/(double)ADC_COUNT_DC;
  adc_int.A6.real = (double)adc_int.A6.dc_sum_bak/(double)ADC_COUNT_DC;

  // calculate B0 DC quantity
  val = (double)adc_int.A0.dc_sum_bak/(double)ADC_COUNT_DC;
  adc_int.B0.real = ADC_REAL_SC * val;
}

/*============================================================================*/

// Calibrates the adc for gain and offset using the reference inputs.

See spra989a.pdf for calibration details
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```c
/*
 * History:
 * 07/10/05 AM - initial creation
 */

void calibrate_adc(void)
{
    // char
    // str[60];

    yHA = (double)adc_int.yHA.dc_sum_bak/(double)ADC_COUNT_CAL;
    yLA = (double)adc_int.yLA.dc_sum_bak/(double)ADC_COUNT_CAL;
    yHB = (double)adc_int.yHB.dc_sum_bak/(double)ADC_COUNT_CAL;
    yLB = (double)adc_int.yLB.dc_sum_bak/(double)ADC_COUNT_CAL;

    cal_gain_A = (xH - xL)/(yHA - yLA);
    cal_offset_A = yLA * cal_gain_A - xL;

    cal_gain_B = (xH - xL)/(yHB - yLB);
    cal_offset_B = yLB * cal_gain_B - xL;

    // sanity check on gains
    if ( ( (cal_gain_A > 0.94) && (cal_gain_A < 1.05) )
        && ( (cal_gain_B > 0.94) && (cal_gain_B < 1.05) )
        && ( (cal_offset_A > -80.0) && (cal_offset_A < 80.0) )
        && ( (cal_offset_B > -80.0) && (cal_offset_B < 80.0) ) )
    {
        cal_gainA = (int16)(cal_gain_A*(double)(1<<14));
        cal_gainB = (int16)(cal_gain_B*(double)(1<<14));
        cal_offsetA = (int16)cal_offset_A;
        cal_offsetB = (int16)cal_offset_B;
    }

    // sprintf(str,"cal:gA=%.3f,oA=%5.1f, gB=%.3f,oB=%5.1f
",cal_gain_A,
    // cal_offset_A,cal_gain_B,cal_offset_B);
    puts_COM1(str);
    // end calibrate_adc */
}
```
A.2.3 DSP Code – Voltage Source Inverter

```c
/* =========================================================================
__Definitions()
============================================================================ */
#define __SQRT2 1.4142135624
#define __SQRT3 1.7320508075
#define __PI 3.1415926535
#define __PI_2 __PI/2.0
#define __INVPI 1/__PI
#define __INVPI_2 1/__PI_2
#define SYSCLK_OUT (150e6)
#define HSPCLK (SYSCLK_OUT)
#define LSPCLK (SYSCLK_OUT/4)

/* =========================================================================
__State_Simple_Definitions()
============================================================================ */
/** Simple State Machine Type */
typedef void (* funcPtr)(void);
typedef struct
{
    funcPtr f;
    unsigned int call_count;
    unsigned char first;
} type_state;

/* Simple State Handling Macros */
#define SS_NEXT(_s_,_f_) { _s_.f = (funcPtr)_f_; 
                          _s_.call_count = 0; 
                          _s_.first = 1; }
#define SS_IS_FIRST(_s_) (_s_.first == 1)
#define SS_DONE(_s_) { _s_.first = 0; }
#define SS_DO(_s_) { _s_.call_count++; 
                    ((*(_s_.f))()); }
#define SS_IS_PRESENT(_s_,_f_) (_s_.f == (funcPtr)_f_)

/* =========================================================================
__Grab_Code_Definitions()
============================================================================ */
#define GRAB_INCLUDE
#ifdef GRAB_INCLUDE
// grab array size
#define GRAB_LENGTH 200
#define GRAB_WIDTH 3

// modes
#define GRAB_GO 0
#define GRAB_WAIT 1
#define GRAB_TRIGGER 2
#define GRAB_STOPPED 3
#define GRAB_SHOW 4

#define GrabStart() grab_mode = GRAB_TRIGGER;
#define GrabStop() grab_mode = GRAB_STOPPED;
#define GrabRun() grab_mode = GRAB_GO;
#define GrabShow() grab_mode = GRAB_SHOW;
```

APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```c
#define GrabClear() { grab_mode = GRAB_WAIT; \
    grab_index = 0; }

#define GrabTriggered() (grab_mode == GRAB_TRIGGER)
#define GrabRunning() (grab_mode == GRAB_GO)
#define GrabStopped() (grab_mode == GRAB_STOPPED)
#define GrabAvail() (grab_mode >= GRAB_STOPPED)
#define GrabShowTrigger() (grab_mode == GRAB_SHD)

#define GrabStore(_loc_,_data_) grab_array[grab_index][_loc_] = _data_; 
#define GrabStep() { grab_index++; \
    if (grab_index >= GRAB_LENGTH) \
        grab_mode = GRAB_STOPPED; }

// variables
extern int16 step, grab_mode, grab_index, set_vref;
extern long volt_req, wo;
extern double grab_array[GRAB_LENGTH][GRAB_WIDTH];

// functions
void GrabDisplay(int16 index);
void GrabInit(void);

#endif
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

APPENDIX A. SIMULATION & EXPERIMENTAL CODE

1 /**
2 \file
3 \brief System software for the DA-2810 Demo code
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 \ 02/11/10 DS - Load Step for Bidirectional DC-DC Converter
14 \ 16/03/11 DS - Grid Connected H-Bridge, with DC links supplied by a
15 \ Bidirectional DC-DC Converter
16 */
17
18 // compiler standard include files
19 #include <stdlib.h>
20 #include <stdio.h>
21 #include <math.h>
22
23 // processor standard include files
24 #include <DSP281x_Device.h>
25 #include <DSP281x_Examples.h>
26
27 #ifdef COM0_CONSOLE
28 #include <bios0.h>
29 #endif
30
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 _Hash_Definitions()
50 ============================================================================= */
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 /* =========================================================================
69 __Typedefs()
70 ============================================================================= */
71
72 /// Time related flag type
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;
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

#include

#define BUILD_RAM
 ifndef BUILD_RAM
 // These are defined by the linker (see F2812.cmd)
 extern Uint16 RamfuncsLoadStart;
 extern Uint16 RamfuncsLoadEnd;
 extern Uint16 RamfuncsRunStart;
 #endif

// state determination
extern int16 Vdc_fixed;
extern int16 load_enable;
extern double mag_serial;

// Background variables
Uint16 quit = 0; ///< exit flag

// timing variable
type_time_flag time =
{
    0, 0, 0, 0 // flags
};

Uint32 idle_count = 0, ///< count of idle time in the background
idle_count_old = 0, ///< previous count of idle time
idle_diff = 0; ///< change in idle time between low speed tasks

char str[40]; // string for displays

if (initial) { // to display correctly
    str[40] = 0;
}

void com_display(void);
void display_help(void);
void com_keyboard(void);

/* =========================================================================
__Grab_Variables()
============================================================================ */
#ifdef GRAB_INCLUDE
#pragma DATA_SECTION(grab_array, "bss_grab")
int16 step, grab_mode = GRAB_STOPPED,
grab_index,
set_vref = 0;
long volt_req = 10,
wo = 314;
double grab_array[GRAB_LENGTH][GRAB_WIDTH];
#endif

/* =========================================================================
__Serial_input_variables()
============================================================================ */
//VSI Modulation Depth Variation
double mod_serial = 0.0;

/* =========================================================================
__Variables()
============================================================================ */
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

161 //VSI Reference Current Variation
162 double Imag_serial=0.0;
163 //VSI fundamental frequency variation
164 double ffund_serial=50.05;
165 //BIDC Phase Shift Variation
166 double phase_serial=0.0;
167 //external debug variables. so they can be displayed
168
169 /* ========================================================================= */
170 /* Main */
171 /* ========================================================================= */
172 void main(void)
173 {
174 static int
175 i = 0;
176 // initial=0;
177
178 // Disable CPU interrupts
179 DINT;
180 // Initialise DSP for PCB
181 lib_mini2810_init(150/*MHz*/,37500/*kHz*/,150000/*kHz*/,LIB_EVAENCLK
182 |LIB_EVBENCLK|LIB_ADCENCLK|LIB_SCIAENCLK|LIB_SCIBENCLK|LIB_MCBSPENCLK);
183
184 InitGpio();
185 spi_init(MODE_CPLD);
186 // SpiaRegs.SPICCR.bit.SPILBK = 1; //Set SPI on loop back for testing
187 cpld_reg_init();
188 giib_init();
189
190 // Initialize the PIE control registers to their default state.
191 InitPieCtrl();
192 // Disable CPU interrupts and clear all CPU interrupt flags:
193 IER = 0x0000;
194 IFR = 0x0000;
195 // Initialize the PIE vector table with pointers to the shell Interrupt
196 // Service Routines (ISR).
197 // This will populate the entire table, even if the interrupt
198 // is not used in this example. This is useful for debug purposes.
199 // The shell ISR routines are found in DSP281x_DefaultIsr.c.
200 // This function is found in DSP281x_PieVect.c.
201 InitPieVectTable();
202
203 #ifndef BUILD_RAM
204 // Copy time critical code and Flash setup code to RAM
205 // The RamfuncsLoadStart, RamfuncsLoadEnd, and RamfuncsRunStart
206 // symbols are created by the linker. Refer to the F2810.cmd file.
207 MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);
208 // Call Flash Initialization to setup flash waitstates
209 // This function must reside in RAM
210 InitFlash();
211 #endif
212
213 // Initialize CODE port
214 #ifdef BUILD_RAM
215 // Copy time critical code and Flash setup code to RAM
216 // The RamfuncsLoadStart, RamfuncsLoadEnd, and RamfuncsRunStart
217 // symbols are created by the linker. Refer to the F2810.cmd file.
218 MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);
219 // Call Flash Initialization to setup flash waitstates
220 // This function must reside in RAM
221 InitFlash();
222 #endif
223 // Initialize 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;
IER |= M_INT1; // Enable CPU Interrupt 1
vsi_init();
EnableInterrupts();
//waste some time, so that the program can finish writing to the screen

#ifdef GRAB_INCLUDE
GrabInit();
#endif
spi_init(MODE_DAC);
spi_set_mode(MODE_DAC);
dac_init();
dac_set_ref(DAC_MODULE_D1,DAC_INT_REF);
dac_power_down(DAC_MODULE_D1,0x0F);
dac_write(DAC_MODULE_D1,DAC_WRn_UPDn,DAC_ADDR_ALL,2047);
spi_set_mode(MODE_CPLD); //Use mode setting for CPLD for SPI to initialize SPI setting
DISABLE_CPLD();
spi_set_mode(MODE_DAC);

/*
void main_loop(void)
*/
while(quit == 0)
{
com_keyboard(); // process keypresses
if (time.msec != 0) // millisecond events
{
time.msec = 0;
vsi_state_machine();
}
else if (time.msec10 != 0) // ten millisecond events
{
time.msec10 = 0;
}
else if (time.sec0_1 != 0) // tenth of second events
{
time.sec0_1 = 0;
switch(initial)
{ /* case 0 never happens */
case 1: puts_COM1("\n	 Single-phase Bidirectional DC-DC Converter");break;
case 2: puts_COM1("\n	 Supplying a Grid Connected VSI");break;
case 3: puts_COM1("\n	 Dinesh Segaran 2011");break;
#ifdef OL_VSI
case 4: puts_COM1("\n	 M/m - VSI modulation depth up/down");break;
#endif
#ifdef CL_VSI
case 5: puts_COM1("\n	 A/a - VSI Current Ref up/down");break;
#endif
case 6: puts_COM1("\n	 l/L - Load Switch OFF/ON");break;
case 8: puts_COM1("\n	 e/d - VSI Enable/Disable
");break;
case 10: puts_COM1("\n	 g/h - Start/Display Grab");break;
case 11: puts_COM1("\n	 H - Display Help");break;
default: break;
}
if (initial<20) initial++;

if(GrabShowTrigger() && i < GRAB_LENGTH){
//GrabDisplay(0xFFFF);
GrabDisplay(i);
i++;
//GrabStop();
}
else if(GrabShowTrigger() && i == GRAB_LENGTH){
GrabStop();
i = 0;
}
}
else if (time.sec != 0) // update every 1sec
{
// puts_COM1("\n counter:");
pst_d(initial);
time.sec = 0;
idle_diff = idle_count - idle_count_old;
idle_count_old = idle_count;
if (initial>=15) com_display();
}
else // low priority events

APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```c
321 {  
322     idle_count++;  
323 } /* end while quit == 0 */
324
325 // DISABLE_PWM();
326 EvaRegs.TICON.bit.TENABLE = 0;
327 EvaRegs.ACTRA.all = 0x0000;
328 DINT;
329 }
330} /* end main */

331 // DISABLE_PWM();
332
333
334 /* =========================================================================
335 __Local_Functions()
336 ============================================================================= */
337
338 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
339
340 void com_display(void)  
341 {  
342 Uint16 status;
343
344 if(GrabShowTrigger()){
345     status = vsi_get_status();
346     if (status == VSI_FAULT)
347         putc_COM1('F');
348     else if (status==0)
349         puts_COM1(" Init ");
350     else if (status==1)
351         puts_COM1(" Gate Charge ");
352     else if (status==2)
353         puts_COM1(" Ramp ");
354     else if (status==3)
355         puts_COM1(" Run ");
356     else if (status==4)
357         puts_COM1(" Settled ");
358     else if (status==5)
359         puts_COM1(" Idle ");
360     else if (status==6)
361         puts_COM1(" FAULT ");
362     else putx(status);
363     puts_COM1("VSI: ");
364     #ifdef OL_VSI
365         puts_COM1("OL ");
366         putxx(mod_serial,2);
367     #endif
368     #ifdef CL_VSI
369         puts_COM1("CL ");
370         putdbl(Imag_serial,2);
371     #endif
372     if (load_enable==1) puts_COM1(" Load ON ");
373     else puts_COM1(" Load OFF ");
374 }
375 
376 if (status==0)  
377     puts_COM1(" Init ");
378 else if (status==1)  
379     puts_COM1(" Gate Charge ");
380 else if (status==2)  
381     puts_COM1(" Ramp ");
382 else if (status==3)  
383     puts_COM1(" Run ");
384 else if (status==4)  
385     puts_COM1(" Settled ");
386 else if (status==5)  
387     puts_COM1(" Idle ");
388 else if (status==6)  
389     puts_COM1(" FAULT ");
390     putx(status);
391 }
392 
393 #ifdef OL_VSI
394 puts_COM1("OL ");
395 putxx(mod_serial,2);
396 #endif
397 
398 #ifdef CL_VSI
399 puts_COM1("CL ");
400 putdbl(Imag_serial,2);
401 #endif
402 
403 if (load_enable==1) puts_COM1(" Load ON ");
404 else puts_COM1(" Load OFF ");
405 }
406 */ end com_display */
407
408 */ * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
409 */ void com_keyboard */
410 Parameters: none
```

