Distributed Current-Regulated Vector Control of High-Performance Modular PM Synchronous Machine Drives

by

Adam Shea

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

University of Wisconsin–Madison

2018

Date of oral examination: June 14, 2018

The dissertation is approved by the following members of the Final Oral Committee:
T. M. Jahns, Professor, Electrical Engineering
R. D. Lorenz, Professor, Mechanical Engineering
Giri Venkataramanan, Professor, Electrical Engineering
Bulent Sarlioglu, Professor, Electrical Engineering
Bernard Lesieutre, Professor, Electrical Engineering

© Copyright by Adam Shea 2018

All Rights Reserved

ABSTRACT

The integrated modular motor drive (IMMD) architecture is based on the concept of a physically integrated motor drive consisting of n identical phase-drive units. Each of these phase-drive units includes a segmented machine stator pole, its winding, and a single-phase inverter to excite the winding. This IMMD architecture opens intriguing opportunities for improved fault tolerance because each of the n phase-drive modules is equipped with its own controller. The objective of this research program has been to design and implement high-performance current regulation and torque control for the IMMD using a modular, distributed control architecture without requiring a dedicated master controller or high-speed synchronized communications. The distributed control architecture poses special challenges for maintaining coordination of the n individual phase controllers since the controllers no longer have the same natural symmetry as conventional motor drives to help limit the effects of asymmetries among the phase modules or machine phases. A combination of closed-form analysis and digital simulation has been used to identify and characterize the most promising distributed control algorithm. This analytical work has shown that it is possible to achieve high-performance current regulation in three-phase machines with floating-wye or delta winding connections using only two measured currents supplied to each phase-drive unit. A closed-loop feedback technique has been developed that uses the neutral point voltage to correct for system asymmetry and provide the required compliance to the over-constrained distributed control problem; this results in stable operation without sacrificing the performance of the complex vector current regulator. A laboratory demonstrator version of the phase-module controller has been used to

implement and verify the proposed distributed control algorithm applied to an experimental 3-phase PM motor drive. The distributed control algorithm has been extended to continue controlling the machine torque after open-circuit faults. By adding proportional feedforward functions to the current command generation, the distributed controller can operate post-fault with no change in tuning or structure. The distributed controller has been shown to be compatible with well-known methods for adjusting the current commands to reduce the torque ripple in a faulted motor drive or to restore the original pre-fault average torque value.

ACKNOWLEDGMENTS

This work was supported by the generous funding and facilites of the Wisconsin Electric Machines and Power Elecronics Consortium. This thesis also would never have been possible without the help and support of many people.

Thanks go out to so many of the WEMPEC students, scholoars, staff, and professors who have been invalulable for insights, support, and assistance.

A special thanks to my advisor and mentor Professor Tom Jahns who guided me through the long road to my dissertation and spent many long nights helping me prepare this document.

Finally, none of this would be possible without the unending support of my wife Leslie. Thank you for supporting me through all the late nights, ruined breaks, and consumed weekends. I couldn't have done it without you.

Finally, this dissertation is dedicated to my daughter Minerva and my late grandfather Jack. I'm proud to be the third generation to graduate from UW-Madison and hope Mia can continue the tradition.

TABLE OF CONTENTS

Page

ABSTRACT i

LIST OF FIGURES vii

1 Introduction 1

1.1 Background and Problem Statement 1
1.1.1 Integrated Motor Drives 1
1.1.2 Modular Motor Drives 3
1.1.3 Integrated Modular Motor Drives 3
1.2 Problem Statement and Approach 4
1.3 PhD Research Program Objectives 6
1.4 Document Organization 7

2 State-of-the-Art Review 9

2.1 Integrated and Modular Drives 9
2.2 Modular High-Power Drives 13
2.3 High Reliability Drives and Converters 15
2.4 Sensor and Machine Faults 17
2.5 Open-End Winding Machines 25
2.6 Modular High-Power Machines 27
2.7 Hot-plugging and DC-Link Isolation 29
2.8 Distributed System Synchronization 29
2.9 Fault Modeling and Fault Tolerance 30
2.10 Conclusion 31

3 Three-Phase Drives with Monolithic Controller 34

3.1 Terminology and Symbols 34
3.2 Machine Model 37
3.2.1 Coordinate Transforms 39
3.3 Three-Phase Drive Model with Ideal Sensors 41
3.3.1 Simulation 41

Page

3.3.2 State Space Model . . . 49
3.4 Current Sensor Errors . . . 54
3.5 Conclusions . . . 60

4 Machine Drives with Distributed Controllers . . . 61
4.0.1 Introduction . . . 61

4.1 Independent Per-Phase Wye-Connected Drives . . . 61
4.1.1 Sensing Needs . . . 63
4.1.2 Speed Control . . . 64
4.1.3 Simulation . . . 64
4.1.4 Single-Phase RL Load Current Regulation Example . . . 67
4.1.5 Polyphase State-Space Model . . . 74
4.2 Neutral Point Voltage Control . . . 80
4.3 Observer-Based Stabilization . . . 87
4.4 Open Winding Arrangement . . . 88
4.5 PWM Synchronization . . . 95
4.6 Conclusions . . . 98

5 High Phase Order Generalization . . . 100

5.1 Coordinate Transforms and Terminology . . . 100
5.2 High Phase-Order Machine Model . . . 105
5.3 Modular Phase Drives . . . 107
5.3.1 Simulation . . . 108
5.4 Faulted Mode Operation . . . 111
5.5 Conclusions . . . 121

6 Experimental Verification . . . 123

6.1 Compact Drive Module . . . 123
6.1.1 Power Supply . . . 125
6.1.2 Sensors . . . 127
6.1.3 Communications . . . 129
6.2 Reconfigurable Polyphase Motor . . . 130
6.3 Firmware Description . . . 136
6.3.1 RTOS . . . 136
6.3.2 Bootloader . . . 137
6.3.3 Control loop structure . . . 138
6.3.4 Communications . . . 140

Appendix

Page

6.3.5 PC Support Software 141
6.4 Experimental Results 142
6.4.1 3-Phase Distributed Drive with Neutral Voltage Sensing 142
6.5 Conclusions 146

7 Contributions, Conclusions, and Future Work 150

7.1 Key Results Summary 150
7.2 Contributions 151
7.3 Future Work 155
7.4 Minor Conclusions 157

APPENDIX Controller Schematics 159

APPENDIX Bootloader Code 168

APPENDIX Recommended Controller Software Improvement Tasks 236

LIST OF FIGURES

Figure Page

1.1 Diagram of distributed modular motor controller with measured signals shown. 4

2.1 Rendered views of 5-phase SMC modular motor drive. 10

2.2 Six-phase distributed control IMMD mounted to end of machine with integrated cooling. 12

2.3 Six-phase distributed control IMMD with drive module components labeled. 13

2.4 Five-phase SRM machine with modular drive and semi-modularized control[8]. 14

2.5 Serial bus and controller layout for semi-modular 5-phase SRM machine control [8]. . 15

2.6 2hp IMMD using series-stacked 3-phase GaN inverters and an induction machine [7]. 16

2.7 Figure 7 from [16] showing Fictive-Axis control of a single phase inverter. 18

2.8 Relative gain error estimation using three positive sequence current measurements from [22]. 21

2.9 Continuous torque star-delta starter using open-end winding machine from 1944 [36]. 26

2.10 Open-end winding machine drive with common mode choke to suppress zero-sequence currents [38]. 27

2.11 Modular Flux-Switching Permanent-Magnet machine for a large wind turbine. 28

2.12 Optimal current waveforms for a star-connected five-phase PM machine with (top) phase a open, (middle) phases a and c open, and (bottom) phases a and b open[62]. . . 32

3.1 Overall System Block Diagram 39

3.2 Axes and sign conventions for coordinate frame transforms. 40

Figure Page

3.3 Schematic of a standard monolithic drive controlling a three-phase machine. 41

3.4 Block diagram for 3-phase PM Machine in the synchronous reference frame. 42

3.5 Block diagram of the complex vector current regulator used in this research. The operating frequency input into this model must be the electrical rotation rate of the reference frame it is applied in as this speed is used to decouple speed voltages. 44

3.6 Simulation of three-phase vector controlled drive with ideal current sensing. Torque and machine terminal voltages are shown. 45

3.7 Simulation of three-phase vector controlled drive with ideal current sensing. Torque, terminal currents, and controller command voltages are shown. 46

3.8 Simulation of three-phase vector controlled drive with a ten percent gain error in one current sensor. Torque and machine terminal voltages are shown. 47

3.9 Simulation of three-phase vector controlled drive with a ten percent gain error in one current sensor. Torque, terminal currents, and controller command voltages are shown. 48

3.10 Synchronous reference frame current sensor response as a function of current vector angle with 1% gain error in one sensor. 59

4.1 Diagram showing a three phase drive broken up into half-bridge distributed segments. 62

4.2 Schematic of a distributed drive controlling a three-phase machine with two current sensors per phase controller. 63

4.3 Distributed three-phase drive with ten percent current sensor error in one phase operating without any stabilization. Torque and machine terminal voltages are shown. 65

4.4 Distributed three-phase drive with ten percent current sensor error in one phase operating without any stabilization. Torque, terminal currents, and controller command voltages are shown. 66

4.5 Distributed three-phase drive with voltage limits and ten percent current sensor error in one phase operating without any stabilization. Torque and machine terminal voltages are shown. 68

Appendix
Figure | Page

4.6 Distributed three-phase drive with voltage limits and ten percent current sensor error in one phase operating without any stabilization. Torque, terminal currents, and controller command voltages are shown. . . . 69

4.7 Close-up of phase voltages during 5 Nm distributed drive operation with phase voltage limits. . . . 70

4.8 Circuit model for single phase R-L load driven by two voltage output current controllers with quantities labeled. . . . 70

4.9 Root locus diagram for two independent current controllers controlling a 1mH, 1ohm R-L load. The controller time constant and neutral point gain is varied to show tuning effects. . . . 75

4.10 Control block diagram showing voltage command error control loop. . . . 80

4.11 Three-phase current regulator phase voltage loci in synchronous reference frame with 0.1% current sensor error showing increasing common mode offset. . . . 81

4.12 Three-phase current regulator phase voltage loci in synchronous reference frame with 0.1% current sensor error showing increasing controlled common mode offset when a Neutral voltage controller is employed. . . . 82

4.13 Distributed drive phase controller block diagram with neutral-point based compensation. 84

4.14 Distributed three-phase drive with ten percent current sensor error in one phase operating with Neutral-Point based linear stabilization. Torque and machine terminal voltages are shown. . . . 85

4.15 Distributed three-phase drive with ten percent current sensor error in one phase operating with Neutral-Point based linear stabilization. Torque, terminal currents, and controller command voltages are shown. . . . 86

4.16 Simulation of a three-phase open-end winding distributed drive with asymmetries where each drive measures all three currents. . . . 89

4.17 Simulation controller states of a three-phase open-end winding distributed drive with asymmetries where each drive measures all three currents. . . . 90

4.18 Simulation of an open winding distributed drive with asymmetries in which each H-bridge drive measures its phase winding current. . . . 91

Appendix
Figure | Page

4.19 Simulation of an open winding distributed drive with asymmetries in which each H-bridge drive measures its phase winding current. 92

4.20 Simulation of an open-end winding distributed drive with asymmetries where each drive measures only two currents. 93

4.21 Simulation controller states of an open-end winding distributed drive with asymmetries where each drive measures only two currents. 94

4.22 Simulated waveforms showing a three-phase inverter driving an R-L load with synchronized PWM carriers (top two plots) and with non-synchronized PWM carriers (bottom two plots). 97

5.1 Axes and sign conventions for logical and physical phase transforms in a three-phase system. 101

5.2 Axes and sign conventions for logical and physical phases transforms in a six-winding system with sixty degrees between windings. 102

5.3 Axes and sign conventions for logical and physical phases transforms in a five phase system. 103

5.4 Schematic of a distributed drive controlling a five-phase machine. 109

5.5 Block diagram implementing equation 5.13 for arbitrary phase orders. 110

5.6 Simulink block diagram implementing neutral-point stabilized vector control for arbitrary phase orders with programmable number of input current sensing channels. . . . 111

5.7 Five-phase distributed drive operating without correction with a ten percent current sensor gain error and only two phases current feedback to each distributed drive. Torque and machine terminal voltages are shown. 112

5.8 Five-phase distributed drive operating without correction with a ten percent current sensor gain error and only two phases current feedback to each distributed drive. Torque, terminal currents, and controller command voltages are shown. 113

5.9 Five-phase distributed drive operating with neutral point feedback correction with a ten percent current sensor gain error and only two phases current feedback to each distributed drive. Torque and machine terminal voltages are shown. 114

Appendix
Figure | Page

5.10 Five-phase distributed drive operating with neutral point feedback correction with a ten percent current sensor gain error and only two phases current feedback to each distributed drive. Torque, terminal currents, and controller command voltages are shown. 115

5.11 Simulated current waveforms for a five-phase machine showing: 1) healthy operation ("Healthy"); 2) one open-circuited phase winding ("Faulted"), 3) a redistribution of the excitation current vector phase angles (but not amplitudes) in the remaining four healthy phases ("Smoothed"); and 4) an adjustment of the current amplitudes and angles in the remaining healthy phases in order to restore the average torque to its pre-fault value ("Equal Torque"). 117

5.12 Instantaneous torque waveforms for a five-phase machine for each of the current excitation conditions from Figure 5.11. 118

5.13 Block diagram implementing neutral-point stabilized vector control for arbitrary phase orders with a programmable number of input current sensing channels. This model includes both the post-fault current and voltage command injection to operate after a phase fault. 119

5.14 Voltage waveforms for the smoothed torque condition including the neutral point voltage offset for the remaining healthy phases shown as a dashed line in the upper figure. 120

6.1 Fairchild FSBB30CH60CT integrated power module. 124

6.2 View of compact drive module. 125

6.3 Power supply arrangement for the converter. 126

6.4 Experimental reconfigurable machine with open terminal box. 131

6.5 Experimental reconfigurable machine winding layout with 9-phase connection shown [67]. 132

6.6 Back-EMF waveform (open-circuit voltage) for a single coil in the experimental waveform showing fundamental component. 135

6.7 Back-EMF waveform (open-circuit voltage) for three adjacent coils in series as used in the tests described in the next chapter. 136

Appendix
Figure | Page

6.8 Unfiltered neutral point voltage waveform showing distributed controller operation with PWM for the full duration of the test (top). The lower waveform is a time zoomed axis showing the PWM steps. Both axes are enumerated in seconds with the upper being 2.6 seconds long and the lower being 500 microseconds long. 144

6.9 Measured phase currents and neutral point voltage waveforms (filtered) demonstrating successful distributed controller operation. 145

6.10 Distributed three-phase drive approximating the experimental setup with Neutral-Point based linear stabilization. Torque and machine terminal voltages are shown. 147

6.11 Distributed three-phase drive approximating the experimental setup with Neutral-Point based linear stabilization. Torque, terminal currents, and controller command voltages are shown. 148

Chapter 1

Introduction

1.1 Background and Problem Statement

In a conventional motor drive system, a separate machine and drive are located in their own packages connected by a cable. Often the drive is located a long distance from from the machine in a dedicated control cabinet. This traditional configuration can be attributed to several factors. First, the power electronics has typically been as large or larger than the machine it controls which makes separate packaging more practical. Second, power electronics is limited to a significantly lower maximum temperature of less than 150°C while machines are routinely designed for maximum temperatures up to 200°C. Finally, power electronics has historically suffered from a reputation for significantly lower reliability than machines. This has raised concerns among both drive manufacturers and their customers about the risks of premature failures in the power electronics that could make it necessary to scrap both the machine and drive if the two were tightly integrated into the same structure.

1.1.1 Integrated Motor Drives

The integrated motor drives (IMD) is not a new idea. Early partial integration attempts even predate solid-state power electronics, such as the Monarch Model 10EE lathe that used a built-in thyratron-based drive with a dc motor to power its spindle. The IMD concept has experienced

significant success in several niche markets for many years. The ability to combine a motor and drive into a single packaged unit significantly eases the integration of the motor system into a new products such as washing machines and other white goods where the motor drive system is replacing an older fixed-speed prime mover. This integrated packaging technique is also being widely used to allow motor manufacturers to meet new efficiency requirements for standard motors since the variable-speed drive enables more efficient operation at lower speeds with partial loads. Integrated drives also make it possible for small synchronous motors to replace shaded-pole induction motors in low-power applications without significantly changing the rest of the product design. Integrated Motor Drives also have provided reduced cost, volume, and mass as the combined packaging eliminates several of the EMI and reliability drawbacks and complicated shielding of the multi-phase cable connecting the drive and machine in electric vehicles [1]. The EMI benefits are growing in importance as faster-switching 4th generation silicon IGBTs and wide-bandgap devices are introduced into motor drives, since the high $\frac{dV}{dt}$ necessitates either extensive filtering or tight shielding of the drive output and motor with very short phase leads.

Today Integrated Motor Drives are drawing increasing interest for a few reasons. Wide-bandgap power switching devices have the ability to operate at significantly higher temperature than Silicon with devices capable of $200°$C junction temperature operation already in production [2]. This increased temperature capability makes it significantly more practical to integrate the drive and machine without compromising the machine's design due to lower temperature limits. Wide-bandgap devices also allow for faster switching which reduces the required size of power converter passives. Finally, as computation and general electronics packaging have decreased in cost while increasing in integration, the non-power components of the motor drive have become a smaller fraction of the size and cost of a motor drive system.

1.1.2 Modular Motor Drives

Modularity in motor drives has also been recognized to be a desirable feature for many years. In large multi-MW motor drives, the high power ratings have made it more practical and often necessary to break the drive power electronics into sections with ratings within the limits of the available power electronic technology. Furthermore, the sheer size of the conductors and cooling systems required by some large drives forces modularity in order to be able to ship and install the drive system. Opportunities for improved fault tolerance have long been recognized in applications where the increased cost and complexity can be tolerated [3], [4]. However, even in these fault-tolerant applications, the system modularity has been limited to the power electronics. The controls have remained a single monolithic unit.

1.1.3 Integrated Modular Motor Drives

Combining integration and modularity, the Integrated Modular Motor Drive (IMMD) concept was introduced about 15 years ago, looking ahead to a time when both the controls and power electronics would become sufficiently inexpensive and environmentally robust to combine with individual phases of the machine while not compromising system performance. This concept also incorporates separate modular controllers for each modular phase drive unit in order to provide increased modularity and fault tolerance. Early developments in this area, like the larger modular motor drives, focused primarily on the modularity of the power electronics, leaving the controls centralized [5]–[7].

While some research has been done to allow distributed control of motor drives, it has taken a lock-step approach which requires each drive module to agree on system state of the speed of the fastest control loop in the system (typically the machine phase current). While this approach technically can do away with the central controller, it requires communications bandwidth in the system scaling with the factorial of the number of modular controllers. This lock-step control also

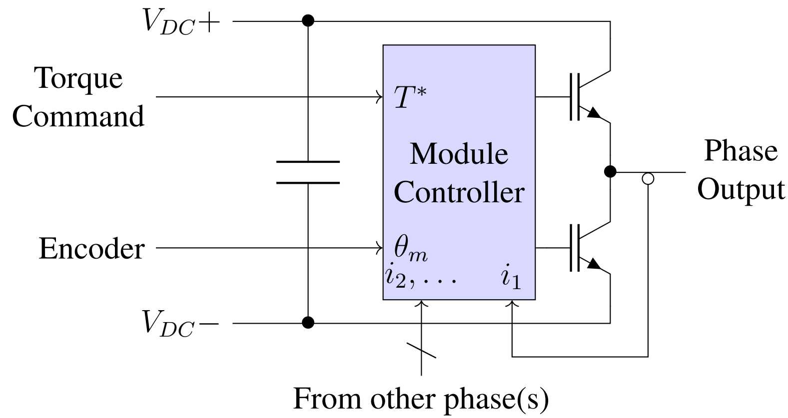


Figure 1.1 Diagram of distributed modular motor controller with measured signals shown.

complicates efforts to introduce fault tolerance since the system depends on prompt communications from all parts [8].

1.2 Problem Statement and Approach

This research seeks to develop a distributed current-regulated vector control architecture for a modular PM motor drive that requires minimal communication between phase modules during normal operation while still providing high-performance machine control. The baseline high-performance control algorithm is assumed to be synchronous-frame complex vector field-oriented control that is designed to provide significant torque control bandwidth. Since fault tolerance and serviceability are key attributes of the integrated modular motor drive, each controller must run identical software.

Each modular phase-drive unit in this IMD is assumed to share the same DC power supply as well as the same torque / current command input. In the baseline configuration, each phase-drive unit excites a single phase of the machine through a half-bridge PWM phase leg as shown in Figure 1.1. Each phase-drive module is assumed to sense its own output current and at least one other current in order to determine the machine's stator current vector in the fundamental reference

frame. It is assumed that each phase-drive current regulator does not sense all of the machine phase currents since this approach becomes increasingly impractical as the number of phases increases in a high-phase-order implementation. Position feedback is assumed to be provided through a common encoder signal. In future work, this requirement could be relaxed by introducing a self-sensing approach, but that is beyond the scope of this research program.

In order to have the overall system respond in the same way as a monolithic vector current controller, each modular controller must run its own vector current control algorithm. This duplication of the vector current controller causes the control system to be overconstrained as the machine still only has one current state per phase, less one if it has a floating neutral point or a delta connection. This suggests that a solution can be found by using an open-end winding machine with each controller having an H-bridge output controlling one phase. This works for most sensing configurations but introduces its own complications in that the zero sequence path is no longer isolated and this added impedance interacts with the neutral point controller requiring higher neutral voltage feedback gain. The results presented in this document focus on the case of the floating neutral using the half-bridge distributed modular motor controller shown in Figure 1.1.

The duplicated controller also introduces new states formed by the difference between the modules' internal control states. These states are fed by any asymmetry between phases including but not limited to winding asymmetry, current sensor errors, shaft misalignment, cabling asymmetry, and slot asymmetry. Attention in this research is focused on the impact of current sensor errors, but it should be understood that this error source is a surrogate that is intended to reflect the impact of other types of drive system asymmetry as well.

The work that follows investigates the effects of this asymmetry in a modular motor drive, and then presents a feedback-based approach to help mitigate these effects.

1.3 PhD Research Program Objectives

The overarching objective of this PhD research program is to address the major challenges associated with implementing distributed current-regulated vector control of high-performance modular PM synchronous machine drives. More specifically, this research aims to accomplish the following:

1. *Develop a current-regulated vector control strategy for a modular motor drive in which each module controls a subset of machine phases.* This control strategy must operate without requiring centralized coordination since the central controller forms a single point of failure which is unwanted in a fault-tolerant design. This controller design must also aim to limit the required sensors and communications since these are major cost drivers as well as sources of additional faults.

2. *Provide a theoretical grounding for the distributed motor control algorithm developed while addressing the first objective in order to provide insight into controller tuning and sensing requirements*. This analytical model will also be used to provide a quantitative performance comparison between well-known standard drives with centralized controllers and this new distributed modular drive.

3. *Develop a flexible hardware platform for experimental verification of the current-regulated vector control techniques developed for distributed PM motor drives.* This hardware platform must provide flexible sensor configuration and complete software flexibility in order to verify the theoretical results developed while addressing the previous objective. Furthermore, due to the large number of relevant control parameters in higher-phase order systems, this hardware platform must provide high-speed data acquisition of both physical and controller states in order to facilitate reporting results.

4. *Verify the distributed motor control strategy in a variety of machine and drive arrangements including 3-phase machines, higher-phase-order machines, arrangements with distributed control modules controlling unbalanced phase sets, and faulted-mode operation.*

1.4 Document Organization

The remainder of this document is organized into seven chapters:

- Chapter 2 presents a review of the state-of-the-art of integrated and modular motor drives as well as relevant literature that provide the foundation for this research program. Attention is focused on previous work on modular and fault-tolerant machine controls as well as control techniques to cope with sensor non-idealities, sensor failures, and cooperative control of power converters.

- Chapter 3 presents a baseline model of a standard monolithic three-phase motor and drive to serve as a foundation for the remainder of this document. Terminology and sign conventions are discussed and the non-ideal sensor model used in the remainder of this work is introduced. The effects of non-ideal sensors are discussed and set as a benchmark for standard drive performance for the development of a distributed system.

- Chapter 4 discusses how to break the three-phase drive into three identical, independent drive sections, each controlling a single phase. Simulation results are presented showing both identical current performance to the standard drive and unstable zero-sequence voltage response due to the additional control states. A state-space model for this system in presented in addition to an explanation of the introduced coupling into the zero-sequence of any mismatch between phase controllers. A neutral-voltage feedback method is presented to stabilize the zero-sequence voltage.

- Chapter 5 extends the distributed drive architecture from Chapter 4 to higher-phase-order machines. Simulation results are presented for a 60-degree six-phase machine. Coupling into higher-order spatial frames is discussed. An observer-based alternative to neutral-voltage sensing is presented but is not perfected due to parameter sensitivity.

- Chapter 6 describes the hardware developed to provide experimental verification of the theoretical work and simulations presented in the preceding chapters. Design goals are discussed and the features of the custom-built compact motor drive modules are explained. The experimental machine is described and winding plans for various phase orders are enumerated. Experimental results are then presented showing the operation of a distributed current regulator in hardware.

- Chapter 7 provides a summary of the key results in this dissertation and identifies the new technical contributions of this work that provide the basis for this PhD research program. Suggested areas of future work are also presented.

Chapter 2

State-of-the-Art Review

This state-of-the-art review is presented in three sections. First, a summary of previous work on integrated and modular motor drives is presented of which this work is a direct descendant. Then, an overview is given of various work on fault tolerance, sensor error handling, and asymmetric drives which provided inspiration for the development of the distributed drive architecture that is the subject of this research program. Finally, a few application areas are covered in which a distributed drive could make an impact due to its unique properties.

2.1 Integrated and Modular Drives

An early work on integrated modular drives was done by Prof. Jahns in his PhD work [9]. This work focused primarily on how to maintain torque production under faulted conditions in high-phase-order drives. As power electronic drives at the time were primarily naturally-commutated inverters, control design was limited to adjusting the timing of firing angles. The key insights which apply to the work presented here are, first, that in a modular drive the angle between phase voltages is not limited to match the physical angle of the stator coils in the drive. Second, in order to maintain smooth torque in the event of a missing machine phase ,the current vector contribution that would have been provided by that phase must be distributed amongst the remaining healthy phases. For a machine drive with no current or voltage limits, the phase currents can be distributed in an infinite variety of ways. However, given realistic system limitations, one might choose instead to

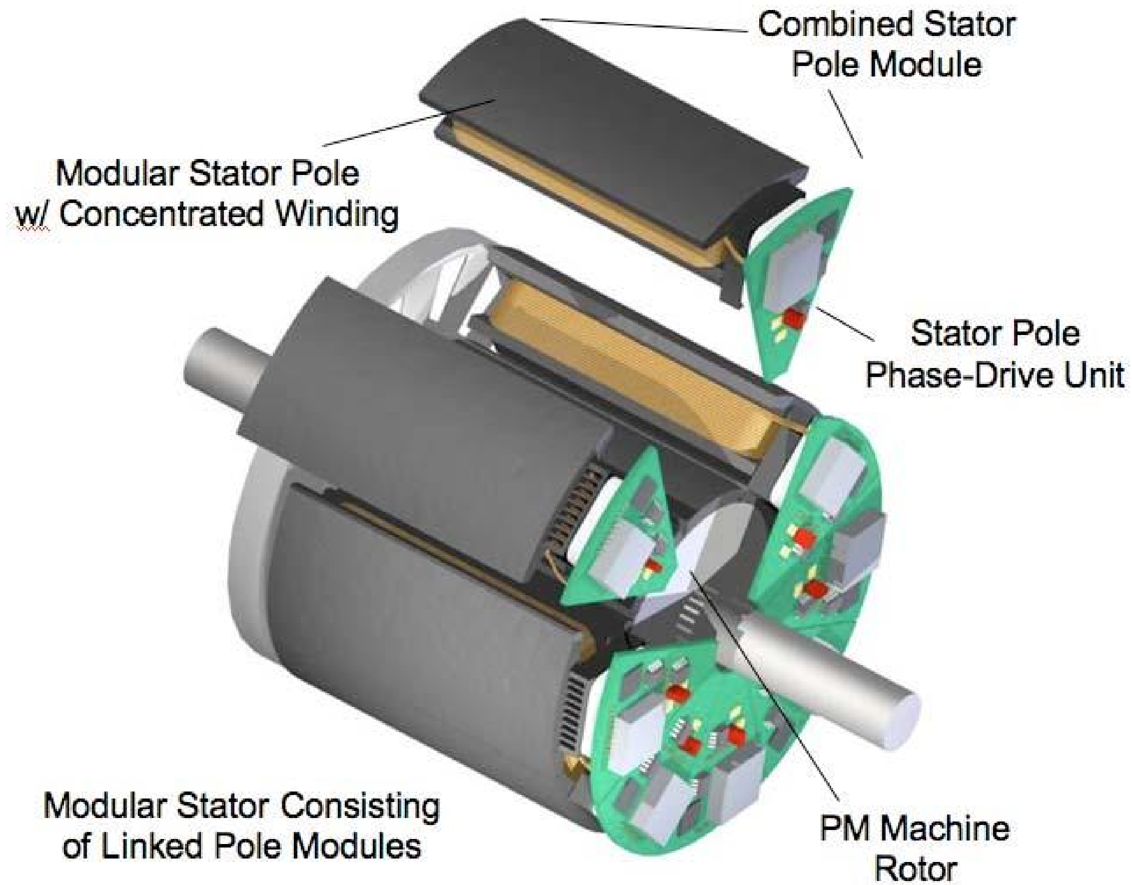


Figure 2.1 Rendered views of 5-phase SMC modular motor drive.

impose a constraint that the healthy phase current magnitudes are equal. This can be accomplished by choosing a weighting for the decomposition proportional to the sine of the angle between the healthy phases and the failed phase. The result of this weighting is that the target current vectors for the remaining healthy phases will again be evenly distributed around the machine.

In 2007 at WEMPEC [10], Nate Brown and others demonstrated a fully modular machine consisting of tooth-wound pole pieces constructed using soft magnetic composite (SMC) material and coupled to a modularized inverter.

The entire machine and drive were air-cooled by drawing air first through the inverter poles then through the machine airgap and slots. The tightly integrated drive on this machine included modular

sections consisting of an IGBT half-bridge, bootstrap gate drive, and snubber. These sections were coupled to a centralized electrolytic DC-link and passive rectifier. The controls for this machine were fully centralized with current sensing of all five phases using LEM hall-effect closed-loop sensors. A single DSP mounted off-board digitized the phase currents and provided all ten PWM gating signals for the drive. Position feedback was provided by an incremental encoder mounted to the shaft amidst the phase modules.

This modular inverter also demonstrated the difficulties in modular drive interconnect as its phase and power connections developed intermittent faults after a relatively short operating life despite being directly soldered using flexible wire without connectors.

An early predecessor of this document's work was developed several years ago with a particular focus on drive integration [11]. In that work, the machine was a 12-slot, 10-pole surface permanent magnet machine designed to 1/3rd the stack length required for a US Drive 55 kW (peak) tracton machine specification. The machine was wired into six equally distributed phases (60 electrical degrees between phases) with each phase consisting of two adjacent teeth. The drive was mounted axially to the end of the machine and consisted of six drive modules, each controlling a single phase of the machine as shown in Figure 2.1. Each phase module contained a 16-bit DSP, open-loop hall effect current sensing, gate drives, half-bridge power stage, and a polypropylene film DC-Link capacitor as labeled in Figure 2.1. The drive was cooled by per-phase waterblocks plumbed in series upstream of the machine's water jacket. The DC-link was interconnected with both star and ring connections along with several decades of capacitance spaced successively closer to the switching phase leg in order to attenuate switching harmonics without needing large capacitors.

While this IMMD prototype was well integrated both mechanically and electrically, the interconnect was not sufficiently robust to the vibration environment close to the machine to allow for extended testing. Furthermore, design choices regarding how sensing was arranged limited the control to V/Hz excitation only.

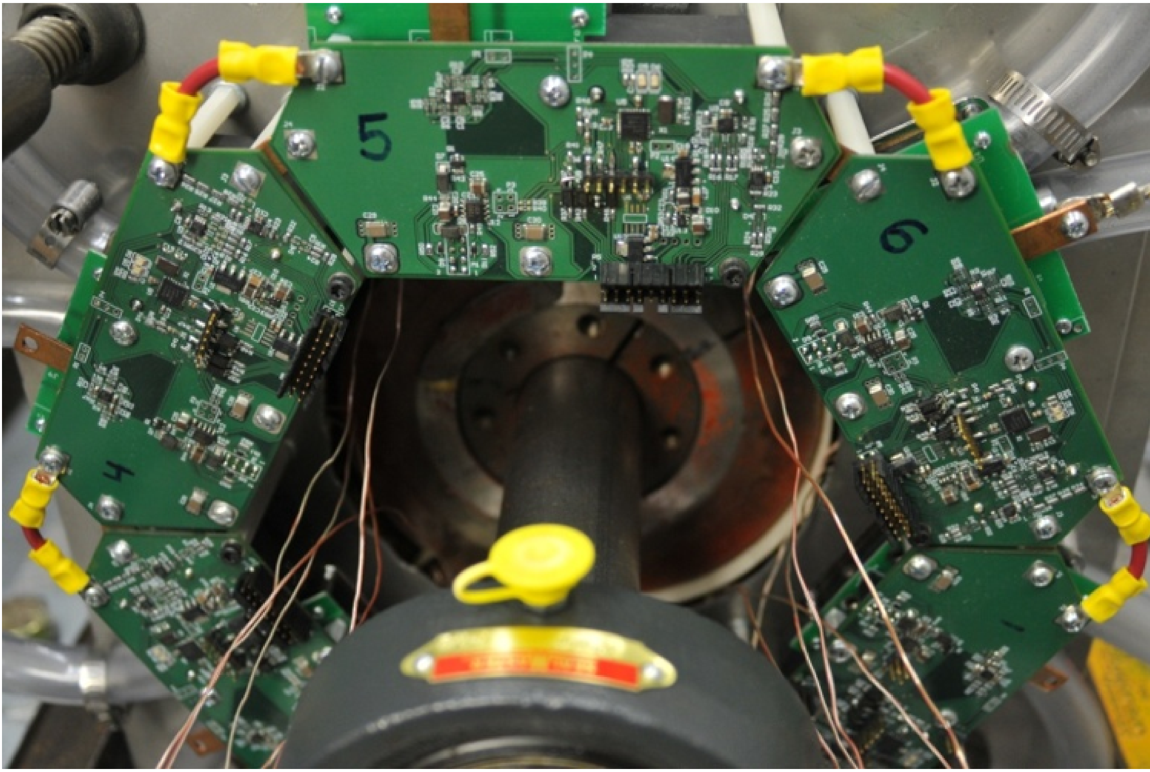


Figure 2.2 Six-phase distributed control IMMD mounted to end of machine with integrated cooling.

Recently at Aachen, a research group at RWTH demonstrated a modular motor drive for a synchronous reluctance traction machine [8]. This machine was a 5-phase, 67kW 16/20 reluctance machine with unipolar phase drives.

The machine had a central controller providing duty ratio commands to individual phase drives which performed PWM modulation on their respective phase windings while sending current measurements back to the central controller. This information was passed between controller and phase drives through a synchronous serial bus as shown in Figure 2.1. This arrangement provided a significant first step towards a practical distributed drive but the lock-step communications mean that significant effort in error correction and signal integrity was needed.

Most recently, at the University of Wisconsin - Madison, a WEMPEC student Jiyao Wang implemented an integrated modular motor drive using GaN FETs with a series-stacked DC-link

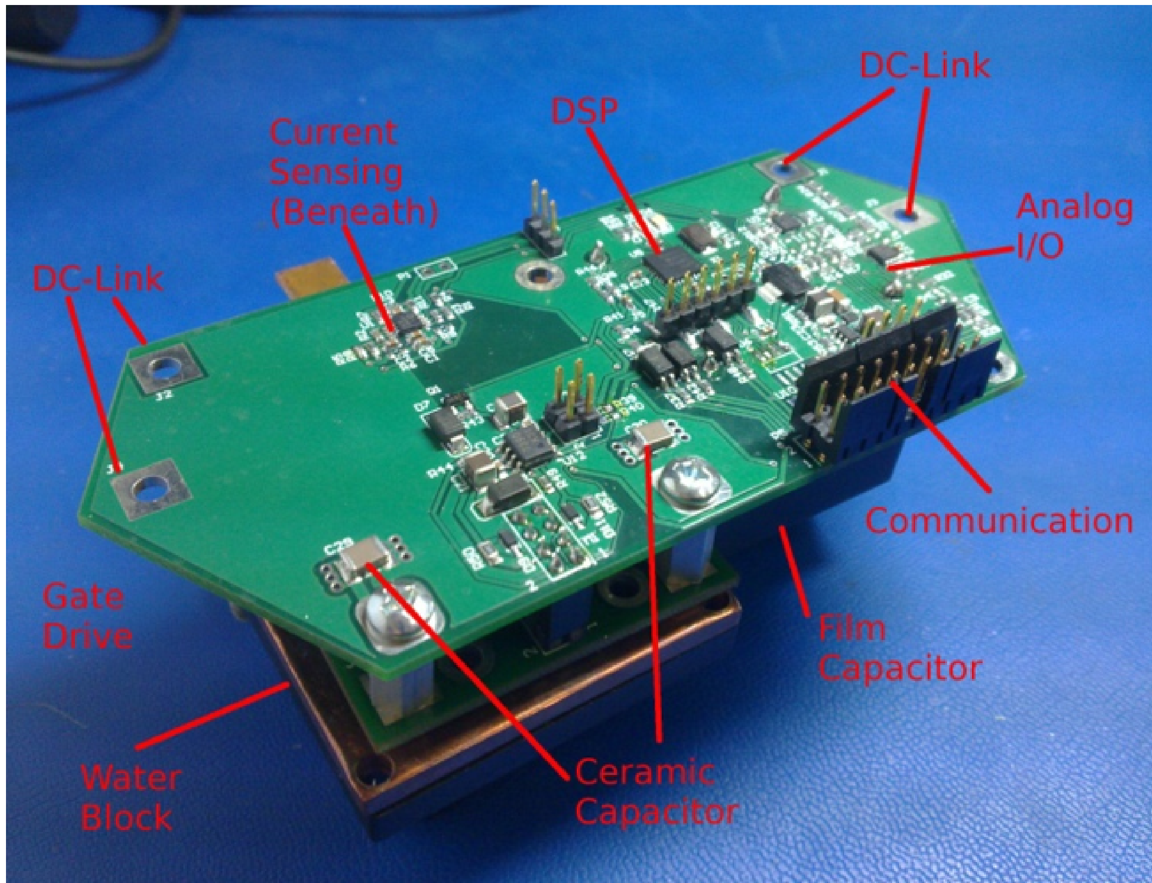


Figure 2.3 Six-phase distributed control IMMD with drive module components labeled.

[7]. Each of the two series-stacked three-phase drive sections drives a separate three-phase set of windings in the machine in order to allow low-voltage GaN FETs to drive a 2hp induction machine. This machine is shown in Figure 2.1. This work explicitly calls out the EMI and machine overvoltage advantages of the tight drive and machine coupling present in an IMMD.

2.2 Modular High-Power Drives

Modular machine drives are not limited to low-power high-reliability systems. They also have seen use in high-power systems where the availability of commodity three-phase drives makes paralleled systems attractive from a cost-efficiency standpoint. In a 2014 paper, Ditmanson and

Figure 2.4 Five-phase SRM machine with modular drive and semi-modularized control[8].

Kolb presented a distributed control system for a wind turbine system consisting of 12 paralleled three-phase drives [12]. In their system, each modular drive performs local current control of its three-phase set, and sends all of its sensor information over an Ethernet network to the centralized controller. This centralized controller performs all system balancing operations, fault-handling, and distributes references for each control cycle. A significant amount of effort was applied in order to keep the distributed system synchronized without running out of communications bandwidth.

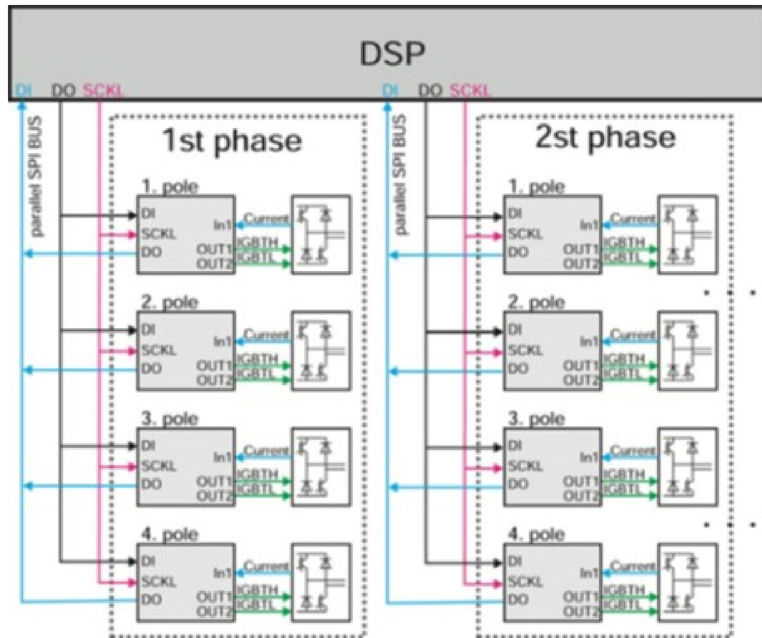


Figure 2.5 Serial bus and controller layout for semi-modular 5-phase SRM machine control [8].

2.3 High Reliability Drives and Converters

Modularization and integration has long been looked to as a route to improve the reliability of motor drives and machines [13]. The key idea behind this approach is that, by adding modularization, there are more redundant components so that any single failure will not cause loss of system function. In machines, this approach has shown significant success in segmenting the machine and magnetic circuits so that a failure in one part of the machine has minimal effect on other parts of the machine [3] [14]. Techniques such as concentrated windings, more than three phases, doubled windings, integral fusing, and single-layer tooth windings are quite effective in reducing fault propagation in machines.

When these techniques are combined with power electronic drive modularization, such as individual H-bridge per phase connections or extra phase legs to drive a neutral point if needed, many of the possible faults in both the drive and machine are survivable [15]. Sadly, all these

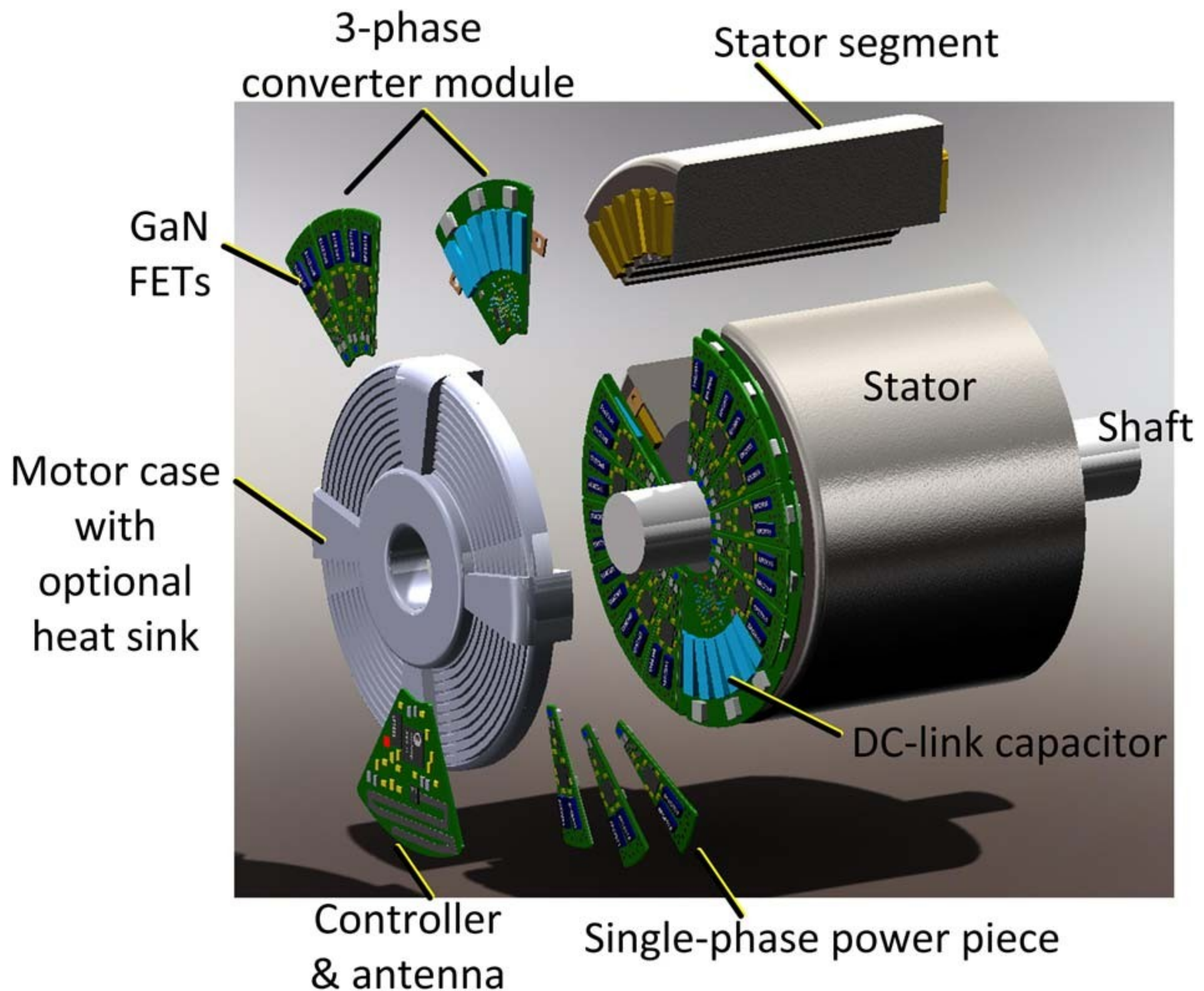


Figure 2.6 2hp IMMD using series-stacked 3-phase GaN inverters and an induction machine [7].

designs typically share two points of failure in that they share a DC-link and power source, and they all share a single central controller to oversee system operation.

Finally, while fault-conscious design can be quite effective in allowing the system to retain function after a fault, the increased complexity of a supposedly fault-tolerant system often increases the fault rate by adding more components sufficiently to reduce the system mean-time-to-failure (MTTF) below the level of standard non-fault-tolerant designs especially if repair is neglected [4]. The work of Argile discusses some of the key tradeoffs associated with the implementation of fault

tolerance, providing several important insights if one is trying to implement a highly reliable and available system. First among these is that fault propagation must be limited as much as possible since the increased system stress from a first fault is the primary instigator of further faults. Second, all single-point failures must be eliminated, if possible. In his paper, an isolated H-bridge topology with independent DC-links is shown to have the greatest potential because the DC-link isolation removes one major source of single-point failures. Removing the central controller would have similar effects in improving reliability. Finally, repair is of utmost importance in a highly reliable system. Modularization significantly eases repair as the system does not need to be completely deconstructed to repair a failed component. If availability is important, being able to repair without down-time may also be important.

2.4 Sensor and Machine Faults

A primary driver of distributed machine control is improved fault tolerance. Fault-tolerant machine control often consists of continuing to create a balanced rotating current vector when faced with limited and/or erroneous feedback and control outputs. Thankfully, this is exactly what a distributed drive must do to handle normal operation. In this way, the literature on fault-tolerant and single-phase control is useful to develop a control algorithm that is able to operate without full sensor availability while only controlling a portion of the output.

In [16], Bahrani *et al* describe what they call Fictive-Axis Emulation in order to allow standard synchronous-frame vector control to be used to control a single-phase inverter. This control technique compensates for the lack of a physical circuit connection in the quadrature phase by modeling a second copy of the physical circuit in the controller and then applying the quadrature stationary phase voltage outputs from the synchronous-frame vector controller. Figure 7 from the paper is reproduced here as Figure 2.4 showing a block diagram of this control technique. This Fictive-Axis method allows for significantly higher controller bandwidth than delay-based

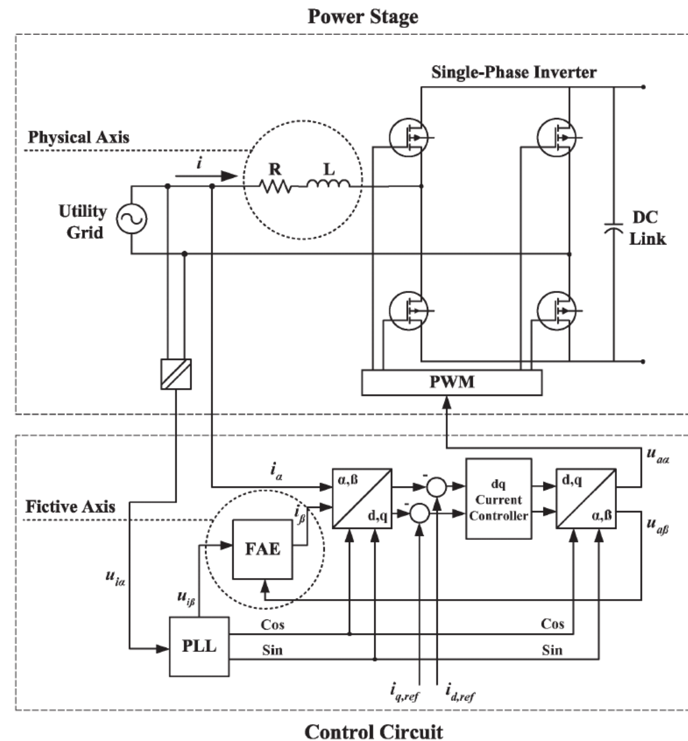


Figure 2.7 Figure 7 from [16] showing Fictive-Axis control of a single phase inverter.

methods of managing the inverter while reducing waveform distortion around command transients and allowing the vector regulator to operate with zero steady-state error. This technique is directly applicable to the current research project since each distributed drive controller can only output a subset of its control outputs to the real system while needing some method to close the loop on its other control outputs.

A similar technique has been applied when paralleling standard three-phase drives to handle a higher power output by Tieyan et. al. [17]. In this case, the zero-sequence current was measured and used to calculate the effective current in a virtual resistance between the two three-phase outputs. This technique was incredibly effective in stopping circulating currents between the two drives while not significantly effecting the control performance or requiring much computation. In this

work, a similar virtual impedance technique will be used to stabilize the free integrators introduced by distributed control.

In a paper on sensorless PMSM control, Bisheimer *et al* describe a rather standard non-linear observer which tracks PM flux in order to run vector control without a dedicated position sensor [18]. This paper however is particularly interesting in that they use a peak detector and vector reconstruction method to handle the case of current sensor failure. Their method uses only two current sensors for a standard three-phase drive in the normal healthy case. During normal operation, a peak detector is used to ensure that the peak amplitudes of each of the current feedback signals is within a given bound of the other. If one of the peak current values falls below the other, it is marked as faulty and the error reconstruction replaces it with a synthesized sinusoid of the correct phase with the same amplitude as the remaining healthy current signal. The paper shows experimental results demonstrating this fault detection and faulted operation. This peak-detection-based fault handling method however poses serious constraints on control tuning. Their observer is formed in such a way that a missing current sensor does not affect the position observer other than a lack of disturbance rejection near the healthy current sensor zero-crossings. The main current regulator, however, is limited in that it cannot be tuned faster than the peak detector circuit which, while acceptable for pumping applications, means that this technique is inadequate for high-bandwidth torque control.

While not directly a control paper, [19] is a nice short paper which describes delay-based formulation of what is essentially an FIR resonant complex bandpass filter for separating positive and negative sequence three phase or complex waveforms. The filter is the standard delayed-difference filter with a compensating gain but has a phase shifted version of the delayed signal added to compensate for the sub-cycle delay filter. This method is mathematically identical to the reference frame transform-based methods but is much more mathematically stable. The delay used can be tuned to trade off detection time with the amplitude of the transient glitch when the

waveform changes quickly and provides a nice knob for tuning if this filter is to be used for fault detection. Originally a filter a filter like this was attempted to null out current sensor errors but the delay formulation caused instability in the distributed current controller.

On the fault diagnosis side, a 2009 paper by Caseiro et al. details a method of mapping the zero-sequence current into the synchronous reference frame for use in fault detection a 3-phase AC drive. In their work, the zero-sequence current is calculated as the average of the individual measured phase currents. This zero-sequence current is placed through the park transform as the real input component in order to determine the phase of the zero sequence current. Faults were detected by monitoring the magnitude and phase derivative of the rotated zero sequence current. In normal operation the phase was constantly changing and the magnitude was low. When a fault occurred, the phase would remain constant and the magnitude would increase. This fault information is then used to trigger a thyristor in the appropriate phase, connecting it to the DC-link midpoint to compensate for the lack of phase connection. While the work presented here is not controlling a drive with a backup SCR midpoint clamp, and the distributed controllers do not have the full-order phase information to calculate a zero-sequence current by direct sum, the use of a Park transformed neutral or zero-sequence variable to detect asymmetry is applicable. In this PhD work, the controller lacks the natural symmetry of a monolithic inverter and thus must use some feedback mechanism to hold each modular inverter section to be symmetric with the others. Neutral-point feedback is a simple and natural way to do this.

Fault diagnosis provides considerable inspiration in how to deal with incomplete sensor information. In 2013, Goruz, Sbita, and Boussak presented a method for determining current sensor faults by monitoring residuals between a set of current vector observers and the measured currents [20]. Each of these observers uses one current sensor for feedback while monitoring the other. When the magnitude of the residual error seen by one of the observers crosses a threshold, the controller can easily mark that sensor as faulty and continue to operate only on the remaining sensor. This paper is

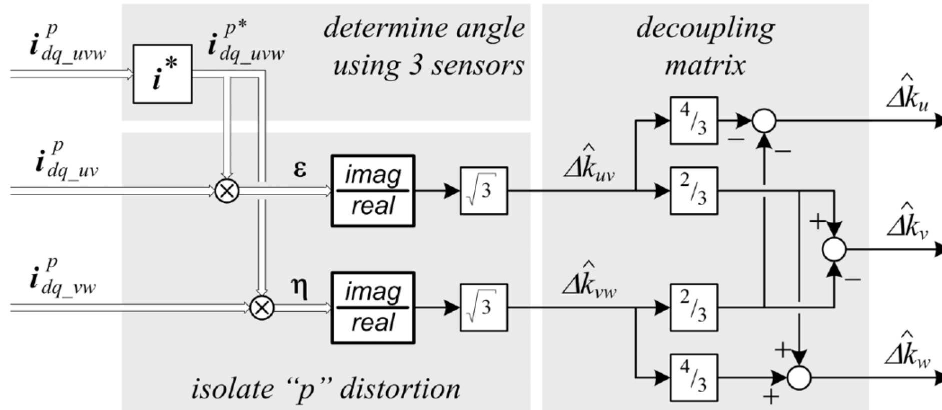


Figure 2.8 Relative gain error estimation using three positive sequence current measurements from [22].

helpful for this work directly in terms of the remaining work of fault handling, but also directly in terms of attempting to build an error model of the current sensor based on an observer model of the ideal system with a subset of sensors for feedback. This observer-based structure is used later in this work as an alternative to sensor-based feedback methods to stabilize a distributed controller in the face of system asymmetries.

With frequency separation, observers can directly be used in order to estimate an accurate signal from imperfect sensors. Yamaguchi et. al. Specifically used periodic disturbance observer to detect and decouple the second harmonic torque error caused by current sensor gain imbalance [21]. This technique caused some limitations in their current regulator tuning but seems promising as a technique for stabilizing a distributed motor drive.

These harmonic separation techniques were perfected by Harke [22] as a way to remove one source of error when attempting to use salient PM machine drives for self sensing. In his method, three current sensors are required. All gain errors in the current sensors are compensated by first forming three measurements of the synchronous frame current, one using all three sensors, one using the first two of the sensors, one using the last two. These measurements are then passed through a pair of matched synchronous reference frame filters to select only the positive sequence and fully

reject the negative sequence. The positive sequence measurement from the full three sensors is used to determine the current angle which is then used to isolate the positive sequence distortion in the pair of two sensor measurements. This stage is shown in Figure 2.8 These distortions are then proportional to the gain difference between the two sensors used to form that respective current measurement. These two distortions are then decoupled into per-phase gain offsets and passed to a PI estimation regulator which converges on a set of sensor gains that is consistent. This method is extremely effective, being both insensitive to saliences and asymmetries and allowing perfect relative gain error decoupling from a standstill, but does have a few drawbacks which make it unsuitable to this work. First, it requires a transient-free fixed frequency voltage excitation in order to make a smoothly rotating current. In Harke's work this is done with a pilot voltage signal. Constructing such a signal in a distributed drive system requires at least somewhat accurate timing synchronization between drives. It cannot be used with the main torque-producing excitation as the current sensor gain change is in the loop for the current regulator. Second, it requires a full set of current sensors in order to obtain accurate angle information while also having two independent two phase measurements to obtain distortion measurements. In the development of this method, he also comments that the relative phases of positive and negative sequence components may be used for a similar purpose which bears some investigation for this distributed application.

In 1996, Chung, Sul, and Lee published a method of compensating for current sensor offset and gain errors by using the speed variation of the machine as a feedback signal [23]. Their method operated by calculating the magnitude and phase of the speed error in the first and second harmonic synchronous reference frames by direct averaging over one electrical cycle then using these complex vectors to derive a current correction value based on the machine parameters and the gains of the controller. The experimental setup used a 1% scaling error in the two current sensors for a three phase system. This caused an effective ripple current of 0.2A and 0.4A to be seen by the drive these

two phases. This method was parameter sensitive, but provided effective attenuation of the torque and speed ripple caused by current sensor errors in a three-phase drive.

Hwang et. al. applied a similar technique to position sensing in order to compensate for amplitude imbalance in a resolver based position sensor by using a cross-correlation method between the position signal and the d-axis current waveform as position error will map into a similar harmonic in the d-axis current [24]. This harmonic cross-correlation method is directly applicable to this work as current sensor errors introduce a second harmonic waveform into the control loop which, if it can be isolated, can be used to determine the system unbalance.

Recently, Secrest et al. used extended this harmonic decoupling based technique to correct for various errors not just in amplitude but in linearity, offset, and gain in a hall-effect based shaft position sensor [25]. In their work, the position sensor measurement is corrected by adding a modeled error vector that is based of an observer's position state. This modeled error is calibrated online by comparing the corrected position sensor output to a unit-vector representation of the observer's position output. This comparison vector is then rotated and run through PI closed loop filters to effectively low- pass the various harmonic components of the sensor error. In the end, this method has great promise for the correction of sensor errors in a sensor limited system because it does not require model feedback from another sensor like Chung et al.'s work, instead depending on the frequency separation of error sources from the system's excitation.

More directly applicable to current sensing, Pat Schneider here at WEMPEC has worked on integrating current sensors directly into IGBT power modules using GMR point field detectors [26]. In order to obtain meaningful current measurements from closely integrated field detectors, significant sensor decoupling is required as each field detector sees fields from currents in several phases as well as any impinging fields from the environment [27].

While this work does not yet tackle faulted operation of the distributed drive, it is good to ensure that the control structure is amenable to the requirements of faulted operation. When operating

after losing control of one or more phases, there are several choices of modulation that can be used to either minimize torque ripple, minimize extra losses, or maximize available remaining torque. Adding to the results in [9], Che et. al. show specifically how to control a 60 degree separation six-phase machine wired as two independent three-phase wye sets after a fault [28]. Their results provide a good starting point for designing faulted operation into a controller from the beginning.

Moving beyond standard current controllers, Schulting et. al. in Aachen show a repetitive controller which is capable of handling current harmonics induced by an external system through the use of a specialized filter to modify the control loop stiffness at undesired harmonics of the fundamental [29]. This is important to this work because any system asymmetries cause second and higher order harmonic errors in feedback and eventually in output. In order to allow for a distributed controller without specialized sensing requirements, the controller must be able to attenuate induced harmonics in the control system.

In the work described in this document, position sensing handled by a shared incremental shaft encoder. This choice represents a single point of failure in the distributed system so future work will want to investigate self-sensing methods of position determination.

Green et al. [30] has shown that a single-phase flux observer is capable of providing a position estimate with only a single voltage output and current input for a poly-phase machine. His results however show significant errors when using single-phase sensing of as much as 7 degrees.

Recently, several injection-based self sensing methods have become popular[31] including sinusoidal[32], square wave[33], mixed-pulse[34], and multi-frequency[35] injections in order to track the rotor saliency. While the sinusoidal carrier methods would present challenges in the limited feedback environment of a distributed drive, the multi-tone and pulse methods may be feasible for distributed position sensing without the drive modules interfering with each other while also providing a communications path independent of the torque production to help with commissioning and fault detection. If each distributed drive is injecting a unique signal, the other drives can use

their own signal for position tracking while motoring the others for topology determination and can detect faults by sensing the loss of other modules injected signals.

2.5 Open-End Winding Machines

The distributed motor drive presented in this work shares several characteristics with open-end winding machines.

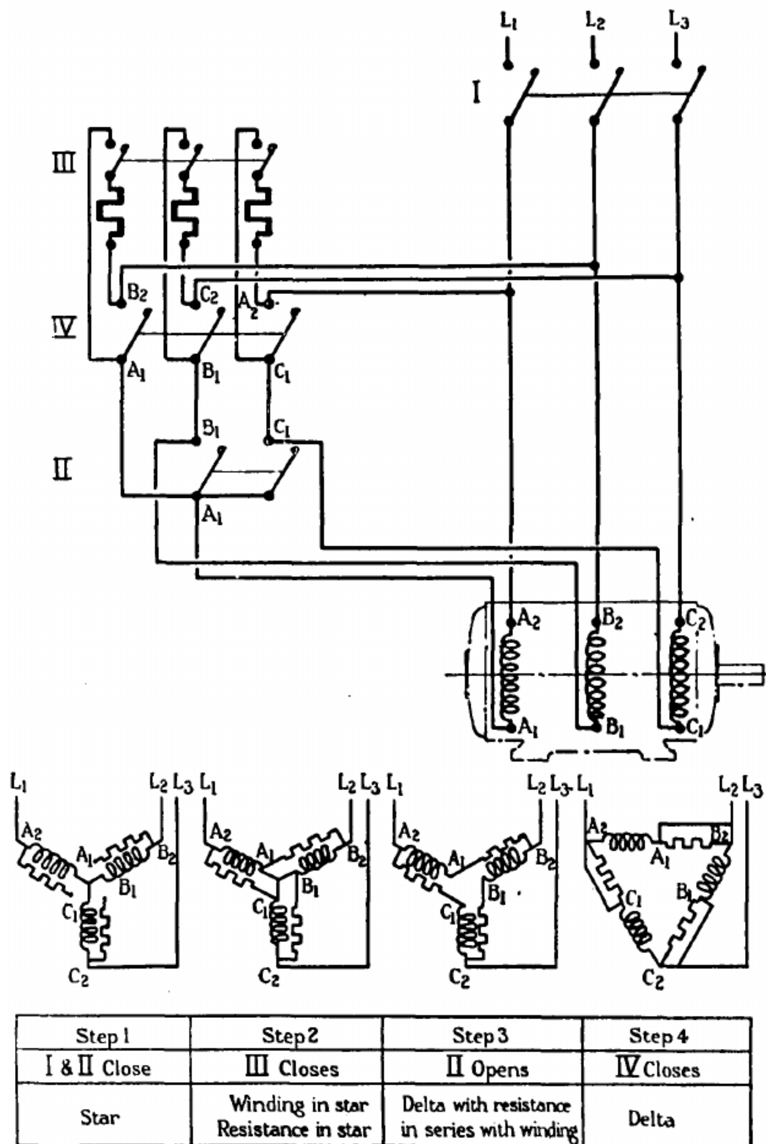
This machine arrangement for three phase machines predates solid state drives as it allowed an induction machine to be mechanically switched between delta and wye configurations during starting in order to allow reduced voltage during startup thus reducing stress on the machine and the line connection while allowing near constant torque though the use of switched series resistors [36]. This arrangement is shown in Figure 2.9.

When applied to power electronic drives, the open winding configuration provides opportunities for increased winding voltage headroom from a given DC-link[37].

Typically, each side of the drive is driven by its own three-phase bridge. If both sides are driven by the same DC power supply, zero-sequence currents are able to circulate through the machine and the DC-link connection. This can be partially handled by adding hardware in the form of a common-mode choke in series with the machine as shown in Figure 2.10 [38].

The zero-sequence current can be mitigated by careful selection of zero-state placement between the two drives which creates an average zero voltage across the zero-sequence during each full switching cycle [39], [40]. This approach however reduces the available linear voltage range by up to 15%.

The either side of the machine is driven by an independent voltage source, the selection of a 2:1 ratio between the voltages on either side makes this drive system isomorphic to a four-level inverter [41].



Step 1	Step 2	Step 3	Step 4
I & II Close	III Closes	II Opens	IV Closes
Star	Winding in star Resistance in star	Delta with resistance in series with winding	Delta

Figure 2.9 Continuous torque star-delta starter using open-end winding machine from 1944 [36].

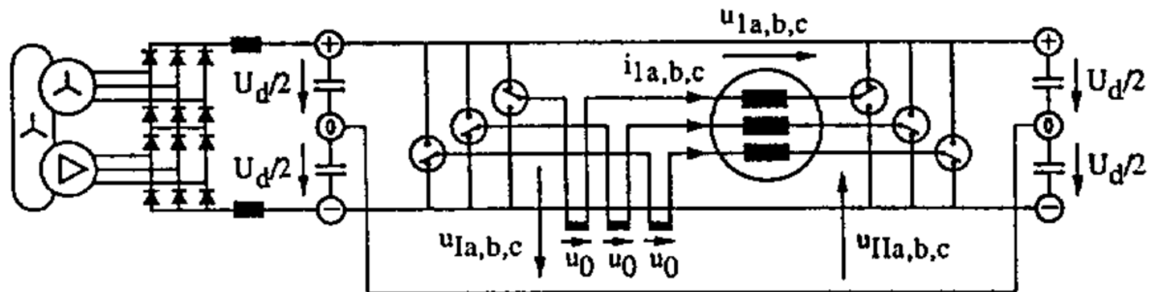


Figure 2.10 Open-end winding machine drive with common mode choke to suppress zero-sequence currents [38].

If the two sides are left floating, they can be used independently to provide or absorb real or reactive power without regard to any zero-sequence worries. This can be used for instance to improve the speed range and efficiency of a PM drive by proving reactive power only with only a capacitor on one side of the drive [42], [43]. At the other extreme, the open winding machine can pass power through the machine as well as produce torque, allowing the two drives to be used as two ends of a DC-DC converter for battery management [44].

The open winding configuration also provides many opportunities for fault-tolerant control due to the added redundant space vectors afforded by the added drive [45].

2.6 Modular High-Power Machines

From the application side, integrated modular motor drives have most often been considered for high-reliability applications. Introducing distributed control however introduces a new high-power application space where modularity is essential. In high power machines, the machine drive and often the machine itself is built in a modular fashion due to the limitations in constructing, handling, and shipping very large monolithic machines and drives.

One example application is in wind turbines. Wind turbines have unique limitations in that a MW-scale machine is mounted atop a narrow tower far from dedicated infrastructure. Machine modularity in this case allows for significantly simplified shipping and maintenance logistics while

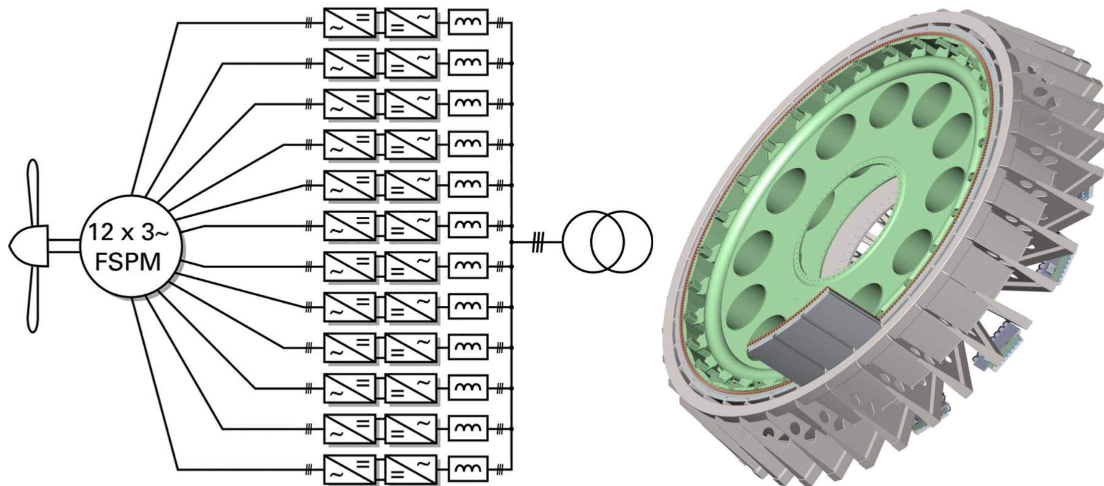


Figure 2.11 Modular Flux-Switching Permanent-Magnet machine for a large wind turbine.

allowing the machine models to be lifted without the assistance of an external crane. In 2014, Ditmanson et. al. presented a modular flux switching machine which included both a modular machine and modular drive [12]. The drive in this case consisted of standard three-phase back-to-back converters with a centralized controller. Renders of both the machine and its controller layout are shown in figure 2.11.

Modular machine designs also allow for much more flexibility in magnetic circuit layout compared to more conventional machines. This can allow for better utilization of magnetic materials and opportunities for improved cooling and simplified winding design. In large machines, these designs are inherently high phase order. In 2015, Husain et. al. presented a novel axial-flux machine where the stator was composed of individually wound E-cores [46]. Their paper showed that the E-core based design can more nearly double the mass-relative torque density of a machine. This highly modular design, with fully disconnected stator iron lends itself well to a distributed machine and drive system designed for ease of repair.

2.7 Hot-plugging and DC-Link Isolation

One final application possibility enabled by modular distributed drives is that of hot-plugging. In a system that is already configured to gracefully handle the loss of individual phases, a system designer can introduce the capability of deliberately removing and adding phases from the system. In doing this, the AC side of the modular controller is safely taken out of the equation, commanding zero current after the removal command is received. For the DC side of the drive module, there has been significant work in managing DC power hot-plugging in the solar [47] and data center spaces [48]. These methods use high-frequency DC-DC converters to manage their power supply which allows no-notice hot-plugging without risking the damage caused by high intensity DC contact opening [49].

2.8 Distributed System Synchronization

Time synchronization in distributed computing systems has been a topic of significant research since nearly the dawn of the computer age [50]. Leslie Lamport pioneered significant early work in how to feasibly synchronize distributed clocks in the presence of errors [51].

Due to the well studied nature of discrete time controls, significant effort has gone into making distributed systems operate in a time-triggered manner [52]. Time triggered control systems even have a formal proof of their limitations [53].

CAN bus has seen significant use in the automotive and industrial sectors because when coupled with time-triggered protocols, it allows for guaranteed performance even in the face of faults while providing it's own synchronization path [54], [55].

As networks require more bandwidth, more control systems are moving to Ethernet. Ethernet was originally designed with no specific guarantees of timing or throughput. In order to address this limitation and allow Ethernet to be used real-time controls and audio, the IEEE defined a Precision

Timing Protocol which allows for timing synchronization to tens of nanoseconds over existing Ethernet [56]. This protocol requires very little overhead and has open-source implementations [57].

2.9 Fault Modeling and Fault Tolerance

A primary reason to modularize and distribute machine control is to remove single points of failure and improve fault tolerance. There has been significant research in how both how to model a faulted motor and on how to excite a motor after a fault has occurred.

On the modeling front, the greatest difficulty arises when modeling open-circuit faults as the machine model commonly used is based on terminal voltages applied as inputs and torque and machine currents being outputs. This usually leads to closed-form solutions for a single choice of machine type, phase-order, and healthy-phase configuration. An excellent example of this modeling effort is described by Welchko et al. [58] where the one phase open-circuit model of a three-phase IPM (Interior Permanent Magnet) machine is derived to be

$$\frac{d\rho}{dt} = \left[-r_s\rho + \omega_e\rho\left[L_d - \frac{L_q' + L_d}{2}\right]\sin(2\theta_e) - \omega_e\Psi_{mag}\sin(\theta_e) - v_d^s\right]\left[\frac{1}{L_q'\sin^2\theta_e + L_d\cos^2\theta_e}\right] \tag{2.1}$$

where

$$\rho = \frac{i_q^s}{\sin\theta_e} \tag{2.2}$$

This describes a single-phase machine that is equivalent to the two-terminal machine that remains after open-circuiting one of the phases. This sort of model is also calculated for six-phase machines in in the same style where the structure of the machine equations changes depending on the fault [59].

While this sort of model is useful, especially for accurate simulation, it is limiting in that it has a significantly different structure than the model of the healthy machine and it only applies to

a single machine type and fault type. A start at a more generalized approach based on harmonic decomposition is covered by [60]. In this work, an induction machine is controlled with any number of phase faults by only controlling harmonic reference frames up to the number of healthy phases thus allowing the machine to continue providing a smooth MMF to the rotor.

There are several strategies to optimize the excitation of a machine after a fault. The first are aimed at mitigating further damage in case of a fault and often aim to short remaining healthy phases in order to save magnets and power electronic components from damage [61]. If the faults can be converted to open-circuit faults, there is ample opportunity to continue to produce torque with varying degrees of smoothness and power capability depending on the amount of overdesign in the drive and machine and the sophistication of the excitation command. At one extreme, there are optimized waveforms such as those developed in [62] for a five-phase machine and shown in Figure 2.12. These currents produce minimal torque ripple but are difficult to use for dynamic applications as they are rich in harmonic content.

At the other end of the spectrum are very effective solutions for the dynamic case which merely change the angle and amplitude of the remaining phases in proportion to their healthy excitation [63]. These solutions may not be optimal and are lacking higher harmonic components needed for perfect torque ripple cancellation but they have an advantage of not changing the fundamental-harmonic controller structure which is helpful for this work.

2.10 Conclusion

To conclude, previous research has made significant efforts to develop modularized motor drives both for improved integration and fault tolerance. These efforts have also begun to extend to the drive controller but have been hampered by communications scaling issues.

There is also excellent prior art in handling motor drives with limited sensing using imperfect sensors which is an essential foundation for modularizing a motor controller without significantly

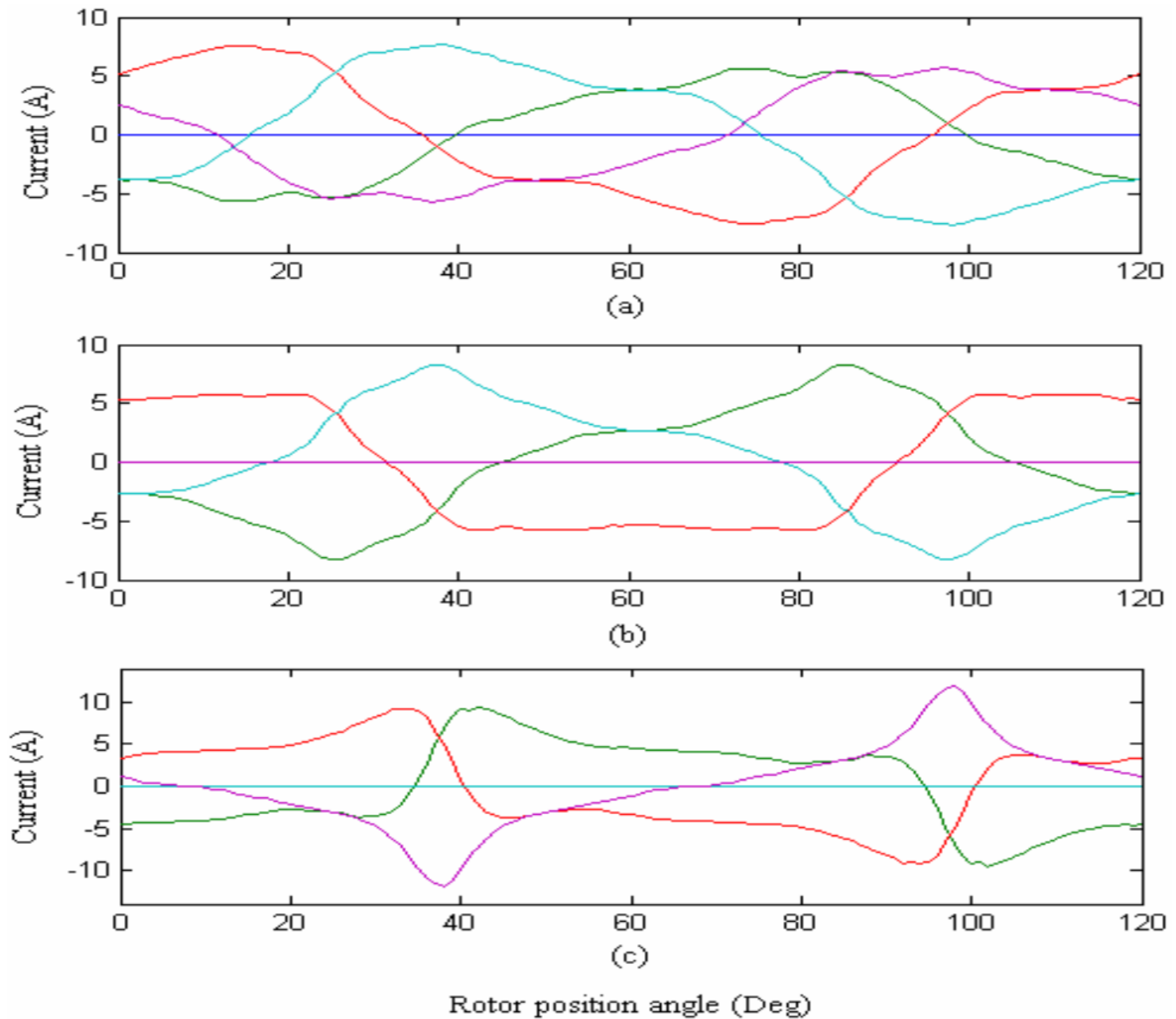


Figure 2.12 Optimal current waveforms for a star-connected five-phase PM machine with (top) phase a open, (middle) phases a and c open, and (bottom) phases a and b open[62].

increasing the cost of sensors. In computer science, synchronization of distributed systems is a well studied problem and there are many low-bandwidth techniques to achieve system synchronization even in the face of faults and errors.

Finally, fault tolerant operation is typically handled as one-off special cases for each faulted motor configuration. There is however some work which maintains the control structure even after faults thus allowing a more generalized post-fault control approach.

These prior research efforts have set the stage for the work presented in this dissertation which aims to develop a fully distributed drive controller which has fault-tolerant capabilities in order to support fault-tolerant modular drives without requiring significant increases in sensors or communications bandwidth.

Chapter 3

Three-Phase Drives with Monolithic Controller

The purpose of this chapter is to first review the analysis and behavior of a baseline three-phase machine drive consisting of a three-phase PM synchronous machine with a floating-wye stator winding configuration, a six-switch full-bridge inverter, and a monolithic controller. It is assumed that the drive is equipped with two high-bandwidth current sensors to provide feedback information for two of the three phase currents. After first considering the performance of this drive using a complex vector current regulator in the reference frame with ideal current sensors, the analysis is then extended to determine how the performance is affected if either the gain or offset of one of the two current sensors is incorrect. The results of this analysis for a standard three-phase machine drive using a conventional centralized (monolithic) controller provides the baseline for investigating how the behavior changes if a distributed controller architecture is adopted in the next chapter.

3.1 Terminology and Symbols

The following terminology and symbols will be used in the development of the distributed motor control.

Motor drive Both the controller and power electronics to excite an electric machine. These may be monolithic where both are fully connected or either the control or power electronics may be modularized or both.

Controller The low-level signal processing components in a motor drive which receive feedback from sensors, calculate internal states, and determine gating commands for power stages.

Monolithic Controller A motor drive with a single controller receiving all feedback and producing all gating signals.

Distributed Controller A motor drive with multiple controllers where each controller receives a subset of feedback signals and produces a subset of gating signals.

λ_{pm} PM machine flux.

R_v A virtual resistance.

$\hat{X}$ The estimated or measured value of quantity X.

X^* The commanded value of quantity X.

$\vec{x}_{\alpha\beta n}$ A vector quantity x in the two-phase equivalent stator reference frame.

$\vec{x}_{abc}$ A vector quantity x in terms of physical phase quantities.

$\vec{x}_{123}$ A vector quantity x in terms of logical phase quantities.

$\vec{x}_{dq0}$ A vector quantity x in the synchronous reference frame.

n The number of phases in a motor drive system.

P The number of poles in an electric machine.

ω_e The electrical frequency of the motor in radians per second.

ω_m The mechanical frequency of the motor in radians per second.

τ_{PI} The time constant of a PI regulator in seconds.

K_p The proportional gain of a PI or P regulator.

$\mathbf{R}(\theta)$ A rotation matrix that describes a Park Transform by angle θ.

$\mathbf{P}$ A polarity matrix that converts from logical phase quantities (where the phases evenly cover only the first π radians) to physical phase quantities including phases occupying the same logical vector.

$\mathbf{T}$ A matrix converting terminal voltages to phase winding voltages by taking into account star or delta connections.

$\mathbf{C}$ A constant-power Clarke Transform matrix between a set of two-phase equivalent quantities and logical phase quantities.

$\mathbf{K}$ A matrix describing the feedback controller in state space form.

$\mathbf{B}^*$ A matrix containing the feedforward terms of the controller in state space form.

$\mathbf{A}$ The feedback matrix for the natural response of the physical system.

$\mathbf{A}^+$ The feedback matrix incorporating the response of physical system and the feedback portion of the controller.

$\mathbf{B}$ The feedforward matrix for the physical motor system.

$\mathbf{B}^+$ The feedforward matrix incorporating the response of physical system and the feedforward portion of the controller.

L_d The d-axis inductance including stator leakage and magnetizing inductance. This may have a further numerical subscript in the case of higher phase orders to denote higher harmonic planes.

L_q The q-axis inductance including stator leakage and magnetizing inductance. This may have a further numerical subscript in the case of higher phase orders to denote higher harmonic planes.

L_{ls} A leakage inductance.

G A sensor gain matrix which may have a subscript denoting the reference frame in which it is parameterized.

O A sensor offset matrix which may have a subscript denoting the reference frame in which it is parameterized. When combined with the **G** matrix this describes an affine transformation which approximates the sensor errors.

3.2 Machine Model

The derivation that follows is based on the dq model of a PM synchronous machine in the rotor reference frame as described in [64], [65] and shown in equations 3.1 and 3.2.

$$v_d = r_s i_d + L_d \frac{di_d}{dt} - \omega_e L_q i_q \tag{3.1}$$

$$v_q = r_s i_q + L_q \frac{di_q}{dt} + \omega_e L_d i_d + \omega_e \lambda_{pm} \tag{3.2}$$

$$v_0 = r_s i_0 + L_s \frac{di_0}{dt} \tag{3.3}$$

where r_s is the stator resistance, L_d and L_q are the d-axis and q-axis inductance, λ_{PM} is the PM flux, P is the number of magnetic poles, n is the number of phases (three in this case).

For ease of analysis, these systems of equations will be represented in a state-space format described by two equations 3.4 and 3.5.

$$\frac{d\vec{x}}{dt} = A\vec{x} + B\vec{u} \tag{3.4}$$

$$\vec{y} = C\vec{x} + D\vec{u} \tag{3.5}$$

Typically, equation 3.5 can be omitted if the only concern is the dynamics of the system in question rather than any specific output.

Therefore, equations 3.1 to 3.3 can be rewritten in space format as:

$$[\mathbf{A}] = \begin{bmatrix} -\frac{r_s}{L_d} & \omega_e \frac{L_q}{L_d} & 0 \\ -\omega_e \frac{L_d}{L_q} & -\frac{r_s}{L_q} & 0 \\ 0 & 0 & -\frac{r_s}{L_{ls}} \end{bmatrix} \quad x = \begin{bmatrix} i_d \\ i_q \\ i_0 \end{bmatrix} \quad [\mathbf{B}] = \begin{bmatrix} \frac{1}{L_d} & 0 & 0 \\ 0 & \frac{1}{L_q} & 0 \\ 0 & 0 & \frac{1}{L_{ls}} \end{bmatrix} \quad u = \begin{bmatrix} v_d \\ v_q - \omega_e \lambda_{pm} \\ v_0 \end{bmatrix} \tag{3.6}$$

For use in simulation, these equations are rearranged to model the machine in terms of flux linkages since it is easier to manage when using physical variables, especially for high-phase-order models. The flux linkages for the three-phase system are defined as

$$\lambda_d = L_d i_d + \lambda_{pm} \tag{3.7}$$

$$\lambda_q = L_q i_q \tag{3.8}$$

The machine equations then become

$$\frac{d\lambda_d}{dt} = v_d - r_s i_d + \omega \lambda_q \tag{3.9}$$

$$\frac{d\lambda_q}{dt} = v_q - r_s i_q - \omega \lambda_d \tag{3.10}$$

$$T_e = \frac{P}{2}\frac{n}{2}\left(\lambda_d i_q - \lambda_q i_d\right) \tag{3.11}$$

Using this flux-linkage-based form of the equations means that the state-space formulation no longer has divisions by inductances everywhere which significantly simplifies the Simulink block diagram. Also, as the analysis moves to high-phase-order machines, the flux-linkage-based term allows a simple vector-based model in both the torque equation and the electrical parameters.