230
Returns: nothing

Description: Process characters from COM0.

Notes:

History:
22/06/05 AM - initial creation
27/11/07 PM - added in testing of the digital I/O

*/

void com_keyboard(void)
{
    char c;
    // int temp=0;
    // puts_COM1("KEY");
    if (kbhit_COM1())
    {
        c = getc_COM1();
        switch (c)
        {
            case 'e': vsi_enable();
            puts_COM1("e");
            break;
            case 'd': vsi_disable();
            puts_COM1("d");
            break;
            #ifdef OL_VSI
            //step change in VSI Modulation depth
            case 'M': if (mod_serial < (2.0-VSI_MODSTEP)) mod_serial+=VSI_MODSTEP;
            else mod_serial=2.0;
            vsi_set_mod(mod_serial);
            break;
            case 'm': if (mod_serial > VSI_MODSTEP) mod_serial-=VSI_MODSTEP;
            else mod_serial=0.0;
            vsi_set_mod(mod_serial);
            break;
            #endif
            #ifdef CL_VSI
            //step change in VSI Current Reg Reference
            case 'A': if (Imag_serial< (MAX_CURR-VSI_CURRSTEP)) Imag_serial+=VSI_CURRSTEP;
            else Imag_serial=MAX_CURR;
            vsi_set_Iref_mag(Imag_serial);
            break;
            case 'a': if (Imag_serial > VSI_CURRSTEP) Imag_serial-=VSI_CURRSTEP;
            else Imag_serial=0.0;
            vsi_set_Iref_mag(Imag_serial);
            break;
            #endif
            //Load step
            case 'l': load_enable=0; //Load off
            break;
            case 'L': load_enable=1; //load on
            break;
            case 'H': // write help info
            initial=0;
            break;
            #ifdef GRAB.Include
            case 'g': /* grab interrupt data */
            GrabClear();
            GrabStart();
            GrabInit();
            break;
            case 'b':
            puts_COM1("\n\nGrab Display\nIndex\n");
            GrabShow();
            break;
            case 'c': /* stop grab display */
            GrabClear();
            break;
            #endif
            default: break;
        }
    }
}
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

```c
/* second CPU timer interrupt. */

#define BUILD_RAM

#pragma CODE_SECTION(isr_cpu_timer0, "ramfuncs");

interrupt void isr_cpu_timer0(void)
{
    static struct {
        Uint16 msec,
        msec10,
        msec100,
        sec;
    } i_count = {
        0, 0, 0
    };

    /*for (ii=0; ii<WD_TIMER_MAX; ii++)
    {
        if (wd_timer[ii] > 0)
            wd_timer[ii]--;
    }*/

    i_count.msec++;
    if (i_count.msec >= 10)
        i_count.msec = 0;
        i_count.msec10++;
        if (i_count.msec10 >= 10)
            i_count.msec10 = 0;
            i_count.msec100++;
            if (i_count.msec100 >= 10)
                i_count.msec100 = 0;
                time.sec = 1;
                time.msec10 = 1;

    // Acknowledge this interrupt to receive more interrupts from group 1
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}

/* and isr_cpu_timer0 */

/*for (ii=0; ii<WD_TIMER_MAX; ii++)
{
    if (wd_timer[ii] > 0)
        wd_timer[ii]--;
}*/

void GrabInit(void)
{
    Uint16 i, j;
    for (i=0; i<GRAB_LENGTH; i++)
    {
        for (j=0; j<GRAB_WIDTH; j++)
        {
            grab_array[i][j] = 0;
        }
    }
}
```

232
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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("index");
575 for (i=0; i<GRAB_WIDTH; i++)
576 {
577 puts_COM1("g");
578 put_d(i);
579 }
580 }
581 else
582 {
583 put_d(index);
584 for (i=0; i<GRAB_WIDTH; i++)
585 {
586 puts_COM1("\t");
587 putdbl(grab_array[index][i],3);
588 }
589 }
590 puts_COM1("\n");
591 }
592 #endif
593 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

233
/* =========================================================================
___Includes()
========================================================================= */

/* =========================================================================
__Macros()
============================================================================ */

#define SW_ENABLE() {
  CPLD.EVACOMCON.bit.ENA = 1;
  cpld_write(ADD_EVACOMCON,CPLD.EVACOMCON.all);
}

#define SW_DISABLE() {
  CPLD.EVACOMCON.bit.ENA = 0;
  cpld_write(ADD_EVACOMCON,CPLD.EVACOMCON.all);
}

#define VSI_DISABLE() {
  EvaRegs.ACTRA.all = 0x0000;
}

#define VSI_ENABLE() {
  CPLD.EVACOMCON.bit.ENA = 1;
  cpld_write(ADD_EVACOMCON,CPLD.EVACOMCON.all);
} // single phase only

#define VSI_GATE_CHARGE() EvaRegs.ACTRA.all = 0x0000

#define SIN_TABLE_READ(PHASE,SIN_VAL){
  SIN_VAL = (int16)sin_table[((Uint16)PHASE>>6)|0x0001];
  VAL_DIFF = (sin_table[((Uint16)PHASE>>6)+2]) - SIN_VAL;
  SIN_VAL += (int16)( ((int32)((Uint16)PHASE&0x007F)*(int32)VAL_DIFF)>>6 );}

// phase is a 16bit number, but the index is only 10 (513 values). The whole sine wave is represented in 16bits (0->2^32),
// shift right by 6 to know where to aim in the sine table. interpolate using the last 7 bits.