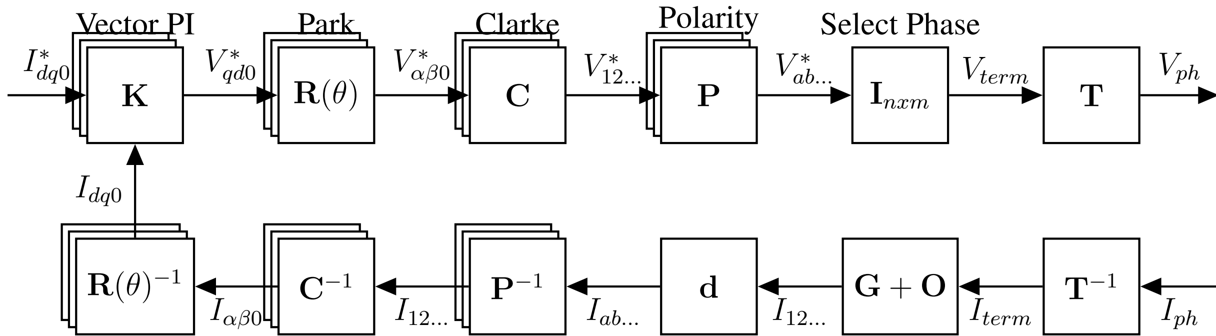


Figure 3.1 Overall System Block Diagram

3.2.1 Coordinate Transforms

This synchronous reference frame model can be decomposed into individual phases by application of the Clarke transform [66] and Park transform. For this work, a constant-power convention is chosen when changing variables as this is more consistent when generalizing to higher phase-order numbers [67]. This choice also allows the Clarke transform to be more easily inverted as the leading coefficient is the same. Dr. Rockhill's [67] naming convention of phase orders is chosen where all winding vectors are generalized to the first half of the machine and a polarity matrix is used to couple these logical windings to physical winding quantities.

This convention not only provides a way to resolve the ambiguity presented by some higher phase-order machines such as the 6-phase model which will described later, it also allows several physical windings to share the same logical axis which is important for several highly fault-tolerant topologies or windings in very large machines. Thus, for the three-phase system the various reference frames are described by the following set of relations where x is a generalized variable

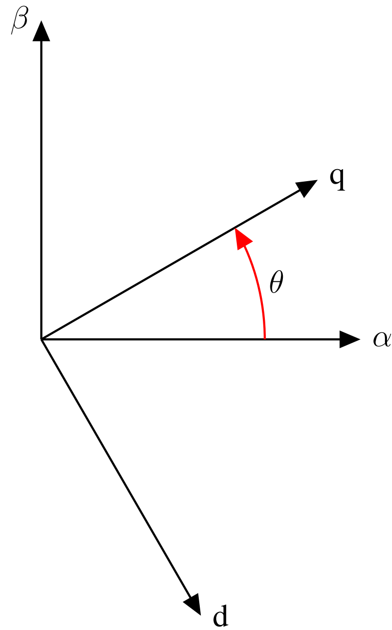


Figure 3.2 Axes and sign conventions for coordinate frame transforms.

(i.e., current or voltage).

$$\mathbf{C} = \sqrt{\frac{2}{3}} \begin{bmatrix} 1 & 0 & \frac{1}{\sqrt{2}} \\ \cos\frac{2\pi}{3} & \sin\frac{2\pi}{3} & \frac{1}{\sqrt{2}} \\ \cos\frac{4\pi}{3} & \sin\frac{4\pi}{3} & \frac{1}{\sqrt{2}} \end{bmatrix} \tag{3.12}$$

$$\vec{x}_{abc} = \mathbf{C}\vec{x}_{\alpha\beta n} \tag{3.13}$$

$$\mathbf{R}(\theta) = \begin{bmatrix} \sin\theta & \cos\theta & 0 \\ -\cos\theta & \sin\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

$$\vec{x}_{\alpha\beta n} = \mathbf{R}(\theta)\vec{x}_{dqn} \tag{3.15}$$

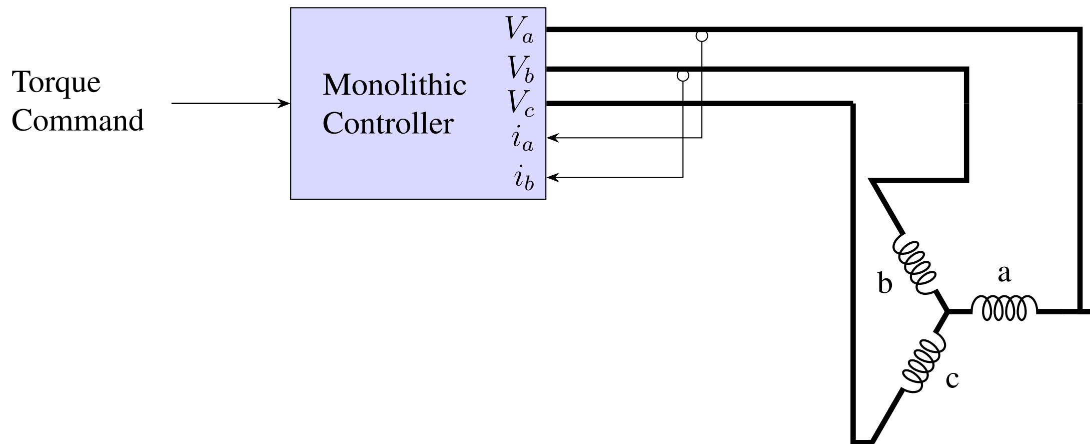


Figure 3.3 Schematic of a standard monolithic drive controlling a three-phase machine.

3.3 Three-Phase Drive Model with Ideal Sensors

In order to provide a baseline, a three-phase machine connected to a standard three-phase drive implementing current-regulated vector control is considered. This is currently the most widely implemented drive control strategy and is the standard benchmark of acceptable general-purpose drive performance. A schematic representation of this drive and its sensors is shown in figure 3.3. It is assumed that the current sensors used in this section are ideal with no offsets or gain mismatches.

3.3.1 Simulation

For simulation, the machine described above is modeled in Simulink. The Simulink block diagram of this machine model in the synchronous frame is based on the standard block diagram shown in Figure 3.4 The machine parameter values for the 5 hp interior PM machine that is being used during experimental tests in this research program (see Sec. 6.2) are presented in Table 3.1. These parameters are used in the simulations presented later in this chapter and the following chapters. This does not match the three-phase setup in the hardware chapter as this model is based on a wiring with three of the nine-phase configuration used with the other six phases left open.

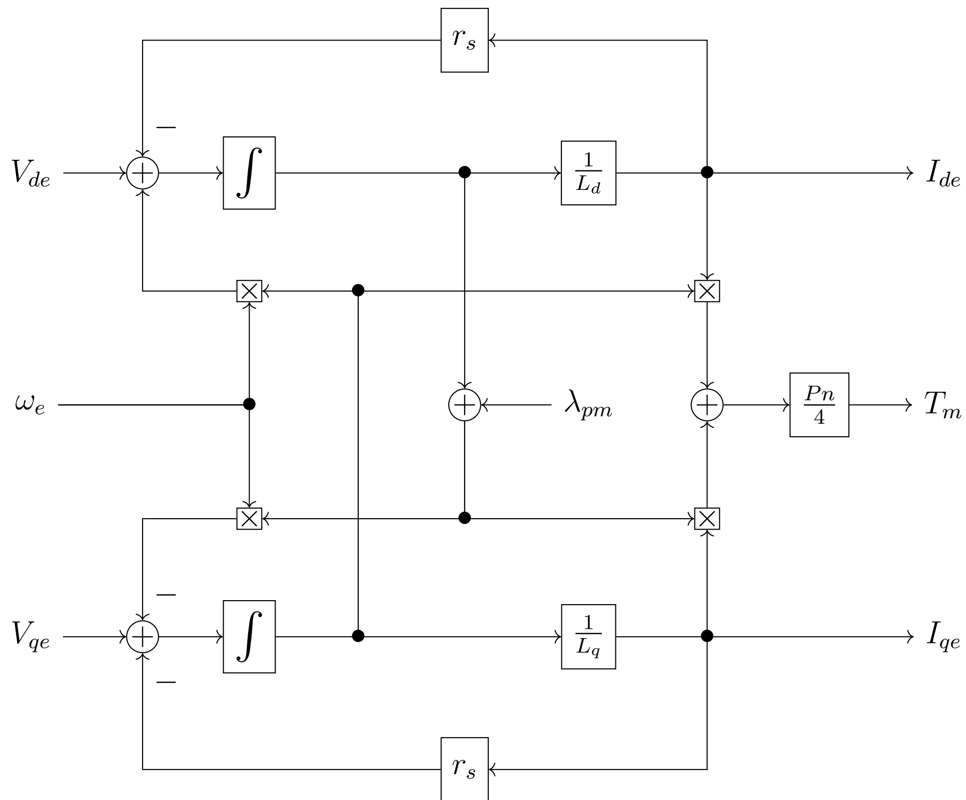


Figure 3.4 Block diagram for 3-phase PM Machine in the synchronous reference frame.

Table 3.1 Machine parameters for the simulated 3-phase machine.

Parameter	Value	Per-Unit
V_{ph}	132 V	1.00
I_{ph}	5 A	1.00
f_{rated}	60 Hz	1.00
R_s	989 mΩ	0.04
L_d	44.0 mH	0.62
L_q	177.3 mH	2.51
λ_{PM}	509mWb	1.02

For the simulation results that follow this machine model is combined with coordinate transforms as described in equations 3.12 and 3.14 to handle terminal quantities in the stationary phase reference frame. Machine speed is controlled by a PI speed controller to model a dynamometer system with a speed-regulated prime mover. Each simulation starts at rest, then the dynamometer ramps speed up to the target then regulates speed with a slow feedback loop as the torque from the test machine model changes. This simulation and all that follow in this report are not fixed-speed and have finite inertia.

The current regulator for both this single monolithic drive and the distributed drives to follow is a complex vector PI current controller in the synchronous reference frame. This is different from the standard PI current regulator presented in [64]. The complex vector PI current controller uses the off-axis pole placement described in [68] in order to ensure that system response is stable with changing rotor speed. A virtual stator resistance is also added as described in [69] in order to passivate the system and reduce parameter dependence. This model is shown in Figure 3.5 along with the coordinate transforms needed to convert its input and output terminals to and from the synchronous reference frame. Position feedback is modeled by a shaft encoder since position self-sensing control is beyond the scope of this research project.

A simplified schematic diagram of this three-phase machine drive using a monolithic (centralized) controller with two phase current sensors is presented in figure 3.3.

The model is first run with ideal current sensing and no asymmetries in order to get a baseline for system performance. The torque command and output waveforms, along with the current and voltage waveforms are shown in Figure 3.6 and Figure 3.7. The neutral voltage is also shown since this will be important later. The test waveform is a five-second series of torque steps where the i_d^* and i_q^* current commands are generated by a table-based maximum-torque-per-amp function tuned to the machine. The simulation runs with the dynamometer regulating speed. This choice of test functions is entirely arbitrary but could be seen to represent some manner of traction application.

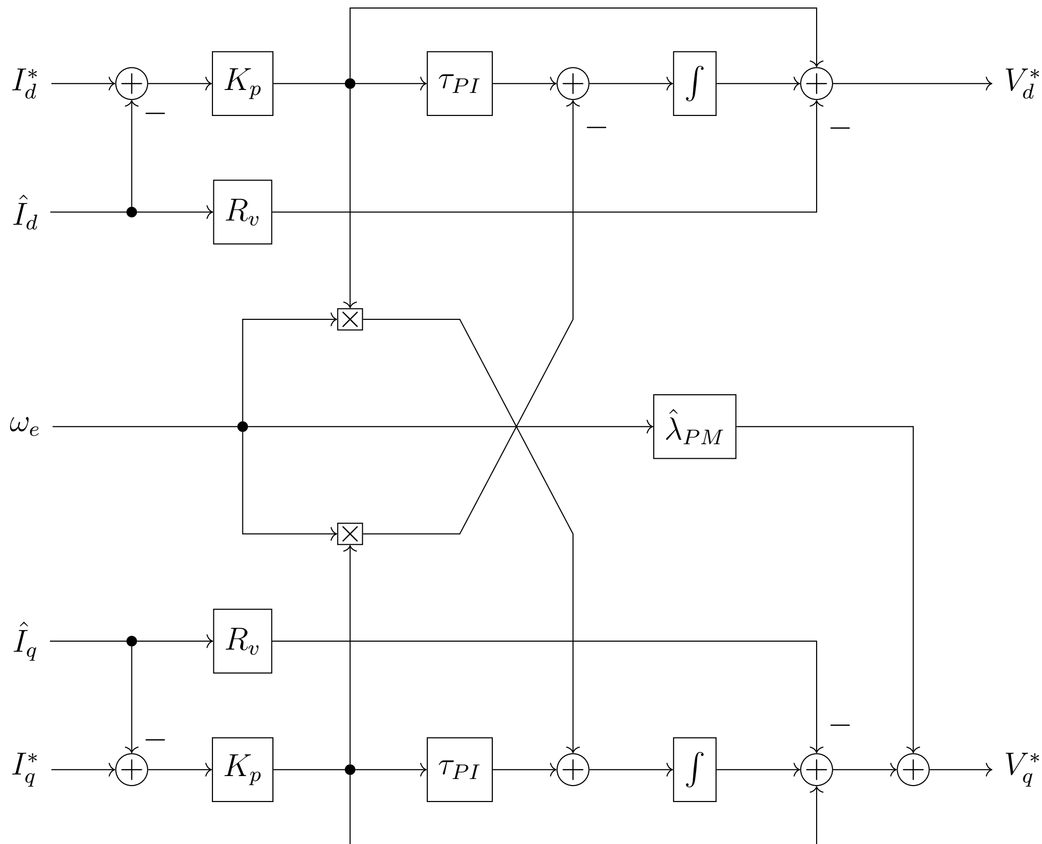


Figure 3.5 Block diagram of the complex vector current regulator used in this research. The operating frequency input into this model must be the electrical rotation rate of the reference frame it is applied in as this speed is used to decouple speed voltages.

Next, the system is simulated with a more realistic current sensor system with ten percent error in one of the current feedback paths. The same command sequence is then run again with the results presented in Figure 3.8 and Figure 3.9.

The system is only using two current sensors since is common for a three phase drive system. Since the controller enforces current summation to zero at every time instant (i.e., no zero-sequence current), an error in one of the current sensors forces the calculated current of the unmeasured phase to also be in error. The torque output generally follows the command well but has a ripple of similar magnitude to the relative current sensor error with a frequency of twice the fundamental since the current regulator in the drive tracks the measured current rather than the actual current. This ripple

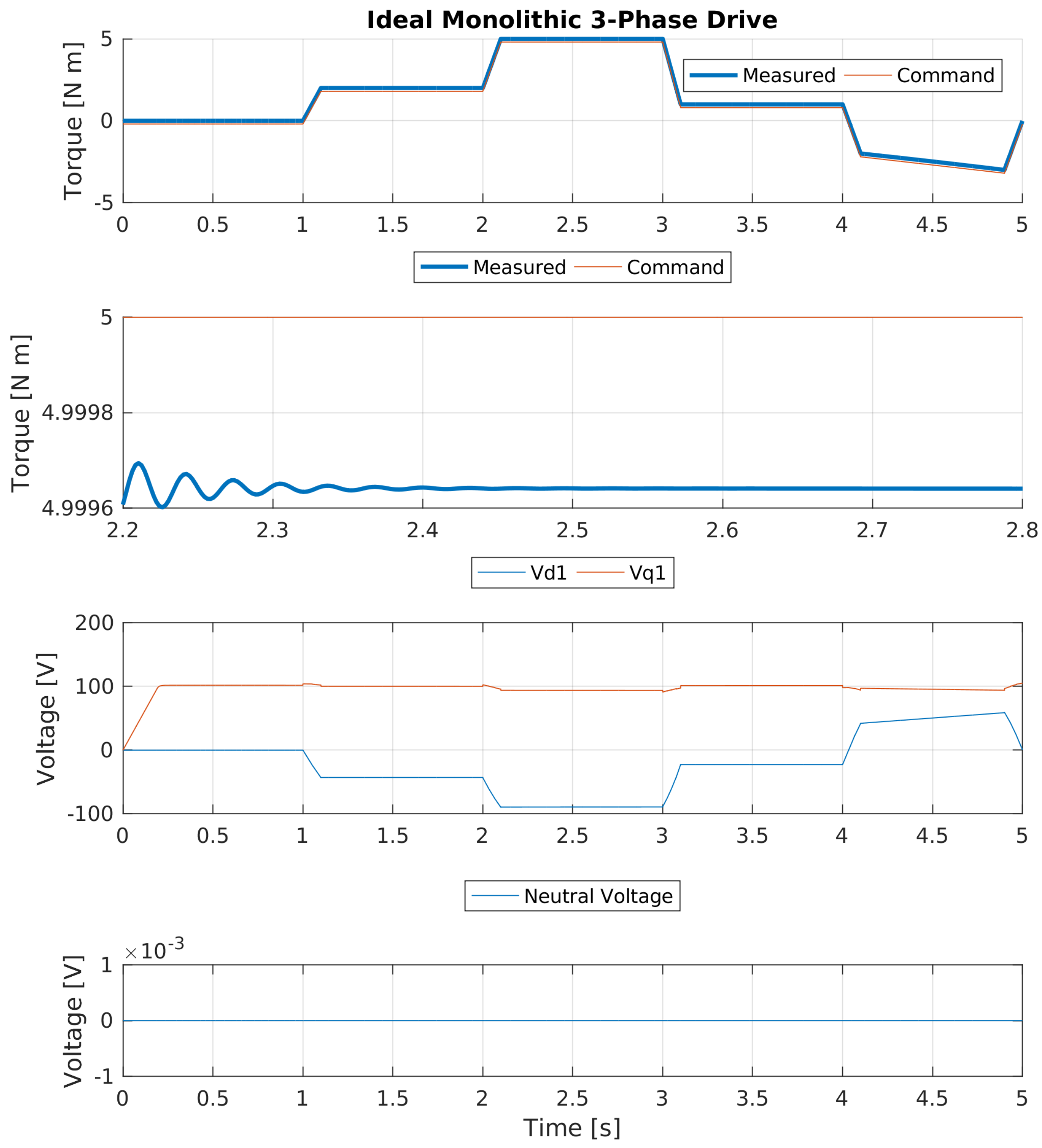


Figure 3.6 Simulation of three-phase vector controlled drive with ideal current sensing. Torque and machine terminal voltages are shown.

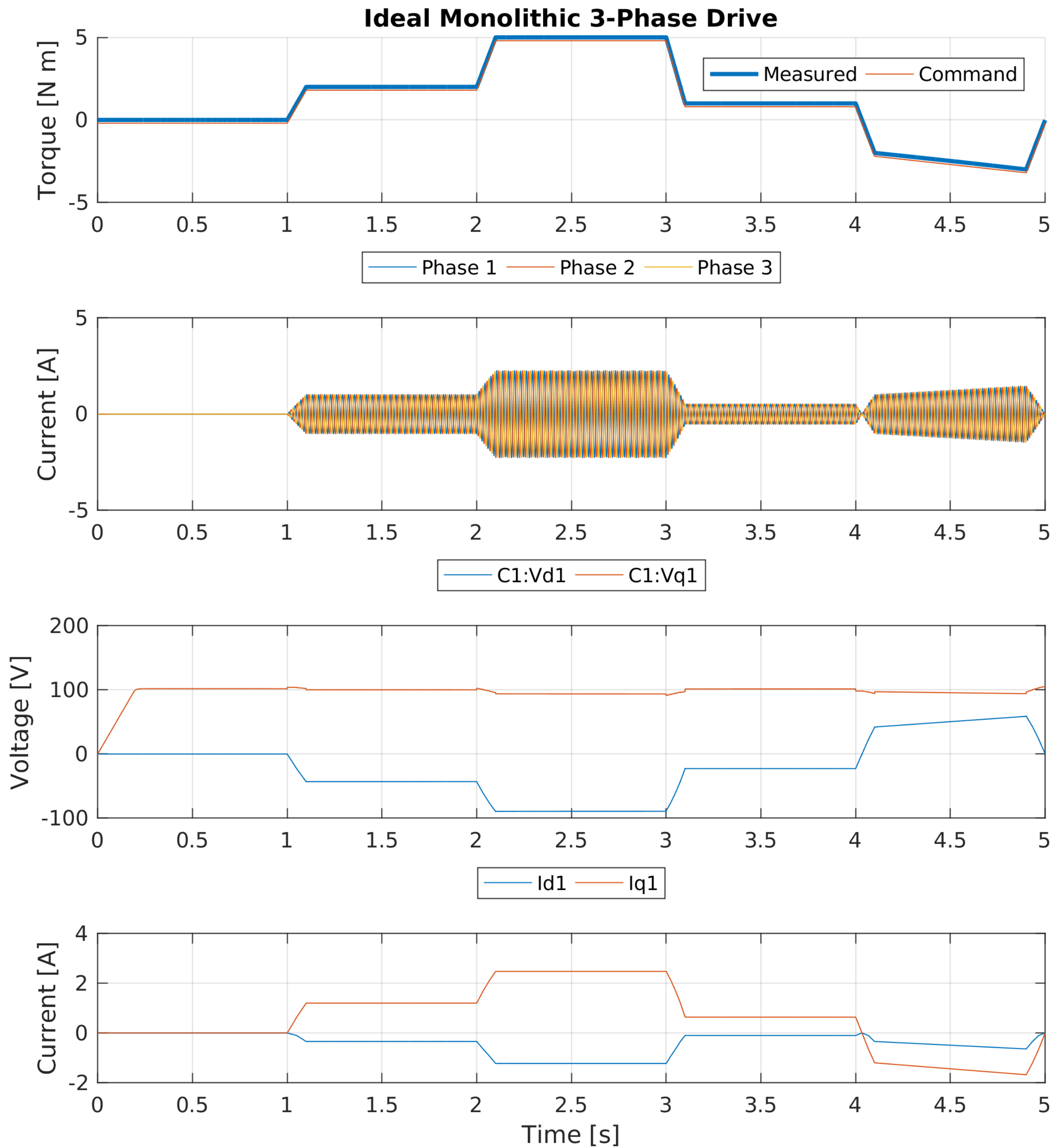


Figure 3.7 Simulation of three-phase vector controlled drive with ideal current sensing. Torque, terminal currents, and controller command voltages are shown.

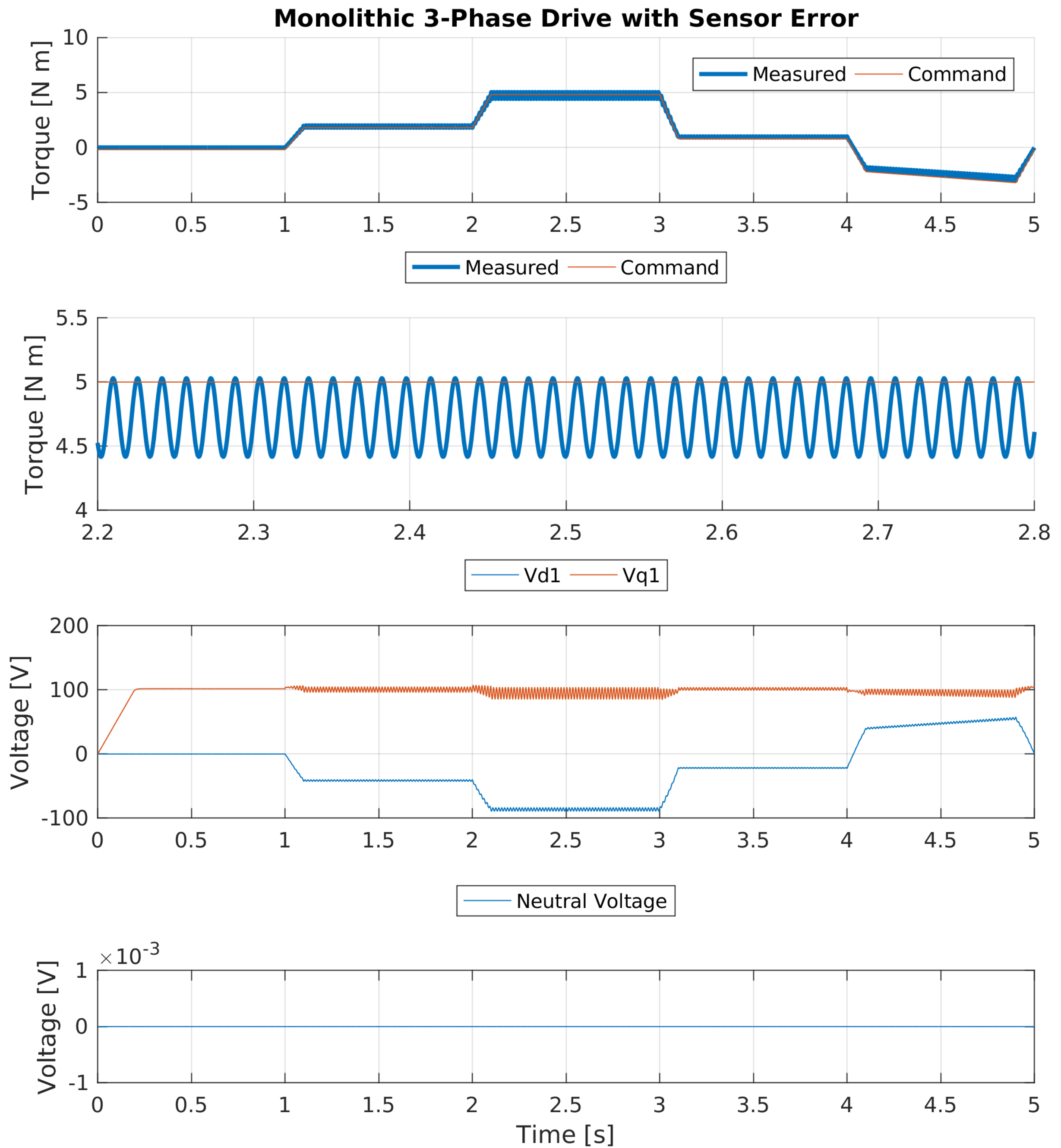


Figure 3.8 Simulation of three-phase vector controlled drive with a ten percent gain error in one current sensor. Torque and machine terminal voltages are shown.

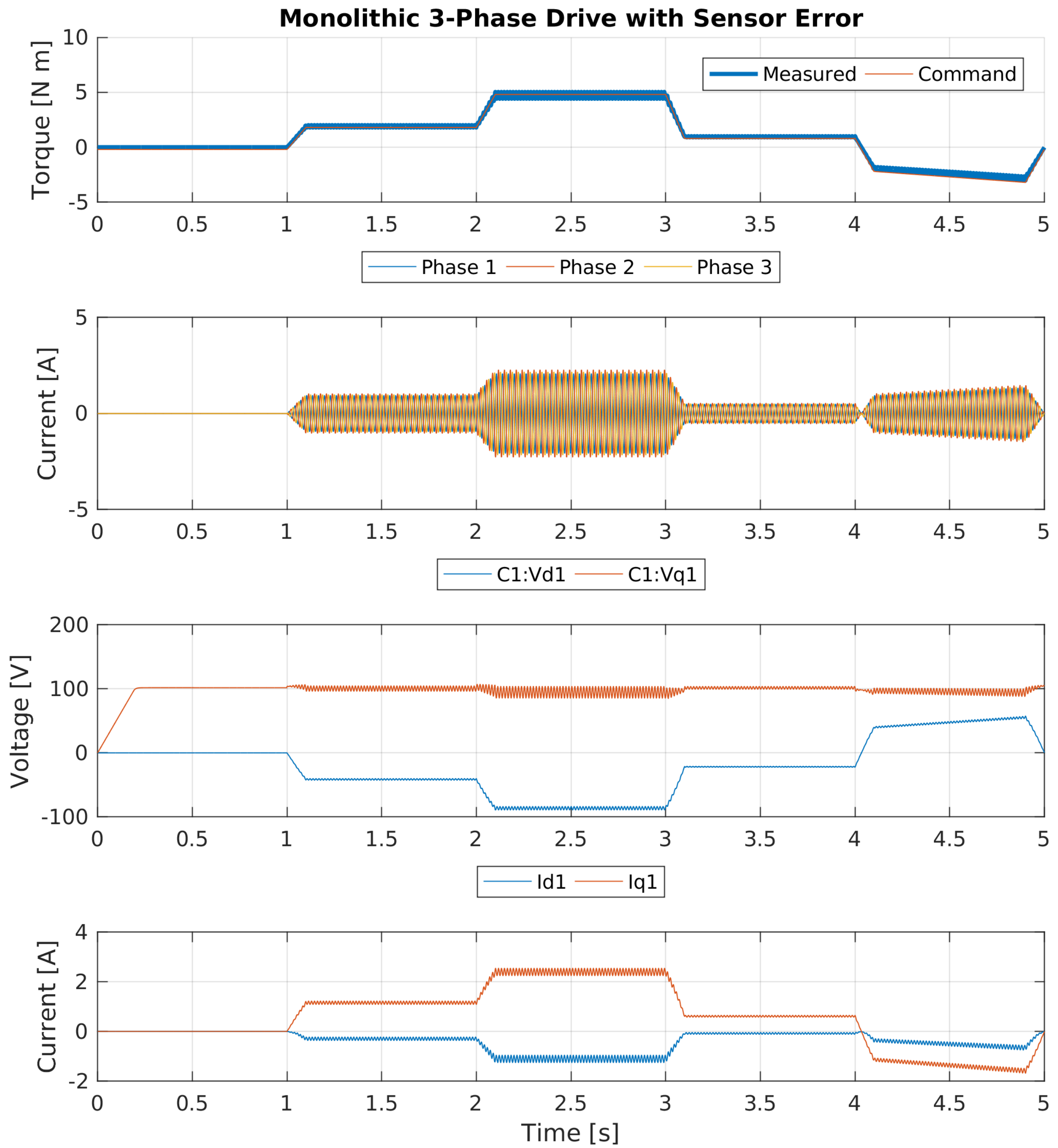


Figure 3.9 Simulation of three-phase vector controlled drive with a ten percent gain error in one current sensor. Torque, terminal currents, and controller command voltages are shown.

can also be seen in the current output since one of the phases has low and another, high current values compared to the nominal case presented in Figure 3.6. This current variation directly stems from the gain mismatch.

3.3.2 State Space Model

This machine is typically controlled by a complex vector synchronous frame PI current regulator with inputs of vector current error and outputs of command voltage. In equation form these equations add another pair of states (the d-axis and q-axis integrators) while incorporating several feedback and coupling terms. These cross-coupling terms disappear when transformed into the stationary frame since they derive from speed voltages. A virtual resistance is also added to the controller. This helps swamp changes in actual machine resistance and allows for more robust tuning of the control system.

$$V_d^* = -R_v I_d + K_p(I_d^* - I_d) + K_p \int \big(\tau_{PI}(I_d^* - I_d) - \omega_e(I_q^* - I_q)\big)\, dt \tag{3.16}$$

$$V_q^* = -R_v I_q + K_p(I_q^* - I_q) + K_p \int \big(\tau_{PI}(I_q^* - I_q) + \omega_e(I_d^* - I_d)\big)\, dt + \omega_e \lambda_{pm} \tag{3.17}$$

where R_v is the virtual resistance.

The aim is to have a state space model of this current regulator where

$$\begin{bmatrix} V_d^* \\ V_q^* \\ V_0^* \end{bmatrix} = -\mathbf{K}\left(\mathbf{G}\begin{bmatrix} i_d \\ i_q \\ V_{dint} \\ V_{qint} \end{bmatrix} + \mathbf{O}\right) + \mathbf{B}^*\mathbf{x}^* \tag{3.18}$$

with the V^* variables are the voltage commands applied to the machine, $\mathbf{K}$ is the control feedback matrix which takes a current sensor input described by the gain matrix $\mathbf{G}$ and sensor offset $\mathbf{O}$. The $\mathbf{K}$ matrix includes new rows beyond those used for the system model to handle the controller

integration states. The matrix $\mathbf{B}^*$ is used for coupling commands into the system. The lower right corner of $\mathbf{B}^*$ is an identity since any value can be input into the controller states.

$$\mathbf{K} = \begin{bmatrix} K_p + R_v & 0 & -1 & 0 \\ 0 & K_p + R_v & 0 & -1 \\ K_p \tau_{PI} & K_p - \omega & 0 & 0 \\ K_p \omega & K_p \tau_{PI} & 0 & 0 \end{bmatrix} \tag{3.19}$$

$$\mathbf{B}^* = \begin{bmatrix} K_p & 0 \\ 0 & K_p \\ K_p \tau_{PI} & K_p - \omega \\ K_p \omega & K_p \tau_{PI} \end{bmatrix} \tag{3.20}$$

Note that these matrices have the gain K_p repeated in several places. In order to simplify things for later analysis, this gain can be shifted out of the integration path with a simple change of variables a new $\mathbf{K}$ and $\mathbf{B}^*$.

$$\mathbf{K} = \begin{bmatrix} K_p + R_v & 0 & -K_p & 0 \\ 0 & K_p + R_v & 0 & -K_p \\ \tau_{PI} & -\omega & 0 & 0 \\ \omega & \tau_{PI} & 0 & 0 \end{bmatrix} \tag{3.21}$$

$$\mathbf{B}^* = \begin{bmatrix} K_p & 0 \\ 0 & K_p \\ \tau_{PI} & -\omega \\ \omega & \tau_{PI} \end{bmatrix} \tag{3.22}$$

Combining this with the above system gives a new $\mathbf{A}$ matrix including extra zeros for the new states.

$$\mathbf{A} = \begin{bmatrix} -\frac{r_s}{L_d} & \omega\frac{L_q}{L_d} & 0 & 0 \\ -\omega\frac{L_d}{L_q} & -\frac{r_s}{L_q} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \frac{1}{L_d} & 0 & 0 & 0 \\ 0 & \frac{1}{L_q} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.23}$$

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} - \mathbf{B}\mathbf{K}\mathbf{x} + \mathbf{B}\mathbf{B}^{*}\mathbf{x}^{*} \tag{3.24}$$

For the purpose of analysis this can be combined into a single matrix which contains all the feedback dynamics of the machine and drive.

$$\mathbf{A}^{+} = \mathbf{A} - \mathbf{B}\mathbf{K} = \begin{bmatrix} -\frac{K_p+R_v+r_s}{L_d} & \omega\frac{L_q}{L_d} & \frac{K_p}{L_d} & 0 \\ -\omega\frac{L_d}{L_q} & -\frac{K_p+R_v+r_s}{L_q} & 0 & \frac{K_p}{L_q} \\ -\tau_{PI} & \omega & 0 & 0 \\ -\omega & -\tau_{PI} & 0 & 0 \end{bmatrix} \tag{3.25}$$

$$\mathbf{B}^{+} = \mathbf{B}\mathbf{B}^{*} = \begin{bmatrix} \frac{K_p}{L_d} & 0 \\ 0 & \frac{K_p}{L_q} \\ \tau_{PI} & -\omega \\ \omega & \tau_{PI} \end{bmatrix} \tag{3.26}$$

Combining these regulators gives an overall state-space system which can be used check eigenvalues for stability and determine state controllability.

For controllability, simply ensure that the matrix formed by concatenating together the product of $\mathbf{B}^{+}$ and all powers of $\mathbf{A}^{+}$ is full rank as stated in equation 3.27.

$$\text{rank}(R) = \text{rank}\left(\begin{bmatrix} B^{+} & A^{+}B^{+} & {A^{+}}^{2}B^{+} & {A^{+}}^{3}B^{+} \end{bmatrix}\right) = 4 \tag{3.27}$$

In this case, the full matrix can be written as shown in equation 3.28 using the shorthand $R_{tot} = K_p + R_v + r_s$.

$$
R = \begin{bmatrix}
\frac{K_p}{L_d} & 0 & \frac{K_p \tau_{PI}}{L_d} - \frac{K_p R_{tot}}{L_d^2} & 0 & \frac{K_p \omega^2}{L_d} + \frac{K_p}{L_d}\left(-\frac{K_p \tau_{PI}}{L_d} - \omega^2 + \frac{R_{tot}^2}{L_d^2}\right) - \frac{K_p R_{tot}}{L_d^2}\tau_{PI} & \frac{K_p}{L_q}\left(\frac{K_p \omega}{L_d} - \frac{R_{tot}\omega}{L_d} - \frac{L_q R_{tot}}{L_d^2}\omega\right) + \frac{K_p \omega}{L_d}\tau_{PI} + \frac{K_p R_{tot}}{L_d^2}\omega & \frac{K_p \omega}{L_q}\left(\frac{K_p \omega}{L_d} - \frac{R_{tot}\omega}{L_d} - \frac{L_q R_{tot}}{L_d^2}\omega\right) + \frac{K_p \tau_{PI}}{L_d}\left(-\frac{K_p \tau_{PI}}{L_d} - \omega^2 + \frac{R_{tot}^2}{L_d^2}\right) + \frac{K_p}{L_d}\left(-\frac{K_p \omega^2}{L_d} + \frac{K_p R_{tot}}{L_d^2}\tau_{PI} - \frac{L_d \omega}{L_q}\left(\frac{K_p \omega}{L_d} - \frac{R_{tot}\omega}{L_d} - \frac{L_q R_{tot}}{L_d^2}\omega\right) - \frac{R_{tot}}{L_d}\left(-\frac{K_p \tau_{PI}}{L_d} - \omega^2 + \frac{R_{tot}^2}{L_d^2}\right)\right) & \frac{K_p \tau_{PI}}{L_q}\left(\frac{K_p \omega}{L_d} - \frac{R_{tot}\omega}{L_d} - \frac{L_q R_{tot}}{L_d^2}\omega\right) + \frac{K_p}{L_q}\left(-\frac{K_p \omega}{L_d}\tau_{PI} - \frac{K_p R_{tot}}{L_d^2}\omega - \frac{R_{tot}}{L_q}\left(\frac{K_p \omega}{L_d} - \frac{R_{tot}\omega}{L_d} - \frac{L_q R_{tot}}{L_d^2}\omega\right) + \frac{L_q \omega}{L_d}\left(-\frac{K_p \tau_{PI}}{L_d} - \omega^2 + \frac{R_{tot}^2}{L_d^2}\right)\right) - \frac{K_p \omega}{L_d}\left(-\frac{K_p \tau_{PI}}{L_d} - \omega^2 + \frac{R_{tot}^2}{L_d^2}\right) \\
0 & \frac{K_p}{L_q} & 0 & \frac{K_p \tau_{PI}}{L_q} - \frac{K_p R_{tot}}{L_q^2} & -\frac{K_p \omega}{L_q}\tau_{PI} - \frac{K_p R_{tot}}{L_q^2}\omega + \frac{K_p}{L_d}\left(-\frac{K_p \omega}{L_q} + \frac{L_d R_{tot}}{L_q^2}\omega + \frac{R_{tot}\omega}{L_q}\right) & \frac{K_p \omega^2}{L_q} + \frac{K_p}{L_q}\left(-\frac{K_p \tau_{PI}}{L_q} - \omega^2 + \frac{R_{tot}^2}{L_q^2}\right) - \frac{K_p R_{tot}}{L_q^2}\tau_{PI} & \frac{K_p \omega}{L_q}\left(-\frac{K_p \tau_{PI}}{L_q} - \omega^2 + \frac{R_{tot}^2}{L_q^2}\right) + \frac{K_p \tau_{PI}}{L_d}\left(-\frac{K_p \omega}{L_q} + \frac{L_d R_{tot}}{L_q^2}\omega + \frac{R_{tot}\omega}{L_q}\right) + \frac{K_p}{L_d}\left(\frac{K_p \omega}{L_q}\tau_{PI} + \frac{K_p R_{tot}}{L_q^2}\omega - \frac{L_d \omega}{L_q}\left(-\frac{K_p \tau_{PI}}{L_q} - \omega^2 + \frac{R_{tot}^2}{L_q^2}\right) - \frac{R_{tot}}{L_d}\left(-\frac{K_p \omega}{L_q} + \frac{L_d R_{tot}}{L_q^2}\omega + \frac{R_{tot}\omega}{L_q}\right)\right) & \frac{K_p \tau_{PI}}{L_q}\left(-\frac{K_p \tau_{PI}}{L_q} - \omega^2 + \frac{R_{tot}^2}{L_q^2}\right) + \frac{K_p}{L_q}\left(-\frac{K_p \omega^2}{L_q} + \frac{K_p R_{tot}}{L_q^2}\tau_{PI} - \frac{R_{tot}}{L_q}\left(-\frac{K_p \tau_{PI}}{L_q} - \omega^2 + \frac{R_{tot}^2}{L_q^2}\right) + \frac{L_q \omega}{L_d}\left(-\frac{K_p \omega}{L_q} + \frac{L_d R_{tot}}{L_q^2}\omega + \frac{R_{tot}\omega}{L_q}\right)\right) - \frac{K_p \omega}{L_d}\left(-\frac{K_p \omega}{L_q} + \frac{L_d R_{tot}}{L_q^2}\omega + \frac{R_{tot}\omega}{L_q}\right) \\
\tau_{PI} & -\omega & -\frac{K_p \tau_{PI}}{L_d} & \frac{K_p \omega}{L_q} & \frac{K_p \omega^2}{L_q} - \frac{K_p \tau_{PI}^2}{L_d} + \frac{K_p}{L_d}\left(-\frac{L_d \omega^2}{L_q} + \frac{R_{tot}\tau_{PI}}{L_d}\right) & \frac{K_p \omega}{L_q}\tau_{PI} + \frac{K_p}{L_q}\left(-\frac{R_{tot}\omega}{L_q} - \frac{L_q \omega}{L_d}\tau_{PI}\right) + \frac{K_p \omega}{L_d}\tau_{PI} & \frac{K_p \omega}{L_q}\left(-\frac{R_{tot}\omega}{L_q} - \frac{L_q \omega}{L_d}\tau_{PI}\right) + \frac{K_p \tau_{PI}}{L_d}\left(-\frac{L_d \omega^2}{L_q} + \frac{R_{tot}\tau_{PI}}{L_d}\right) + \frac{K_p}{L_d}\left(-\frac{K_p \omega^2}{L_q} + \frac{K_p \tau_{PI}^2}{L_d} - \frac{L_d \omega}{L_q}\left(-\frac{R_{tot}\omega}{L_q} - \frac{L_q \omega}{L_d}\tau_{PI}\right) - \frac{R_{tot}}{L_d}\left(-\frac{L_d \omega^2}{L_q} + \frac{R_{tot}\tau_{PI}}{L_d}\right)\right) & \frac{K_p \tau_{PI}}{L_q}\left(-\frac{R_{tot}\omega}{L_q} - \frac{L_q \omega}{L_d}\tau_{PI}\right) + \frac{K_p}{L_q}\left(-\frac{K_p \omega}{L_q}\tau_{PI} - \frac{K_p \omega}{L_d}\tau_{PI} - \frac{R_{tot}}{L_q}\left(-\frac{R_{tot}\omega}{L_q} - \frac{L_q \omega}{L_d}\tau_{PI}\right) + \frac{L_q \omega}{L_d}\left(-\frac{L_d \omega^2}{L_q} + \frac{R_{tot}\tau_{PI}}{L_d}\right)\right) - \frac{K_p \omega}{L_d}\left(-\frac{L_d \omega^2}{L_q} + \frac{R_{tot}\tau_{PI}}{L_d}\right) \\
\omega & \tau_{PI} & -\frac{K_p \omega}{L_d} & -\frac{K_p \tau_{PI}}{L_q} & -\frac{K_p \omega}{L_q}\tau_{PI} - \frac{K_p \omega}{L_d}\tau_{PI} + \frac{K_p}{L_d}\left(\frac{L_d \omega}{L_q}\tau_{PI} + \frac{R_{tot}\omega}{L_d}\right) & -\frac{K_p \tau_{PI}^2}{L_q} + \frac{K_p}{L_q}\left(\frac{R_{tot}\tau_{PI}}{L_q} - \frac{L_q \omega^2}{L_d}\right) + \frac{K_p \omega^2}{L_d} & \frac{K_p \omega}{L_q}\left(\frac{R_{tot}\tau_{PI}}{L_q} - \frac{L_q \omega^2}{L_d}\right) + \frac{K_p \tau_{PI}}{L_d}\left(\frac{L_d \omega}{L_q}\tau_{PI} + \frac{R_{tot}\omega}{L_d}\right) + \frac{K_p}{L_d}\left(\frac{K_p \omega}{L_q}\tau_{PI} + \frac{K_p \omega}{L_d}\tau_{PI} - \frac{L_d \omega}{L_q}\left(\frac{R_{tot}\tau_{PI}}{L_q} - \frac{L_q \omega^2}{L_d}\right) - \frac{R_{tot}}{L_d}\left(\frac{L_d \omega}{L_q}\tau_{PI} + \frac{R_{tot}\omega}{L_d}\right)\right) & \frac{K_p \tau_{PI}}{L_q}\left(\frac{R_{tot}\tau_{PI}}{L_q} - \frac{L_q \omega^2}{L_d}\right) + \frac{K_p}{L_q}\left(\frac{K_p \tau_{PI}^2}{L_q} - \frac{K_p \omega^2}{L_d} - \frac{R_{tot}}{L_q}\left(\frac{R_{tot}\tau_{PI}}{L_q} - \frac{L_q \omega^2}{L_d}\right) + \frac{L_q \omega}{L_d}\left(\frac{L_d \omega}{L_q}\tau_{PI} + \frac{R_{tot}\omega}{L_d}\right)\right) - \frac{K_p \omega}{L_d}\left(\frac{L_d \omega}{L_q}\tau_{PI} + \frac{R_{tot}\omega}{L_d}\right)
\end{bmatrix} \quad (3.28)
$$

This of course quickly becomes unnecessarily tedious since the first four columns of this matrix provide the full rank needed which shows that all machine and controller states can be fully controlled from the i_d^* and i_q^* inputs.