/* =========================================================================
___Hash_Definitions()
========================================================================= */

#define FIXED_Q 11
#define FIXED_Q_SCALE (long)2048

/** @name VSI Status bit definitions */
//@{
#define VSI_INIT 0x0000
#define VSI_GATECHARGE 0x0001 ///< VSI is running
#define VSI_RAMP 0x0002 ///< VSI is running
#define VSI_RUNNING 0x0003 ///< VSI is running
}
#define VSI_SETTLED 0x0004 ///< set when target reached
#define VSI_STOP 0x0005 ///< VSI is running
#define VSI_FAULT 0x0006 ///< set when fault present in VSI system
//@}

/** @name Fault Codes */
//@{
#define FAULT_VSI_IAC_OL 0x0001
#define FAULT_VSI_IAC_OC 0x0002
#define FAULT_VSI_VDC_OV 0x0004
#define FAULT_VSI_VDC_UV 0x0008
#define FAULT_VSI_PDPINT 0x0010
#define FAULT_VSI_SPI 0x0020
//@}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Zero crossing states */
#define ZX_LOST 0 ///< No idea of anything
#define ZX_EST 1 ///< Initial fundamental frequency estimation
#define ZX_SYNC 2 ///< nudges the phase to stay synchronised
#define ZX_FREQ 3 ///< nudges the freq (phase_step) for persistent err
#define ZX_LOCK 4 ///< tests to see if system is locked into sync
#define ZX_MISC 5 ///< load levelling calculation state

/* Zero crossing constants */
/* Sync lost if no ZX in ~3.5 cycles */
#define ZX_MAX_COUNT ((Uint16)(3.5*FSAMPLE_BIDC/F_FUND)) // 1050
#define ZX_CYCLE_AVG 64 /* Number of cycles for frequency estimate */
#define ZX_SYNC_LIMIT 10 /* Number of cycles in sync */
#define ZX_BIG_ERR (400*65536) /* ~2.2 degrees */
#define ZX_PHASE_ERR (3600*65536) // ~20 degrees - maximum sync phase error
#define ZX_FREQ_ERR (100*65536) // Persistent phase error for freq change
#define ZX_FREQ_ERR_BIG (200*65536) // Persistent phase error for freq change
#define ZX_OFFSET_POS (-4500*65536) // trim phase for +ve phase seq
#define ZX_OFFSET_NEG (6700*65536) // trim phase for -ve phase seq

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

//Topology parameters
{"code_lines":null}
#define C 20e-6 //20uF
#define L 132e-6
#define R_L 0.1
#define R_L_2 R_L*R_L
#define LVSI (16e-3)
#define OMEGA_BIDC_L (OMEGA_BIDC*L)
#define OMEGA_BIDC_L_2 (OMEGA_BIDC_L*OMEGA_BIDC_L)
#define NPRI 10.0
#define NSEC 11.0
#define NPRI_NSEC ((double)(NPRI/NSEC))
#define VIN 200.0
#define VDCPRI (VIN/2.0)
#define VDC_VSI 200.0
#define VBUS_NOM_FIXED ((int32)(VDC_VSI*FIXED_Q_SCALE))
#define MAX_CURR (12.0)
#define MAX_CURR_FIXED ((long)(MAX_CURR*FIXED_Q_SCALE))

//VSI Parameters
{"code_lines":null}
#define SW_FREQ_VSI 5000.0
#define PERIOD_VSI ((Uint16)((double)HSPCLK/SW_FREQ_VSI)/2.0)) // Carrier timer half period in clock ticks
#define FRAGMENT_VSI ((SW_FREQ_VSI+1.0)
#define TSAMPLE_VSI (1.0/FRACTMHz)
#define T_DELAY_VSI (1.5*TSAMPLE_VSI)
#define F_FUND_MAX 60.0
#define F_FUND 50.0
#define F_FUND_MIN 40.0
#define OMEGA_C_VSI (PI_2-(40*DEG_TO_RAD))/(T_DELAY_VSI) //40 deg phase margin
#define KP_CONST ((int32)(LVSI*OMEGA_C_VSI*FIXED_Q_SCALE))
#define PHASE_STEP (Uint16)(65536.0*F_FUND/SW_FREQ_VSI/2.0)
#define PHASE_STEP (Uint32)(4294967296.0/(double)F_FUND/(double)SW_FREQ_VSI/2.0)

APPENDIX A. SIMULATION & EXPERIMENTAL CODE
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

161 #define KP_VSI (OMEGA_C_VSI*LVSI/VDC_VSI)
162 #define KI_VSI (OMEGA_C_VSI/10/FSAMPLE_VSI)
163 /// Maximum VSI switching time in clock ticks
164 #define MIN_VSI_TIME 1e-6
165 #define MIN_VSI_COUNT (HSPCLK*MIN_VSI_TIME)
166 #define MAX_VSI_TIME (int16)(PERIOD_2_VSI-MIN_VSI_COUNT)
167
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
184 //DAC hash defines
185 //
186 #define DAC_SCALE_VREF 2048.0/50.0
187 #define DAC_SCALE_PHASE 2048.0/100.0
188 #define DAC_SCALE_VA ((long)(FIXED_Q_SCALE/(double)PERIOD_2_VSI)) //scaled by FIXED_Q
189 //
190
192 // sine table hash definitions
193 //
194 // COUNT_TO_SINTABLE (32768.0/PERIOD_BIDC)
195 #define COUNT_TO_SINTABLE (long)(32768.0/PERIOD_BIDC)
196 #define COUNT_TO_RAD PI/3750.0
197 #define RAD_TO_COUNT 3750.0/PI
198 #define DEG_TO_RAD PI/180.0
199 //
200 /***************
201 //ADC calibration time
202 //
203 #define ADC_CAL_TIME 1// seconds
204 #define ADC_COUNT_CAL (Uint16)(ADC_CAL_TIME * 20000 * 2.0)
205 #define ADC_REAL_SC 1
206 //
207 /***************
208 /*******************/
209 // ADC Scaling
210 //
211 #define ADC_DA_SCALE_MULT (long)3 //3.0/4096.0 - scaled by FIXED_Q+5 cos num is so small
212 #define ADC_DA_SCALE_SHIFT 12
213 #define ADC_DA_SHIFT 4
214 //
215 // GIIB Scaling Resistors
216 #define RFB_GIIB_VAC (long)10000 //10000.0 //feedback resistor on GIIB board
217 #define RIN_GIIB_VAC (long)(150000+150000+150000) //150000.0+180000.0+180000.0 //preloaded input resistor on GIIB board
218 #define RFB_GIIB_VDC (long)10000 //10000.0 //feedback resistor on GIIB board
219 #define RIN_GIIB_VDC (long)(150000+180000+180000) //150000.0+180000.0+180000.0 //preloaded input resistor on GIIB board
220
223
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 // GIIB Scaling Resistors
234 //
235 #define NFB_GIIB_VAC (long)10000 //10000.0 //feedback resistor on GIIB board
236 #define NFB_GIIB_VAC (long)(150000+150000+150000) //150000.0+180000.0+180000.0 //preloaded input resistor on GIIB board
237 //
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

// AC Voltage Inputs
#define RIN_GIIB_ADD_VAC (long)0 //NO additional scaling resistor on GIIB board
#define RIN_GIIB_TOTAL_VAC RIN_GIIB_VAC //Input resistor to GIIB op amp stage
#define VAC_GIIB_GAIN (long)((-1.0*(double)RFB_GIIB_VAC*FIXED_Q_SCALE)/(double)RIN_GIIB_TOTAL_VAC) //scaled by FIXED_Q
#define VAC_GIIB_GAIN_INV (long)(((double)FIXED_Q_SCALE*(double)FIXED_Q_SCALE)/(double)VAC_GIIB_GAIN) //scaled by 2^9

// DC Voltage Inputs
#define RIN_GIIB_ADD_VDC (long)150000 //NO additional scaling resistor on GIIB board
#define RIN_GIIB_TOTAL_VDC RIN_GIIB_VDC
#define VDC_GIIB_GAIN ((-1.0*(double)RFB_GIIB_VDC)/(double)RIN_GIIB_TOTAL_VDC) //scaled by FIXED_Q
#define VDC_GIIB_GAIN_INV (long)(FIXED_Q_SCALE/VDC_GIIB_GAIN) //scaled by 2^9

// Mini2810 Scaling Resistors
#define RUP_MINI1 (long)6800
#define RUP_MINI2 (long)4700
#define RUP_MINI_TOTAL (long)((RUP_MINI1*RUP_MINI2)/(RUP_MINI1+RUP_MINI2))
#define RDWN_MINI (long)6800
#define RDOWN_MINI_TOTAL (long)((RDWN_MINI*RIN_MINI)/(RDWN_MINI+RIN_MINI))

// Mini2810 ADC Scaling
#define ADC_MINI_GAIN (((double)(RUP_MINI_TOTAL*RDOWN_MINI_TOTAL))/((double)((RUP_MINI_TOTAL+RDOWN_MINI_TOTAL)*RIN_MINI))) //is a double
#define ADC_MINI_GAIN_INV (long)(FIXED_Q_SCALE/ADC_MINI_GAIN) //scaled by FIXED_Q
#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
#define ADC_OFFSET (long)(((MINI_LEVEL_SHIFT<<ADC_DA_SCALE_SHIFT)>>FIXED_Q)/ADC_DA_SCALE_MULT) //in counts

// Voltage Overall Gain
#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)) //scaled by FIXED_Q 1/(VAC_GIIB_GAIN*ADC_MINI_GAIN)
#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))

// Current Inputs
#define CT_RATIO 2000.0 //For LA 100P SP13, it is 1000, for LA 100P - 2000
#define CT_TURNS 2.0
#define BURDEN_R 380.0 //Burden resistor
#define LEM_GAIN ((CT_TURNS*BURDEN_R)/CT_RATIO)
#define LEM_GAIN_INV (1.0/LEM_GAIN) //is a double
#define RIN1_GIIB_I 10000.0 //Input resistor to GIIB op amp stage
#define RIN2_GIIB_I 10000.0
#define RIN_GIIB_TOTAL_I ((RIN1_GIIB_I*RIN2_GIIB_I)/(RIN1_GIIB_I+RIN2_GIIB_I))
#define I_GIIB_GAIN (-1.0*RFB_GIIB_I/RIN_GIIB_TOTAL_I) //Voltage gain of amplifier on GIIB for current
#define I_GIIB_GAIN_INV (1.0/I_GIIB_GAIN) //Voltage gain of amplifier on GIIB for current
#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)
#define GAIN_OFFSET_CURRENT 1.0 //this is to account for the differences in scaling resistors

// Scaling for reading VGEN - takes a +/-10V signal
#define VREF_MAX 205 //101
#define VREF_MIN 10
#define VREF_STEP_S 1
#define VREF_STEP_L 10
#define VREF_OFFSET_CURRENT 1.0/1.08
FunctionPrototypes()

============================================================================ */

void vsi_init(void);

void vsi_state_machine(void);

void vsi_enable(void);

void vsi_disable(void);

Uint16 vsi_set_fund(double f);

void vsi_set_mod(double mod_serial);

void vsi_set_Iref_mag(double mag_serial);

void vsi_set_vref(int16 vref);

Uint16 vsi_get_status(void);

Uint16 vsi_get_faults(void);

void vsi_clear_faults(void);

void get_state(void);

void calibrate_adc(void);

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

1 /
2 /**
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 measurement. The carrier timer for the VSI generation also triggers the internal ADC conversion at the peak of the carrier. The end of conversion then triggers this interrupt. Its tasks are:
7 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 \par History:
18 15/03/09 AM - initial creation
19 "Seeded"
20 11/11/09 Ds - Turning this into a GIIB-Based Bidirectional DC-DC Converter
21 28/08/10 Ds - Fixed Point implementation of the Adaptive Controlled Bidirectional DC-DC Converter
22 15/03/11 Ds - Grid Connected H-Bridge, with DC links supplied by a
23 Bidirectional DC-DC Converter
24
25 */
26
27 /************
28 CODE_TASKS()
29 ************/
30
31 15/03/2011 - Trying to implement a single phase VSI on the
32 E-10 Gate Driver Board (GDB), whose DC link is
33 supplied by a Single Phase Bi-directional DC-DC
34 Converter
35 - PWM to be generated on EVB,i.e CMPR4&5, and routed out
36 through CPLD (Needs to be coded), and to the GDB.
37 - First, run a 10kHz interrupt. open loop VSI
38 20/3/2011 - Gate Drive Resistors = 27 ohms.
39 21/3/2011 - Scaling Resistors:
40 - AC Voltage - Standard scaling to read +/- 450VAC
41 - DC Voltage - Scaled to read 510Vdc
42 - AC Current - Scaled for +/- 15A - 2 turns - Burden Resistor = 270ohm
43 - DC Current - Scaled for +/- 15A - 2 turns - Burden Resistor = 270ohm
44
45 6/4/2011 - ADCs fully tested
46 - Grid Synch code included (not yet working)
47 - phase now a 32-bit number
48 - Change in strategy
49 - PWM for H-bridge comes from PWMA
50 - PWM for Sec Bridge comes from PWMB
51 - 2 lines sent to Master Bridge: 1) Synch Pulse
52 - 2) Fault trigger
53 8/4/2011 - Unable to use CPLD to route gate signals, because it will use up capture port
54 - Instead, use hysteresis inputs to route gate drives for Sec Side
55 - To synth, still use two lines: 1) Synch Pulse
56 - 2) Enable/Disable
57 13/4/2011 - Open Loop H-bridge running at 200V operational
58 - DC voltage measurements set at +/- 450VAC
59 - AC voltage measurements set at +510V - no scaling resistors
60 - AC current measurements set at 95 ohms - 2 turns - so +/- 20A
61 21/4/2011 - Closed Loop H-bridge and a closed loop bidirectional DC-DC converter work.
62 - Need to implement feed-forward compensation. For this, need to synch switching and send mod depth info across.
63 - Stage 1: - Synchronise Carriers. use zaki's code.
64 - Synchronise the VSI to the BiDC because the BiDC uses a lot of DIGIO pins already.
65 - Use the shielded ribbon cable for this. Build Loopback function and test.
66 - Loopback cable = GPIOB4-6 (PWMB4-6) are routed back into DIG16 on the BiDC, send out a synch pulse at 5kHz (1 every 8 interrupts) on GPIOB4.
67 - This is DIGOUT5, pin 5 on the 20-pin header.
68 - On the VSI, bring the sync pulse into CP2P. this is on DIG16, which is pin 16. is connect pins 5 & 16.
69
70 PORTED OVER TO OPEN GIIB!!!!!!!
71 For this experiment I want a current regulated H bridge running from a bidirectional that supplies 200V
72 23/04/09 AM - initial creation
73 25/04/09 AM - initial creation
74 26/04/09 AM - initial creation
75 27/04/09 AM - initial creation
76 28/04/09 AM - initial creation
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

- Also connect all the GNDs on the 20-pin header together. i.e., leave pins 18 & 20.
- Disconnect VCC, i.e., cut pins 17 & 19
- That lets you lift synch code from GridCon set, and also the fault trigger when synch is lost.

Stage 2: Phase & Modulation depth information. Via SPI or via DAC?
22/4/2011 - Bridges Synchronised. NEED TO ADD IN EMERGENCY STOP CODE
- able to send va over DAC, will be read in by Vgen. needs a mascon header
- uses the shielded ribbon, cable tied to the synch pulse cable. Connect the AGND as well. seems to work fine for now.

*/
// compiler standard include files
#include <math.h>

// processor standard include files
#include <DSP281x_Device.h>

#ifdef COM0_CONSOLE
#include <bios0.h>
#endif
#ifdef COM1_CONSOLE
#include <bios1.h>
#endif

// board standard include files
#include <lib_mini2810.h>
#include <dac_ad56.h>
#include <lib_cpld.h>
#include <lib_giib.h>

// local include files
#include "main.h"
#include "conio.h"
#include "vsi_InvLoad.h"

/*===========================================================================*/

#define ROM_TABLE_SIZE 512
#define ROM_TABLE_PEAK 16384

/*===========================================================================*/

typedef struct
{
  int16 raw, ///< raw ADC result from last sampling
  filt; ///< decaying average fast filter of raw data
} type_adc_ch;

typedef struct
{
  int16 raw, ///< raw ADC result from last sampling
  filt; ///< decaying average fast filter of raw data
  int16 type_adc_ch;
}

typedef struct
{
  int16 raw, ///< raw ADC result from last sampling
  filt; ///< decaying average fast filter of raw data
  int16 type_adc_ch;
}

typedef struct
{
  int16 raw, ///< raw ADC result from last sampling
  filt; ///< decaying average fast filter of raw data
  int16 type_adc_ch;
}

typedef struct
{
  int16 raw, ///< raw ADC result from last sampling
  filt; ///< decaying average fast filter of raw data
  int16 type_adc_ch;
}
count_dc, ///< counter for DC averaging
count_dc_bak, ///< background copy of DC counter
flag_cal, ///< flag set to trigger background calibration averaging
flag_rms, ///< flag set to trigger background RMS averaging
flag_dc; ///< flag set to trigger background DC averaging

A0, ///< ADC channel A0
A1, ///< ADC channel A1
A2, ///< ADC channel A2
A3, ///< ADC channel A3
A4, ///< ADC channel A4
A5, ///< ADC channel A5
A6,
B0, ///< ADC channel B0
B1, ///< ADC channel B1
B2, ///< ADC channel B2
B3, ///< ADC channel B3
B4, ///< ADC channel B4
B5, ///< ADC channel B5
yHA, ///< bank A high reference
yLA, ///< bank A low reference
yHB, ///< bank B high reference
yLB; ///< bank B low reference

} type_adc_int;

/** @name Internal ADC Variables */
//@{


// ADC calibration variables
int16 cal_gainA = 1<<14, // calibration gain factor for A channel
cal_gainB = 1<<14, // calibration gain factor for B channel
cal_offsetA = 0, // calibration offset for A channel
cal_offsetB = 0; // calibration offset for B channel
double cal_gain_A, cal_gain_B,
cal_offset_A, cal_offset_B;

double yHA = 0.0,
yLA,
yHB,
yLB;

} type_adc_int =

{ 0, // count_cal
 0, // count_rms
 0, // count_rms_bak
 0, // count_dc
 0, // count_dc_bak
 0, // flag_cal
 0, // flag_rms
 0, // flag_dc

0, // raw
0, // filt
0, // rms_sum
0, // rms_sum_bak
0, // dc_sum
0, // dc_sum_bak
0.0 // real
}, // #A0

{ 0, 0, OL, OL, OL, OL, 0.0 }, // #B0

{ 0, 0, OL, OL, OL, OL, 0.0 }, // yHA

{ 0, 0, OL, OL, OL, OL, 0.0 }, // yLA

{ 0, 0, OL, OL, OL, OL, 0.0 }, // yLB

};


/* =========================================================================
__Variables()
============================================================================ */

// state machine level variables
Uint16 vsi_status = 0, /// Status of VSI system
is_switching = 0, // flag set if PWM switching is active
vsi_counter = 0, // counter for timing VSI regulation events
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

dac_vref=0,
spi_fail_count,
dac_phasesref=0; //FOR DAC

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 U32 detected_faults = 0,
265 timer_synch_count = 100,
266 first=0; // bits set for faults detected (possibly cleared)
267
268 //DAC Variable
270 U32 data_out;
271 int _spi;

272
273 /***************
274 _Macro_Variables()
275 *****************/

276 //sin table read variables
277 U32 PHASE;
278 U32 SIN_VAL,
279 VAL_DIFF; // interpolation temp variable

280 /****************
281 _DSP_Emulation_Variables()
282 **********************/

283 int16 UF_VSI,
284 int_vsi_count,//to tell which interrupt to run in.
285 int_count=0;

286
287 /*****************
288 _VSI_Modulation_Variables()
289 *******************/

290 int16 va,
291 max_time,
292 t_A,
293 t_B,
294 sin_val,
295 cos_val,
296 val_diff;

297 U32 vsiphase = 0,
298 prev_sin_val = 0,
299 ZZ_vsi_phase=0,
300 phase_step;
301
302 U32 V_Asat = 0,
303 V_Bsat = 0;
304
305 double mod=0.0;
306
307 /**********************/
308 _Grid_Synch_Variables()
309 *********************

310 // @name Zero Crossing Synch Variables @
311 @{
312 Uint16
313 ZX_seen = 0, /// flag set when a zx event is detected
314 in_sync = 0, /// Flag to indicate that sync is achieved
315 ZZ_in_sync = 0, /// ZZ is at the state of the zero crossing synch process
316 ZZ_state = ZZ_LH, /// ZZ_SYNC_LHIT means that sync has been achieved
317 ZZ_count = 0, /// ZZ_count_grab, // for grab code only
318 ZZ_count_grab, // for grab code only
320 }
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

ZX_cycles = 0, ///< Count of number of ZXs during averaging
ZX_sum = 0; ///< Running sum for average
Uint32
ZX_phase_step = PHASE_STEP; ///< Change in phase angle in half a switching cycle
int16
ZX_time = 0; ///< Time of captured ZX in timer units
int32
ZX_time_phase = 0L, ///< Time of captured ZX in phase units
zx_offset = ZX_OFFSET_POS, ///< variable offset for tuning
ZX_phase_scale = 0L, ///< Scale factor between timer and phase units
ZX_phase_err = 0L, ///< Difference in phase units (2^16 == 360deg)
ZX_err_sum = 0L; ///< Integral for frequency control
//@}

__Control_Loop_Variables()

//Interface variables used to receive controller loop parameters from background
//Controller loop turning parameters in real floating pointer number from background

_VSI_Curreg_Variables()

//ADC_Calibration_Variables()