This model is simplified and assumes the rotor speed is constant. If the model is expanded in order to include a variable rotor speed, the augmented matrices then become

$$\mathbf{A}^{+} = \begin{bmatrix} -\frac{1}{L_d}\left(K_p + R_v + r_s\right) & \frac{L_q \omega_o}{L_d} & \frac{K_p}{L_d} & 0 & I_{qo} \\ -\frac{L_d \omega_o}{L_q} & -\frac{1}{L_q}\left(K_p + R_v + r_s\right) & 0 & \frac{K_p}{L_q} & -I_{do} - \frac{\lambda_{pm}}{L_d} \\ -\tau_{PI} & \omega_o & 0 & 0 & 0 \\ -\omega_o & -\tau_{PI} & 0 & 0 & 0 \\ -\frac{I_{qo} L_q}{J} & \frac{1}{J}\left(I_{do} L_d + \lambda_{pm}\right) & 0 & 0 & -\frac{1}{\tau_m} \end{bmatrix} \tag{3.29}$$

$$\mathbf{B}^{+} = \begin{bmatrix} \frac{K_p}{L_d} & 0 & 0 \\ 0 & \frac{K_p}{L_q} & 0 \\ \tau_{PI} & -\omega_o & 0 \\ \omega_o & \tau_{PI} & 0 \\ 0 & 0 & -\frac{1}{J} \end{bmatrix} \tag{3.30}$$

where the added state is the change in rotor speed. This results in a rank matrix which is spanned by its first five columns

$$\mathbf{R}_{1-5} = \begin{bmatrix} \frac{K_p}{L_d} & 0 & 0 & \frac{K_p}{L_d^2}\left(L_d \tau_{PI} - R_{tot}\right) & 0 \\ 0 & \frac{K_p}{L_q} & 0 & 0 & \frac{K_p}{L_q^2}\left(L_q \tau_{PI} - R_{tot}\right) \\ \tau_{PI} & -\omega_o & 0 & -\frac{K_p \tau_{PI}}{L_d} & \frac{K_p \omega_o}{L_q} \\ \omega_o & \tau_{PI} & 0 & -\frac{K_p \omega_o}{L_d} & -\frac{K_p \tau_{PI}}{L_q} \\ 0 & 0 & -\frac{1}{J} & -\frac{I_{qo} K_p L_q}{J L_d} & \frac{K_p}{J L_q}\left(I_{do} L_d + \lambda_{pm}\right) \end{bmatrix} \tag{3.31}$$

thus showing that the system is controllable with similar response even if the rotor speed varys.

Thus far, the model has assumed ideal current sensing. For the analysis that follows, current sensor imbalance will be used to introduce an asymmetry into the system. This choice should not

be assumed to mean that current sensors are the primary source of system imbalance, but merely provide an easily adjustable asymmetry that is sufficient to cause a distributed controller to show the effects of its over constrained structure. In reality the current feedback is often formed using less than the full set of current sensors and these sensors have offset and gain errors.

This can be depicted in the system above by adding a pair of matrices into the feedback equation.

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} - \mathbf{B}\mathbf{K}\left(\mathbf{G}\mathbf{x} + \mathbf{O}\right) + \mathbf{B}\mathbf{B}^{*}\mathbf{x}^{*} \tag{3.32}$$

These error models with gain $\mathbf{G}$ and offset $\mathbf{O}$ will be left alone for a moment noting only that these matrices are position dependent in the synchronous reference frame because the current sensors are stationary.

Thus far all this control development has been strictly in the synchronous dq frame. With the introduction of current sensor errors, it makes sense to translate this into the stationary frame and furthermore, to break it out into phase variables. This can be done by inserting modified Clarke transforms between the states and the controller.

3.4 Current Sensor Errors

Thus far, this analysis has dealt only with ideal feedback and perfect symmetry in the control system. In reality, machines and drives are never symmetrical and parameters are never known exactly. To make matters worse, current sensors are anything but ideal. The remainder of this work will use a more realistic current sensor model incorporating both gain and offset errors:

$$\hat{\imath} = Gi + O \tag{3.33}$$

In this case an ideal sensor has parameters $G = 1$ and $O = 0$. The gain error can be described as a relative term, in this case the the difference between the gain G and 1. When describing the quantities involved in a drive, all of these quantities will be subscripted.

This analysis will start with a full-order system with current sensors in all three phases. In this case the gain and offset are described as a $\mathbf{C}$ and $\mathbf{D}$ state space matrix.

$$\begin{bmatrix} \hat{i_a} \\ \hat{i_b} \\ \hat{i_c} \end{bmatrix} = \begin{bmatrix} G_a & 0 & 0 \\ 0 & G_b & 0 \\ 0 & 0 & G_c \end{bmatrix} \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} + \begin{bmatrix} O_a \\ O_b \\ O_c \end{bmatrix} \tag{3.34}$$

This is directly the gain and offset matrix.

If this gain is transformed into the $\alpha\beta n$ complex reference frame, the current sense errors cause cross-coupling between axes. In the stationary frame, the offset errors only cause a fixed offset in the complex plane.

$$G_{\alpha\beta n} = \mathbf{C}^{-1} G_{abc} \mathbf{C} = \begin{bmatrix} \frac{2G_a}{3} + \frac{G_b}{6} + \frac{G_c}{6} & -\frac{\sqrt{3}G_b}{6} + \frac{\sqrt{3}G_c}{6} & \frac{\sqrt{2}G_a}{3} - \frac{\sqrt{2}G_b}{6} - \frac{\sqrt{2}G_c}{6} \\ -\frac{\sqrt{3}G_b}{6} + \frac{\sqrt{3}G_c}{6} & \frac{G_b}{2} + \frac{G_c}{2} & \frac{\sqrt{6}G_b}{6} - \frac{\sqrt{6}G_c}{6} \\ \frac{\sqrt{2}G_a}{3} - \frac{\sqrt{2}G_b}{6} - \frac{\sqrt{2}G_c}{6} & \frac{\sqrt{6}G_b}{6} - \frac{\sqrt{6}G_c}{6} & \frac{G_a}{3} + \frac{G_b}{3} + \frac{G_c}{3} \end{bmatrix} \tag{3.35}$$

$$O_{\alpha\beta n} = C^{-1} O_{abc} = \begin{bmatrix} \frac{\sqrt{6}O_a}{3} - \frac{\sqrt{6}O_b}{6} - \frac{\sqrt{6}O_c}{6} \\ \frac{\sqrt{2}O_b}{2} - \frac{\sqrt{2}O_c}{2} \\ \frac{\sqrt{3}O_a}{3} + \frac{\sqrt{3}O_b}{3} + \frac{\sqrt{3}O_c}{3} \end{bmatrix} \tag{3.36}$$

In the stationary frame, the most interesting thing about this error is that active rotational currents couple into the zero-sequence and zero-sequence currents couple into the rotational frames. Since there is no neutral connection in a wye-connected system, no true zero-sequence currents can flow so these terms can be used to help detect sensor gain and offset errors in order to improve system tuning.

The controller however, is not operating in the stationary frame, but in the synchronous frame. If these gain and offsets are translated into and out of the synchronous frame, the sensor errors form

a much more interesting set of equations.

$$
\begin{aligned}
G_{dq0} &= \mathbf{R}^{-1}(\theta) G_{\alpha\beta n} \mathbf{R}(\theta) \\
&= \begin{bmatrix}
\frac{1}{3}\left(-G_a \cos(2\theta) + G_b \sin\left(2\theta + \frac{\pi}{6}\right) + G_c \cos\left(2\theta + \frac{\pi}{3}\right)\right) & \frac{1}{3}\left(G_a \sin(2\theta) + G_b \cos\left(2\theta + \frac{\pi}{6}\right) - G_c \sin\left(2\theta + \frac{\pi}{3}\right)\right) & \frac{\sqrt{2}}{3}\left(G_a \sin(\theta) - G_b \sin\left(\theta + \frac{\pi}{3}\right) + G_c \cos\left(\theta + \frac{\pi}{6}\right)\right) \\
\frac{1}{3}\left(G_a \sin(2\theta) + G_b \cos\left(2\theta + \frac{\pi}{6}\right) - G_c \sin\left(2\theta + \frac{\pi}{3}\right)\right) & \frac{1}{3}\left(G_a \cos(2\theta) - G_b \sin\left(2\theta + \frac{\pi}{6}\right) - G_c \cos\left(2\theta + \frac{\pi}{3}\right)\right) & \frac{\sqrt{2}}{3}\left(G_a \cos(\theta) - G_b \cos\left(\theta + \frac{\pi}{3}\right) - G_c \sin\left(\theta + \frac{\pi}{6}\right)\right) \\
\left(-\frac{\sqrt{6}G_b}{6} + \frac{\sqrt{6}G_c}{6}\right)\cos(\theta) + \left(\frac{\sqrt{2}G_a}{3} - \frac{\sqrt{2}G_b}{6} - \frac{\sqrt{2}G_c}{6}\right)\sin(\theta) & \left(\frac{\sqrt{6}G_b}{6} - \frac{\sqrt{6}G_c}{6}\right)\sin(\theta) + \left(\frac{\sqrt{2}G_a}{3} - \frac{\sqrt{2}G_b}{6} - \frac{\sqrt{2}G_c}{6}\right)\cos(\theta) & 0
\end{bmatrix} \\
&+ \begin{bmatrix}
\frac{G_a+G_b+G_c}{3} & 0 & 0 \\
0 & \frac{G_a+G_b+G_c}{3} & 0 \\
0 & 0 & \frac{G_a+G_b+G_c}{3}
\end{bmatrix}
\end{aligned}
\tag{3.37}
$$

$$
O_{dq0} = \mathbf{R}^{-1}(\theta) O_{\alpha\beta n} = \begin{bmatrix}
\frac{\sqrt{6}}{3}\left(O_a \sin(\theta) - O_b \sin\left(\theta + \frac{\pi}{3}\right) + O_c \cos\left(\theta + \frac{\pi}{6}\right)\right) \\
\frac{\sqrt{6}}{3}\left(O_a \cos(\theta) - O_b \cos\left(\theta + \frac{\pi}{3}\right) - O_c \sin\left(\theta + \frac{\pi}{6}\right)\right) \\
\frac{\sqrt{3}O_a}{3} + \frac{\sqrt{3}O_b}{3} + \frac{\sqrt{3}O_c}{3}
\end{bmatrix}
\tag{3.38}
$$

In this reference frame there are some encouraging and some discouraging components. There is a twice-frequency cross coupling between the active currents proportional to the gain mismatch. This can only really be decoupled with a filter or observer with significantly slower bandwidth than the synchronous speed. If this filter is placed in the control loop, the system control dynamics will be severely compromised. This is acceptable in blower and bulk pumping but less so in traction and completely unusable for servo applications. Thankfully, offsets and gain errors also manifest in the zero-sequence and the coupling into and out of the zero sequence from the active currents

is spectrally separated by the synchronous frequency and known to be zero at the synchronous frequency and all low harmonics so the zero sequence coupling can be used to decouple gain errors if no real zero-sequence current exists as described in [22].

In the three-phase case, often only two sensors are used. This case starts with a C and D matrix with affine errors.

$$\begin{bmatrix} \hat{i_a} \\ \hat{i_b} \end{bmatrix} = \begin{bmatrix} G_a & 0 & 0 \\ 0 & G_b & 0 \end{bmatrix} \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} + \begin{bmatrix} O_a \\ O_b \end{bmatrix} \tag{3.39}$$

Using these to reconstruct the three phase currents gives

$$G_{abc} = \begin{bmatrix} G_a & 0 & 0 \\ 0 & G_b & 0 \\ -G_a & -G_b & 0 \end{bmatrix} \qquad O_{abc} = \begin{bmatrix} O_a \\ O_b \\ -O_a - O_b \end{bmatrix} \tag{3.40}$$

When these are transformed back into the synchronous frame, there is a position dependent error term.

$$G_{\alpha\beta n} = \mathbf{C}^{-1} G_{abc} \mathbf{C} = \begin{bmatrix} G_a & 0 & \frac{G_a}{\sqrt{2}} \\ \frac{G_a - G_b}{\sqrt{3}} & G_b & \frac{G_a}{\sqrt{6}} + \frac{\sqrt{2} G_b}{\sqrt{3}} \\ 0 & 0 & 0 \end{bmatrix} \qquad O_{\alpha\beta n} = \begin{bmatrix} \frac{\sqrt{3}}{\sqrt{2}} O_a \\ \frac{O_a + 2O_b}{\sqrt{2}} \\ 0 \end{bmatrix} \tag{3.41}$$

$$G_{dq0} = \mathbf{R}^{-1}(\theta) G_{\alpha\beta n} \mathbf{R}(\theta)$$

$$= \sin 2\theta \begin{bmatrix} \frac{G_a - G_b}{2\sqrt{3}} & 2\frac{G_a - G_b}{3} & 0 \\ 2\frac{G_a - G_b}{3} & \frac{G_a - G_b}{2\sqrt{3}} & 0 \\ 0 & 0 & 0 \end{bmatrix} + \cos 2\theta \begin{bmatrix} -\frac{G_a - G_b}{2} & -\frac{G_a - G_b}{2\sqrt{3}} & 0 \\ -\frac{G_a - G_b}{2\sqrt{3}} & \frac{G_a - G_b}{2} & 0 \\ 0 & 0 & 0 \end{bmatrix} + \tag{3.42}$$

$$\begin{bmatrix} \frac{G_a + G_b}{2} & -\frac{G_a - G_b}{2\sqrt{3}} & \frac{G_a}{\sqrt{2}} \sin\theta - \frac{G_a + 2G_b}{\sqrt{6}} \cos\theta \\ \frac{G_a - G_b}{2\sqrt{3}} & \frac{G_a + G_b}{2} & \frac{G_a}{\sqrt{2}} \cos\theta + \frac{G_a + 2G_b}{\sqrt{6}} \sin\theta \\ 0 & 0 & 0 \end{bmatrix}$$

$$O_{dq0} = \begin{bmatrix} \frac{\sqrt{3}}{\sqrt{2}} O_a \sin\theta - \frac{O_a + 2O_b}{\sqrt{2}} \cos\theta \\ \frac{\sqrt{3}}{\sqrt{2}} O_a \cos\theta + \frac{O_a + 2O_b}{\sqrt{2}} \sin\theta \\ 0 \end{bmatrix} \tag{3.43}$$

Of particular note here is how a real zero-sequence current will be mapped into the active complex frame because the sensing equations assume no zero-sequence current. While this is not a problem for the standard three-phase wye-connected drive, in a fault tolerant system the ability to operate when zero-sequence currents are flowing due to either ground faults or neutral-point control for fault mitigation is needed.

If the average of this set of feedback equations is applied to the $\mathbf{A}^+$ matrix derived above, the following new system equation shown in equation 3.44 is derived.

$$[\mathbf{A}^+_{\mathbf{error}}] = [\mathbf{A}] - [\mathbf{B}][\mathbf{K}] \begin{bmatrix} [\mathbf{G_{dq}}] & [\mathbf{0}] \\ [\mathbf{0}] & [\mathbf{I_{2x2}}] \end{bmatrix}$$

$$= \begin{bmatrix} -\frac{1}{L_d}\left(r_s + \frac{1}{2}(G_a + G_b)(K_p + R_v)\right) & \frac{1}{L_d}\left(L_q\omega + \frac{\sqrt{3}}{6}(G_a - G_b)(K_p + R_v)\right) & \frac{K_p}{L_d} & 0 \\ -\frac{1}{L_q}\left(L_d\omega + \frac{\sqrt{3}}{6}(G_a - G_b)(K_p + R_v)\right) & -\frac{1}{L_q}\left(r_s + \frac{1}{2}(G_a + G_b)(K_p + R_v)\right) & 0 & \frac{K_p}{L_q} \\ \frac{\sqrt{3}\omega}{6}(G_a - G_b) - \frac{\tau_{PI}}{2}(G_a + G_b) & \frac{\omega}{2}(G_a + G_b) + \frac{\sqrt{3}\tau_{PI}}{6}(G_a - G_b) & 0 & 0 \\ -\frac{\omega}{2}(G_a + G_b) - \frac{\sqrt{3}\tau_{PI}}{6}(G_a - G_b) & \frac{\sqrt{3}\omega}{6}(G_a - G_b) - \frac{\tau_{PI}}{2}(G_a + G_b) & 0 & 0 \end{bmatrix} \tag{3.44}$$

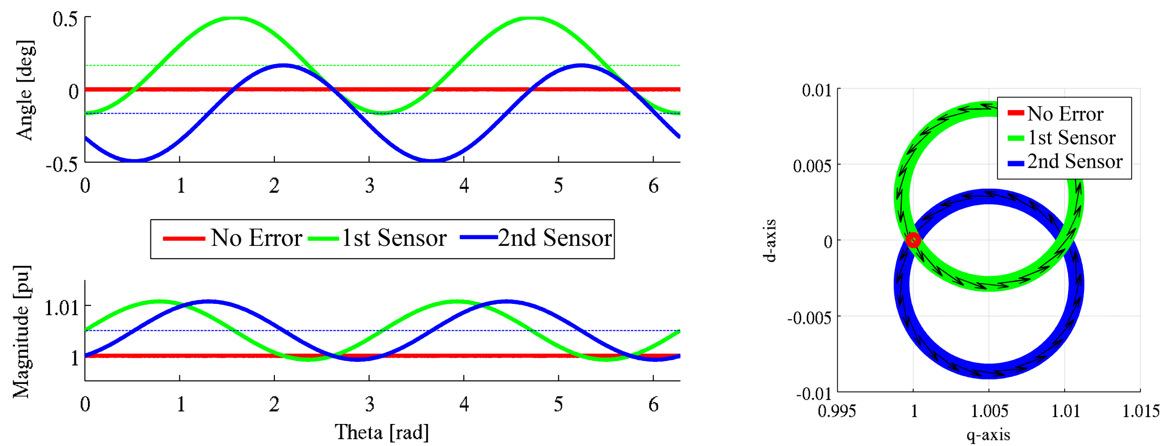


Figure 3.10 Synchronous reference frame current sensor response as a function of current vector angle with 1% gain error in one sensor.

In a standard monolithic 3-phase drive, the apparent neutral currents don't have any significant effect since the neutral circuit is not excited due to how voltages are generated with a fixed zero-sequence output.

In order to help visualize these equations, the result of a one percent gain error in one of the two current sensors is shown in figure 3.10 for a 1pu commanded rotating current vector oriented along the q-axis where either the leading or lagging sensor has a gain 1 percent too high. A third trace is added in each of the figures for the baseline case of errors in either of the sensors. Of particular note here is that the actual current vector amplitude has a second harmonic component for the two cases with a 1 percent sensor error as seen in the current amplitude and phase waveforms on the left side of figure 3.10 ,while the current vector has a constant amplitude of 1 and a constant phase angle of zero deg when the currents sensors are ideal. The right half of Figure 3.10 shows the circular trajectories of the current vector terminus point in the dq phase plane for the two cases with sensor errors, while the red fixed dot corresponds to the case with correct gains. The green and blue circles show that the average current vector is offset both in amplitude and phase due to the gain error while the measured current orbits this average value at twice synchronous frequency.

3.5 Conclusions

This chapter has reviewed and discussed the performance of a monolithic three-phase drive to serve as a baseline for the research results presented in the remainder of this document. The performance of this drive has been shown for both the ideal current feedback cases and more realistic cases that include the presence of current feedback with gain errors.

The effect of current sensor errors was then investigated in various reference frames. The current vector response to a current sensor error viewed in the synchronous reference frame has been shown to suffer from a constant phase and amplitude offset combined with a rotating error component that has an angular frequency twice the synchronous frequency. Despite these error components, the conventional three-phase machine drive with a monolithic controller behaves very stably in the presence of current sensor errors, even with non-zero values of the integral gains in the complex vector synchronous-frame current regulator.

Chapter 4

Machine Drives with Distributed Controllers

4.0.1 Introduction

The task now is to split the vector-controlled drive and its monolithic controller presented in the previous chapter into a distributed set of phase-drive units, each with its own modular controller. Since the goal is to maintain the performance of the monolithic vector controller with a minimum of added complexity or communications, the most straightforward approach is to simply replicate the same controller for each modular phase drive, isolating the appropriate output for each modular phase drive while feeding each of these drive units with the same torque command input.

This controller duplication will cause the overall control system to be over-constrained since there are n modular controllers attempting to control $(n-1)$ independent current variables because of the floating-wye connection. It will be shown that the presence of an integrator term in the each current regulator PI gain stage will create current regulator stability problems under these circumstances.

4.1 Independent Per-Phase Wye-Connected Drives

In order to implement a distributed drive, one can start with the drive structure used in the previous chapter. Instead of a full three-phase bridge, this bridge can be broken into three parts so that each modular drive consists of a single half-bridge phase leg along with its DC-link capacitor as shown in Figure 4.1. The drive modules share a DC-link so their relative voltages are fixed. This

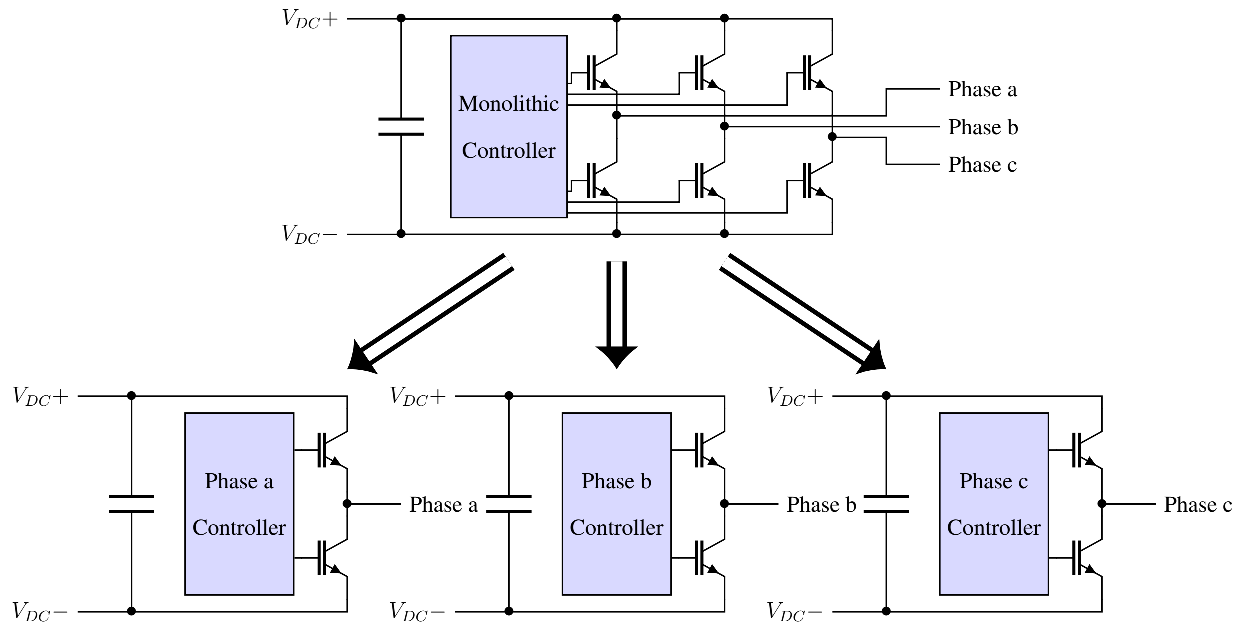


Figure 4.1 Diagram showing a three phase drive broken up into half-bridge distributed segments.

structure is mathematically identical to the combined three-phase drive shown at the top of Figure 4.1 in the case that all three of the controllers are fed the same feedback and command information since the new control states are mathematically identical to the states in the three-phase drive with a monolithic controller discussed in the preceding chapter. A schematic representation of how this drive is structured is shown in figure 4.2. In order to introduce show the over constraint challenge caused by the distributed controller, a 10 percent error is added to one of the current sensors and a separate set of current sensors is chosen for each phase controller. While this decision is arbitrary, it provides sufficient mismatch to show how the newly introduced controller states behave in the presence of a nearly ideal but slightly asymmetric system in a way that can be easily implemented in hardware. The 10 percent current sensor gain error is large specifically in order to swamp the asymmetries caused by other system parameters such as rotor and winding asymmetry.

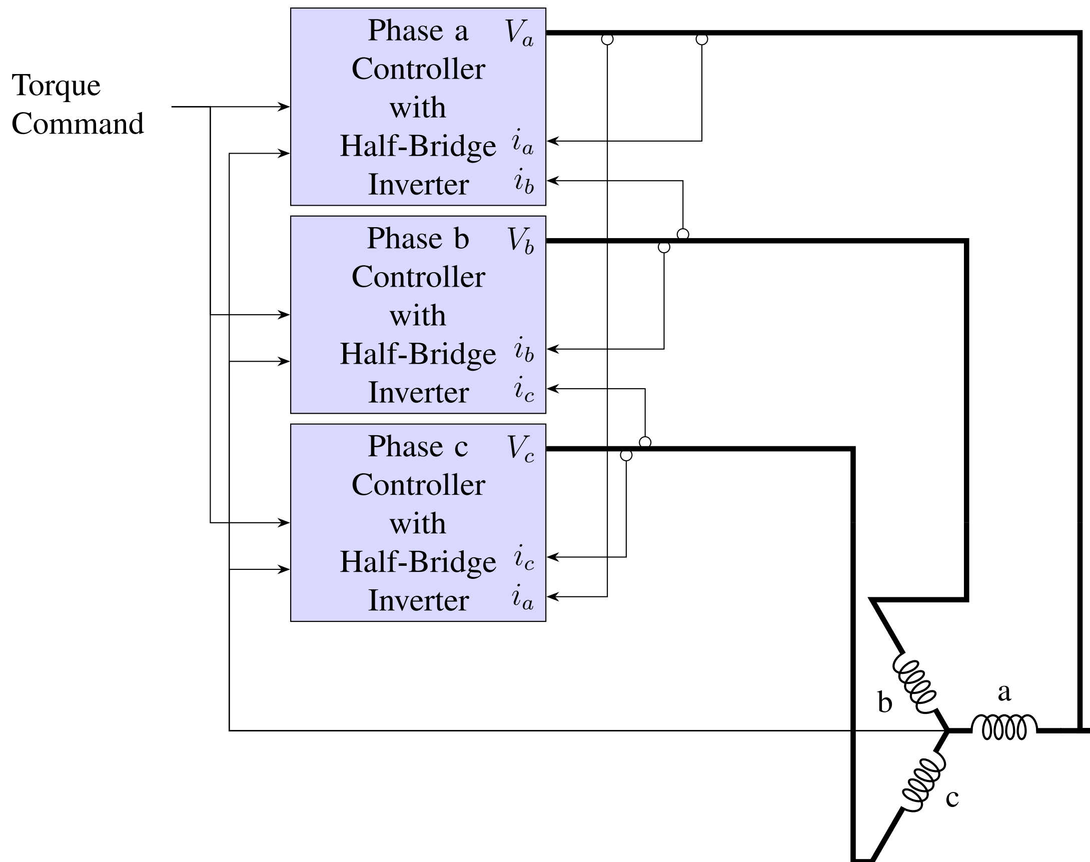


Figure 4.2 Schematic of a distributed drive controlling a three-phase machine with two current sensors per phase controller.

4.1.1 Sensing Needs

While having only a phase module's own current sensor is technically possible through the use of a flux observer structure [30], this research focuses on the case with two current sensors for each phase module in order to obtain both magnitude and phase information. Position feedback is provided by the use of an encoder which is fed to all drive modules in parallel. While working to remove this position sensing requirement would be an interesting topic of research, it is beyond the scope for this document.

4.1.2 Speed Control

The controller development in this work has focused on torque commands being the input to each of the distributed controllers. Often in industrial processes, speed is the more important variable. As this work is depending on shared encoder position feedback, speed control has no difference from a monolithic controller when distributed. Simulations of speed control did not yield any interesting results and aren't included here. That being said, when position self-sensing control is developed in future work, the position feedback mismatch between phases will cause challenges for distributed speed control. This challenge is however somewhat tempered by the slower dynamics and limited states of a speed controller as there is only one shaft and mechanical response in several decades slower than electrical time constants.

4.1.3 Simulation

For the simulation of a distributed drive, start by copying the vector control structure used in the previous chapter as shown in figure 3.5. For the distributed drive, each module simply outputs only one phase voltage and operates with an offset position as discussed in the general case in the next chapter. The same torque command sequence is used as before. For the three phase case, the same machine model as figure 3.4 with the same speed-controlled dynamometer is used in order to provide a stable comparison.

With no added feedback, the machine torque and voltage waveforms are shown in Figure 4.3. The controller command voltages and currents are shown in figure 4.4. The current response shows identical current waveforms to the monolithic drive from chapter 3. This is a hopeful sign that this may be the right path to achieve a high-performance distributed drive.

Sadly, the voltage commands in Figure 4.3 exhibit the characteristics of continuous integration errors in all of the voltage commands for all three of the modular controllers. This error is due to the lack of feedback on the differential states between the controllers which is being driven by

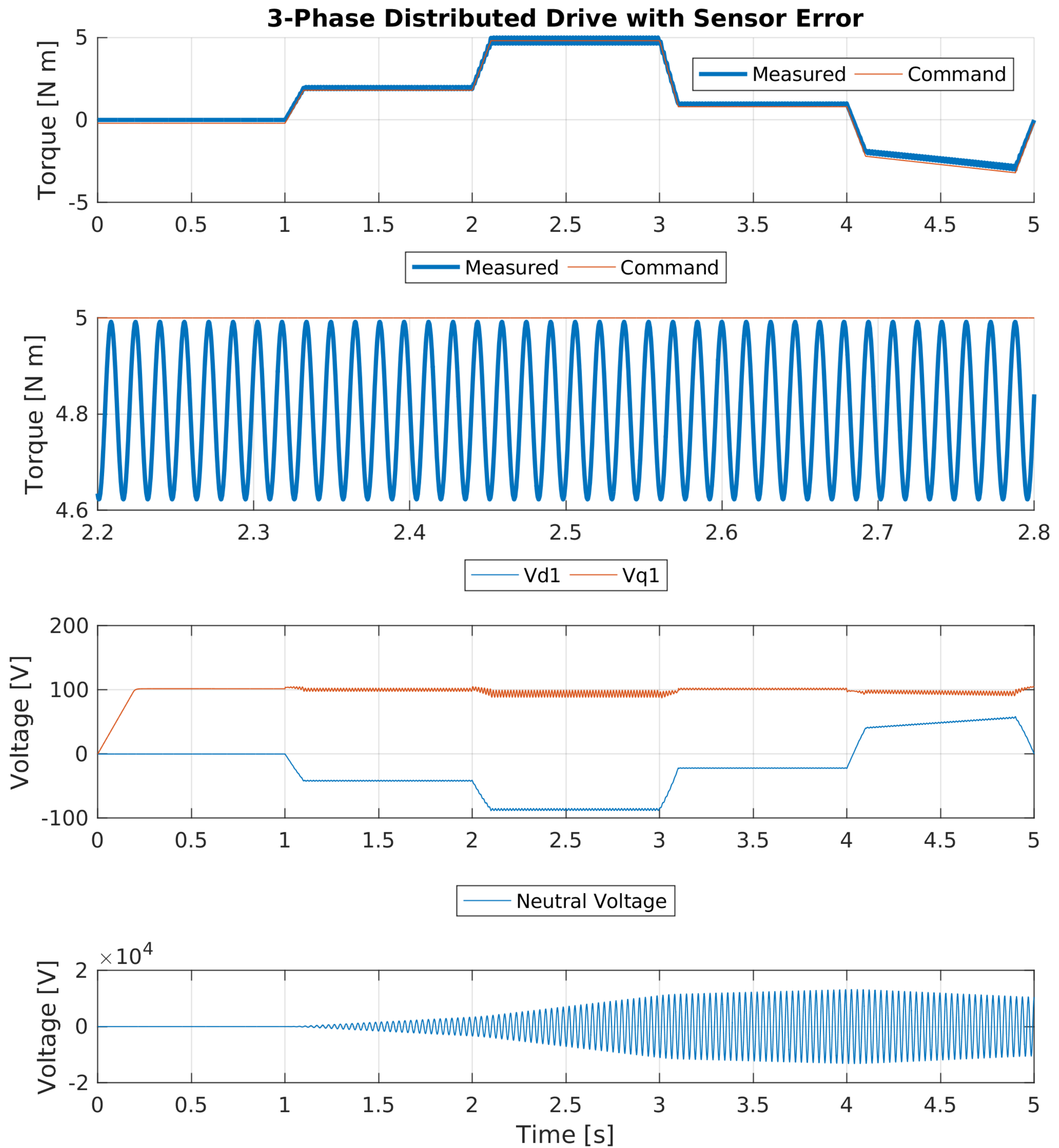


Figure 4.3 Distributed three-phase drive with ten percent current sensor error in one phase operating without any stabilization. Torque and machine terminal voltages are shown.

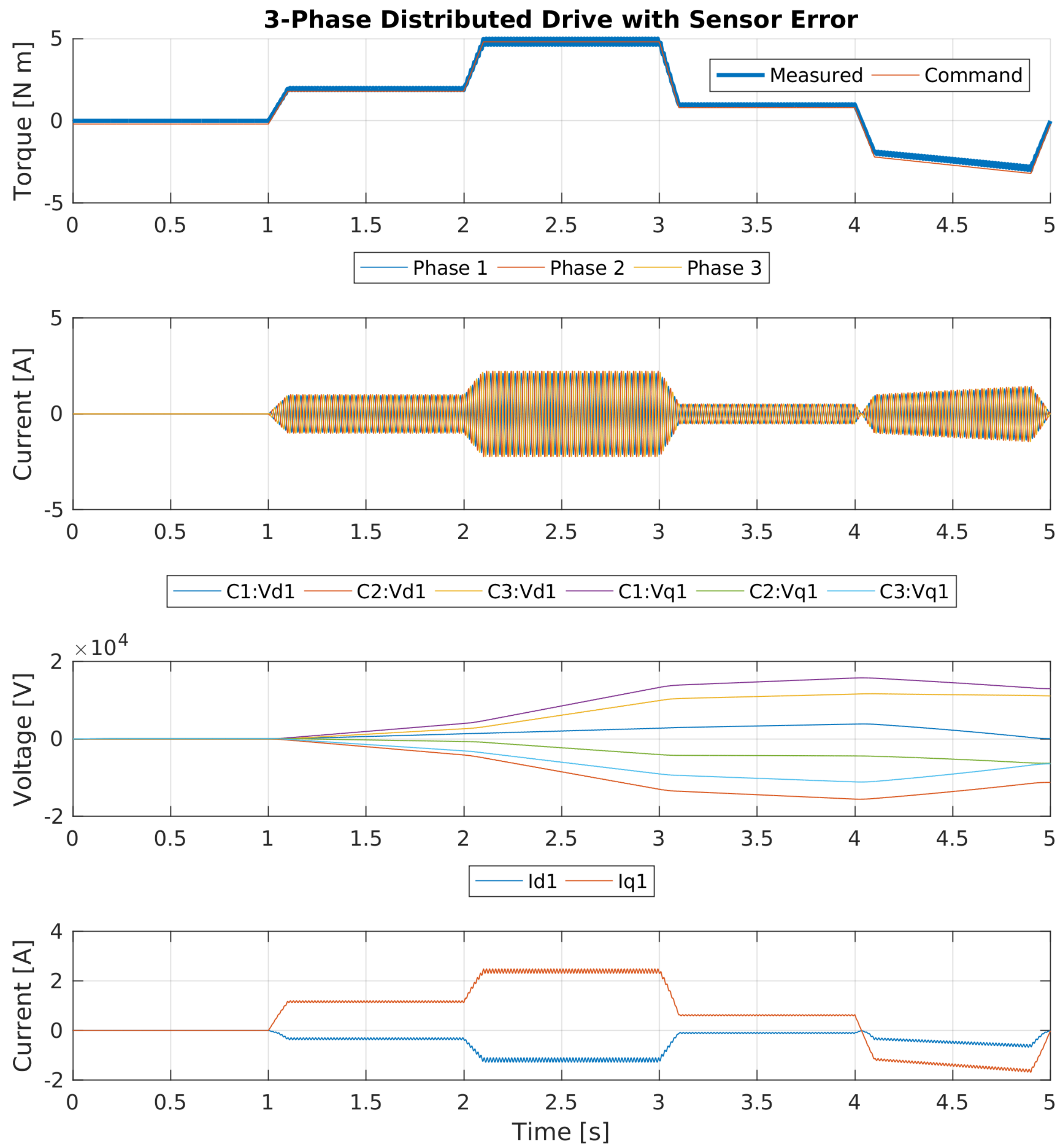


Figure 4.4 Distributed three-phase drive with ten percent current sensor error in one phase operating without any stabilization. Torque, terminal currents, and controller command voltages are shown.

the current feedback mismatch. This voltage command error consists entirely of zero sequence components so it does not affect the machine's phase currents because the machine cannot conduct any zero sequence current due to the floating-neutral connection. Instead, the controllers keep commanding higher and higher oscillating zero-sequence voltages until the inverters run out of voltage headroom. For these particular simulations, the voltage output is artificially not limited, so the commanded voltage waveforms are allowed to extend to several kV instead of losing current control as the drive runs out of voltage at a much lower bus voltage.

If the drive module simulation is implemented with voltage saturation, the waveforms instead change to those shown in figures 4.5 and 4.6. This voltage saturation is implemented as a hard limit to the DC-link voltage for each phase output as well as a saturation limit to the full DC-link magnitude for the controller integration states. When these limits are implemented, the controller internal voltage commands become erratic due to integrators dropping out of the system when they are at their limits. These erratic mode switching effects can clearly be seen in the voltage waveforms in Figure 4.7. The torque and current ripple rises to approximately 50% as the non-linear behavior causes the system to no longer match the monolithic case in the differential circuit path. As a result of this instability, the neutral-point voltage swings widely from rail to rail.

4.1.4 Single-Phase RL Load Current Regulation Example

First, in order to explain some of the basic principles of the behavior shown above, consider the simplified example case of regulating current in a simple single-phase RLcircuit using an independent voltage controller at both terminals of this ac load. This model is the minimum model in which PI regulators are used to ensure zero steady-state error at DC while the use of multiple controllers causes the control system to be over-constrained. A labeled schematic is given in Figure 4.8. In this schematic, V_{cm} is the voltage at the midpoint of the RL load and V_d is the voltage across the RL load. For the case where only a single controller and single sensor is used, the state space

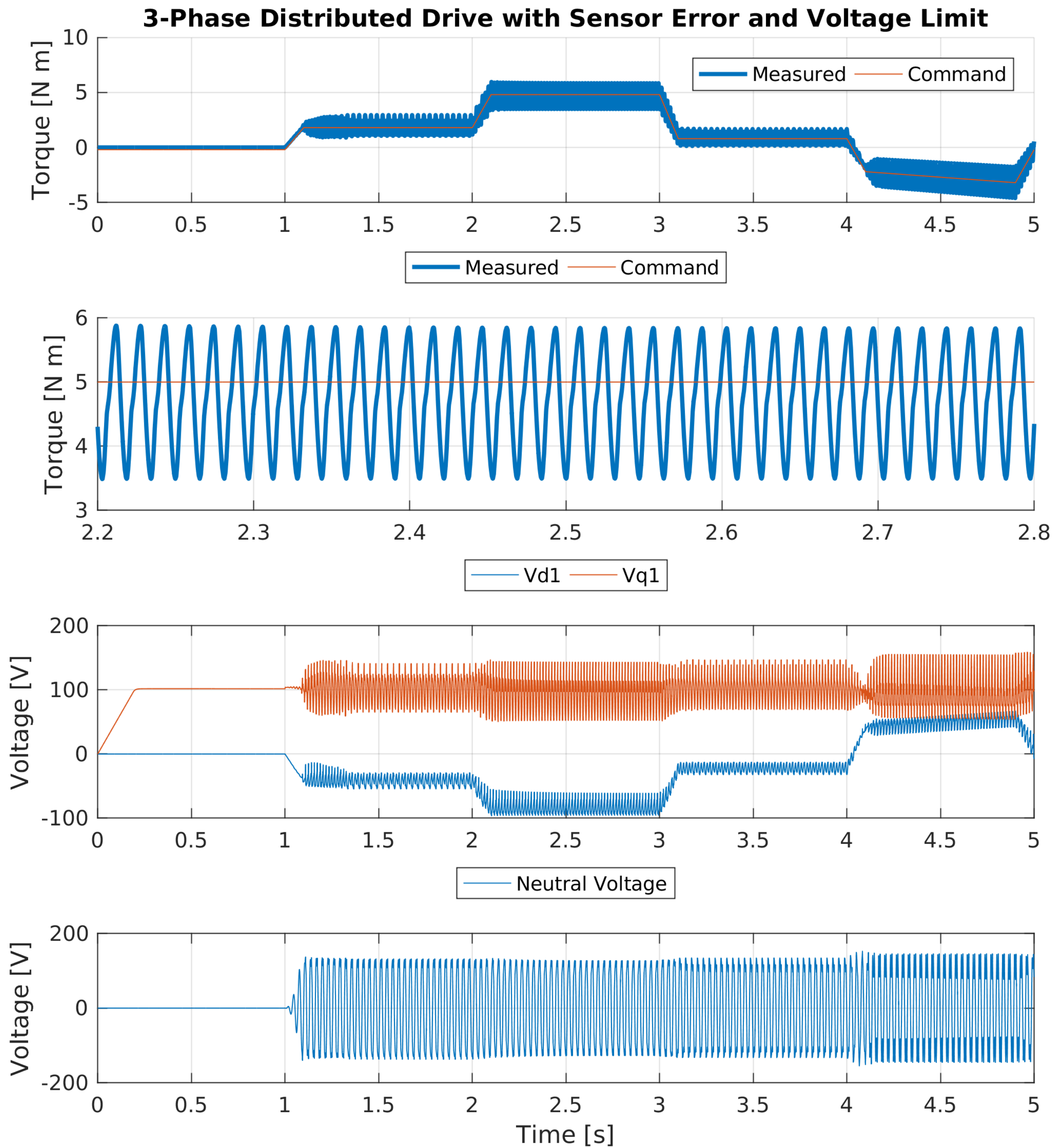


Figure 4.5 Distributed three-phase drive with voltage limits and ten percent current sensor error in one phase operating without any stabilization. Torque and machine terminal voltages are shown.

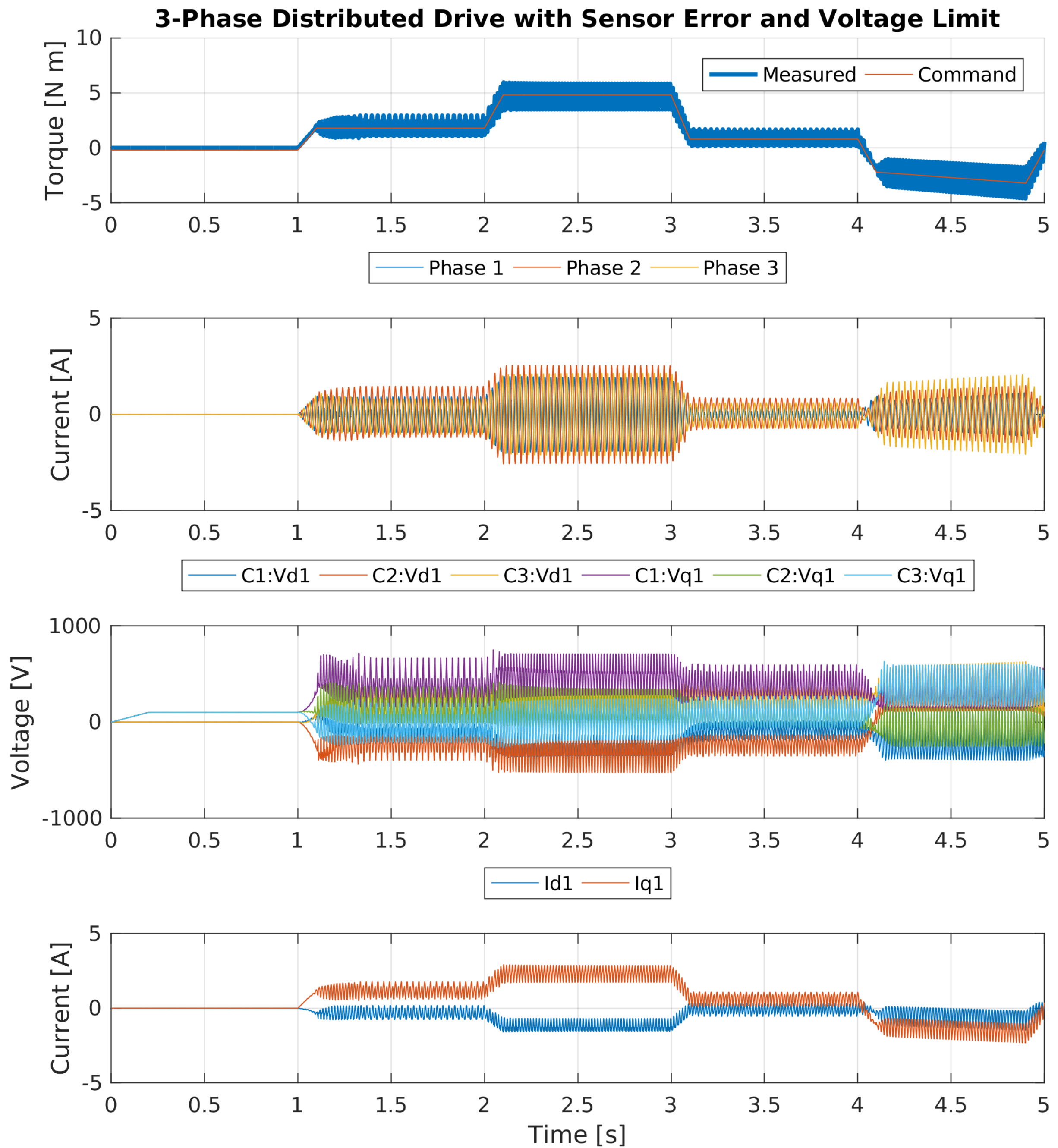


Figure 4.6 Distributed three-phase drive with voltage limits and ten percent current sensor error in one phase operating without any stabilization. Torque, terminal currents, and controller command voltages are shown.

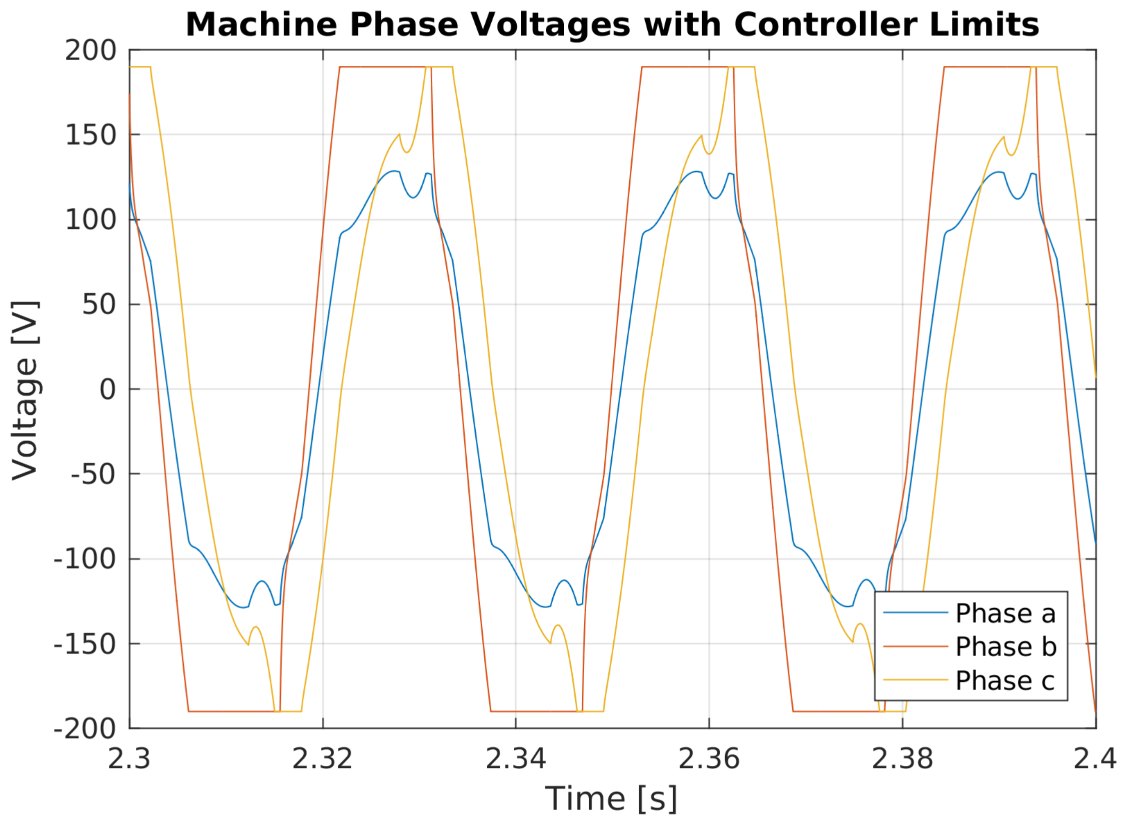


Figure 4.7 Close-up of phase voltages during 5 Nm distributed drive operation with phase voltage limits.

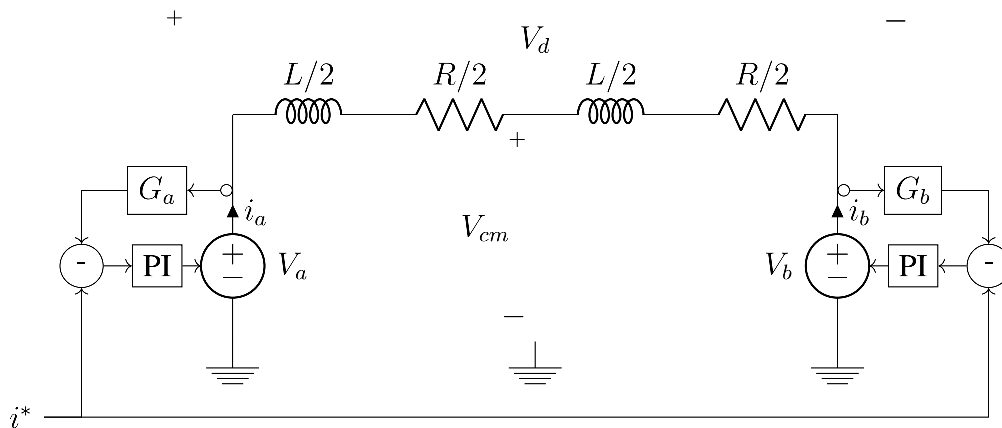


Figure 4.8 Circuit model for single phase R-L load driven by two voltage output current controllers with quantities labeled.

model can be formed as shown in equation 4.1.

$$\frac{d}{dt}\begin{bmatrix} I \\ V_I \end{bmatrix} = \begin{bmatrix} -\frac{R+G_iK_p}{L} & \frac{1}{L} \\ -G_i\tau_{PI} & 0 \end{bmatrix} \begin{bmatrix} I \\ V_I \end{bmatrix} + \begin{bmatrix} \frac{K_p}{L} \\ \tau_{PI} \end{bmatrix} I^* \tag{4.1}$$

In this model, I is the inductor current, V_I is the PI-regulator integration state, G_i is the current feedback gain and is ideally 1, K_p is the proportional gain, τ_{PI} is the PI-regulator corner frequency time constant, and R and L are the physical parameters of the RL circuit.

If a second identical PI controller is added to the other terminal of the RL load with its own current sensor and gain, the system model becomes becomes.

$$\frac{d}{dt}\begin{bmatrix} I \\ V_{Ia} \\ V_{Ib} \end{bmatrix} = \begin{bmatrix} -\frac{R+\frac{G_{ia}K_p}{2}+\frac{G_{ib}K_p}{2}}{L} & \frac{K_p}{2L} & -\frac{K_p}{2L} \\ -G_{ia}\tau_{PI} & 0 & 0 \\ -G_{ib}\tau_{PI} & 0 & 0 \end{bmatrix} \begin{bmatrix} I \\ V_{Ia} \\ V_{Ib} \end{bmatrix} + \begin{bmatrix} \frac{K_p}{L} \\ \tau_{PI} \\ \tau_{PI} \end{bmatrix} I^* \tag{4.2}$$

Under these conditions, if the two current feedback gains are identical, the two equations in the bottom two rows can be reduced to be the same as equation 4.1 using the change of variables. The new states $V_{Ia}$and V_{Ib} are the controller integration states equivalent to V_I in equation 4.1. V_{Ia} is the internal integration state of the side a controller, while V_{Ib} is the internal integration state of the side b controller. The new gains are G_a and G_b which are ideally the same and take the place of G_i above.

Each controller senses the phase current at its own output. If the current feedback gains are not identical, this causes an extra state with no feedback but with an input proportional to the current through the load. This can be shown analytically by calculating the eigenvalues of the A^+ matrix for this system. These eigenvalues are listed in equation 4.3.

$$\begin{aligned} \text{eigs}\left(\mathbf{A}^+\right) = \Big[& 0, \\ & -\frac{1}{4L}\left(G_aK_p + G_bK_p + 2R\right) + \frac{1}{4L}\sqrt{\begin{matrix} G_a^2K_p^2+2G_aG_bK_p^2-8G_aK_pL\tau_{PI}+4G_aK_pR \\ +G_b^2K_p^2-8G_bK_pL\tau_{PI}+4G_bK_pR+4R^2 \end{matrix}}, \\ & -\frac{1}{4L}\left(G_aK_p + G_bK_p + 2R\right) - \frac{1}{4L}\sqrt{\begin{matrix} G_a^2K_p^2+2G_aG_bK_p^2-8G_aK_pL\tau_{PI}+4G_aK_pR+ \\ G_b^2K_p^2-8G_bK_pL\tau_{PI}+4G_bK_pR+4R^2 \end{matrix}} \Big] \end{aligned} \tag{4.3}$$

Since the first of these is zero, there is an open integrator in the system. Because this system includes both the physical system and controller, this open integrator is the result of system over constraint as there are more controllers than controlled states. Removing the integration state by setting $\tau_{PI} = 0$ removes this zero eigenvalue, simplifying the system to single-order, but this change removes the zero-steady-state error property and causes much lower system stiffness in the presence of disturbances.

In order to more clearly demonstrate this, the system can be rewritten in terms of common-mode and differential-mode voltages. The following change of variables to split this combined PI controller into a common mode and differential mode shows that the two integration states in the controller can be thought of as shared between the two controllers.

$$\begin{bmatrix} I \\ V_{Ia} \\ V_{Ib} \end{bmatrix} = \begin{bmatrix} I \\ V_{Icm} + \frac{V_{Id}}{2} \\ V_{Icm} - \frac{V_{Id}}{2} \end{bmatrix} \tag{4.4}$$

$$\frac{d}{dt}\begin{bmatrix} I \\ V_{Id} \\ V_{Icm} \end{bmatrix} = \begin{bmatrix} -\frac{R+\frac{G_{ia}+G_{ib}}{2}K_p}{L} & \frac{K_p}{L} & 0 \\ -\frac{G_{ia}+G_{ib}}{2}\tau_{PI} & 0 & 0 \\ -(G_{ia}-G_{ib})\tau_{PI} & 0 & 0 \end{bmatrix}\begin{bmatrix} I \\ V_{Id} \\ V_{Icm} \end{bmatrix} + \begin{bmatrix} \frac{K_p}{L} \\ \tau_{PI} \\ 0 \end{bmatrix} I^* \tag{4.5}$$

Again there are two integration states, which are now V_{Id} for the differential mode and V_{Icm} for the common mode. All other variables are the same as above.

Thus, there is a differential circuit which is the same as the single controller case, but also a common mode circuit with no feedback that has an integration state and is fed by the sensor

mismatch.

$$\begin{bmatrix} V_a \\ V_b \end{bmatrix} = \begin{bmatrix} -\frac{G_a K_p}{2} & \frac{K_p}{2} & 0 \\ \frac{G_b K_p}{2} & 0 & -\frac{K_p}{2} \end{bmatrix} \begin{bmatrix} I \\ V_{Ia} \\ V_{Ib} \end{bmatrix} + \begin{bmatrix} \frac{K_p}{2} \\ -\frac{K_p}{2} \end{bmatrix} I^* \tag{4.6}$$

$$\begin{bmatrix} V_d \\ V_{cm} \end{bmatrix} = \begin{bmatrix} -\frac{G_a+G_b}{2} K_p & K_p & 0 \\ -\frac{(G_a-G_b)K_p}{4} & 0 & K_p \end{bmatrix} \begin{bmatrix} I \\ V_{Id} \\ V_{Icm} \end{bmatrix} + \begin{bmatrix} K_p \\ 0 \end{bmatrix} I^* \tag{4.7}$$

Since real controllers do not have infinite voltage headroom, this common mode voltage loop must be controlled somehow. The most straight forward way to do this is to take a voltage feedback from the midpoint of the RL load (or generate a virtual midpoint with a resistive divider). If this feedback path is added, the system shown in equation 4.8 is created where none of the states has an open loop.

$$\frac{d}{dt} \begin{bmatrix} I \\ V_{Id} \\ V_{Icm} \end{bmatrix} = \begin{bmatrix} -\frac{R+\frac{G_{ia}+G_{ib}}{2}K_p}{L} & \frac{K_p}{L} & 0 \\ -\frac{G_{ia}+G_{ib}}{2}\tau_{PI} & 0 & 0 \\ -(G_{ia}-G_{ib})\tau_{PI} & 0 & \frac{-K_n \tau_{PI}}{4} \end{bmatrix} \begin{bmatrix} I \\ V_{Id} \\ V_{Icm} \end{bmatrix} + \begin{bmatrix} \frac{K_p}{L} \\ \tau_{PI} \\ 0 \end{bmatrix} I^* \tag{4.8}$$

The eigenvalues for this system and all the systems that follow were verified by use of the computer algebra system SymPy [70]. These eigenvalues, shown in 4.9, show that the system with added feedback in the common mode has all negative-real components and, thus, will tend to zero steady-state error.

$$\begin{aligned} \operatorname{eigs}(\mathbf{A}^+) = \Bigg[& -\frac{K_n K_p \tau_{PI}}{4}, \\ & -\frac{1}{4L}\left(G_a K_p + G_b K_p + 2R - \sqrt{\begin{array}{c} G_a^2 K_p^2 + 2G_a G_b K_p^2 - 8G_a K_p L \tau_{PI} \\ +4G_a K_p R + G_b^2 K_p^2 - 8G_b K_p L \tau_{PI} + 4G_b K_p R + 4R^2 \end{array}} \right), \\ & -\frac{1}{4L}\left(G_a K_p + G_b K_p + 2R + \sqrt{\begin{array}{c} G_a^2 K_p^2 + 2G_a G_b K_p^2 - 8G_a K_p L \tau_{PI} \\ +4G_a K_p R + G_b^2 K_p^2 - 8G_b K_p L \tau_{PI} + 4G_b K_p R + 4R^2 \end{array}} \right) \Bigg] \end{aligned} \tag{4.9}$$

These eigenvalues are plotted in Figure 4.9 showing a current controller tuned to 100Hz break frequency with a 1mH inductance, one ohm real resistance, and ten percent current sensor error. The neutral point correction gain starts at zero an then is varied from reciprical 10 ohms to reciprical 10 milliohms. The current regulator time constant is varied between 0.1x and 10x the physical system break frequency. As is shown here, the system is stable, though slightly oscillitory when the PI regulator is tuned faster than the physical system time constant. Without any neutral point gain, the neutral voltage pole is precisely on the origin which causes the system to integrate any disturbance including current sensor offsets. As the gain is increased, the neutral circut becomes passivated with a reduced settling time.

If the system in equation 4.8 is translated back into per-controller variables, it gives the system in equation 4.10.

$$\frac{d}{dt}\begin{bmatrix} I \\ V_{Ia} \\ V_{Ib} \end{bmatrix} = \begin{bmatrix} -\frac{R+\frac{G_{ia}K_p}{2}+\frac{G_{ib}K_p}{2}}{L} & \frac{K_p}{2L} & -\frac{K_p}{2L} \\ -G_{ia}\tau_{PI} & -\frac{K_nK_p\tau_{PI}}{4} & -\frac{K_nK_p\tau_{PI}}{4} \\ -G_{ib}\tau_{PI} & -\frac{K_nK_p\tau_{PI}}{4} & -\frac{K_nK_p\tau_{PI}}{4} \end{bmatrix} \begin{bmatrix} I \\ V_{Ia} \\ V_{Ib} \end{bmatrix} + \begin{bmatrix} \frac{K_p}{L} \\ \tau_{PI} \\ \tau_{PI} \end{bmatrix} I^* \tag{4.10}$$

This shows how the feedback path can be added directly to the PI regulator feedback path, where K_n has units of inverse ohms, making it a type of virtual common-mode resistance.

4.1.5 Polyphase State-Space Model

Before deriving a model for the three-phase version, some new notation is needed that will be used in order to make the matrices below fit in the page width. These notations are not quite standard, but should be clear enough to avoid confusion. First, **0** is a square zero matrix, distinct from 0. It is only used when building block-based matrices and its size can be deduced from the other matrices present.

Second, the notation $\mathbf{I}_{nxm}$ is used to denote a non-square identity matrix with dimensions n and m columns. This matrix has the minimum of n and m ones along the elements closest to the

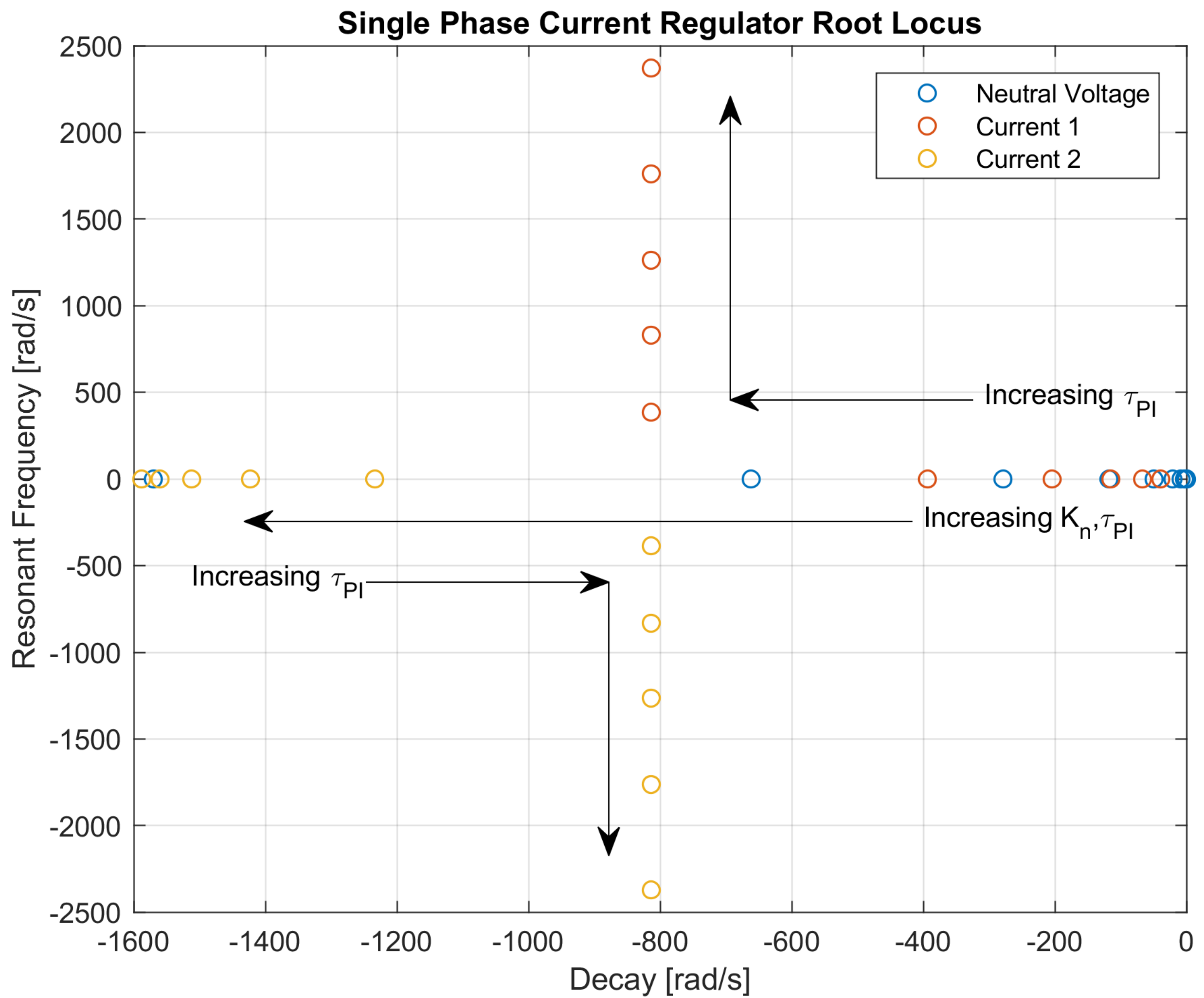


Figure 4.9 Root locus diagram for two independent current controllers controlling a 1mH, 1ohm R-L load. The controller time constant and neutral point gain is varied to show tuning effects.

diagonal with zeros elsewhere. This matrix is used to select n elements from an $m = kn$ element vector where k is an integer such that the first element is selected from the first n elements, the second element is selected from the second n elements, etc. $\mathbf{I}_{nxn}$ is identical to the standard identity

matrix of dimension n. The example often used in this chapter is

$$\mathbf{I}_{3x9} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.11}$$

In order to form a state-space model for the distributed controller, a coupling model must be derived showing how the current sensing and output voltage recombination affects the system. As was shown in the previous chapter for the case of the three-phase machine drive with a monolithic controller, calibration errors in the current sensors cause ripple and offset, but the currents and voltages are stable due to the lack of closed-loop feedback in the zero-sequence. In the distributed controller case, this zero sequence isolation is no longer the case.

This zero-sequence response is the result of the independent nature of the phase controllers. Even though each phase controller generates its output voltage using a Clarke transform with no zero-sequence component, the combination of phases generating a voltage component with non-identical d and $q$components will generate a zero-sequence component when combined.

To begin, assume that each of the three phase controllers is delivering a stationary frame complex voltage vector $V_{\alpha n}$, $V_{\beta n}$ where n is the index number of the phase. This voltage is then translated into phase voltages and the appropriate phase voltage for each phase winding is then chosen for delivery at the module's output terminals.

$$V_{abc} = \mathbf{I}_{3x9} \begin{bmatrix} [\mathbf{P}][\mathbf{C}] & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & [\mathbf{P}][\mathbf{C}] & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & [\mathbf{P}][\mathbf{C}] \end{bmatrix} V_{\alpha\beta n} \tag{4.12}$$

$$V_{abc} = \mathbf{I}_{3x9}\sqrt{\frac{2}{3}}\begin{bmatrix} 1 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} & \frac{\sqrt{3}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} & -\frac{\sqrt{3}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \frac{\sqrt{2}}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & \frac{\sqrt{3}}{2} & \frac{\sqrt{2}}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & -\frac{\sqrt{3}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}\begin{bmatrix} V_{\alpha 1} \\ V_{\beta 1} \\ 0 \\ V_{\alpha 2} \\ V_{\beta 2} \\ 0 \\ V_{\alpha 3} \\ V_{\beta 3} \\ 0 \end{bmatrix} \tag{4.13}$$

$$V_{abc} = \begin{bmatrix} \frac{\sqrt{6}V_{\alpha 1}}{3} \\ -\frac{\sqrt{6}V_{\alpha 2}}{6} + \frac{\sqrt{2}V_{\beta 2}}{2} \\ -\frac{\sqrt{6}V_{\alpha 3}}{6} - \frac{\sqrt{2}V_{\beta 3}}{2} \end{bmatrix} \tag{4.14}$$

When these phase voltages are then recombined to form the stationary-frame voltages as seen by the machine, a zero-sequence term emerges.

$$V_{\alpha\beta n\text{machine}} = (\mathbf{PC})^{-1}V_{abc} \tag{4.15}$$

$$V_{\alpha\beta n\text{machine}} = \begin{bmatrix} \frac{2V_{\alpha 1}}{3} + \frac{V_{\alpha 2}}{6} + \frac{V_{\alpha 3}}{6} - \frac{\sqrt{3}V_{\beta 2}}{6} + \frac{\sqrt{3}V_{\beta 3}}{6} \\ -\frac{\sqrt{3}V_{\alpha 2}}{6} + \frac{\sqrt{3}V_{\alpha 3}}{6} + \frac{V_{\beta 2}}{2} + \frac{V_{\beta 3}}{2} \\ \frac{\sqrt{2}V_{\alpha 1}}{3} - \frac{\sqrt{2}V_{\alpha 2}}{6} - \frac{\sqrt{2}V_{\alpha 3}}{6} + \frac{\sqrt{6}V_{\beta 2}}{6} - \frac{\sqrt{6}V_{\beta 3}}{6} \end{bmatrix} \tag{4.16}$$

In order to make more sense of this outcome, it helps to rewrite the voltages in terms of a common-mode average and two offset vectors which capture the difference between the first two phases' controllers and the average. This mathematical abstraction causes no loss of generality but lends insight into what is happening when these phase quantities are applied to the machine.

Then, terms can be replaced to separate the common-mode and differential voltages. In the following two equations the vectors of $V_{\alpha n}$ and $V_{\beta n}$ commands, which are the stationary first-harmonic reference frame voltages as described in the previous chapter, output by each of the three phase controllers is decomposed into an average output V_α and V_β plus a pair of difference vectors $(V_{\alpha d1}, V_{\beta d1})$ and $(V_{\alpha d2}, V_{\beta d2})$.

$$\begin{bmatrix} V_{\alpha 1} \\ V_{\alpha 2} \\ V_{\alpha 3} \end{bmatrix} = \begin{bmatrix} V_\alpha + V_{\alpha d1} \\ V_\alpha + V_{\alpha d2} \\ V_\alpha - V_{\alpha d1} - V_{\alpha d3} \end{bmatrix} \tag{4.17}$$

$$\begin{bmatrix} V_{\beta 1} \\ V_{\beta 2} \\ V_{\beta 3} \end{bmatrix} = \begin{bmatrix} V_\beta + V_{\beta d1} \\ V_\beta + V_{\beta d2} \\ V_\beta - V_{\beta d1} - V_{\beta d3} \end{bmatrix} \tag{4.18}$$

When these substitutions are applied to equation 4.16, the common average of the command voltages does get through along with an error term in each of the coordinates.

$$V_{\alpha\beta n} = \begin{bmatrix} V_\alpha \\ V_\beta \\ 0 \end{bmatrix} + \begin{bmatrix} \frac{V_{\alpha d1}}{2} - \frac{\sqrt{3}V_{\beta d1}}{6} - \frac{\sqrt{3}V_{\beta d2}}{3} \\ -\frac{\sqrt{3}V_{\alpha d1}}{6} - \frac{\sqrt{3}V_{\alpha d2}}{3} - \frac{V_{\beta d1}}{2} \\ \frac{\sqrt{2}V_{\alpha d1}}{2} + \frac{\sqrt{6}V_{\beta d1}}{6} + \frac{\sqrt{6}V_{\beta d2}}{3} \end{bmatrix} \tag{4.19}$$

These error terms each show a structure reminiscent of the vector sum of the three phases, but their content is still somewhat opaque.

This becomes clearer when calculated in the $dq0$ reference frame. The calculation starts with a command vector coming from the phase controllers of the common mode average (V_d and V_q) coupled with two differential mode vectors to capture the difference between each phase and the

average.

$$V^*_{dq0} = \begin{bmatrix} V^*_d + V^*_{d1} \\ V^*_q + V^*_{q1} \\ 0 \\ V^*_d + V^*_{d2} \\ V^*_q + V^*_{q2} \\ 0 \\ V^*_d - V^*_{d1} - V^*_{d2} \\ V^*_q - V^*_{q1} - V^*_{q2} \\ 0 \end{bmatrix} \tag{4.20}$$

This is then sent through the phase splitting process as shown in equation 4.21.

$$\begin{aligned} V^*_{dq0\text{machine}} &= [\mathbf{R}(\theta)]([\mathbf{P}][\mathbf{C}])^{-1}\mathbf{I}_{3x9} \begin{bmatrix} [\mathbf{P}][\mathbf{C}] & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & [\mathbf{P}][\mathbf{C}] & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & [\mathbf{P}][\mathbf{C}] \end{bmatrix} \begin{bmatrix} [\mathbf{R}(\theta)] & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & [\mathbf{R}(\theta)] & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & [\mathbf{R}(\theta)] \end{bmatrix} V^*_{dq0} \\ &= \begin{bmatrix} V^*_d - \frac{\sqrt{3}V^*_{d1}}{3}\sin\left(2\theta + \frac{\pi}{3}\right) - \frac{\sqrt{3}V^*_{d2}}{3}\sin\left(2\theta\right) + \frac{\sqrt{3}V^*_{q1}}{3}\cos\left(2\theta + \frac{\pi}{3}\right) + \frac{\sqrt{3}V^*_{q2}}{3}\cos\left(2\theta\right) \\ V^*_q + \frac{\sqrt{3}V^*_{d1}}{3}\cos\left(2\theta + \frac{\pi}{3}\right) + \frac{\sqrt{3}V^*_{d2}}{3}\cos\left(2\theta\right) + \frac{\sqrt{3}V^*_{q1}}{3}\sin\left(2\theta + \frac{\pi}{3}\right) + \frac{\sqrt{3}V^*_{q2}}{3}\sin\left(2\theta\right) \\ \frac{\sqrt{6}}{3}\left(V^*_{d1}\sin\left(\theta + \frac{\pi}{6}\right) + V^*_{d2}\cos\left(\theta\right) - V^*_{q1}\cos\left(\theta + \frac{\pi}{6}\right) + V^*_{q2}\sin\left(\theta\right)\right) \end{bmatrix} \end{aligned} \tag{4.21}$$

This shows that, in the synchronous frame, the zero-sequence component consists of a oscillating component proportional to the error between the phases' synchronous frame voltage command and their overall average, while the phase of this oscillating component lags the error by 30 degrees. The frequency of this zero-sequence component is the same as the fundamental electrical frequency. The active d-axis and q-axis components also show an oscillating error component proportional to the synchronous frame voltage error, but this error component rotates at twice the synchronous

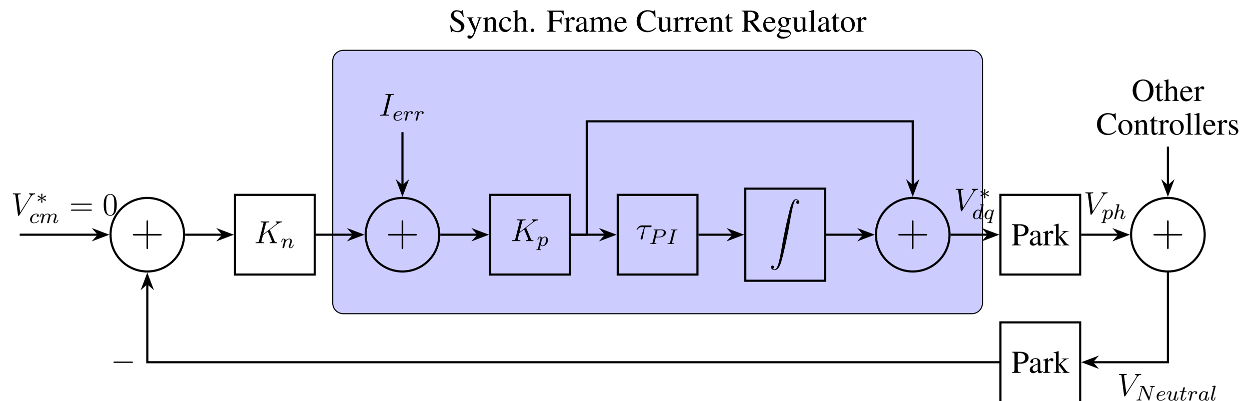


Figure 4.10 Control block diagram showing voltage command error control loop.

frequency and thus does not affect the average torque, only causing torque ripple. If this error is driven to zero, the second harmonic component will not be a factor.

4.2 Neutral Point Voltage Control

As the neutral point voltage of the machine now contains a voltage proportional to the controller voltage mismatch, and the neutral voltage is oscillating at the synchronous frequency, it can be used along with some coordinate transforms to close the loop on the differential mode states between controllers. The condensed block diagram of how this part of the control loop is shown in Figure 4.10.

This controller only needs to tame the integration pole at low frequencies in order to keep system imbalance disturbances from becoming a problem. These disturbances come from either virtual sources such as the sensor imbalance described above or from physical sources such as winding asymmetry, resistance asymmetry, dead-time effects, or rotor asymmetry. The current error acts like an injection at the synchronous frame summing junction. The others act as voltages at the stationary neutral summing junction. These asymmetries interact with the Park transforms in the neutral voltage loop to become a constant component which must be attenuated and a twice synchronous

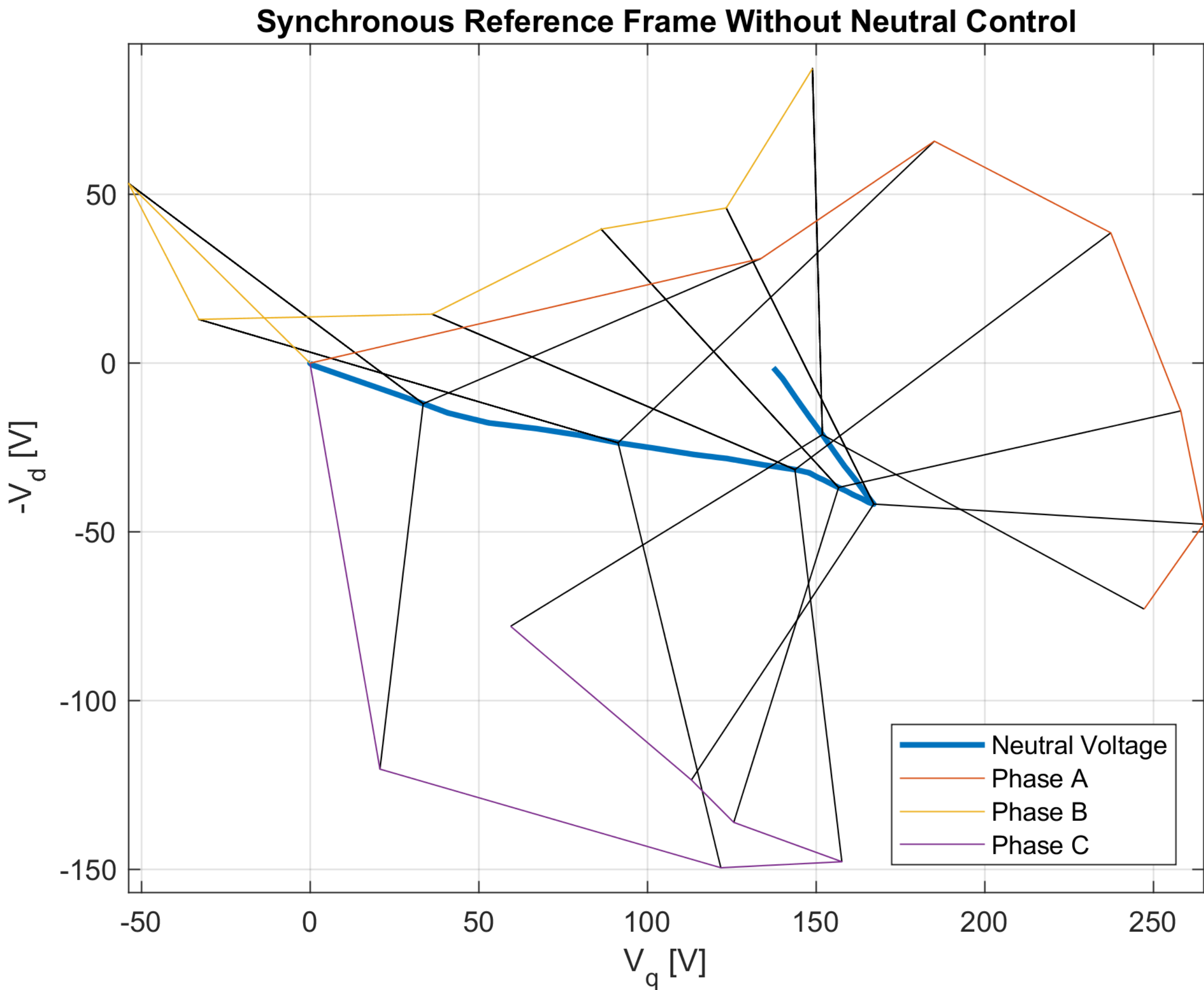


Figure 4.11 Three-phase current regulator phase voltage loci in synchronous reference frame with 0.1% current sensor error showing increasing common mode offset.

harmonic component. So long as K_n is high enough ensure no drive hits a voltage rail, the value is only limited by sensor noise.