//DAC_Variables()
# Appendix A. Simulation & Experimental Code

```
401 /**************************************************************************
402 _LoadStep_Variables()
403 /**************************************************************************/
404 int16 load_enable=0,
405 prev_load_enable;
406
407 /**************************************************************************
408 __Local_Function_Prototypes()
409 /**************************************************************************/
410 void
411 st_vsi_init(void), // initialises CPPF regulator
412 st_vsi_stop(void), // waiting for start trigger
413 st_vsi_gate_charge(void), // delay to charge the high side gate drivers
414 st_vsi_ramp(void), // ramping to target mod depth
415 st_vsi_run(void), // maintaining target mod depth
416 st_vsi_fault(void); // delay after faults are cleared
417
418 // ADC and VSI interrupt
419 interrupt void isr_adc(void);
420
421 // capture port interrupt
422 interrupt void isr_cap2(void);
423
424 // Gate fault (PDPINT) interrupt
425 interrupt void isr_gate_fault(void);
426
427 /**************************************************************************
428 /* State Machine Variable */
429 /**************************************************************************
430 type_state
431 vsi_state =
432 {
433 &st_vsi_init,
434 1
435 },
436
437 /**************************************************************************
438 __Exported_ADC_Functions()
439 /**************************************************************************/
440 int
441 result_registers[31];
442
443 /**************************************************************************
444 This function initialises the ADC and VSI interrupt module. It sets the
445 internal ADC to sample the DA-2810 analog inputs and timer1 to generate a PWM
446 carrier and the event manager A to generate the VSI switching. It also
447 initialises all the relevant variables and sets up the interrupt service
448 routines.
449
450 This function initialises the ADC unit to:
451 - Trigger a conversion sequence from timer 1 overflow
452 - Convert the appropriate ADC channels
453
454 Result registers as follows:
455 - ADCRESULT0 = ADCHB0
456 - ADCRESULT1 = ADCHB1
457 - ADCRESULT2 = ADCHB2
458 - ADCRESULT3 = ADCHB3
459 - ADCRESULT4 = ADCHB4
460 - ADCRESULT5 = ADCHB5
461 - ADCRESULT6 = ADCHB6
462 - ADCRESULT7 = ADCHB7
463 - ADCRESULT8 = ADCHR0
464 - ADCRESULT9 = ADCHR1
465 - ADCRESULT10 = ADCHR2
466 - ADCRESULT11 = ADCHR3
467 - ADCRESULT12 = ADCHR4
468 - ADCRESULT13 = ADCHR5
469 - ADCRESULT14 = ADCHR6
470 - ADCRESULT15 = ADCHR7
471
472 It initializes the Event Manager A unit to:
473 - drive PWM1-4 as PWM pins not GPIO
474 - a 0.48ns deadtime between the high and low side pins
475 - Timer 1 as an up/down counter for the PWM carrier
476 It initializes the PIE unit to:
477 - Take PDPINTA as a power stage interrupt
478 - Use the internal ADC completion interrupt to trigger the main ISR
```
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

481 \author A. McIver
482 \par History:
483 \% 12/10/07 AM - initial creation
484 \% 26/08/10 DS - Fixed Point Bidirectional DC-DC Converter
485 */
486 void vsi_init(void)
487 {
488 //EVA
489 EvaRegs.ACTRA.all = 0x0000;
490 EvaRegs.GPTCONA.all = 0x0000;
491 EvaRegs.EVTIMMA.all = 0x0000;
492 EvaRegs.EVTIPRA.all = BIT0;
493 EvaRegs.COMCMA.all = 0x0000;
494 // Set up ISRs
495 EALLOW;
496 PieVectTable.ADCINT = &isr_adc;
497 PieVectTable.CAPINT2 = &isr_cap2;
498 PieVectTable.PDPINTA = &isr_gate_fault;
499 EDIS;
500 // Set up compare outputs
501 EALLOW;
502 OpicMuxRegs.GPOMUX.all = BIT0;
503 //EVA
504 OpicMuxRegs.GPOMUX.bit.PWM1_GPIOA0 = 1; // enable PWM3 pin
505 OpicMuxRegs.GPOMUX.bit.PWM3_GPIOA2 = 1; // enable PWM3 pin
506 OpicMuxRegs.GPOMUX.bit.PWM4_GPIOA3 = 1; // enable PWM4 pin
507 OpicMuxRegs.GPOMUX.bit.PWM5_GPIOA4 = 1; // enable PWM4 pin
508 OpicMuxRegs.GPOMUX.bit.PWM6_GPIOA5 = 1; // enable PWM3 pin
509 //DEADBAND CONTROL
510 //EVA
511 EvaRegs.DBTCONA.bit.DBT = 5; //1.0us deadtime
512 EvaRegs.DBTCONA.bit.EDBT1 = 1;
513 EvaRegs.DBTCONA.bit.EDBT2 = 1;
514 EvaRegs.DBTCONA.bit.EDBT3 = 1;
515 EvaRegs.DBTCONA.bit.DBTPS = 6;
516 //COMPARE REGISTERS
517 //EVA - Current Reg H-bridge
518 EvaRegs.CMPR2 = PERIOD_2_VSI;
519 EvaRegs.CMPR3 = PERIOD_2_VSI;
520 // Setup and load COMCON
521 //EVA
522 EvaRegs.COMCMA.bit.ACTRD = 1; // reload ACTR on underflow or period match
523 EvaRegs.COMCMA.bit.SVENABLE = 0; // disable space vector PWM
524 EvaRegs.COMCMA.bit.CLD = 1; // reload on underflow & period match
525 EvaRegs.COMCMA.bit.FCOMPOE = 1; // full compare enable
526 EvaRegs.COMCMA.bit.CENABLE = 1; // enable compare operation
527 // Setup Timer 1
528 EvaRegs.T1CON.all = 0x0000;
529 EvaRegs.T1PR = PERIOD_VSI;
530 EvaRegs.T1CNT = 0x0000;
531 EvaRegs.T1CON.bit.TMODE = 1; // continuous up/down count mode
532 EvaRegs.T1CON.bit.TCMPOE = 1; // full compare enable
533 EvaRegs.T1CON.bit.T2SWT1 = 1; // Use TENABLE bit of GP Timer 1
534 // Setup Timer 2
535 EvaRegs.T2CON.all = 0x0000;
536 EvaRegs.T2PR = PERIOD_VSI;
537 EvaRegs.T2PR = 0;
538 EvaRegs.T2CONF = 0;
539 EvaRegs.T2CONF.bit.TMODE = 1; // continuous up/down count mode
540 EvaRegs.T2CONF.bit.TCMPOE = 1; // full compare enable
541 EvaRegs.T2CONF.bit.T2SWT1 = 1; // Use TENABLE bit of GP Timer 1
542 // Setup capture port 2
543 EvaRegs.CAPCMA.all = 0x0000;
544 EvaRegs.CAPCMA.bit.TCMP = 0;
545 EvaRegs.CAPCMA.bit.T2CMP = 0; // use own period register
546 // Set up capture port
547 EvaRegs.CAPCMA.bit.T2SEL = 1;
548 EvaRegs.CAPCMA.bit.T2SEL = 0;
549 EvaRegs.CAPCMA.bit.T2SEL = 1;
550 EvaRegs.CAPCMA.bit.T2SEL = 0; // continuous up mode
551 EvaRegs.CAPCMA.bit.T2SEL = 0; // input clock prescaler
552 EvaRegs.CAPCMA.bit.T2SEL = 1; // disable time compare
553 EvaRegs.CAPCMA.bit.T2SEL = 1; // use own period register
554 // Set up capture port
555 EvaRegs.CAPCMA.bit.T2SEL = 1;
556 // Set up capture port
557 EvaRegs.CAPFIFOA.all = 0x0000;
558
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

561  // Capture 2 gets Timer 1 on rising edge
562  EvaRegs.CAPCONA.bit.CAPRES = 1; // Release from reset
563  EvaRegs.CAPCONA.bit.CAP12TSEL = 0; // Select Timer 2
564  EvaRegs.CAPCONA.bit.CAP12EN = 1; // Enable captures 1 and 2
565  EvaRegs.CAPCONA.bit.CAP2EDGE = 1; // detects rising edge on Capture 2
566  // keep an initial value in Capfifo register so that the first edge is indeed captured
567  EvaRegs.CAPFIFOA.bit.CAP2FIFO = 1;
568  GpioMuxRegs.GPAMUX.bit.CAP1Q1_GPIOA8 = 0; // select GPIO
569  GpioMuxRegs.GPAMUX.bit.CAP2Q2_GPIOA9 = 1; // select capture port 2 - synch
570  GpioMuxRegs.GPAMUX.bit.CAP3Q1_GPIOA10 = 0; // select GPIO
571  GpioMuxRegs.GPBMUX.bit.CAP4Q1_GPIOB8 = 0; // select GPIO
572  GpioMuxRegs.GPBMUX.bit.CAP5Q2_GPIOB9 = 0; // select GPIO
573  GpioMuxRegs.GPBMUX.bit.CAP6Q1_GPIOB10 = 0; // select GPIO
574  GpioMuxRegs.GPBMUX.bit.CAP7Q2_GPIOB11 = 0; // select GPIO
575  // Set up ADC
576  // Setup and load GPTCONA
577  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
578  // these are being done in A/B pairs
579  AdcRegs.ADCMAXCONV.all = 0x0007; // Setup 8 conv's on SEQ1 // To Oversample?
580  AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0x0; // (A0/B0) - ADCRESULT0 - ADCINA0 - APOT1/I3 - SW_A - default I3 - 1 ADCINB0 - VDC2 - VDC
581  AdcRegs.ADCCHSELSEQ1.bit.CONV01 = 0x1; // (A1/B1) - ADCRESULT2 - ADCINA1 - VDC3/Vac3 - SW_A - default Vac3 - 3 ADCINB1 - I5
582  AdcRegs.ADCCHSELSEQ1.bit.CONV02 = 0x2; // (A2/B2) - ADCRESULT4 - ADCINA2 - I1 - IAC OUT - 5 ADCINB2 - I4
583  AdcRegs.ADCCHSELSEQ1.bit.CONV03 = 0x3; // (A3/B3) - ADCRESULT6 - ADCINA3 - Vac1 - ZX - 7 ADCINB3 - VDC1 - VDC
584  AdcRegs.ADCCHSELSEQ2.bit.CONV04 = 0x4; // (A4/B4) - ADCRESULT8 - ADCINA4 - I2 - 9 ADCINB4 - APOT2/I6 - SW_B - default I6 - 11 ADCINB5 - Vgen/VDC4 - VDC - 13 ADCINB6 - 2.5V ref
585  AdcRegs.ADCCHSELSEQ2.bit.CONV05 = 0x5; // (A5/B5) - ADCRESULT10 - ADCINA5 - Vac2 - 15 ADCINB7 - 2.5V ref
586  AdcRegs.ADCCHSELSEQ2.bit.CONV06 = 0x6; // (A6/B6) - ADCRESULT12 - ADCINA6 - 2.5V ref - 17 ADCINB8 - 1.25V ref
587  AdcRegs.ADCCHSELSEQ2.bit.CONV07 = 0x7; // (A7/B7) - ADCRESULT14 - ADCINA7 - 1.25V ref - 19 ADCINB9 - 1.25V ref
588  AdcRegs.ADCTRL1.bit.ACQ_PS = 1; // lengthen acq window size
589  AdcRegs.ADCTRL1.bit.SEQ_CASC = 1; // cascaded sequencer mode
590  AdcRegs.ADCTRL2.bit.EVA_SOC_SEQ1 = 1; // EVA manager start - enabled
591  AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = 1; // interrupt enable
592  AdcRegs.ADCTRL3.bit.SMODE_SEL = 1; // simultaneous sampling mode
593  AdcRegs.ADCTRL3.bit.ADCCLKPS = 0x04; // ADCLK = HSPCLK/8 (9.375MHz)
594  // Enable interrupts
595  IER |= M_INT1; // Enable CPU Interrupts 1
596  IER |= M_INT3; // Enable CPU Interrupts 3
597  EINT;
598  AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1; // clear interrupt flag from ADC
599  PieCtrlRegs.PIEIER1.bit.INTx1 = 1; // enable interrupt 1
600  PieCtrlRegs.PIEIER1.bit.INTx3 = 1; // enable interrupt 3
601  AdcRegs.EVAINA.all = 0; // disable all interrupts
602  // Enable PDPINTA: clear PDPINT flag,
603  EvaRegs.EVAIFRA.all = 0; // disable all interrupts
604  EvaRegs.EVAINA.all = 1; // enable interrupt 1
605  EvaRegs.EVAIFRA.all = 1; // enable interrupt 3
606  EvaRegs.EVAIFRA.bit.PDPINTA = 1; // enable interrupt 1
607  // Capture port interrupts
608  EvaRegs.EVAINB.all = 0; // disable all capture port interrupt
609  EvaRegs.EVAINB.bit.PDPINTA = 1; // enable capture port 2 interrupt
610  EvaRegs.EVAINB.all = 1; // enabling capture port 2 interrupt
611  // Enable PDPINTA in PIE: Group 1 interrupt 1
612  PieCtrlRegs.PIESEL1.bit.INTx1 = 1; // enable interrupt 1
613  PieCtrlRegs.PIESEL1.bit.INTx3 = 1; // enable interrupt 3
614  PieCtrlRegs.PIESEL1.bit.INTx0 = 1; // enable interrupt 0
615  // clear interrupt flag from ADC
616  PieCtrlRegs.PIESEL1.bit.INTx1 = 1; // enable interrupt 1
617  PieCtrlRegs.PIESEL1.bit.INTx3 = 1; // enable interrupt 3
618  PieCtrlRegs.PIESEL1.bit.INTx0 = 1; // enable interrupt 0
619  // clear interrupt flag from ADC
620  PieCtrlRegs.PIESEL1.bit.INTx1 = 1; // enable interrupt 1
621  PieCtrlRegs.PIESEL1.bit.INTx3 = 1; // enable interrupt 3
622  // enable all interrupts
623  PieCtrlRegs.PIESEL1.bit.INTx0 = 1; // enable interrupt 0
624  // enable interrupt 1
625  PieCtrlRegs.PIESEL1.bit.INTx1 = 1; // enable interrupt 1
626  PieCtrlRegs.PIESEL1.bit.INTx3 = 1; // enable interrupt 3
627  // enable interrupt 0
628  IER |= M_INT1; // enable CPU interrupts 1
629  IER |= M_INT3; // enable CPU interrupts 3
630  // enable interrupts
631  // determine gains
632  PID_initializations;
633  max_time = MAX_VSI_TIME;
634  phase_step = PHASE_STEP;
635  // determine gains
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