If an integration state is desired, it can be added, but the interaction of the neutral point feedback integrator and the main current loop integrator creates tuning difficulties. An integrator in the neutral voltage feedback loop must have a break frequency at least half a decade below the fundamental frequency in order to keep it from swamping the current regulation. This poses difficulties when

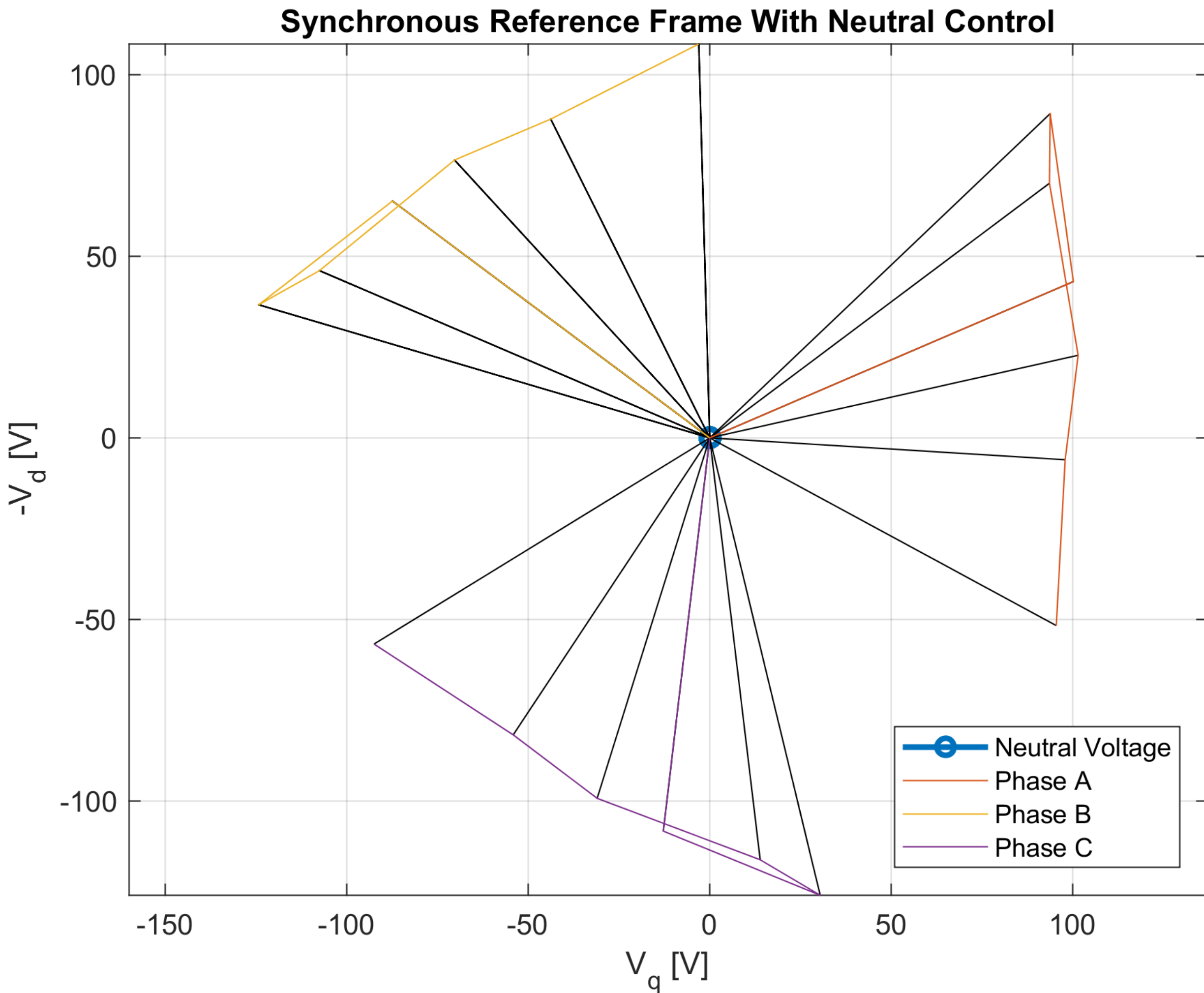


Figure 4.12 Three-phase current regulator phase voltage loci in synchronous reference frame with 0.1% current sensor error showing increasing controlled common mode offset when a Neutral voltage controller is employed.

starting from low speeds though it may be helpful to reduce torque ripple during higher speed operation.

While this neutral point controller may appear to be an independent control path, it is not decoupled from the torque-producing current regulator. If other non-constant neutral voltage commands are desired, there are two options. First, they can be injected as a command into this controller, though care must be taken to synchronize the neutral voltage commands in all phase

modules and the command injection must be decoupled from the outputs of the complex current regulator. The command does not need to be synchronized to the machine current command voltage command. It should not however provide another feedback path between the complex current regulator integration states and the neutral controller input. If the commands are not decoupled, the current regulator will attempt to null out the input command and place itself at risk of losing control of one axis of current regulation. If a higher-bandwidth neutral voltage injection is desired, it can be directly injected at the output voltage of the Clarke transform instead of being injected as a command input to the stabilizing neutral voltage controller. Care should be taken to ensure that the command is filtered appropriately so that it matches the phase response of the neutral voltage sensing circuitry. This higher-bandwidth path is open loop but does allow full PWM output bandwidth.

If the zero-sequence term from equation 4.21 is passed through the first column of $[\mathbf{R}(\theta)]$ in order to rotate it into the synchronous reference frame, it becomes the voltage vector shown in equation 4.22 for theta aligned with the first phase.

$$[\mathbf{R}(\theta)] \begin{bmatrix} V_n \\ 0 \\ 0 \end{bmatrix} = \frac{\sqrt{6}}{12} \left(\begin{bmatrix} \sqrt{3}V_{d1}^* + V_{q1}^* + 2V_{q2}^* \\ -V_{d1}^* - 2V_{d2}^* + \sqrt{3}V_{q1}^* \\ 0 \end{bmatrix} + \cos(2\theta) \begin{bmatrix} -\sqrt{3}V_{d1}^* - V_{q1}^* - 2V_{q2}^* \\ -V_{d1}^* - 2V_{d2}^* + \sqrt{3}V_{q1}^* \\ 0 \end{bmatrix} \right.$$
$$\left. + \sin(2\theta) \begin{bmatrix} V_{d1}^* + 2V_{d2}^* - \sqrt{3}V_{q1}^* \\ -\sqrt{3}V_{d1}^* - V_{q1}^* - 2V_{q2}^* \\ 0 \end{bmatrix} \right) \tag{4.22}$$

This shows that when rotated back into the synchronous reference frame, the neutral voltage provides a feedback path for the synchronous frame error which can be fed into the complex vector current regulator to close the loop on these free states.

Extracting the eigenvalues and checking controllability on this overall system shows that it is stable and controllable though the terms do not fit on a single page.

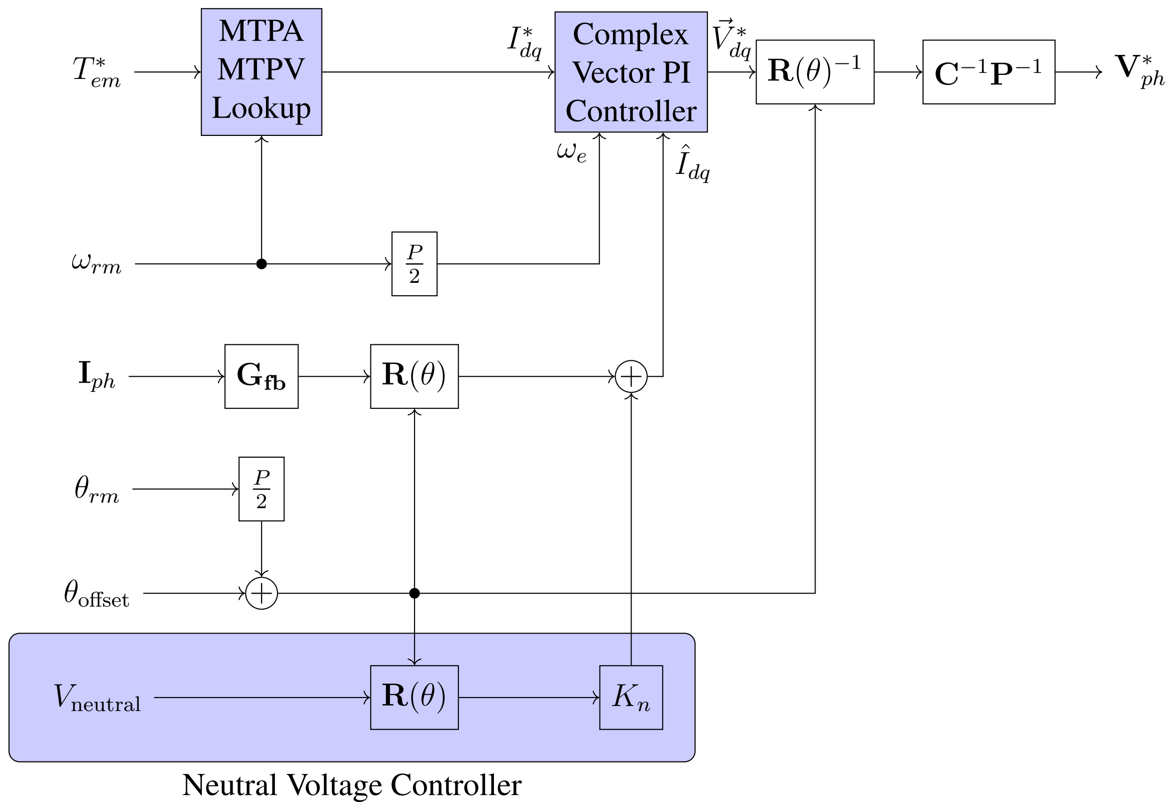


Figure 4.13 Distributed drive phase controller block diagram with neutral-point based compensation.

As the modular phase controller is now getting more complicated, it is helpful to take the common components and encapsulated them in a for-each block in Simulink. This both makes sure that any change to the distributed controller is common to all phases, and allows for flexibility when this controller is applied to higher phase numbers.

Figure 4.13 shows the structure of the neutral-point stabilization method. In order to extract an error signal, take the neutral voltage and rotate it at the synchronous speed. This creates an oscillating vector in the synchronous reference frame that is is on average proportional to the voltage offset for that phase. This oscillating error term can then be multiplied by a gain equivalent to a virtual resistance then added to the synchronous frame current feedback for the main regulator.

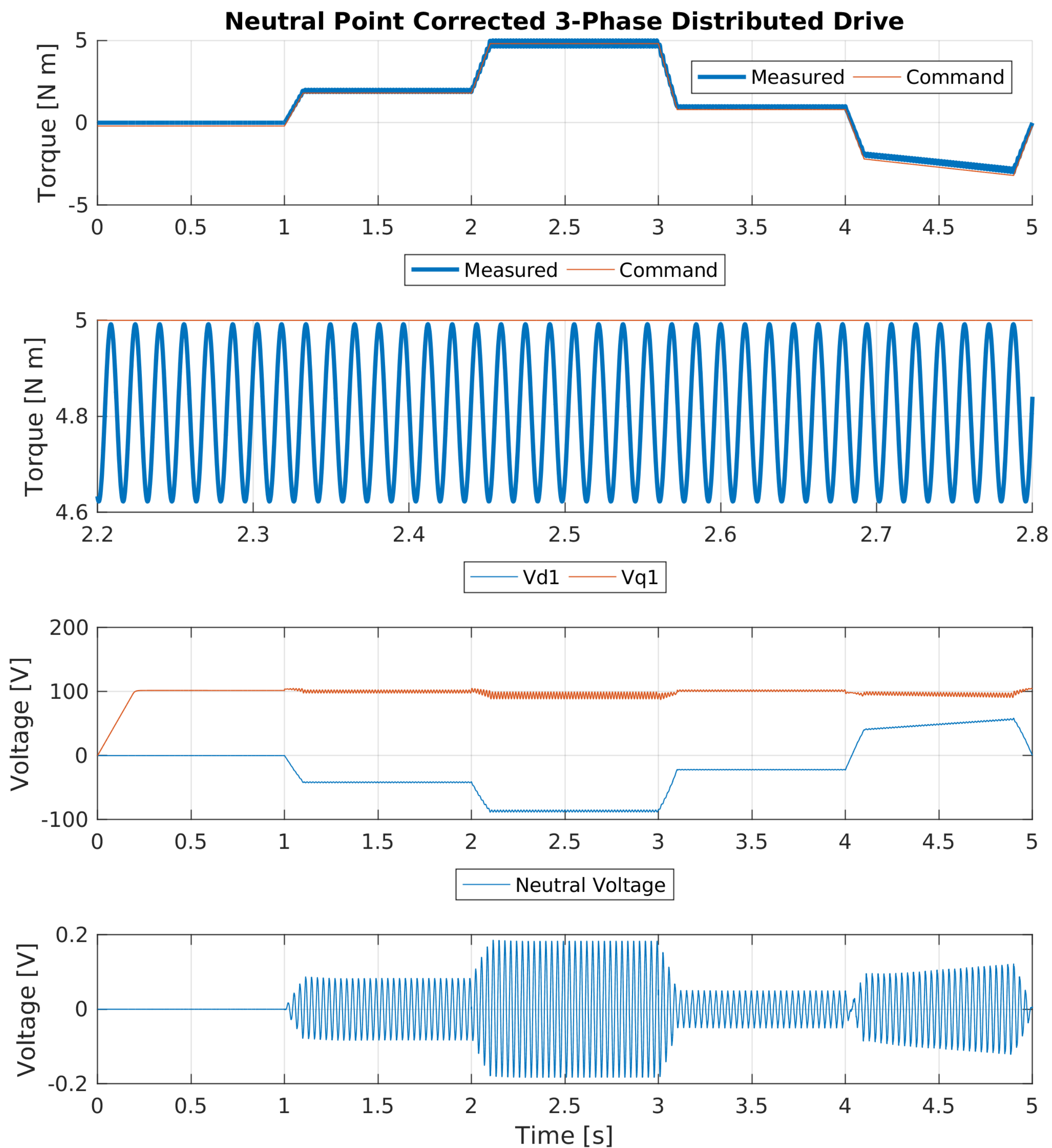


Figure 4.14 Distributed three-phase drive with ten percent current sensor error in one phase operating with Neutral-Point based linear stabilization. Torque and machine terminal voltages are shown.

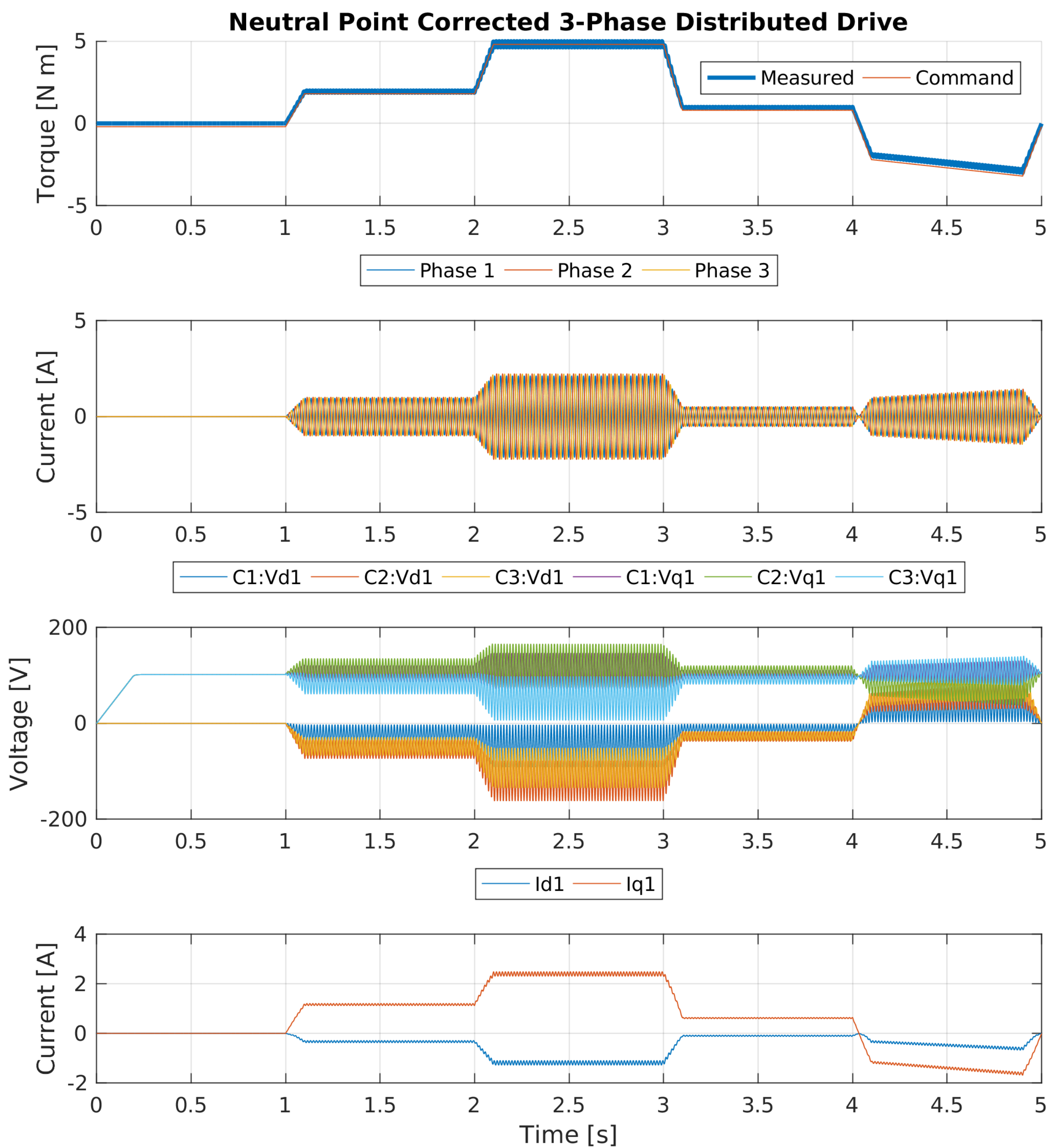


Figure 4.15 Distributed three-phase drive with ten percent current sensor error in one phase operating with Neutral-Point based linear stabilization. Torque, terminal currents, and controller command voltages are shown.

The system is stable with only a proportional gain, but the feed through of second harmonic terms is approximately twice that of the monolithic drive case. This feed through also manifests itself as ripple in the off-axis components of the drive command voltages. These ripples are shown in figures 4.14 and 4.15. Sequence separation techniques like those described in [22] could be used to reduce this error without compromising the system stability.

This shows that a three phase distributed controller can be stabilized with a simple feedback method.

4.3 Observer-Based Stabilization

The correction system shown in the previous section does require an extra voltage sensor. While this voltage does not need to be the actual neutral machine (i.e., a virtual neutral point works just as well), measuring this voltage increases the sensor requirements beyond those of a standard drive. In order to eliminate this extra sensor requirement an observer can be introduced to construct a neutral point voltage. This method is quite similar to the fictive-axis based control proposed by Bahrani et al. [16]. This observer is configured as an open-loop observer to construct a model of the vector PI controller and machine system. This modeled system is fed with the same current command as the controller and is used to generate the other two phases of the drive while taking the phase voltage for the active phase from the actual controller.

While this is relatively robust to inductance and resistance differences, it is not stable in the q-axis with respect to PM-Flux estimate error. This is because the synchronous observer models an offset in the voltages which reflects the PM-flux feed forward in the controller itself meaning that there is no regulation in the PM current path.

4.4 Open Winding Arrangement

Another approach to this problem is to remove the neutral connection in the three-phase winding and excite each of the three open phase windings with its own H-bridge. This allows zero-sequence currents to flow, relaxing the over-constraint problem caused by the floating-wye winding connection. However, this change sets the stage for new problems because the zero-sequence impedances are typically very low, allowing small zero-sequence voltage components in the phase voltages applied by the H-bridges to generate large zero-sequence current components if the current regulators saturate at high operating speeds.

If this approach is used and each modular H-bridge drive is given its own current sensor, the system is stable even in the face of asymmetries since the zero sequence can be correctly decoupled and the controller can operate with the same tunings as the single monolithic controller. Simulation of this operating case is shown in figures 4.16 and 4.17.

By having every phase current sensed, each controller has full knowledge of both the magnitude and phase of the machine MMF as well as the zero-sequence current. This sensing arrangement however, is not scalable to higher phase orders. The final arrangement of a distributed drive must be able to operate with less than full current feedback in order to avoid the problem of interconnections scaling with the number of phases squared.

The other extreme is also possible where each drive only has its own current sense. For accurate operation, this needs to be formed more like a single-phase drive using either a modeled quadrature axis as in [16] using a phase delay method. This drive configuration however is stable even without any accordance for the lack of stator mmf angle information even though the fast current regulator tracks the feedback going do zero. Simulation of this case is shown in figures 4.18 and 4.19.

This single sensor arrangement, while appealing for its simplicity and fault tolerance, is insufficient for high performance control of a wye-connected machine as the phase controllers lack

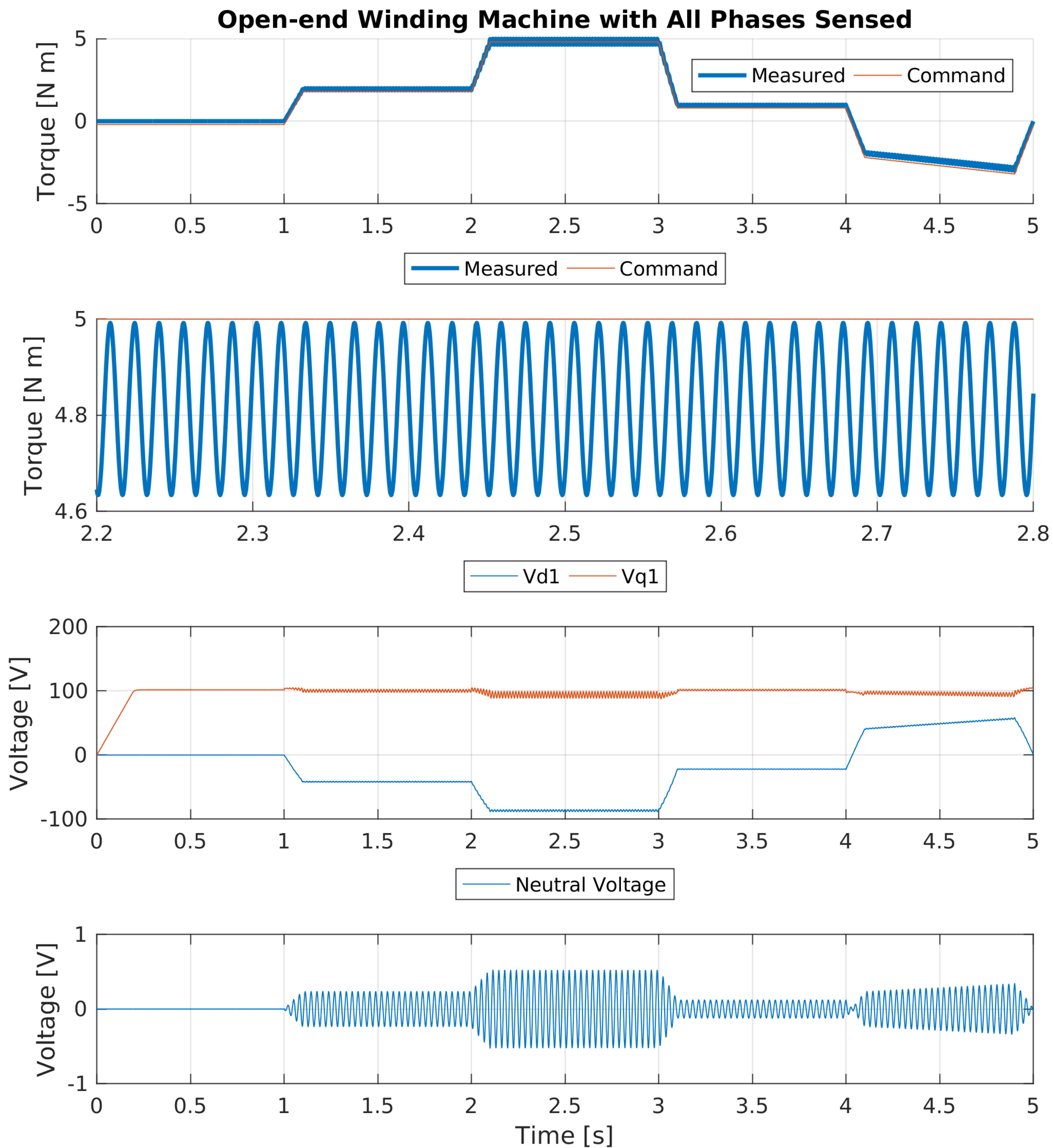


Figure 4.16 Simulation of a three-phase open-end winding distributed drive with asymmetries where each drive measures all three currents.

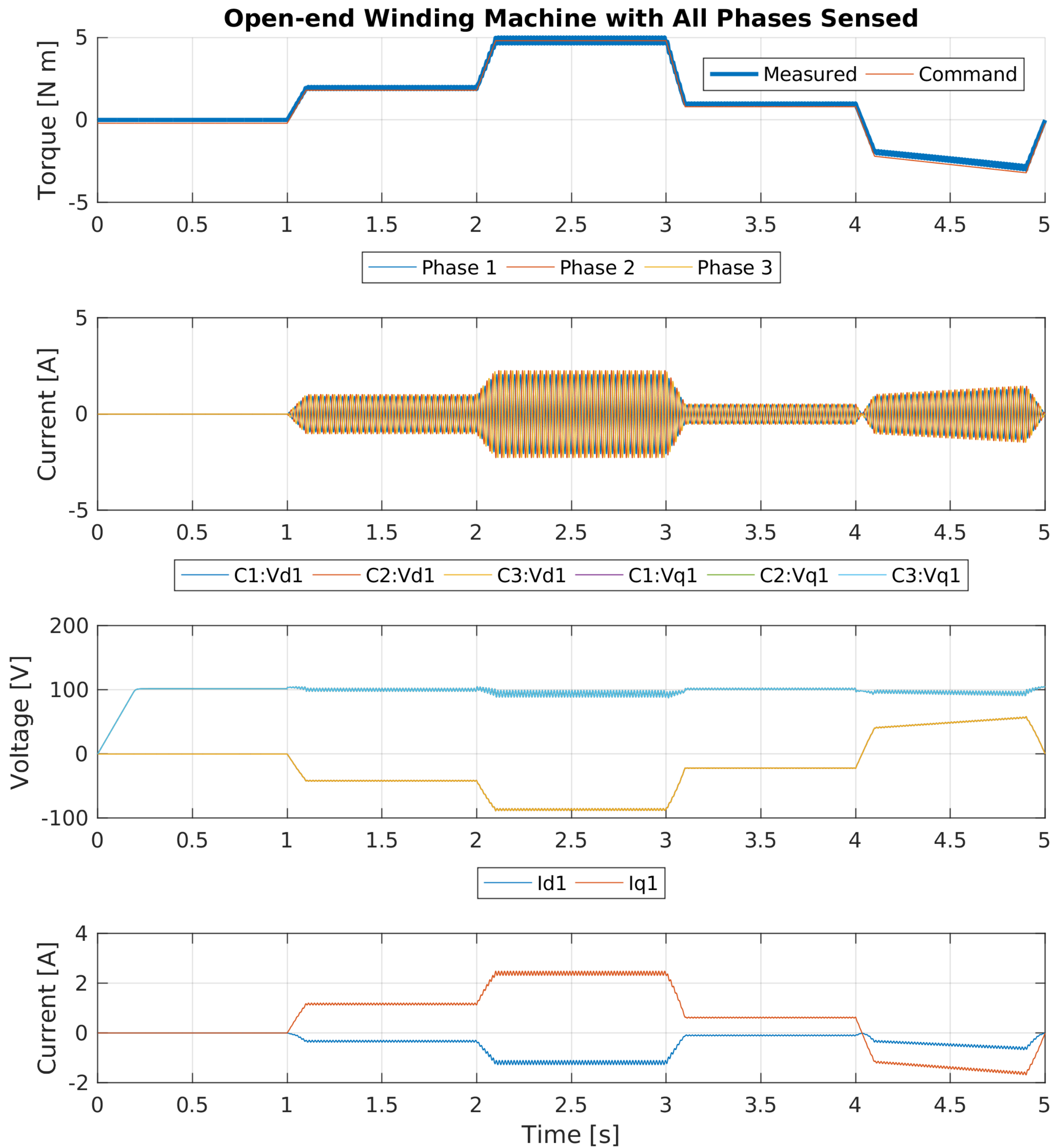


Figure 4.17 Simulation controller states of a three-phase open-end winding distributed drive with asymmetries where each drive measures all three currents.

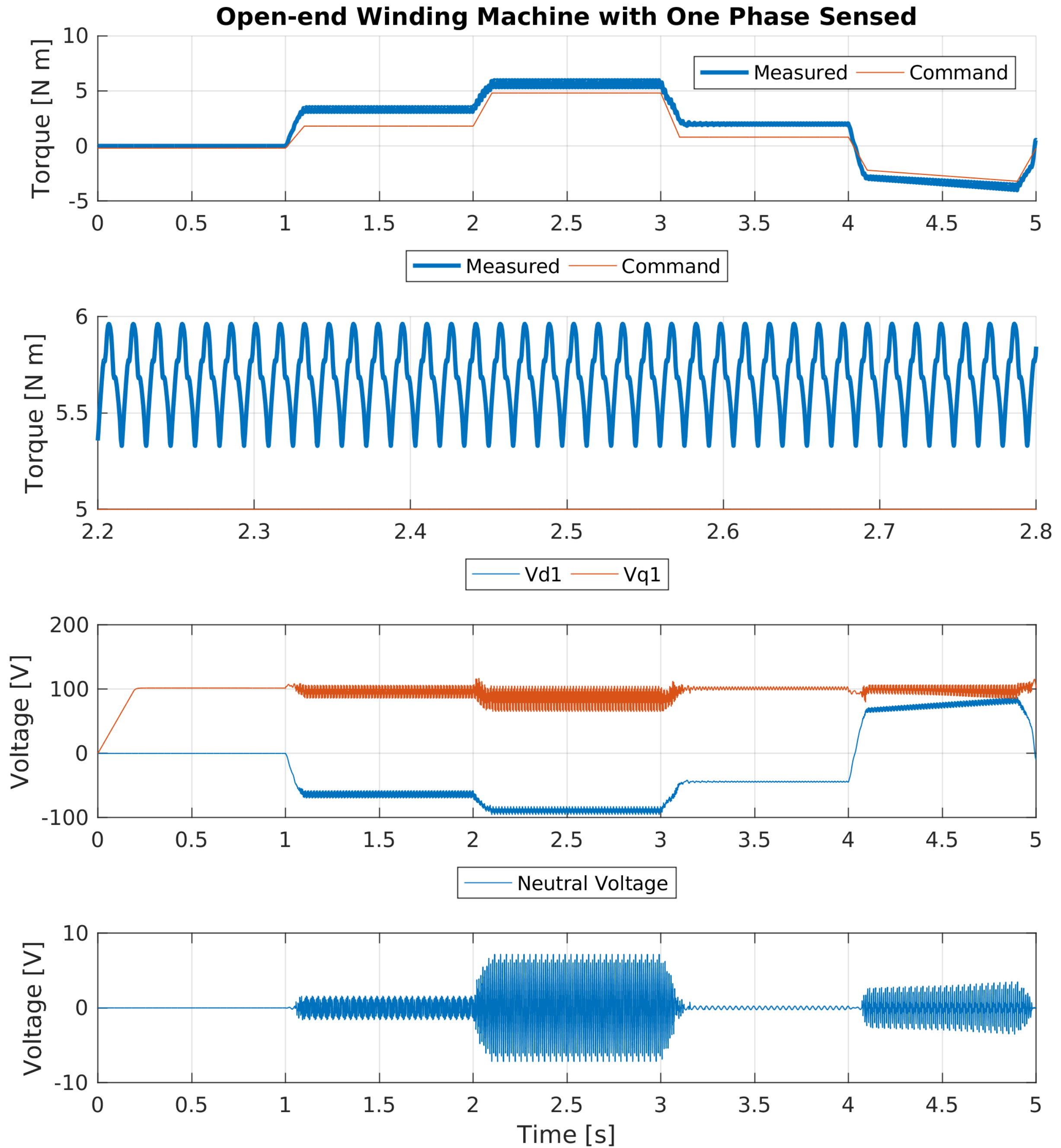


Figure 4.18 Simulation of an open winding distributed drive with asymmetries in which each H-bridge drive measures its phase winding current.

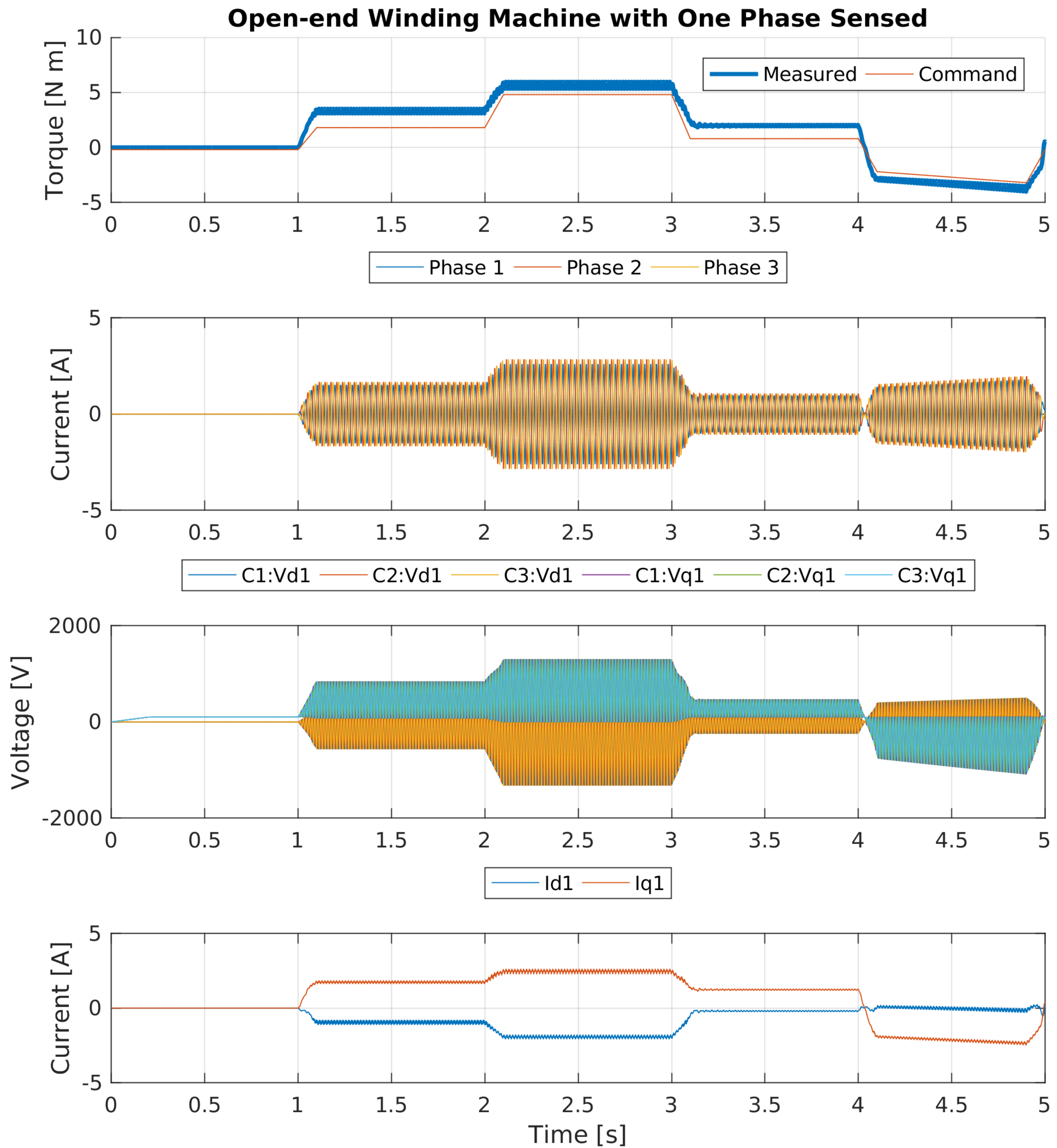


Figure 4.19 Simulation of an open winding distributed drive with asymmetries in which each H-bridge drive measures its phase winding current.

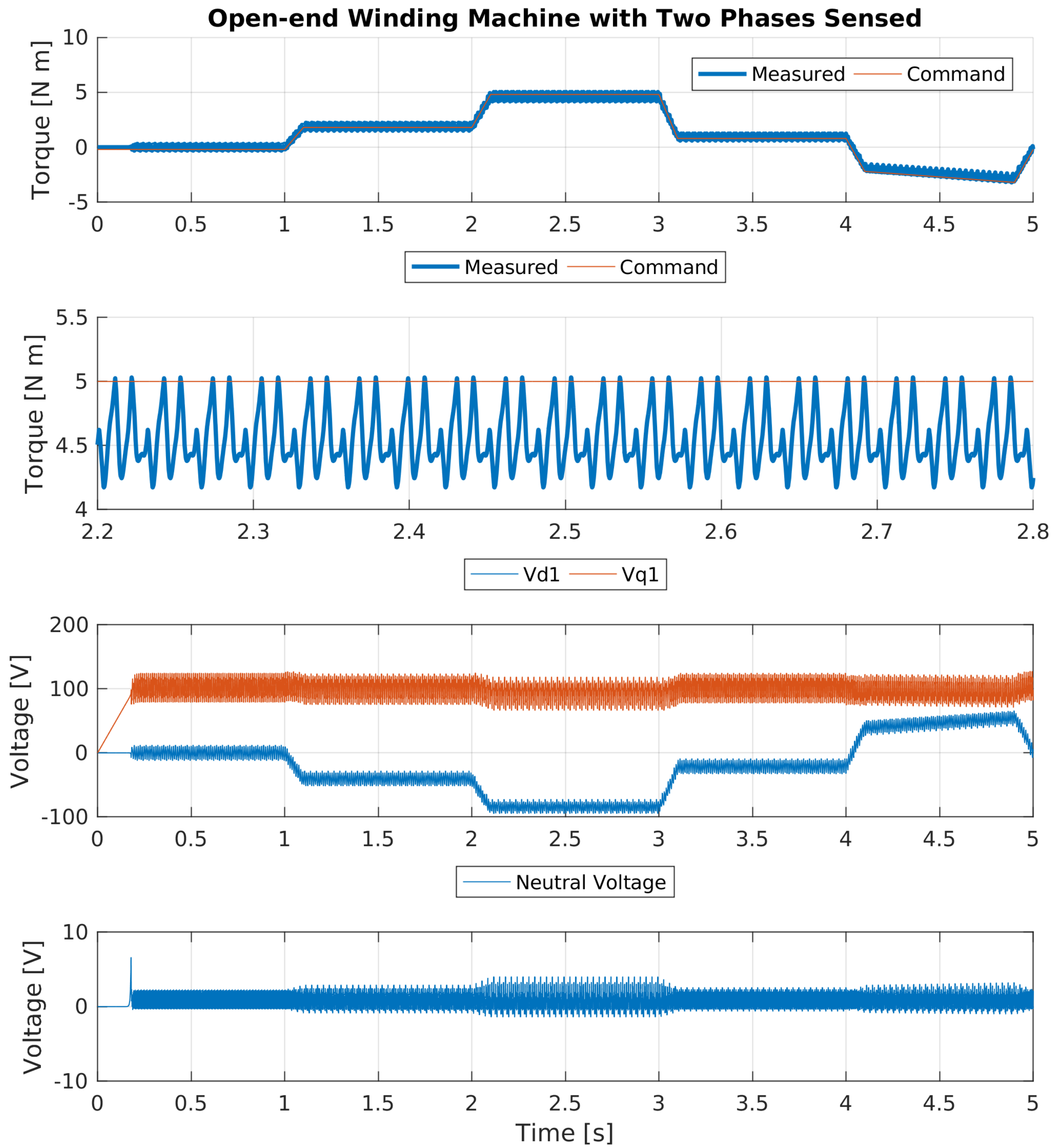


Figure 4.20 Simulation of an open-end winding distributed drive with asymmetries where each drive measures only two currents.

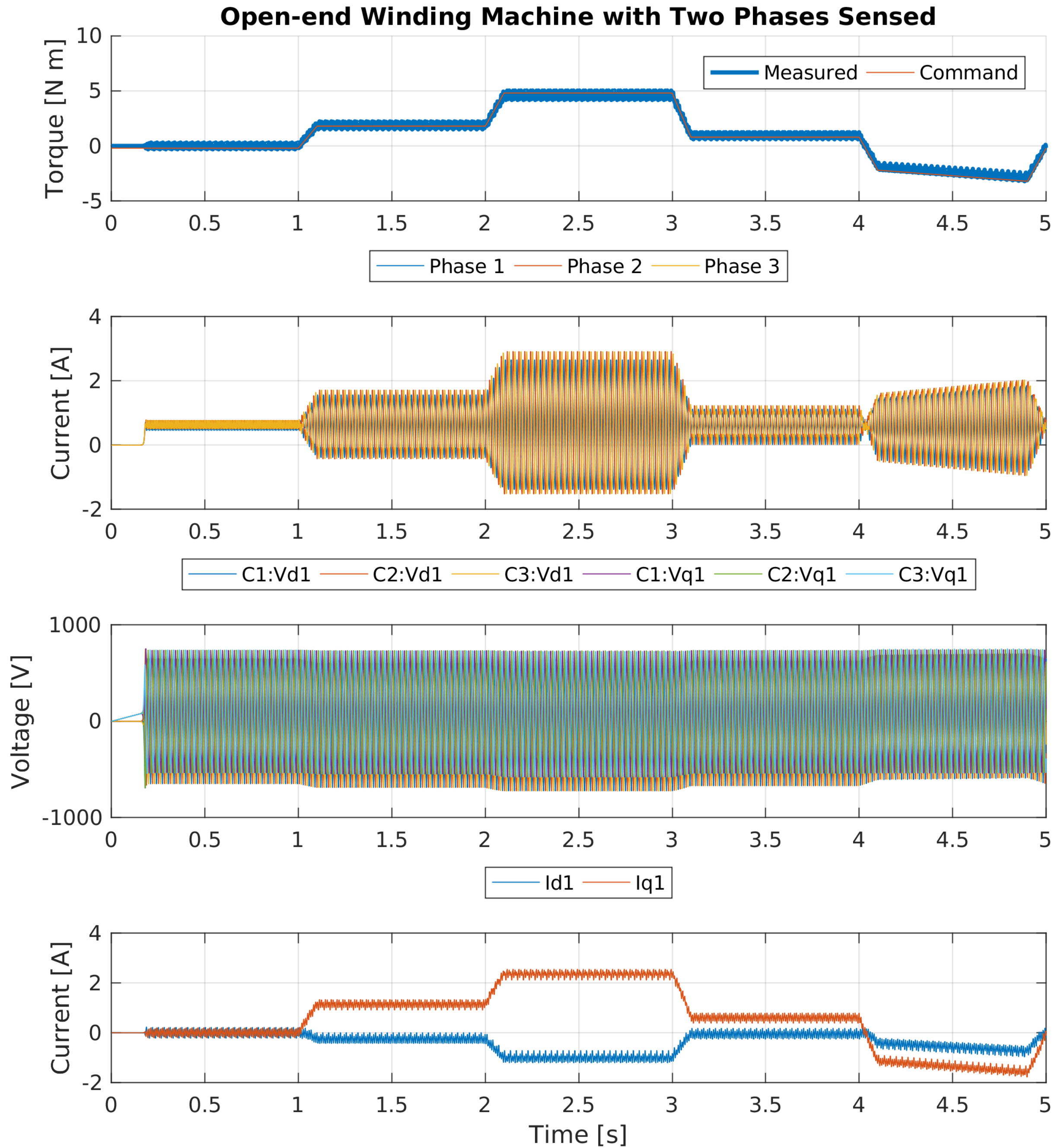


Figure 4.21 Simulation controller states of an open-end winding distributed drive with asymmetries where each drive measures only two currents.

current angle feedback and must operate solely on amplitude information. In order to determine the current angle, at least two sensors are required. However, when two sensors are used with an open winding machine configuration, the zero-sequence currents end up being mapped into the synchronous reference frame since they cannot be independently resolved. In order to resolve both the fundamental harmonic rotating reference frame current and the zero sequence current, at least three current sensors are needed. In the distributed drive system, this couples with the persistent angle error caused by sensor imbalance to drive the controllers into their voltage limits as shown in figures 4.20 and 4.21.

4.5 PWM Synchronization

Up to this point in the document, the question of how the issue of PWM generation and control synchronization in the modular controllers has not been addressed. In a monolithic drive, all PWM outputs share a common carrier which is usually a triangle waveform such that all output pulses are centered in the switching interval. This brings several benefits which include allowing sampling at the peaks of this carrier to cancel out most of the switching harmonics. As a result, the switching instants are rarely aligned between phases which reduces the average dV/dt in the machine, reducing both the EMI and capacitively-driven currents which damage bearings and winding insulation. Synchronization of the PWM carrier waveforms for all of the machine phases also helps to reduce the PWM-induced losses in the machine caused by higher-order switching harmonics.

This synchronized PWM method is not the only choice. PWM excitation works based on the frequency separation of the switching transients from the modulated waveforms. Electric machines and other *L-R* loads have a strong low-pass characteristic which means that if the controller bandwidth and controlled variables are sufficiently separated from the switching frequency, the current is independent of the precise PWM strategy. In [71] Holmes and Lipo derive an analytical

closed form for the harmonic content of a PWM output waveform for both triangular and sawtooth carriers. Both these waveforms have no difference in the baseband spectrum, only changes in the high-frequency content. They then go on to derive the three-phase output case by summing the single phase spectrum with appropriate phase shifts. These results mirror the single-output case in that the baseband waveform is unchanged by carrier choice, only the switching harmonics change. This is demonstrated in Figure 4.22 where both the synchronized carrier case and the non-synchronized carrier case show a three phase PWM power stage driving an identical R-L load. The ripple in the non-synchronized carrier case is at most double that of the synchronized carrier as it shifts the ripple down by an octave. This seems to be on the order of 5% of total losses when compared with modern switching loss minimization as described in [72], [73].

This result means that the controllers do not need phase synchronized PWM carriers in order to regulate a poly-phase current. If the carriers are not synchronized, the individual module carriers will drift in and out of phase on the order of several seconds to several minutes due to variation in the timing circuitry. This will cause increased voltage stress on the machine when the carriers line up such that switching instants align, but will not affect the fundamental voltages and currents.

This observation that carrier waveform synchronization is not required does not mean that it is not desirable. Although not addressed in this document, there are certainly opportuities for slower auxiliary control algorithms to be implemented in each modular controller that would cause the carrier waveforms for all of the modular phase drives to remain synchronized during normal operation. However, this discussion is intended to point out that synchronism is not an absolute requirement for continued operation of the drive. As a result, the occurrence of a control problem in one of the modular drives that disturbs the synchronism of its PWM carrier waveform with the other phases does not cause any immediate problem that will disturb the ability of the complete drive to continue its normal operation with high-performance torque control.

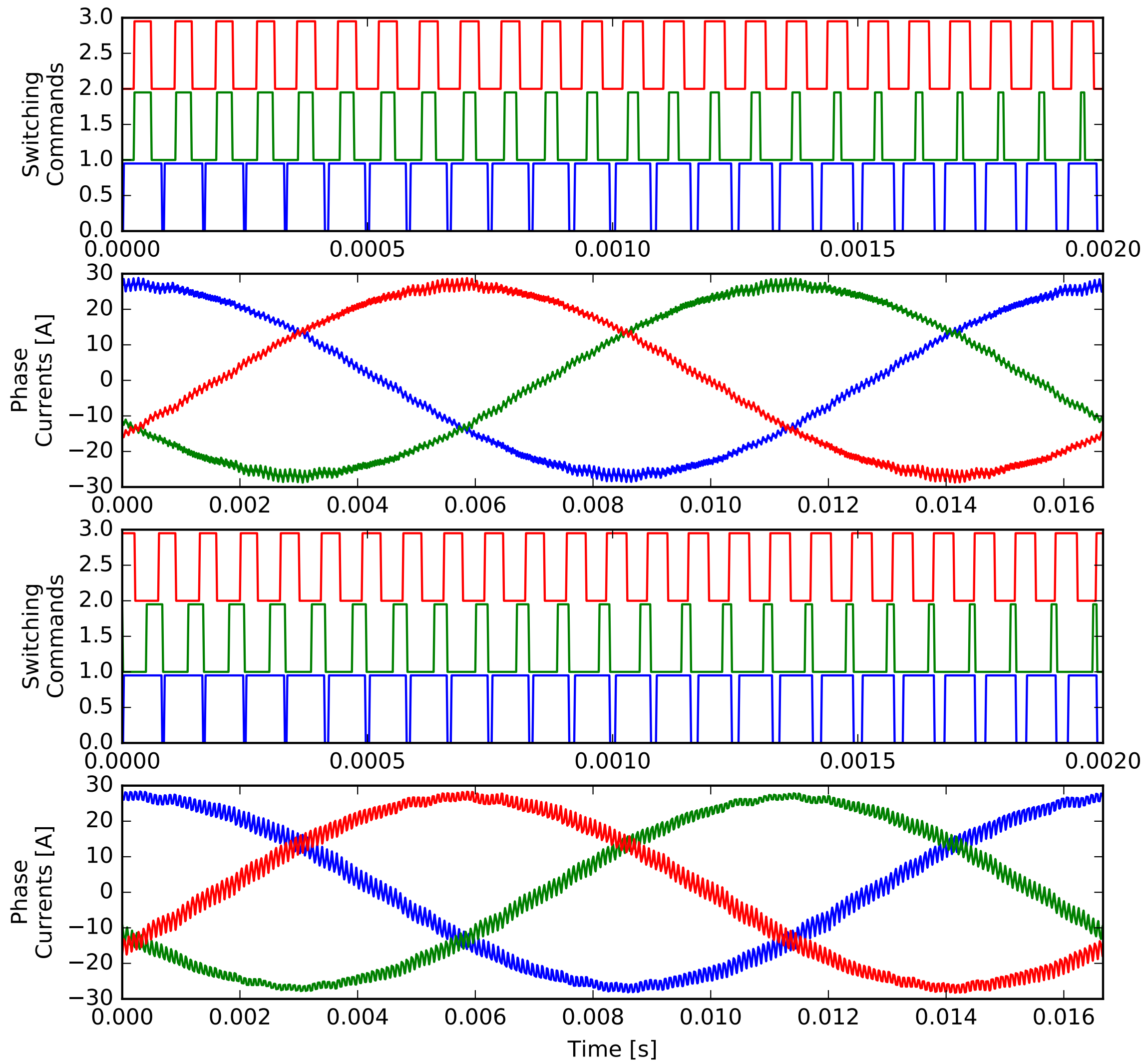


Figure 4.22 Simulated waveforms showing a three-phase inverter driving an R-L load with synchronized PWM carriers (top two plots) and with non-synchronized PWM carriers (bottom two plots).

These slight changes in carrier and control loop timing will have a small effect on control gains and bandwidths. However, since the timing is controlled by quartz crystal clocks, these errors are on the same relative order as the oscillator tolerance of a few tens of ppm which in negligible compared to parameter uncertainty or thermal parameter changes. The lack of a common control timebase also means that this report is currently limited to continuous time modeling for the moment as varying the controller phase shift in a discrete time model is a serious complication.

4.6 Conclusions

This chapter introduced the distributed current regulator. First, a single phase R-L load was presented, controlled by two voltage sources with local PI current regulation. This provided a tractable example in order to introduce the problem of distributed control causing an over constrained control problem. The added controller integration states need extra feedback in order to be stable. The example system was decomposed into differential-mode and common-mode control loops which clearly showed where the extra state arose and indicated a practical method of feedback. Neutral-point or common-mode feedback was then applied to bring the extra states under control.

Next, this concept was applied to a three phase machine where each phase was controlled by its own independent controller with two phases of current measurement in order to obtain a view of the rotating current vector in the machine. The system was then simulated with no additional feedback and showed that the system current response was comparable to a monolithic drive. Without any extra feedback, however this system was simulated to have an uncontrolled integration state which caused the zero-sequence voltage to run away.

The neutral point feedback method was again applied, this time with the necessary coordinate transforms to map the uncontrolled neutral voltage back into the integration state errors it is meant to measure. This feedback method was then simulated to show that it tamed the uncontrolled

integration leading to a bounded neutral point voltage waveform while maintaining good current control.

Finally, an observer based approach was proposed which models the machine system along with an ideal drive in order to derive signal that approximates the neutral voltage. This modeled controller succeeded in taming the system unbalance but introduced a critical parameter sensitivity that needs further investigation.

Chapter 5

High Phase Order Generalization

Only three-phase machines have been considered up to this point in this document. In this chapter, these results are extended to higher phase-order numbers.

First, the symmetrical-winding six-phase machine that was used in previous work must be acknowledged as being a three-phase machine for the purposes of analysis. In that case, even though there are six physical phase windings, they occupy slots that cause pairs of them to generate the same electrical winding function. This means that even though there are more than three phases, there are only three control axes available to the machine (i.e., a first spatial harmonic and a zero sequence). This is not a bad thing from a control standpoint since the higher-order reference frames do not need to be considered and cannot cause losses or instability. However, a fault-tolerant machine having independent electrical axes allows for enhanced sensing opportunities and for smoother torque production during faulted conditions. Furthermore, as machines are extended to higher numbers of physical phases, especially non-triplen phase counts, the access to higher-order control axes becomes inevitable.

5.1 Coordinate Transforms and Terminology

Until this point, this document has dealt with both machine and controller models as scalar variables. When there is only a single harmonic reference frame and only three phases this is

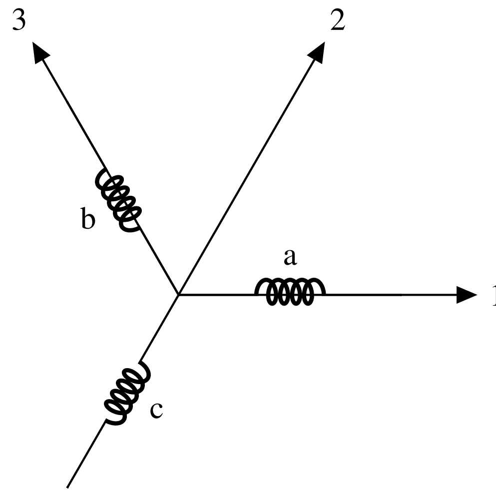


Figure 5.1 Axes and sign conventions for logical and physical phase transforms in a three-phase system.

tractable, but as the analysis is generalized to higher phase-orders, it must be switched to a vector-based notation in order to clearly describe the control equations.

First, a terminology and sign convention must be defined for describing polyphase machines and their windings. The discussion that follows will use the terminology from [67] where phase order is the count of independent current axes, with the axes numbered only in the first 180 degrees of electrical angle. Different winding configurations are distinguished by a polarity matrix mapping physical phases to logical phases. This is in contrast to the typical terminology which divides the full 360 degrees of electrical angle evenly into the number of phases. This slightly more tedious terminology is used to resolve the ambiguity between different winding arrangements in higher phase-order machines and to make explicit the number of controllable harmonic spaces. Figure 5.1 shows the three control axes and the location of the three windings in machines considered thus far. For this arrangement, the polarity matrix is defined in equation 5.1 which translates between physical winding quantities and logical phase quantities. In this equation, x can refer to either currents or voltages. The inverse of this matrix is equal to its transpose and is used to translate

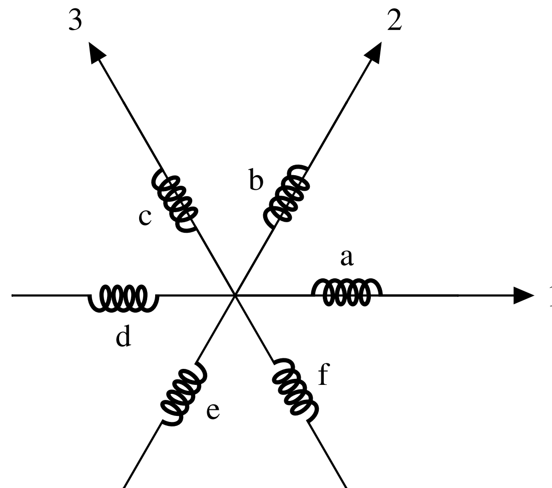


Figure 5.2 Axes and sign conventions for logical and physical phases transforms in a six-winding system with sixty degrees between windings.

quantities in the other direction.

$$\begin{bmatrix} x_a \\ x_b \\ x_c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{5.1}$$

A common phase arrangement in fault tolerant machines is often called a six-phase machine, but has each phases' electrical axis separated by 60 degrees. This is diagrammed in Figure 5.2. By the use of independent current axes as the phase order, this is still only a three phase machine as it only has windings along three electrical axes. The polarity matrix in this case is shown in equation 5.2.

$$[\mathbf{P_{60^\circ}}] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{5.2}$$

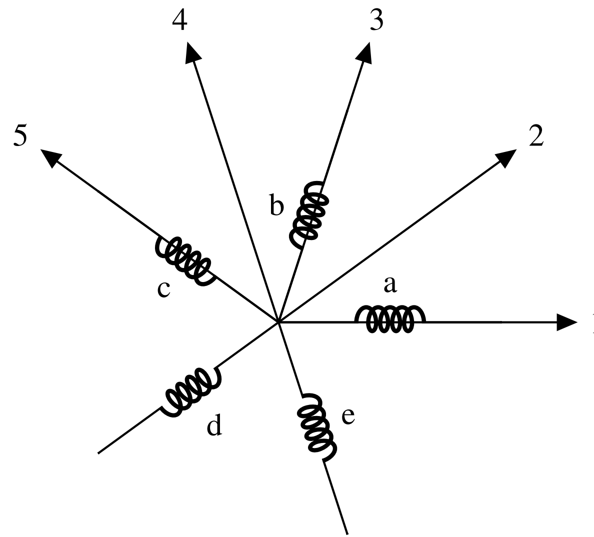


Figure 5.3 Axes and sign conventions for logical and physical phases transforms in a five phase system.

Later, this chapter will describe the control of a five-phase machine. This, like the three phase machine only has one winding arrangement, shown in Figure 5.3. In this case the polarity matrix is defined in equation 5.3.

$$[\mathbf{P_5}] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \end{bmatrix} \tag{5.3}$$

Both the three phase and sixty-degree separation six-winding case above have only three active logical phases so the stationary and synchronous frame vectors are identical to the three phase case above. In the five phase case, has the ability to cause currents in a higher-order spatial harmonic than the fundamental. In general all harmonics modulo the number of independent current vectors are accessible. In the three phase case, the means only the positive and negative sequences of the first spatial harmonic. The third harmonic is equivalent to the zero-sequence. This is used to

increase voltage headroom in some inverters. In the case of the five-phase machine, there is access to the positive and negative first spatial harmonic, as well as the positive and negative third spatial harmonic. The fifth harmonic maps to the zero sequence. This results in the new stationary and synchronous frame vectors shown in equation 5.4. Again, each x can be either a voltage or current.

$$x_{\alpha\beta 0} = \begin{bmatrix} x_{\alpha 1} \\ x_{\beta 1} \\ x_{\alpha 3} \\ x_{\beta 3} \\ x_0 \end{bmatrix} \qquad x_{dq0} = \begin{bmatrix} x_{d1} \\ x_{q1} \\ x_{d3} \\ x_{q3} \\ x_0 \end{bmatrix} \tag{5.4}$$

The transforms between these reference frames again are

$$x_{ab} = [\mathbf{P}]x_{12} \tag{5.5}$$

$$x_{12} = [\mathbf{C}]x_{\alpha\beta 0} \tag{5.6}$$

$$x_{\alpha\beta 0} = [\mathbf{R}(\theta)]x_{dq0} \tag{5.7}$$

$$x_{ab} = [\mathbf{P}][\mathbf{C}][\mathbf{R}(\theta)]x_{dq0} \tag{5.8}$$

These later two again are the Clarke transform, shown in general form in equation 5.9 and the Park Transform shown in equation 5.10. In these cases, the final special column is only present in odd-phase order systems. $\delta = \frac{n}{\pi}$ is the angle between logical phases.

$$[\mathbf{C}] = \begin{bmatrix} 1 & 0 & 1 & 0 & \cdots & \frac{1}{\sqrt{2}} \\ \cos\delta & \sin\delta & \cos 3\delta & \sin 3\delta & \cdots & \frac{-1}{\sqrt{2}} \\ \cos 2\delta & \sin 2\delta & \cos 6\delta & \sin 6\delta & \cdots & \frac{1}{\sqrt{2}} \\ \cos 3\delta & \sin 3\delta & \cos 9\delta & \sin 9\delta & \cdots & \frac{-1}{\sqrt{2}} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \cos(n-1)\delta & \sin(n-1)\delta & \cos(n-3)\delta & \sin(n-3)\delta & \cdots & \frac{1}{\sqrt{2}} \end{bmatrix} \tag{5.9}$$

$$[\mathbf{R}(\theta)] = \begin{bmatrix} \sin(\theta) & \cos(\theta) & 0 & 0 & \cdots & 0 \\ -\cos(\theta) & \sin(\theta) & 0 & 0 & \cdots & 0 \\ 0 & 0 & \sin(3\theta) & \cos(3\theta) & \cdots & 0 \\ 0 & 0 & -\cos(3\theta) & \sin(3\theta) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \tag{5.10}$$

5.2 High Phase-Order Machine Model

Next, at the machine model equations from Chapter 3 must be translated to be applied to machines of arbitrary phase order. In that chapter the machine equations are defined both in current form and in flux form with individual phase variables. In order to allow for more generalized modeling of arbitrary phase orders these equations must be redefined in terms of vectors. First, observe that both $x_{\alpha\beta 0}$ and x_{dq0} vectors can be thought of as complex vectors of $n/2$ elements. In that case, the imaginary unit is defined as a matrix.

$$[\mathbf{j}] = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ -1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ 0 & 0 & -1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{5.11}$$

Again, the final row and column are removed if the system has an even number of phases.

Using this the equations 3.1, 3.2, and 3.3 can be restated as equation 5.12 where λ_{pm} is now a vector showing the magnet flux in each of the spatial reference frames. In a typical machine there will be most flux in the fundamental reference frame direct axis with smaller amounts of flux in

higher order direct axes.

$$v_{dq0} = r_s i_{dq0} + [\mathbf{L_{dq0}}]\frac{di_{dq0}}{dt} + \omega[\mathbf{j}][\mathbf{L_{dq0}}]i_{dq0} + \omega[\mathbf{j}]\lambda_{pm} \tag{5.12}$$

For simulation, use the flux equation and form this in terms of derivatives of fluxes as in equation 5.13.

$$\lambda_{dq0} = [\mathbf{L_{dq0}}]i_{dq0} + \lambda_{pm} \qquad \frac{d\lambda_{dq0}}{dt} = v_{dq0} - \omega[\mathbf{j}]\lambda_{dq0} - r_s[\mathbf{L_{dq0}}]^{-1}\lambda_{dq0} \tag{5.13}$$

$$T = \frac{P}{2}\frac{n}{2}([\mathbf{j}]\lambda_{dq0})^T i_{dq0} \tag{5.14}$$

It is notable that the inductance is now a matrix rather than a set of scalars. Also, the inductance depends on whether the equation is formed in the phase, stationary, or synchronous reference frame. These matters are much more fully treated in chapter 5 of [67] and will only be summarized for the synchronous reference frame here. The matrix $[\mathbf{L_{dq0}}]$ which will be used for this work is a simplified form that only includes the major non-zero components. Along the diagonal are the fundamental inductance components of each spatial harmonic. Only the first spatial harmonic has saliency which is the same as the saliency as in a three phase system. In addition to these reference frame self inductances, there are off-axis cross coupling inductances between adjacent spatial reference frames. These arise as a mixing product of the coils' winding harmonics and the second spatial harmonic of the effective air gap with gives rise to the saliency in the fundamental reference frame. These inter-harmonic coupling inductances are symmetric. As this inductance matrix is band-diagonal and

symmetric, it is trivially invertible. The general form is shown in equation 5.15

$$[\mathbf{L_{dq0}}] = L_{ls}[\mathbf{I}] + \begin{bmatrix} L_{0s_1}+L_{2s_1} & 0 & -L_{2s_3} & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & L_{0s_1}-L_{2s_1} & 0 & -L_{2s_3} & 0 & \cdots & 0 & 0 & 0 \\ -L_{2s_3} & 0 & L_{0s_3} & 0 & -L_{2s_5} & \cdots & 0 & 0 & 0 \\ 0 & -L_{2s_3} & 0 & L_{0s_3} & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & L_{0s_{n-2}} & 0 & -\sqrt{2}L_{2sn}\sin n\theta_r \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & L_{0s_{n-2}} & -\sqrt{2}L_{2sn}\cos n\theta_r \\ 0 & 0 & 0 & 0 & 0 & \cdots & -\sqrt{2}L_{2sn}\sin n\theta_r & -\sqrt{2}L_{2sn}\cos n\theta_r & L_{0s_n} \end{bmatrix} \tag{5.15}$$

In the stationary reference frame and phase reference frame, this inductance matrix depends on the rotor angle as with any salient pole machine. In chapter 6 we present the inductance matrix of the machine we will be using for hardware experiments when wired in a 9-phase and 3-phase configuration.

5.3 Modular Phase Drives

In order to implement modular phase drives, we use the same methods as the previous chapter. First, choose a connection where each phase drive controller is delivering one phase voltage, and is operating with a programmed rotor offset angle such that that phase voltage is at zero rotor angle for each module. We then wire the current sensor from the module itself and the next module around the machine in as current feed back inputs. As this higher-phase order drive cannot reconstruct all phase currents exactly from two measurements, we have to create a conditioning matrix to extract the fundamental current vector from two arbitrary measurements. In this case, we can use the Clarke and Polarity matrices defined above as shown in equation 5.16. The subscript shown here is a block-subscripting where we take the upper square block.

$$[\mathbf{G_{fb}}] = \left(([\mathbf{C}][\mathbf{P}])^{-1}_{1:2,1:2}\right)^{-1} \tag{5.16}$$

This matrix will give us a feedback for the fundamental spatial frame only, but that is sufficient for good performance torque production in most cases.

For the simulation that follows, we will be using a five phase machine controlled by five independent drives. Each drive controls one phase output and senses two current inputs. A schematic of this arrangement is shown in Figure 5.4. The two current sensors are used to reconstruct a first-harmonic reference frame rotating current vector as was described in equation 5.16.

As this machine has five phases, we have access to both the first and third spatial harmonic currents and voltages.

Each phase controller is internally operating using the same equations and form as in Chapter 3. As we are only providing current feedback for the first harmonic reference frame, we don't need to change anything in the controller. If access to other reference frames is wanted, another current sensor can be employed and the controller can be duplicated into the third harmonic reference frame.

5.3.1 Simulation

The simulations that follow are carried out in simulink using the vector formulation of the machine and controller. Figure 5.5 shows the block diagram of machine implementing equation 5.13. The machine model also includes coordinate transforms needed to translate between phase quantities and the synchronous reference frame.

The controller model is formed in the same way as the model from Chapter 3 though now in vector form. This block diagram is shown in Figure 5.6. This vector-based controller allows this model to be generalized to controlling higher-order reference frames if the current feedback needed is included. Of note, the polarity matrix and Clarke transform in the controller is replaced in this case with the $[\mathbf{G_{fb}}]$ matrix from equation 5.16 which take a generalized subset of the phase currents and reconstructs the fundamental current vector.

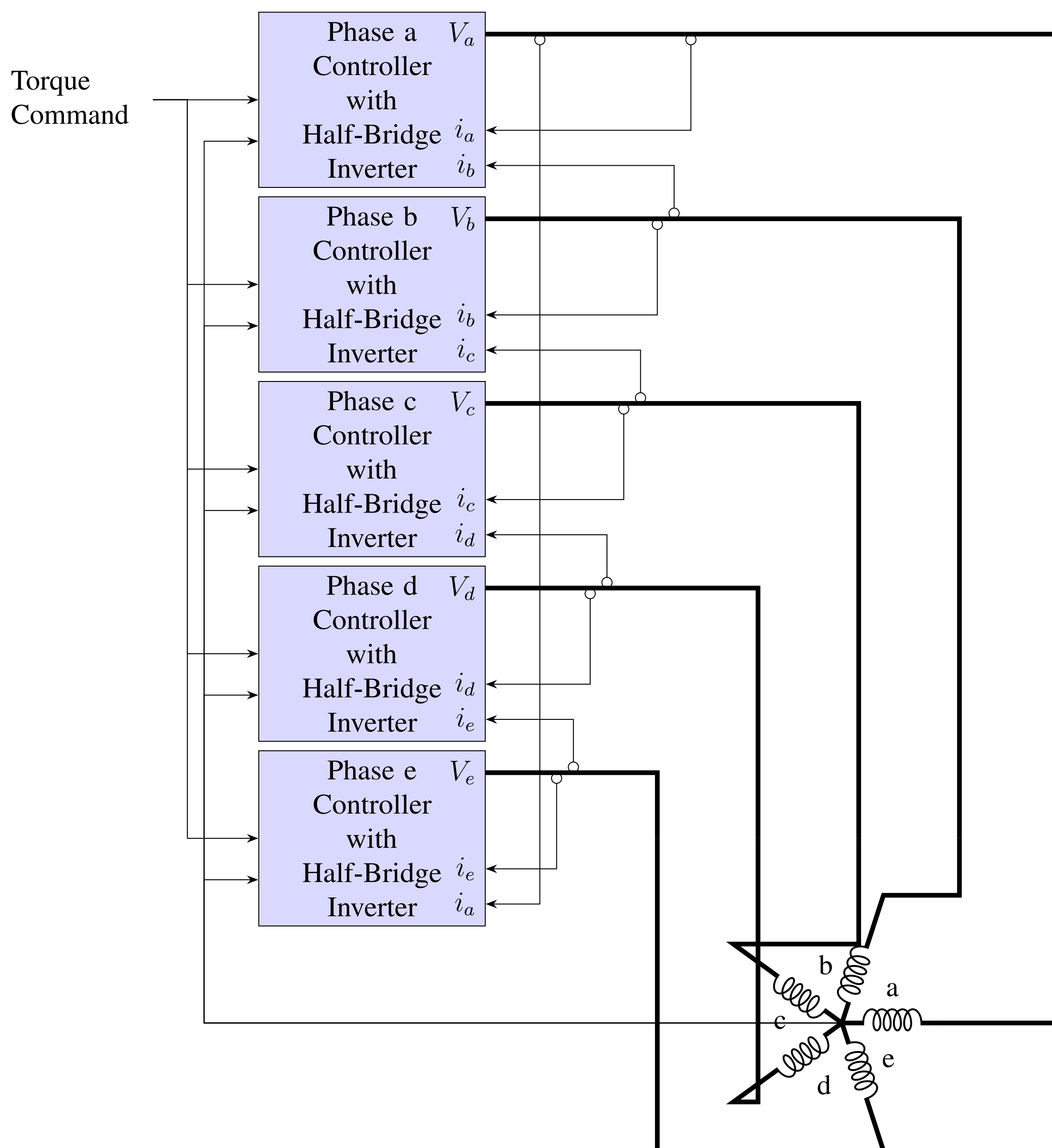


Figure 5.4 Schematic of a distributed drive controlling a five-phase machine.

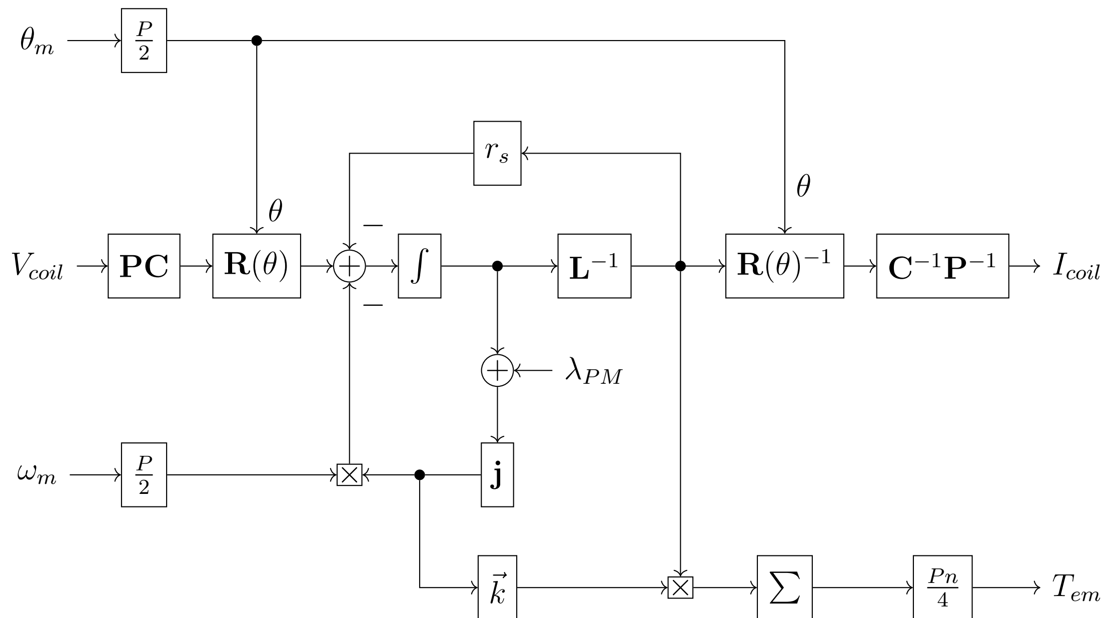


Figure 5.5 Block diagram implementing equation 5.13 for arbitrary phase orders.

For simulation, we use the same torque command series that was used in the previous two chapters. Again, like in Chapter 4 we start with no feedback other than current sensors, and we adjust the current sensor gain so that there is a ten percent gain mismatch. The waveforms for this case are shown in Figure 5.7. Again, like in the three-phase case, the current waveform is well behaved but the voltage output shows an uncontrolled integration state which manifests as an increasing zero sequence voltage. Again, for the sake of simulation we have no voltage limit in order to show how current control is maintained even when the voltage is unstable.

Next, we implement proportional neutral point feedback. Using the same current gain settings and connection as when we generated Figure 5.7 we now run with neutral-point feedback. The waveforms in Figure 5.9 show that neutral point feedback is capable of stabilizing a higher-phase order distributed drive with ripple similar to that of a monolithic drive running with the same sensor arrangement.

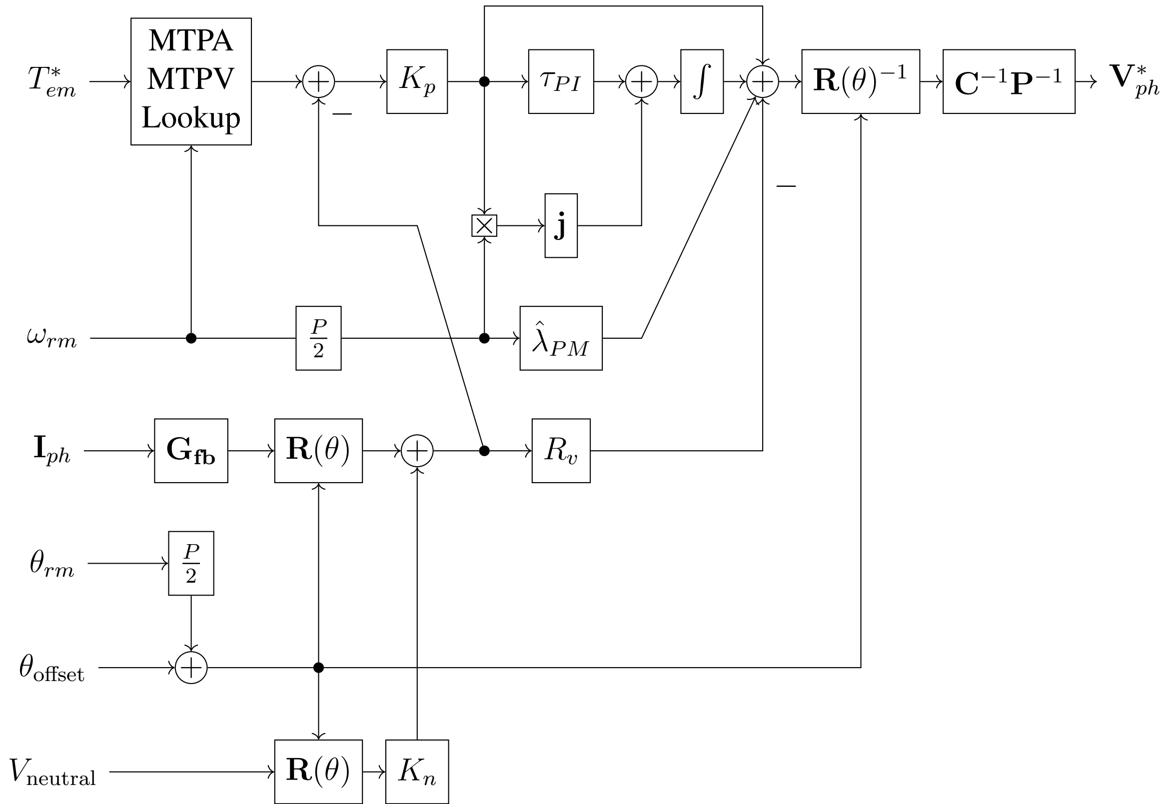


Figure 5.6 Simulink block diagram implementing neutral-point stabilized vector control for arbitrary phase orders with programmable number of input current sensing channels.

5.4 Faulted Mode Operation

One of the major motivations of drive modularization is to reduce the number of single-point failures in a motor drive system. In order to achieve this, a distributed drive must be capable of operating even after loss of one or more drive modules.

While the three-phase machine may be common, it has very limited options when operating after a fault due to the fact that there are only two remaining terminals to excite if the faulted phase is open-circuited. This assumes that the neutral point of the three-phase wye-connected machine winding is floating, which has been the assumption throughout this research program. The availability of only two phase terminals to excite following the fault confines post-fault operation

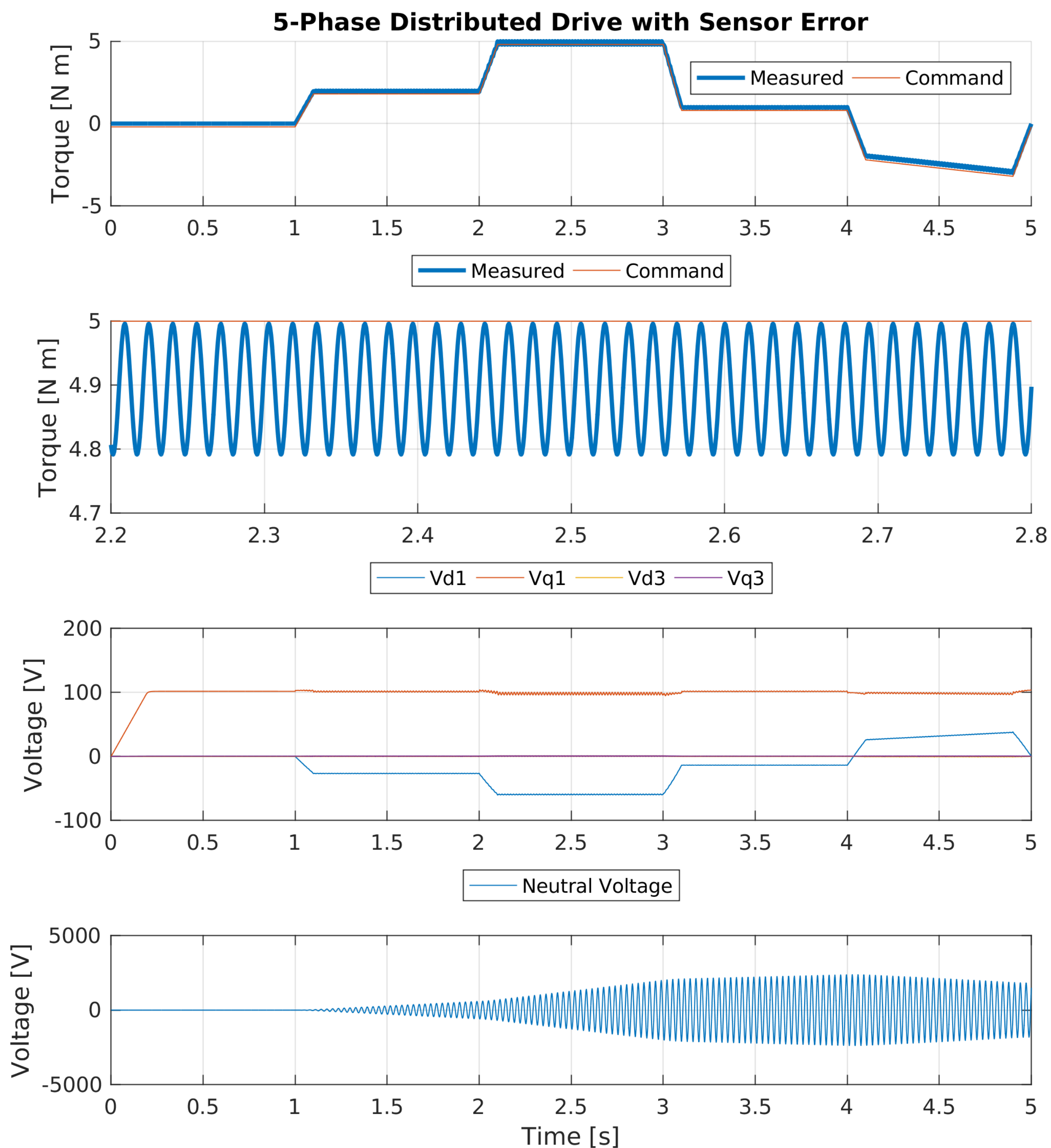


Figure 5.7 Five-phase distributed drive operating without correction with a ten percent current sensor gain error and only two phases current feedback to each distributed drive. Torque and machine terminal voltages are shown.

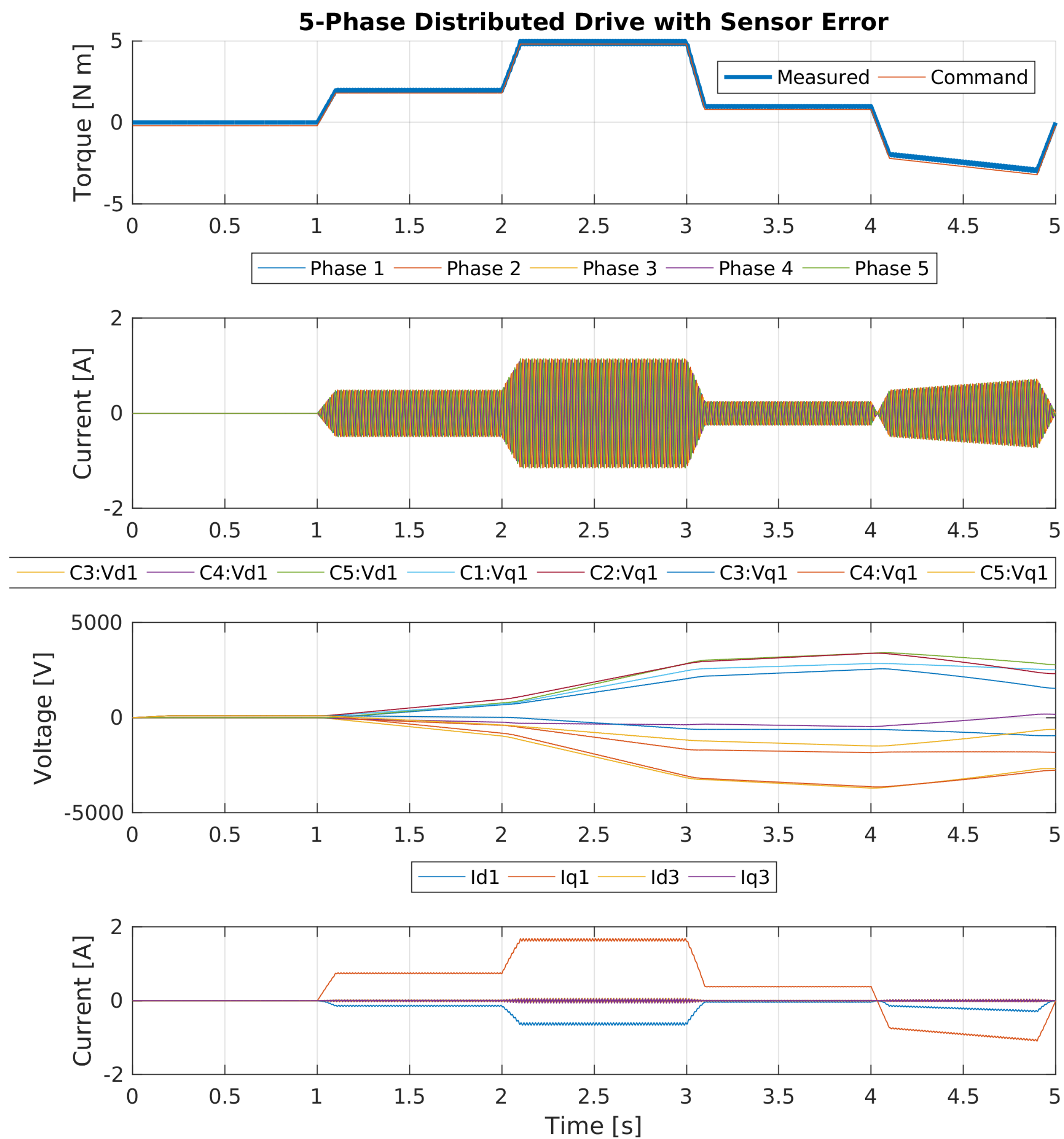


Figure 5.8 Five-phase distributed drive operating without correction with a ten percent current sensor gain error and only two phases current feedback to each distributed drive. Torque, terminal currents, and controller command voltages are shown.