641 kp_VSI_fixed=(int32)(KP_VSI*FIXED_Q_SCALE);
642 ki_VSI_fixed=(int32)(KI_VSI*FIXED_Q_SCALE);
643 DINT;
644 EvaRegs.T1CON.bit.TENABLE = 1; // enable timer1 &2
645 // EvaRegs.T2CON.bit.TENABLE = 1; // enable timer2
646 EINT;
647 // Initialize state machine
648 vsi_state.first = 1;
649 vsi_state.f = &st_vsi_init;
650 } /* end vsi_init */
651 vsi_state.first = 1;
652 vsi_state.f = &st_vsi_init;
653 EINT;
654 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
655 /* This function is called from the main background loop once every millisecond.
656 It performs all low speed tasks associated with running the core interrupt
657 process, including:
658 - checking for faults
659 - calling the VSI state functions
660 - calling internal analog scaling functions
661 \author A.McIver
662 \par History:
663 \li 13/10/07 AM - derived from 25kVA:vsi:vsi.c
664 */
665 void vsi_state_machine(void)
666 { 
667 SS_DO(vsi_state);
668 if (adc_int.flag_cal != 0)
669 {
670 adc_int.flag_cal = 0;
671 calibrate_adc();
672 }
673 } /* end vsi_state_machine */
674 /* =========================================================================
675 __Exported_VSI_Functions()
676 ========================================================================= */
677 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
678 /* This function switches the VSI from the stopped state to a running state.
679 \author A.McIver
680 \par History:
681 \li 13/10/07 AM - derived from 25kVA:vsi:vsi.c
682 */
683 void vsi_enable(void)
684 { 
685 if (detected_faults == 0)
686 { 
687 is_switching = 1;
688 VSI_ENABLE();
689 VSI_int_fixed=0;
690 }
691 } /* end vsi_enable */
692 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
693 /* This function switches the VSI from the running state to a stop state.
694 \author A.McIver
695 \par History:
696 \li 13/10/07 AM - derived from 25kVA:vsi:vsi.c
697 */
698 void vsi_disable(void)
699 { 
700 is_switching=0;
701 VSI_DISABLE();
702 SW_DISABLE();
703 } /* end vsi_disable */
704 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

This function sets the VSI target modulation depth.
The target is passed in ????.

void vset_mod(double mod_serial)
{
    mod=mod_serial;
    if (mod>=2.0)
    {
        mod = 2.0;
    }
    else if (mod<=0)
    {
        mod = 0;
    }
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

This function sets the VSI target modulation depth.
The target is passed in ????.

void vset_Iref_mag(double Imag_serial)
{
    Iref_mag_fixed=(long)(Imag_serial*GAIN_OFFSET_CURRENT*(double)FIXED_Q_SCALE);
    if (Iref_mag_fixed>=MAX_CURR_FIXED)
    {
        Iref_mag_fixed = MAX_CURR_FIXED;
    }
    else if (Iref_mag_fixed<=0)
    {
        Iref_mag_fixed = 0;
    }
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

Set the target Fundamental frequency in Hz.

Uint16 vsi_set_freq(double f)
{
    phase_step = (Uint16)(65536.0*f/SW_FREQ_VSI/2.0);
    return phase_step;
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

This function sets the desired reference Voltage.
The target is passed in ????.

void vset_vref(int16 vref)
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 \li 28/04/08 AM - added event reporting  
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 This function returns the fault word of the VSI module.  
832  
833 \author A.McIver  
834 \par History:  
835 \li 04/03/08 AM - initial creation  
836 \li 28/04/08 AM - added event reporting  
837 \returns The present fault word  
838 */  
839 Uint16 vsi_get_faults(void)  
840 {  
841    return detected_faults;  
842 } /* end vsi_get_faults */  
843
844 /* ========================================================================= */  
845 /* Interrupt Routines */  
846 /* ========================================================================= */  
847 #ifndef BUILD_RAM  
848 #pragma CODE_SECTION(isr_cap2, "ramfuncs");  
849 \#endif  
850
851 Uint16 i;  
852
853 if (detected_faults & FAULT_VSI_PDPINT)  
854 {  
855    for (i=0; i<100; i++)  
856      i++; // delay for fault to clear  
857    EvalRegs.COMCONA.all = 0;  
858    EvalRegs.COMCONA.all = 0xAA00;  
859 }  
860
861 #ifdef BUILD_RAM  
862 \#pragma CODE_SECTION(irq_cap2, "ramfuncs");  
863 \#endif  
864
249
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

Interrupt void isr_cap2(void) // closed loop interrupt structure
int temp;

while (temp<5)
{
  SET_TP13();
temp++;
}
if (first==0) first++;
timer_synch_count++;
CLEAR_TP13();

EvaRegs.EVAIFRC.bit.CAP2INT = 1; // clear T1PINT & T1UFINT interrupt flag
PieCtrlRegs.PIEACK.all = PIEACK_GROUP3; // Acknowledge interrupt to PIE Group 2

This interrupt is triggered by the ADC interrupts.
It then:
- takes the adc measurements (synch sample, throws away every alternate one)
- determines the gains for the adaptive controller
- performs closed loop control calculations
- updates phase angle & calculates switching times

This interrupt is triggered by the ADC interrupts.
It then:
- takes the adc measurements (synch sample, throws away every alternate one)
- determines the gains for the adaptive controller
- performs closed loop control calculations
- updates phase angle & calculates switching times

interrupt void isr_time(void) // Updates VSI and performs closed loop control
This interrupt is triggered by the ADC interrupts.
It then:
- takes the adc measurements (synch sample, throws away every alternate one)
- determines the gains for the adaptive controller
- performs closed loop control calculations
- updates phase angle & calculates switching times

interrupt void isr_adc(void) // closed loop interrupt structure
Vdc1_cal = Vdc1_cal+(AdcRegs.ADCRESULT7-(ADC_OFFSET<<4));
Vdc2_cal = Vdc2_cal+(AdcRegs.ADCRESULT1-(ADC_OFFSET<<4));
Vdc3_cal = Vdc3_cal+(AdcRegs.ADCRESULT2-(ADC_OFFSET<<4));
Vdc4_cal = Vdc4_cal+(AdcRegs.ADCRESULT11-(ADC_OFFSET<<4));
Vac1_cal = Vac1_cal+(AdcRegs.ADCRESULT6-(ADC_OFFSET<<4));
Vac2_cal = Vac2_cal+(AdcRegs.ADCRESULT0-(ADC_OFFSET<<4));
Vac3_cal = Vac3_cal+(AdcRegs.ADCRESULT10-(ADC_OFFSET<<4));
I1_cal = I1_cal+(AdcRegs.ADCRESULT8-(ADC_OFFSET<<4));
I2_cal = I2_cal+(AdcRegs.ADCRESULT9-(ADC_OFFSET<<4));
I3_cal = I3_cal+(AdcRegs.ADCRESULT3-(ADC_OFFSET<<4));
I4_cal = I4_cal+(AdcRegs.ADCRESULT5-(ADC_OFFSET<<4));
I5_cal = I5_cal+(AdcRegs.ADCRESULT7-(ADC_OFFSET<<4));
I6_cal = I6_cal+(AdcRegs.ADCRESULT12-(ADC_OFFSET<<4));

Vdc1_cal = Vdc1_cal>>10;
Vdc2_cal = Vdc2_cal>>10;
Vdc3_cal = Vdc3_cal>>10;
Vdc4_cal = Vdc4_cal>>10;
Vac1_cal = Vac1_cal>>10;
Vac2_cal = Vac2_cal>>10;
Vac3_cal = Vac3_cal>>10;

// Dimash's Calibration
// Calibrate the zero offset of the ADCs by taking 1024 readings at 0V and 0A and finding the average
average = (Vdc1_cal + Vdc2_cal + Vdc3_cal + Vdc4_cal + Vac1_cal + Vac2_cal + Vac3_cal + Vac4_cal + I1_cal + I2_cal + I3_cal + I4_cal + I5_cal + I6_cal)/1024;
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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("CALIBRATION COMPLETE ");
972 }
973 } else
974 { //Use this when running 40kHz interrupt - resets the compare
975   if (EvaRegs.GPTCONA.bit.T1STAT==1) //last interrupt was an underflow
976     {
977       UF_VSI=1;
978       EvaRegs.T1CMPR = period_vsi-1;
979     }
980   else //last interrupt was a compare match
981     {
982     UF_VSI=0;
983     EvaRegs.T1CMPR = 1;
984 }
985 /*************
986 _SYNCH_CODE()
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   if (first!=0) timer_synch_count--;
996   if(carrier < 320 )
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   // We want it to wobble around the original FSW
1013   adjust_time++;
1014   if (adjust_time==0)
1015     {
1016     period_vsi = PERIOD_VSI + carrier_adjust;
1017     period_2_vsi = period_vsi>>1;
1018     EvaRegs.T1PR = period_vsi;
1019     EvaRegs.T2PR = (period_vsi<<1)-1;
1020     adjust_time=0;
1021     }
1022 }
1023 }
1024 //EMERGENCY STOP - if synchronism lost
1025 if (timer_synch_count>5)
1026   {
1027     detected_faults=1;
1028   }
1029 }
1030 /************
1031 _LOAD_STEP()
1032 *************/
1033 _LOAD_STEP()
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     if ((detected_faults==0)&&(CPLD.EVACOMCON.bit.ENA == 0)) SW_ENABLE();
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

1041 }
1042 else CLEAR_TP11();
1043 prev_load_enable=load_enable;
1044 }**********
1045 _VSI_INT();
1046 }**********/
1048 //This section of code looks after the H-bridge
1049 }**********  
1050 _ADC_VSI();
1051 }**********/
1052 //For the current-regulated VSI, three ADC inputs are needed
1053 // - DC bus voltage for compensation
1054 // - Output AC current
1055 // - BackEMF voltage
1056 Vdc2_fixed = (((AdcRegs.ADCRESULT1-Vdc2_cal)>>4)-ADC_OFFSET)*VDC_ANALOG_GAIN;
1057 Vdc1_fixed = (((AdcRegs.ADCRESULT7-Vdc1_cal)>>4)-ADC_OFFSET)*VDC_ANALOG_GAIN;
1058 Vac1_fixed = (((AdcRegs.ADCRESULT6-Vac1_cal)>>4)-ADC_OFFSET)*VAC_ANALOG_GAIN;
1059 I1_fixed = (((AdcRegs.ADCRESULT4-I1_cal)>>4)-ADC_OFFSET)*I_ANALOG_GAIN;
1060 I2_fixed = (((AdcRegs.ADCRESULT8-I2_cal)>>4)-ADC_OFFSET)*I_ANALOG_GAIN;
1061 Vdc_fixed = (Vdc2_fixed+Vdc1_fixed)>>1;
1062 Vac_fixed = Vac1_fixed;
1063 IACout_fixed= (I1_fixed+I2_fixed)>>1;
1064 }**********
1065 _VSI_MODULATOR();
1066 }**********/
1067 //this piece of code tells you when you are at the peak of the sine wave
1068 if ((16384-sin_val)<=10)
1069 {
1070 if (Iref_mag_fixed_timed != Iref_mag_fixed)
1071 {
1072 Iref_mag_fixed_timed = Iref_mag_fixed;
1073 SET_TP11();
1074 }
1075 else
1076 vsiphase+=PHASE_STEP;
1077 sin_val = sin_table[(vsiphase>>22)|0x00000001];
1078 prev_sin_val = sin_val;
1079 }**********
1080 //Need to scale the modulator reference between 0 to period_2
1081 //mod_target*sin_table*period_2
1082 va = (int16)(((int32)(mod*sin_val*period_2_vsi))>>14);
1083 }**********
1084 //in fixed point - scaled by FIXED_Q
1085 //first, generate reference
1086 Iref_fixed = (long)(Iref_mag_fixed_timed*(long)sin_val)>>>(4+FIXED_Q);//scaled by 2^15 from sin table
1087 }**********
1088 //very fast SPI
1089 //SpiRegs.SPIIRQEN = (DAC_CLK_UPD|DAC_ADDR_A)<<8;
1090 dac_iref = ((Iref_fixed*DAC_SCALE_IREF>>FIXED_Q)+2048);
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 #ifdef CL_VSI
1135 va = (VSI_ctrl_fixed*period_2_vsi)>>FIXED_Q;
1136 #endif
1137 1138 /**************************
1139 _VSI_SWITCHING_TIMES()
1140 **************************/
1141 /* Switching duty cycles */
1142 t_A = va;
1143 t_B = -t_A;
1144 1145 /**************
1146 _VSI_DESAT()
1147 ***************/
1148 /* clamp switch times for pulse deletion and saturation */
1149 1150 // A phase
1151 // A phase
1152 if (t_A > max_time)
1153 {
1154    EvaRegs.CMPR2 = 1;
1155    dac_va = max_time>>2;
1156 }
1157 else if (t_A < (-max_time))
1158 {
1159    if ((V_Asat && UF_VSI) EvaRegs.CMPR2 = period_vsi - 1;
1160    else EvaRegs.CMPR2 = period_vsi - 1;
1161    V_Asat = 1;
1162    dac_va = -max_time>>2;
1163 }
1164 else
1165 {
1166    if (V_Asat && UF_VSI) EvaRegs.CMPR2 = period_vsi - 1;
1167    else EvaRegs.CMPR2 = (Uint16)(period_2_vsi - t_A);
1168    V_Asat = 0;
1169    dac_va = va>>2;
1170 }
1171 // B phase
1172 if (t_B > max_time) EvaRegs.CMPR3 = 1;
1173 else if (t_B < (-max_time))
1174 {
1175    if ((V_Bsat && UF_VSI) EvaRegs.CMPR3 = period_vsi - 1;
1176    else EvaRegs.CMPR3 = period_vsi - 1;
1177    V_Bsat = 1;
1178 }
1179 else
1180 {
1181    if (V_Bsat && UF_VSI) EvaRegs.CMPR3 = period_vsi - 1;
1182    else EvaRegs.CMPR3 = (Uint16)(period_2_vsi - t_B);
1183    V_Bsat = 0;
1184 }
1185 // freeze integrator
1186 if((V_Asat==1)||(V_Bsat==1))
1187 {
1188    VSI_int_fixed = VSI_intnow_fixed;
1189 }
1190 // now write va to the DAC to be picked up on the other side for FF purposes
1191 // FAST DAC WRITE
1192 CLEAR_OC_SPI_EN();
1193 SET_SPI_MASTER();
1194 ENABLE_DAC1();
1195 spi_fail_count = 65535;
1196
253
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

1201 //VERY FAST SPI
1202SpiRegs.SPITXBUF = (DAC_WRn_UPDn|DAC_ADDR_B)<<8;
1203SpiRegs.SPITXBUF = (((((dac_va+2048) << DAC_SHIFT)>>8)&0x00FF)<<8);
1204SpiRegs.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 if (ZX_count++ >= ZX_MAX_COUNT) /* Zero crossing signal lost */
1223 {
1224 ZX_seen = 0;
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 else if (ZX_state == ZX_EST) /* Roughly measure period and average */
1245 {
1246 if (ZX_seen != 0)
1247 {
1248 ZX_seen = 0;
1249 ZX_cycles++;
1250 ZX_sum += ZX_count;
1251 ZX_count = 0; /* Reset counter */
1252 }
1253 if (ZX_cycles >= ZX_CYCLE_AVG)
1254 {
1255 ZX_sum = ZX_sum/ZX_CYCLE_AVG;
1256 ZX_phase_step = ((Uint32)(0xFFFF/ZX_sum))<<16; // Approximate frequency
1257 ZX_sum -= ZX_sum/8; /* Also use for glitch filter */
1258 ZX_vsiphase = ZX_phase_step + zx_offset; /* Within phase_step */
1259 ZX_state = ZX_MISC; /* Calculate ZX_phase_scale first */
1260 }
1261 else if (ZX_state == ZX_SYNC) /* Accurately measure phase error */
1262 {
1263 if (ZX_seen != 0)
1264 {
1265 ZX_seen = 0;
1266 if (ZX_count > ZX_sum) /* Ignore glitches */
1267 {
1268 ZX_count_grab = ZX_count;
1269 ZX_count = 0;
1270 /* Rescale to phase units */
1271 ZX_time_phase = zx_offset + ((int32)ZX_time*ZX_phase_scale)>>5); // Calculate phase error captured time */
1272 ZX_phase_err = ZX_vsiphase - ZX_time_phase;
1273 /* Limit size of phase change */
1274 if (ZX_phase_err > ZX_BIG_ERR)
1275 {
1276 ZX_vsiphase = ZX_BIG_ERR;
1277 /* /* /*WO_UPDa|DAC_ADDR_B<<8; //****************
1208 //_GRID_CONNECTION()
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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_vsi phase += ZX_BIG_ERR;
1287 ZX_err_sum = (ZX_err_sum-ZX_BIG_ERR)>>1;
1288 }
1289 else
1290 {
1291 ZX_vsi phase = ZX_phase_err;
1292 ZX_err_sum = (ZX_err_sum+ZX_phase_err)>>1;
1293 }
1294 // vsi phase = ZX_vsi phase;
1295 //
1296 // vsiphase = ZX_vsiphase;
1297 //
1298 //else if (ZX_state == ZX_FREQ) /* Nudge frequency if needed */
1299 {
1300 // /* If too large, nudge freq (phase_step) */
1301 if (ZX_err_sum > ZX_FREQ_ERR)
1302 {
1303 ZX_phase_step -= 100L;
1304 if (ZX_err_sum > ZX_FREQ_ERR_BIG)
1305 {
1306 ZX_phase_step -= 1000L;
1307 }
1308 }
1309 }
1310 else if (ZX_err_sum < -ZX_FREQ_ERR)
1311 {
1312 ZX_phase_step += 100L;
1313 if (ZX_err_sum < -ZX_FREQ_ERR_BIG)
1314 {
1315 ZX_phase_step += 1000L;
1316 }
1317 }
1318 }
1319 //
1320 // else if (ZX_state == ZX_LOCK) /* Test to see if still in sync */
1321 {
1322 if (ZX_in_sync >= ZX_SYNC_LIMIT)
1323 {
1324 if ((ZX_phase_err>ZX_PHASE_ERR)||(ZX_phase_err<-ZX_PHASE_ERR)) { /* Gone out of sync */
1325 // VSI_DISABLE();
1326 ZX_in_sync = 0;
1327 in_sync = 0;
1328 }
1329 else
1330 {
1331 in_sync = 1;
1332 }
1333 }
1334 // else if ((ZX_phase_err<ZX_PHASE_ERR)&amp;(ZX_phase_err&lt;ZX_PHASE_ERR))
1335 {
1336 /* In sync this cycle */
1337 ZX_in_sync++;
1338 }
1339 //
1340 }
1341 else
1342 {
1343 ZX_in_sync = 0;
1344 }
1345 //
1346 // else if (ZX_state == ZX_MISC)
1347 {
1348 if (ZX_phase_scale = (phase_step<<5)/PERIOD_VSI;
1349 ZX_state = ZX_SYNC;
1350 }
1351 //end grid connection
1352 // Finish the DAC write for the Ref Step before starting the next one.
1353 // this is almost the last thing done, to give as much as time as possible
1354 // for the DAC write to complete (because SPI is slow)
1355 while ((SpiRegs.SPIFFRX.bit.RXFFST < 3)&amp;&amp;spi fail_count>0 )
1356 {
1357 spi fail_count--; /* counter to avoid hang up if SPI fails
1358 // wait for ts to finish

255
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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
1374 #ifdef GRAB_INCLUDE
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(7,EvaRegs.CMPR3);
1384 // GrabStore(8,EvaRegs.CMPR2);
1385 grab_index++;
1386 if (grab_index >= GRAB_LENGTH)
1387 grab_mode = GRAB_STOPPED;
1388 }
1389 #endif
1390
1391 // Reinitialize for next interrupt
1392 AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1; // clear interrupt flag
1393 PieCtrlRegs.PIEACK.all = PIEACK_GROUP1; // Acknowledge interrupt to PIE Group 2
1394 CLEAR_TP10(); // timing bit
1395 /* end isr_timer_CL */
1396
1397 /> * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
1398/**
1399Handles the PDPINT interrupt caused by a gate fault.
1400
1401 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1402 #ifndef BUILD_RAM
1403 #pragma CODE_SECTION(isr_gate_fault, "ramfuncs");
1404 #endif
1405 interrupt void isr_gate_fault(void)
1406 {
1407 is_switching = 0;
1408 VSI_DISABLE();
1409 // SET_TP12();
1410 mod_targ = 0;
1411 detected_faults = FAULT_VSI_PDPINT;
1412 GrabClear();
1413 GrabStart();
1414 GrabRun();
1415 // Acknowledge this interrupt to receive more interrupts from group 1
1416 /* end isr_gate_fault */
1417 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

256
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

void st_vsi_init(void)
{
    mod_ref = 0;
    mod_targ = 0;
    EvaRegs.ACTRA.all = 0x0000;
    vsi_status = VSI_INIT;
    VSI_DISABLE();
    SS_NEXT(vsi_state, st_vsi_stop);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

This is the state where the VSI is stopped. There is no switching. It waits
for a start trigger.

void st_vsi_stop(void)
{
    if (SS_IS_FIRST(vsi_state))
    {
        SS_DONE(vsi_state);
        vsi_counter = 0;
        // VSI_GATE_CHARGE();
        // vsi_status = VSI_GATECHARGE;
        // vsi_status |= VSI_RUNNING|VSI_SETTLED;
    }
    if (detected_faults != 0)
    {
        SS_NEXT(vsi_state, st_vsi_fault);
        return;
    }
    if (is_switching != 0) // start trigger
    {
        SS_NEXT(vsi_state, st_vsi_gate_charge);
    }
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

In this state the VSI gates are enabled and the low side gates held on to
charge the high side gate drivers. The next state is either the ramp state.

void st_vsi_gate_charge(void)
{
    if (SS_IS_FIRST(vsi_state))
    {
        SS_DONE(vsi_state);
        vsi_counter = 0;
        // VSI_GATE_CHARGE();
        // vsi_status = VSI_GATECHARGE;
        // vsi_status |= VSI_RUNNING;
    }
    if (detected_faults != 0)
    {
        SS_NEXT(vsi_state, st_vsi_fault);
        return;
    }
    if (is_switching == 0)
    {
        SS_NEXT(vsi_state, st_vsi_stop);
        return;
    }
}
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

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 
1537 \author A. McIver  
1538  
1539 void st_vsi_ramp(void)  
1540 {  
1541 if (SS_IS_FIRST(vsi_state))  
1542 {  
1543 SS_DONE(vsi_state);  
1544 VSI_ENABLE();  
1545 SW_ENABLE();  
1546 vsi_counter = 0;  
1547 vsi_status = VSI_RAMP;  
1548 }  
1549 if (detected_faults != 0)  
1550 {  
1551 SS_NEXT(vsi_state, st_vsi_fault);  
1552 return;  
1553 }  
1554 // check for stop signal  
1555 if (is_switching == 0)  
1556 {  
1557 SS_NEXT(vsi_state, st_vsi_stop);  
1558 return;  
1559 }  
1560 else  
1561 {  
1562 SS_NEXT(vsi_state, st_vsi_run);  
1563 return;  
1564 }  
1565 }  
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 
1577 void st_vsi_run(void)  
1578 {  
1579 if (SS_IS_FIRST(vsi_state))  
1580 {  
1581 SS_DONE(vsi_state);  
1582 vsi_status = VSI_RUNNING;  
1583 }  
1584 if (detected_faults != 0)  
1585 {  
1586 SS_NEXT(vsi_state, st_vsi_fault);  
1587 return;  
1588 }  
1589 // check for stop signal  
1590 if (is_switching == 0)  
1591 {  
1592 SS_NEXT(vsi_state, st_vsi_stop);  
1593 return;  
1594 }  
1595 // check for changes in reference  
1596 if (mod_targ != mod_ref)  
1597 {  
1598 vsi_status &= ~VSI_SETTLED;  
1599  
258
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

1601 SS_NEXT(vsi_state, st_vsi_ramp);
1602 }
1603 /* end st_vsi_run */
1604
1605 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1606 /* void st_vsi_fault(void)
1607 Parameters: none
1608 Returns: nothing
1609 Description: Delays for a while after faults are cleared.
1610 Notes:
1611 History:
1612 03/11/05 AM - initial creation
1613 04/03/08 AM - set vsi_status with fault bit
1614 28/04/08 AM - added event reporting
1615 */
1616 void st_vsi_fault(void)
1617 {
1618     if (SS_IS_FIRST(vsi_state))
1619     {
1620         SS_DONE(vsi_state);
1621         VSI_DISABLE();
1622         SW_DISABLE();
1623         vsi_counter = 0;
1624         vsi_status = VSI_FAULT;
1625         // vsi_status &= ~(VSI_RUNNING|VSI_SETTLED);
1626         putsxx(detected_faults);
1627         puts_COM1("->VSI faults
");
1628     }
1629     if (detected_faults == 0)
1630         vsi_counter++;
1631     else
1632         vsi_counter = 0;
1633     if (vsi_counter > 100)
1634         {
1635             // vsi_status &= ~VSI_FAULT;
1636             SS_NEXT(vsi_state, st_vsi_stop);
1637         }
1638     /* end st_vsi_fault */
1639 }
1640
1641 /* =========================================================================
1642 __Local_Functions()
1643 ============================================================================= */
1644 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1645 /**
1646 This function is called every fundamental period to perform the RMS
1647 calculations and scale the analog quantities to Volts and Amps for use in the
1648 background.
1649 
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 12/10/07 AM - derived from IR25kVA:vsi:adc_scale
1656 21/08/08 AM - added VSI DC offset compensation
1657 12/09/08 AM - added stop_count and moved to floating point data
1658 */
1659 //void scale_adc_rms(void)
1660 {//
1661     double val,
1662     temp;
1663 // calculate A0 RMS quantity
1664 // temp = (double)adc_int.A0.dc_sum_bak/(double)adc_int.count_rms_bak;
1665 // val = (double)adc_int.A0.rms_sum_bak*(double)(1<<ADC_RMS_PS)
1666 // / (double)adc_int.count_rms_bak - temp*temp;
1667 // if (val < 0.0) val = 0.0;
1668 // adc_int.A0.real = ADC_REAL_SC * sqrt(val);
1669 //} /* end scale_adc_rms */
1670
1671 /* =========================================================================
1672 __Local_Functions()
1673 ============================================================================= */
1674 /**
1675 This function is called every ADC_DC_TIME to perform the DC calculations and
1676 scale the analog quantities to Volts and Amps for use in the background.
1677 */
1678 \author A.McIver
1679 \par History:
APPENDIX A. SIMULATION & EXPERIMENTAL CODE

1681 \li 12/10/07 AM - derived from IR25kVA:vsi:adc_scale
1682 /*
1683 void scale_adc_dc(void)
1684 //
1685 // void scale_adc_dc(void)
1686 //
1687 // adc_int.A0.real = (double)adc_int.A0.dc_sum_bak/(double)ADC_COUNT_DC;
1688 // adc_int.A2.real = (double)adc_int.A2.dc_sum_bak/(double)ADC_COUNT_DC;
1689 // adc_int.A4.real = (double)adc_int.A4.dc_sum_bak/(double)ADC_COUNT_DC;
1690 // adc_int.A6.real = (double)adc_int.A6.dc_sum_bak/(double)ADC_COUNT_DC;
1691 //
1692 // calibrate B0 DC quantity
1693 // adc_int.B0.real = ADC_REAL_SC * val;
1694 //
1695 //} /* end scale_adc_dc */
1696 1697 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
1698 /**
1699 Calibrates the adc for gain and offset using the reference inputs.
1700 See spra989a.pdf for calibration details
1701 //
1702 \par History:
1703 07/10/05 AM - initial creation
1704 */
1705 void calibrate_adc(void)
1706 {
1707 // char
1708 // str[60];
1709 1710 yHA = (double)adc_int.yHA.dc_sum_bak/(double)ADC_COUNT_CAL;
1711 yLA = (double)adc_int.yLA.dc_sum_bak/(double)ADC_COUNT_CAL;
1712 yHB = (double)adc_int.yHB.dc_sum_bak/(double)ADC_COUNT_CAL;
1713 yLB = (double)adc_int.yLB.dc_sum_bak/(double)ADC_COUNT_CAL;
1714 1715 cal_gain_A = (xH - xL)/(yHA - yLA);
1716 cal_offset_A = yLA * cal_gain_A - xL;
1717 1718 cal_gain_B = (xH - xL)/(yHB - yLB);
1719 cal_offset_B = yLB * cal_gain_B - xL;
1720 1721 // sanity check on gains
1722 if ( ( (cal_gain_A > 0.94) && (cal_gain_A < 1.05) )
1723 && ( (cal_gain_B > 0.94) && (cal_gain_B < 1.05) )
1724 && ( (cal_offset_A > -80.0) && (cal_offset_A < 80.0) )
1725 && ( (cal_offset_B > -80.0) && (cal_offset_B < 80.0) ) )
1726 { 1727 cal_gainA = (int16)(cal_gain_A*(double)(1<<14));
1728 cal_gainB = (int16)(cal_gain_B*(double)(1<<14));
1729 cal_offsetA = (int16)cal_offset_A;
1730 cal_offsetB = (int16)cal_offset_B;
1731 } 1732 1733 if(vsi_state.f == st_vsi_init){
1734 puts_COM1("INIT ");
1735 } 1736 else if(vsi_state.f == st_vsi_stop){
1737 puts_COM1("STOP ");
1738 } 1739 else if(vsi_state.f == st_vsi_gate_charge){
1740 puts_COM1("GATE ");
1741 } 1742 else if(vsi_state.f == st_vsi_ramp){
1743 puts_COM1("RAMP ");
1744 } 1745 else if(vsi_state.f == st_vsi_run){
1746 puts_COM1("RUN ");
1747 } 1748 else if(vsi_state.f == st_vsi_fault){
1749 puts_COM1("FAULT ");
1750 }
puts_COM1("FAU ");
}
References


REFERENCES


REFERENCES


REFERENCES

in Proc. 2nd Power Electronics, Drive Systems and Technologies Conference (PEDSTC), 2011, pp. 92--97.


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