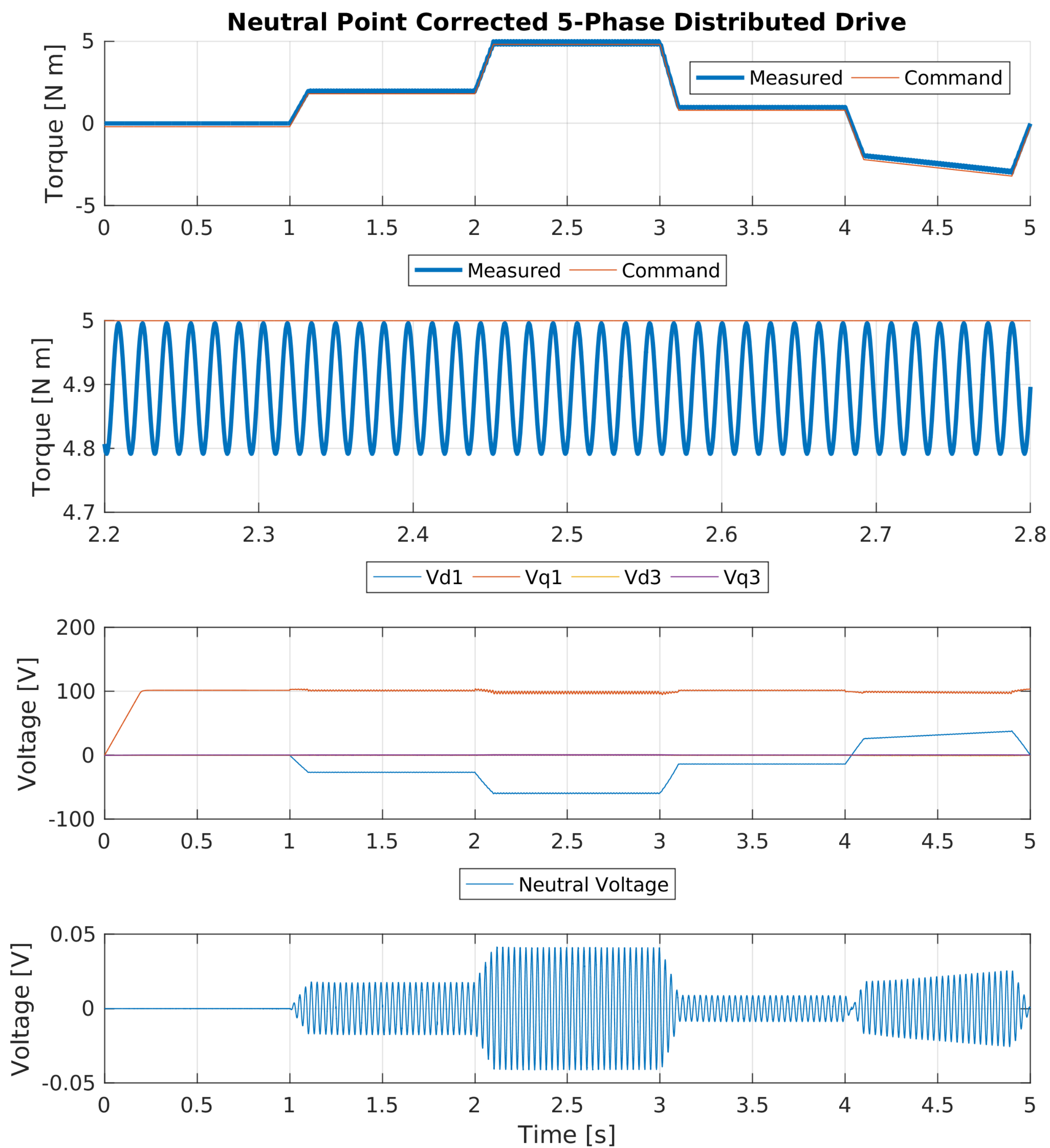


Figure 5.9 Five-phase distributed drive operating with neutral point feedback correction with a ten percent current sensor gain error and only two phases current feedback to each distributed drive. Torque and machine terminal voltages are shown.

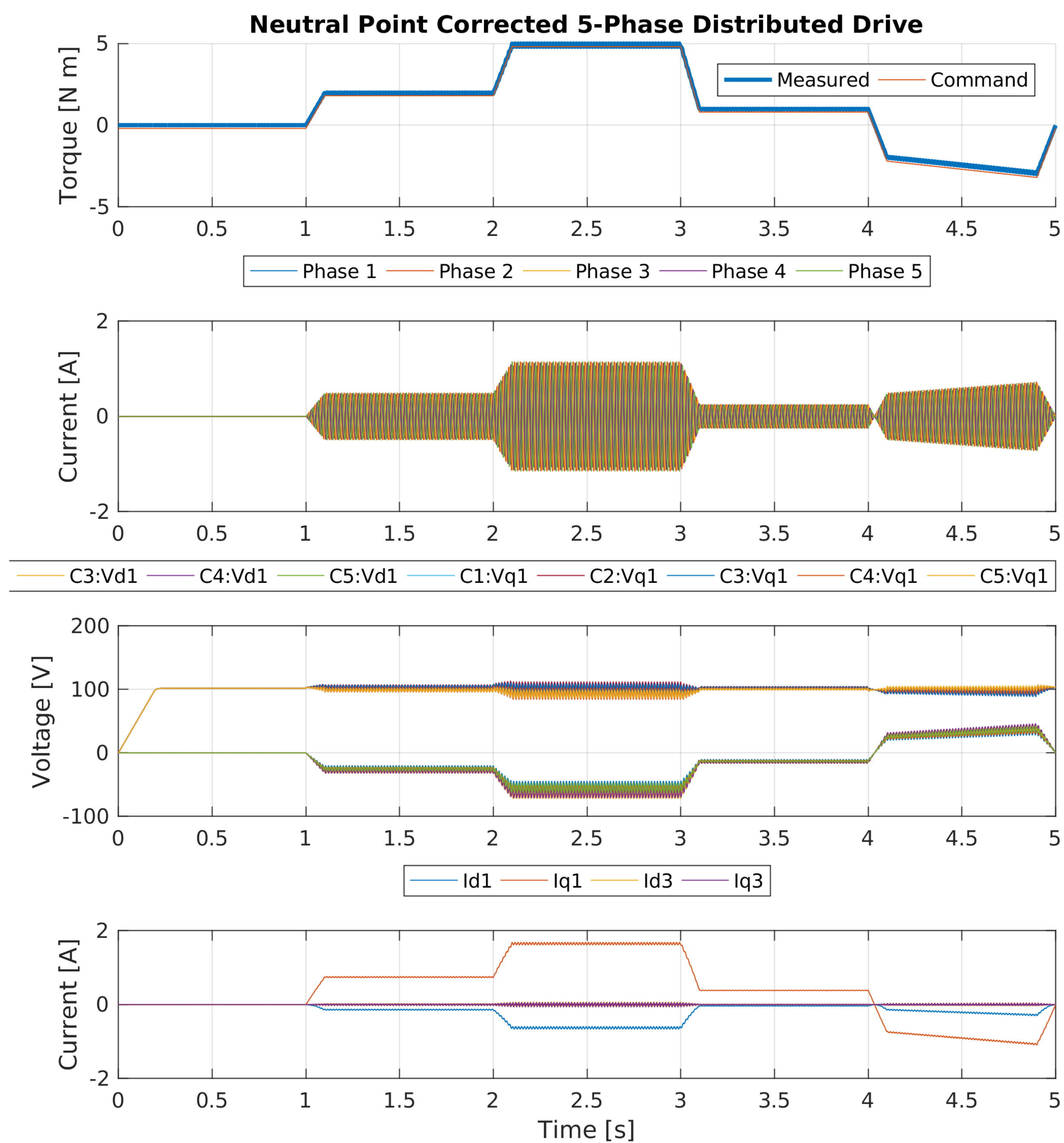


Figure 5.10 Five-phase distributed drive operating with neutral point feedback correction with a ten percent current sensor gain error and only two phases current feedback to each distributed drive. Torque, terminal currents, and controller command voltages are shown.

of a three-phase machine to single-phase excitation of the two healthy phase windings connected in series. As a result, there is no opportunity to maintain smooth torque under these constrained conditions [58]. To circumvent this serious limitation, the investigation of fault-mode operation during this research program has focused on a five-phase machine with a floating wye winding configuration following loss of excitation to one of its phase windings. This five-phase machine was chosen since it is the minimal phase order number for which loss of excitation to one phase winding still allows full control of the fundamental stator MMF which is essential to retaining smooth, controlled torque production.

When considering fault tolerance, there are many options as were covered in in Chapter 2. For this investigation, a method is considered that does not alter the distributed control structure, but merely modifies the individual phase current commands to add the necessary lost MMF from the faulted phase. This approach allows the dynamic performance of the machine under fault-mode conditions to remain the same as that of the unfaulted drive when it is operating in field-oriented vector control. This fault tolerance strategy allows for a wide variety of post-fault exitation options while retaining the same distributed control structure. For this work, the constant fundamental MMF option was chosen as it is well understood, having been derived in closed form by Fu and Lipo over 20 years ago [63]. While this method does not achieve any of the measures of optimality, it does have the virtue of simplicity. If will work even when only one synchronous reference frame can be regulated due to limited sensing, as is likely to be the case with a distributed drive during fault-mode operation.

Figure 5.11 shows the current waveforms for both a healthy five-phase machine, and a machine with one phase-open circuited. The smoothed waveforms shown are the result of adding a set of current components to the remaining healthy phases which combine to deliver the missing MMF from the opened phase. In order to maintain the same MMF amplitude after shifting the phase angle of the healthy phase currents, the current amplitudes must be increased by 40 percent. This is higher

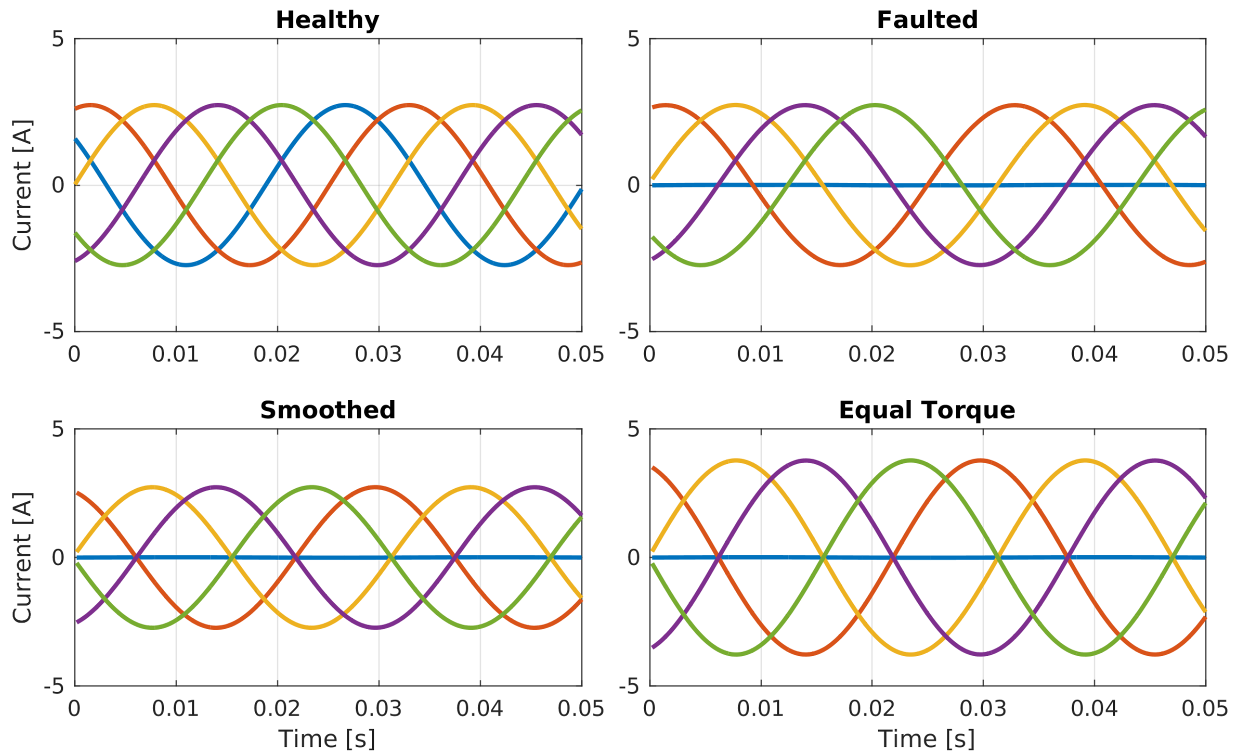


Figure 5.11 Simulated current waveforms for a five-phase machine showing: 1) healthy operation ("Healthy"); 2) one open-circuited phase winding ("Faulted"), 3) a redistribution of the excitation current vector phase angles (but not amplitudes) in the remaining four healthy phases ("Smoothed"); and 4) an adjustment of the current amplitudes and angles in the remaining healthy phases in order to restore the average torque to its pre-fault value ("Equal Torque").

than the expected 25 percent due to the fact that the healthy phases now have their current vectors shifted in such a way that they partially cancel out the MMF contributions of the other healthy phases.

The instantaneous torque waveforms accompanying each of these current excitation conditions is shown in figure 5.12. The loss of excitation to one phase causes nearly 50 percent torque ripple. This ripple is greatly reduced, though not completely eliminated by shifting the currents in the remaining phases to continue producing a constant fundamental MMF. The reason this constant fundamental MMF does not perfectly smooth the torque waveform is that the five-phase machine is capable of producing both magnet and reluctance torque from its third harmonic reference frame,

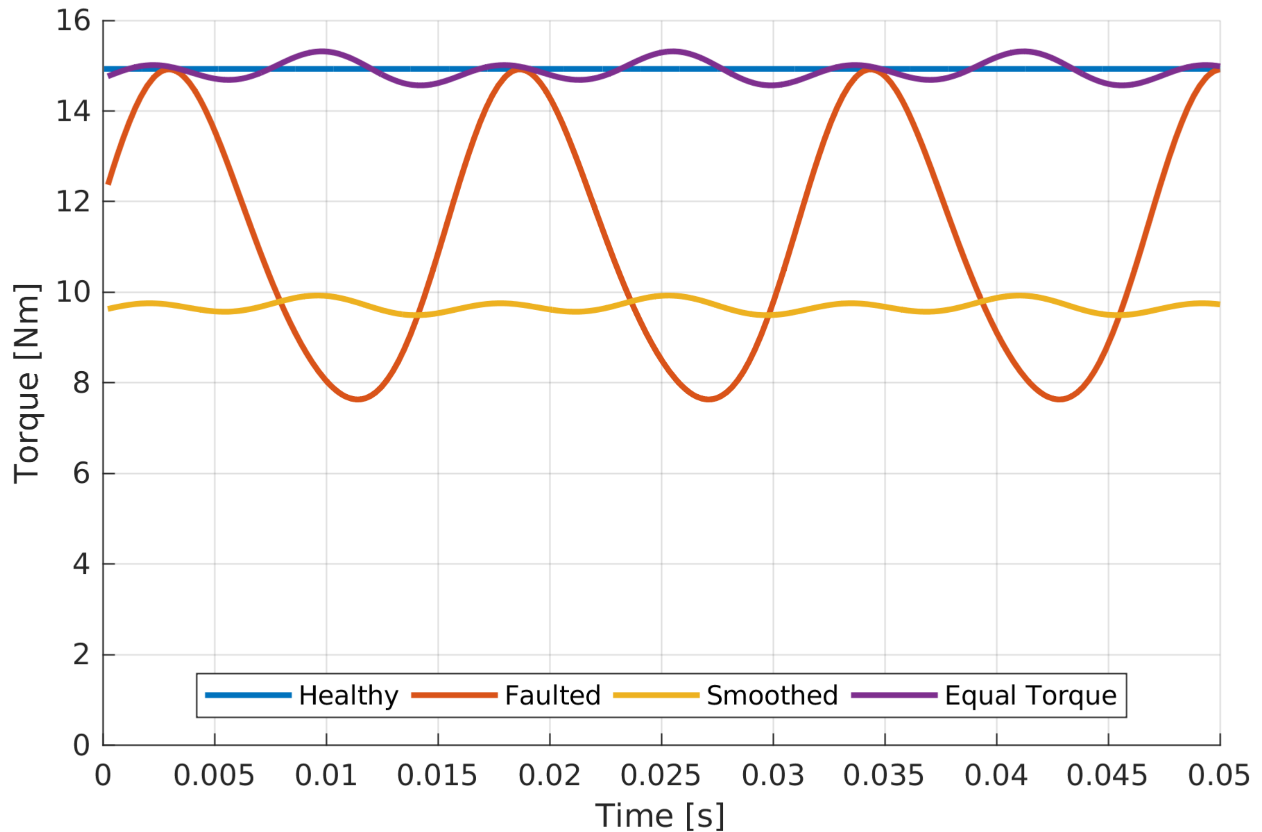


Figure 5.12 Instantaneous torque waveforms for a five-phase machine for each of the current excitation conditions from Figure 5.11.

and the fault-mode operating point does not preserve the zero amplitude of the third harmonic current component of the original balanced excitation operating point. This could be remedied by adding a compensating command to the third harmonic reference frame, but that is not done here as it would require at least three phase current feedback signals per phase controller in order to independently resolve both the first and third harmonic reference frame currents.

With the drive structure presented here, fault-mode operation has one more responsibility. Because the neutral point voltage must be controlled to keep the modular drives synchronized, the impact of the fault on this feedback loop must be addressed as well. Depending on how the neutral voltage is measured, it may or may not include the back-EMF of the faulted phase. This back-EMF may be included if the neutral voltage is formed by a resistive divider at the drive module terminals

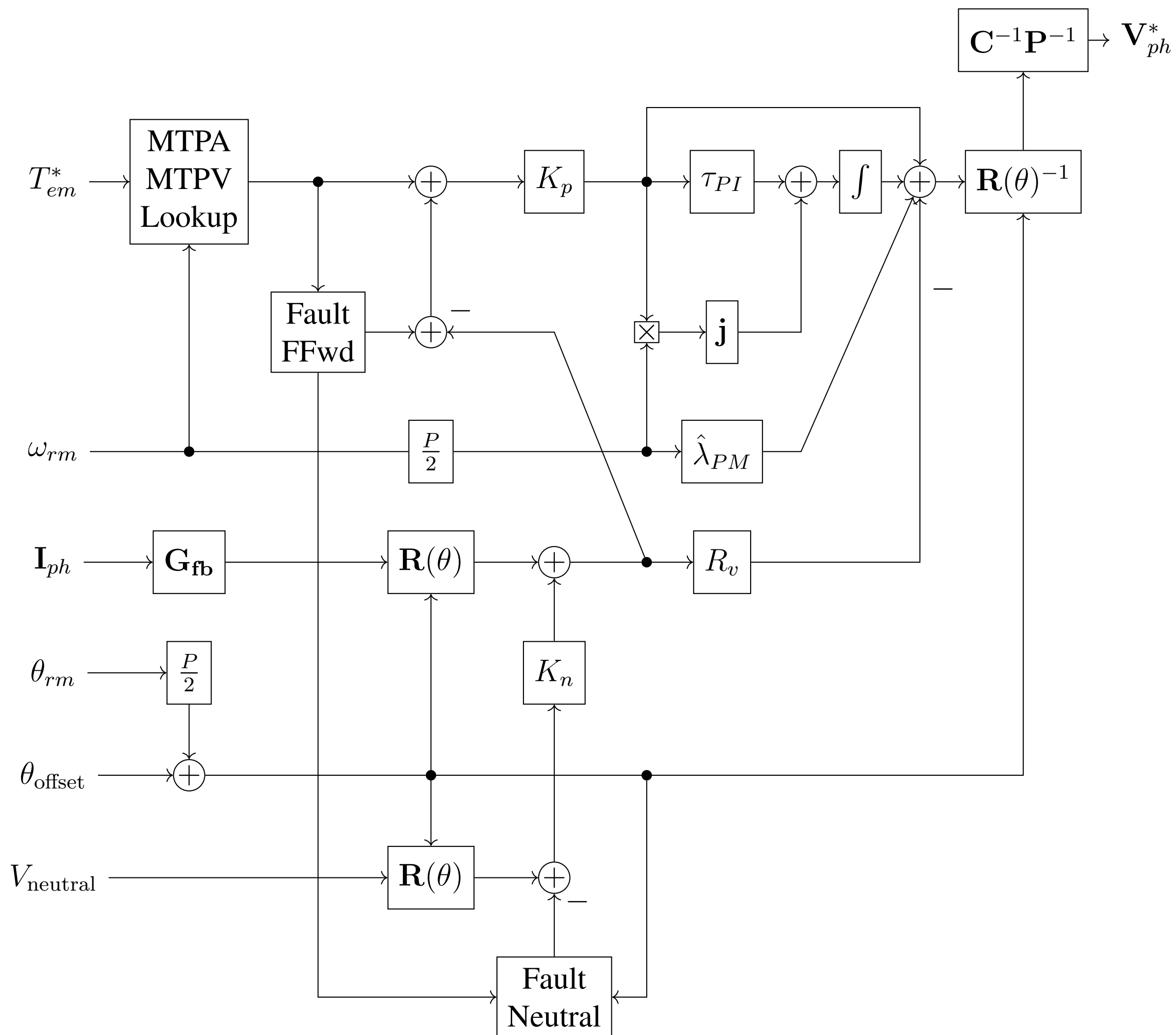


Figure 5.13 Block diagram implementing neutral-point stabilized vector control for arbitrary phase orders with a programmable number of input current sensing channels. This model includes both the post-fault current and voltage command injection to operate after a phase fault.

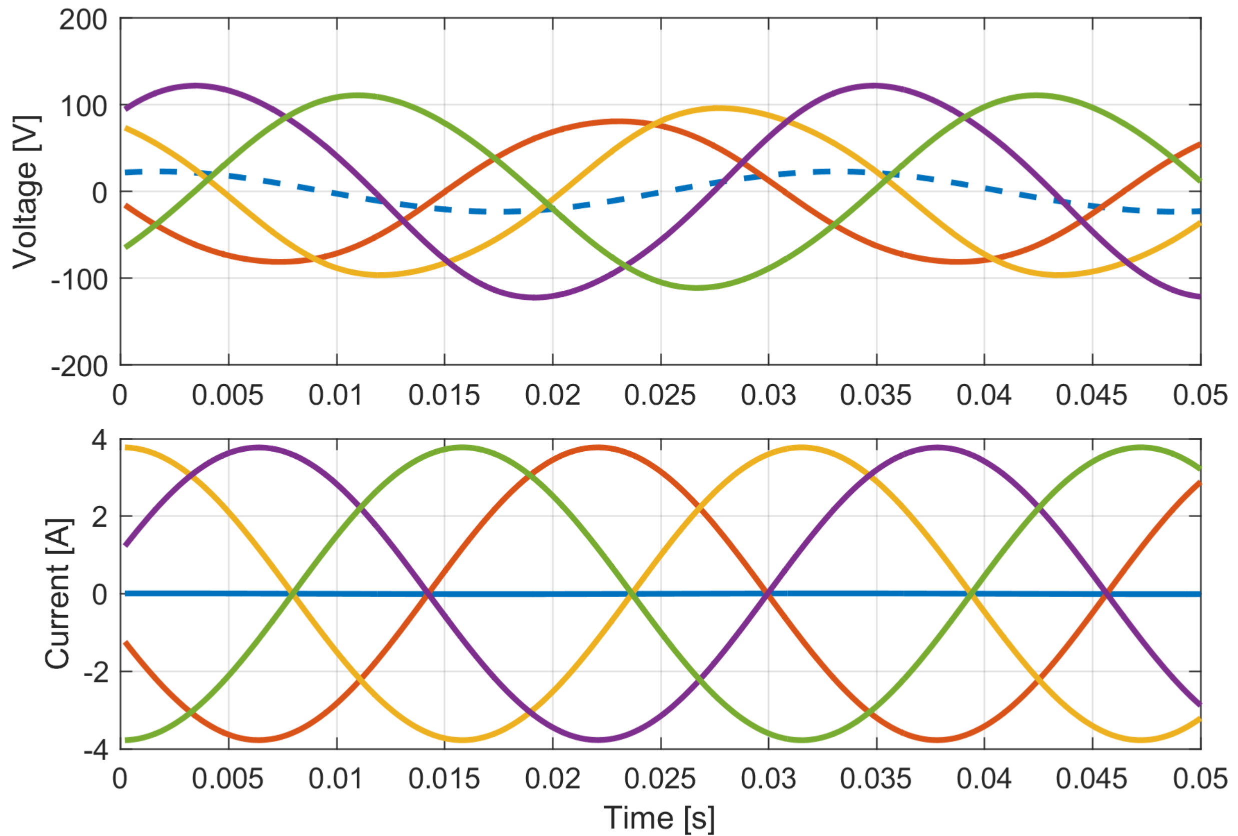


Figure 5.14 Voltage waveforms for the smoothed torque condition including the neutral point voltage offset for the remaining healthy phases shown as a dashed line in the upper figure.

and the faulted phase is a virtual fault where the phase is still healthy but is being controlled to zero current in order to allow modular drive hot-swapping.

For the analysis that follows and the case where the physical neutral voltage is measured, the back-EMF of the faulted phase is not included in the neutral voltage feedback signal. In this case, a non-zero neutral voltage must be calculated based on the excitation applied to the remaining healthy phases and the back-EMF of the machine. For the smoothed torque case where the phase currents are commanded to be phase-shifted sine waveforms at the fundamental frequency, this neutral voltage is a fundamental frequency sine waveform in phase with the missing phase voltage but only 25 percent of its amplitude as shown in figure 5.14. This voltage can either be calculated on a per-module basis on the fly based on the current command, fault ride-through feedforward

model, or a simplified model of the machine. This method requires additional computational time and control integration states so it may pose a limitation for a computationally-constrained control platform.

The other option is to create a table of the neutral voltage corrections based on the number of missing phases, the relative angle of the missing phase to a distributed module, and the operating condition. This table lookup requires minimal computational overhead and can be directly added to the rotated neutral voltage feedback path as shown in Figure 5.13.

The equation for the table lookup is a relatively straightforward application of equations 3.1 and 3.2. For an open circuit, there is no current in the faulted winding. Thus, assuming the controller is still able to regulate the requested current, the neutral voltage that needs to be applied is presented as follows:

$$V_d = \frac{1}{n-1}\omega_e L_q i_q \tag{5.17}$$

$$V_q = \frac{1}{n-1}\omega_e \left(L_d i_d + \lambda_{pm}\right) \tag{5.18}$$

where n is the number of phases in the system. This voltage must be rotated so that the q axis is aligned with the failed phase. This voltage command captures the low-speed dynamics of the stator flux which is all that is needed for the neutral point compensator since the higher-frequency dynamics will interfere with the current regulator.

5.5 Conclusions

This chapter has extended the concepts of a distributed current regulator introduced in chapter 4 to an arbitrary number of phases. First, a standard description and modeling approach for arbitrary phase-order machines was described including machines where more than one physical winding shares a magnetic axis. The vector transforms needed to generalize the electric machine models were

then described and the machine equations were restated in terms of vector quantities of arbitrary phase order.

Next, the distributed current controller was applied to the vectorized machine model with simulation of a five-phase example. With this example, the concept of multiple torque-producing harmonic reference frames was introduced. The limited sensing of the distributed current controller presented a limitation in that only a subset of harmonic reference frames can be determined by the distributed controllers, thus limiting the control structure compared to that of a monolithic drive. The neutral point feedback method was shown to still apply to distributed current regulators operating on machines of arbitrary phase order.

Finally, fault-mode operation of a high-phase-order machine drive with distributed controllers was discussed. The distributed current controller was then applied to control the current in a machine after an open-circuit fault occurs in one of the phases. Without changing the control structure beyond the addition of a feedforward term, the distributed controller was able to suppress the torque ripple arising from the lost phase by adjusting the phase angles of the remaining current commands. Equal post-fault torque generation capability was then shown to be possible provided that sufficient current headroom is available from both the drive and machine. The difficulties of neutral point voltage control in the presence of faulted phases was then discussed and overcome by introduction of non-zero neutral point voltage commands to correct for the back-EMF-induced common-mode voltage arising from the open-circuited phase winding.

Chapter 6

Experimental Verification

In order to verify the results of the simulations from the previous chapter, experimental verification is required. Due to the distributed nature of the controllers in this work, multiple controllers with independent power electronics are required. After requesting quotations from several vendors of off-the-shelf drives and hardware rapid prototyping systems, it was determined that the only feasible approach would be to build custom hardware for this research.

6.1 Compact Drive Module

The controls and power electronics for this research are presented as a compact integrated drive module. Each drive can operate as a standalone DC to AC converter with up to three phases of output.

- Support 400V DC-link.
- High-bandwidth communications.
- Self-contained current and voltage sensing.
- Reconfigurable current and voltage sensing.
- Self-powered off DC-link.
- Hardware input for position encoder.

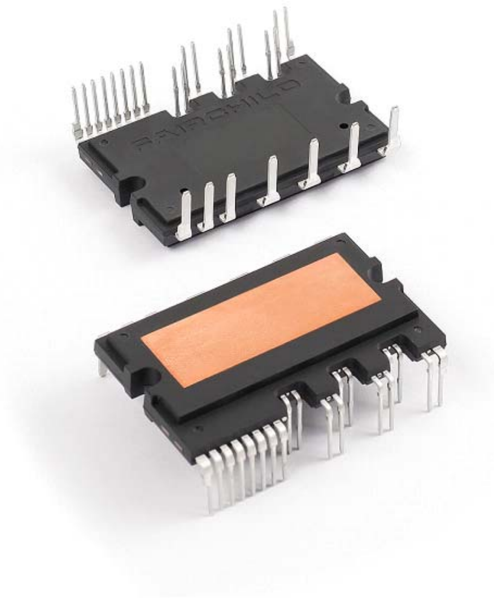


Figure 6.1 Fairchild FSBB30CH60CT integrated power module.

- Ground-referenced communications.

- Floating-point DSP.

Since the purpose of this research was primarily controls development rather than power electronics design or system integration a compact design using as much off-the-shelf integration was chosen. The power switching stage for this drive is a Fairchild Semiconductor integrated power module designed for white-goods and light industrial applications.

This module, shown in Figure 6.1 contains three 600V, 30A IGBT half bridges with independent lower-side emitter pins for low-side resistive current sensing. This module also includes a non-isolated bootstrap gate driver with shoot-through over current protection implemented in hardware. This module is easily cooled for this application by a standard Pentium-3 era CPU cooler. The drive module dissipates just over 120W at full power which is about three times the TDP of a late-model Pentium-3 Coppermine processor. This drive module however has a maximum junction temperature

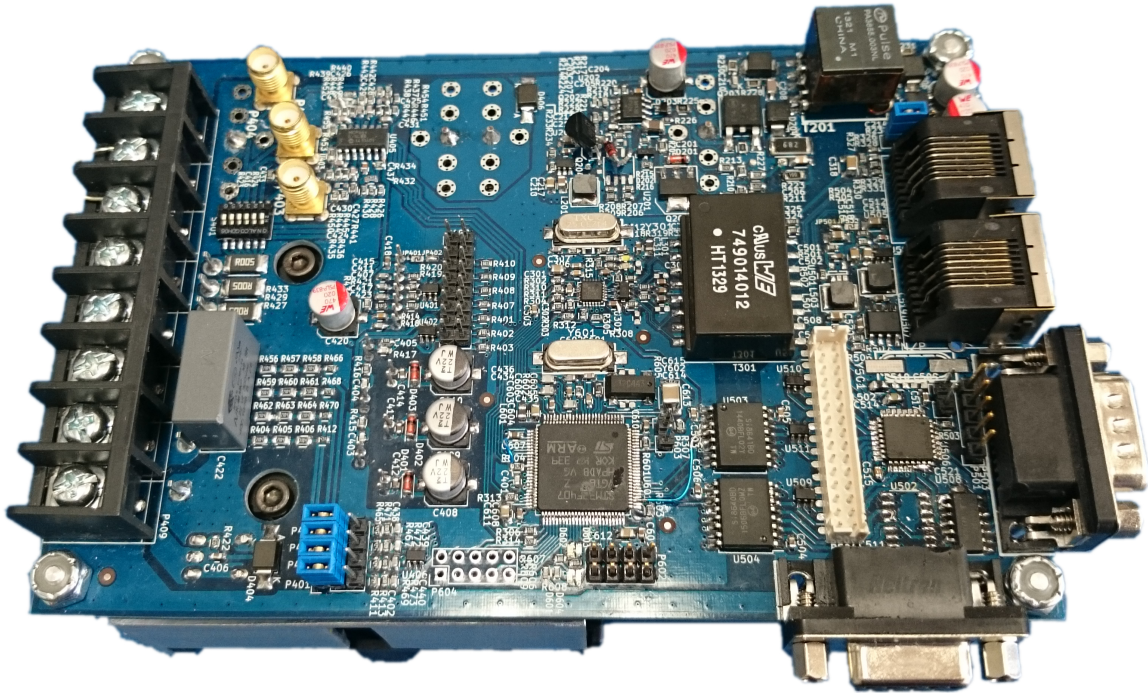


Figure 6.2 View of compact drive module.

30C higher than the processor and significantly lower junction to case thermal resistance. An overall view of the drive module board is shown in Figure 6.1. For more information about the circuitry described below, full schematics are provided in the appendix.

6.1.1 Power Supply

The power supply design for the control and logic power on the drive modules was primarily driven by the difficulties in providing control power from a central off-board power supply in previous work [11]. To remedy this problem, a small flyback supply was designed to power each drive module directly from it's own DC-link. Each module required several power supply rails. The power stage and main DSP required 15V and 3.3V referenced to the negative DC-link rail. The CAN transceiver, isolators, and encoder signal conditions required 5V and 3.3V referenced to case ground. A further complication is that low DC-link testing capability is crucial for a controls

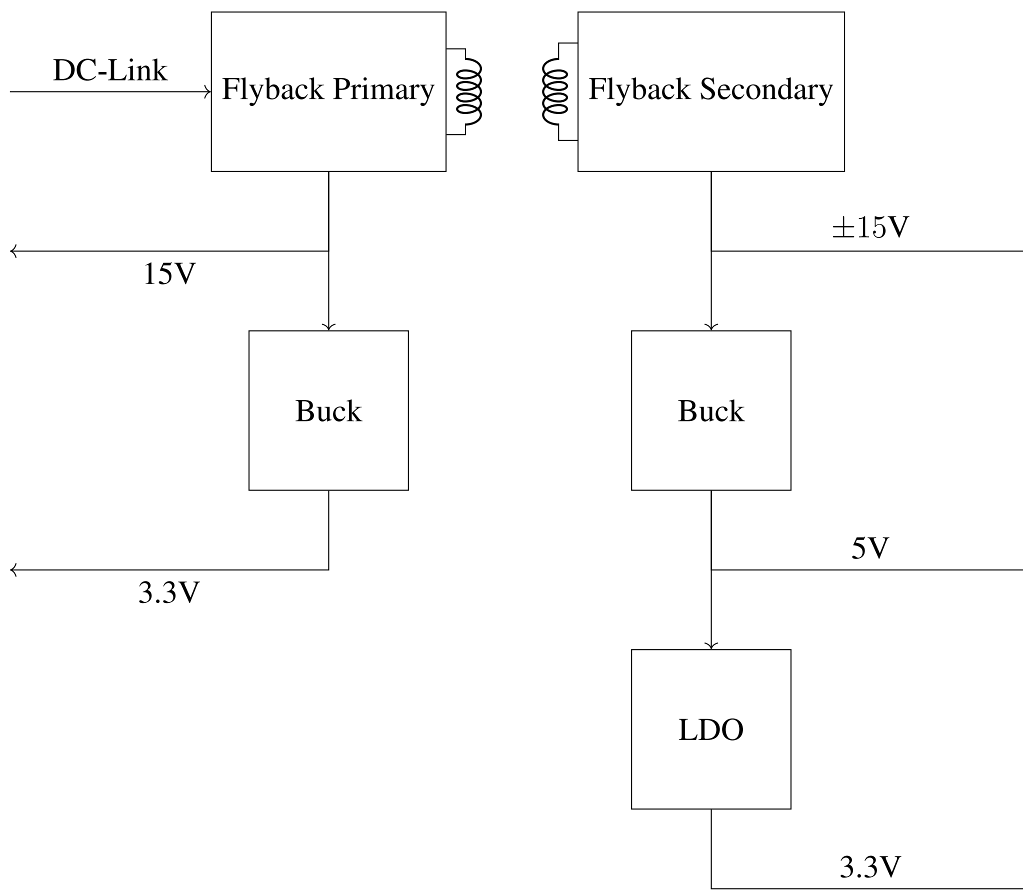


Figure 6.3 Power supply arrangement for the converter.

development platform in order to allow for testing of safety functions and fault handling without risking hardware damage.

To meet these requirements, a multi-output flyback converter was designed based on a UC3842 current-mode switching controller. This converter controller is extremely simple, consisting of a ramp generator, comparator, gate driver, and error amplifier. The requirement for low voltage testing means that the converter must be designed to handle an input voltage range of 40V — 420V (more than 10:1). In order to avoid the need for an extremely oversize flyback transformer, an input voltage feed-forward was implemented to compensate for the turn-off delay of the converter. This method is applicable to any current-mode boost-derived converter. It is implemented by placing a

resistance between the input voltage and the current feedback summing node. This feedforward resistance is chosen such that the current offset injected into this summing node is equivalent to the primary inductor current rise during the turn-off delay of the power switch.

The flyback converter outputs 15V on the DC-link referenced side. This voltage is regulated by the switching controller. The converter also outputs 15V and a user selectable ±15V on the ground-referenced side. This selection allows for more positive rail current to be supplied if the negative rail is unneeded. These isolated rails are not directly regulated, but are dependent on the regulation of the 15V rail on the primary side.

The further rails are derived from these two 15V rails. On the primary side, the 3.3V for the DSP, signal conditioning, isolators, and Ethernet PHY is provided by a TPS560200 synchronous buck converter operating from the 15V rail. The output from this converter is filtered through a 2nd order ferrite-bead and capacitor filter to help reduce switching noise. Power for the analog circuitry has another similar filter. On the secondary side, the same converter is used to derive a 5V rail which powers the CAN transceiver, RS-485 transceiver and the RS-422 receivers used for both communication and encoder conditioning. The secondary side I/O controller is powered by 3.3V derived from the 5V rail by an LDO linear regulator.

6.1.2 Sensors

The sensing requirements for the converter modules are driven by the control requirements determined by theory and simulation. Each module integrates it's own current sensing and voltage sensing such that the module can operate as a standalone drive if needed. On board current sensing consists of three low-side shunt resistors appropriately amplified and fed to the DSP. These shunts are also used for over-current protection even if they are not used for control. As these resistors are only on the lower switch of each phase leg PWM switching must be synchronized between modules if these resistive current sensors are to be shared between controllers.

Table 6.1 Encoder pinout. Numbering is for a standard DE-9. All signals are referenced to earth ground. Differential inputs are RS-422 compatible.

1	2	3	4	5	6	7	8	9
A+	B+	Z+	GND	5V	A-	B-	Z-	GND

The current sense amplifier input channels are wired to SMA connectors with switches to allow resistor voltages to be exported from a board or for another analog current sense signal to be input in place of the onboard sense resistor. For ease of development, this hardware is begin wired with LEM current sensors on the machine phases which will be distributed to the required input channels on each modular controller.

Voltage sensing is integrated directly with a high-value resistor divider and low pass filter at approximately 3kHz. This low pass filter value was determined to allow for accurate reconstruction of the actual output waveform from the PWM voltages [74]. If external voltage sensing is required, this input can be overridden directly by removing a jumper and providing an external voltage signal in the range of 0V to 3.3V relative to the negative DC-Link.

Position feedback is provided by a CUI capacitive encoder. This encoder is a small shaft-mounted encoder which provides quadrature digital outputs and an index pulse with differential signal conditioning. The encoder resolution can be programmed through PC software and is currently set at it's maximum of 4096 counts per revolution. Each modular drive has a ground referenced input for three differential channels which is fed to the main DSP through a capacitive isolator. The connector pinout on the control board is listed in table 6.1. The differential inputs are RS-422 compatible. If a single-ended encoder is being used, the negative differential inputs should be terminated to an appropriate voltage in the connector using a resistive divider from the 5V to ground. Note that 5V power is supplied by each board. If a single encoder is to drive several boards in parallel, this pin should not be connected in order to avoid contention between the buck

converters on each controller. On the DSP the encoder is connected to a sixteen-bit timer peripheral which provides majority-function FIR edge glitch filtering while counting each edge on both A and B phases as well as capturing the count at the index pulse. By allowing the counter to use the full sixteen-bit range and capturing index pulses, the timer/counter hardware can maintain position up eight full revolutions in either direction with no ambiguity without software interaction. The software can then increment or decrement this counter atomically by a multiple of the edge count in order to maintain location synchronization.

If further sensors required, a Cortex-M0 processor is provided on the ground-referenced side with 8 pins broken out which can be used for either analog or digital functions. This processor is linked to the main DSP through a high speed SPI link and if installed is responsible for handling the RS-485 communications interface. In other applications this processor can handle resolver excitation and sensing or other ground-referenced sensors.

6.1.3 Communications

While the primary target of this work is to allow distributed motor control without the need for significant communications, communication interfaces are still required for command input, data acquisition, module synchronization, debugging, fault tolerance.

For this purpose, the motor drive has three digital communications interfaces. The first, and most flexible is a 100Base-TX Ethernet interface. This is wired to the built-in MAC on the main DSP through a LAN8720 Ethernet PHY. As this DSP is referenced to the lower DC-Link, galvanic isolation is provided through means of an industrial 4000V rated Ethernet transformer. For synchronization purposes, this interface is capable of IEEE 1588 PTP time-stamping in hardware.

A CAN interface is provided, also connected to the CAN peripheral on the main DSP. This was chosen since it is a common industrial field bus and was also used on an earlier revision of modular motor drive. CAN can be used for synchronization since it provides a deterministic timing for

receive interrupts. Galvanic isolation for this interface is provided through a Silicon Labs capacitive digital isolator.

Finally, a high-speed RS-485 / RS-422 interface is provided which can either be connected to the ground-referenced Cortex-M0 microcontroller or, if that microcontroller is unpopulated, jumpered to one of the UARTS on the main DSP. This jumpering uses the pins which would have formed the SPI interface to the ground-referenced MCU to directly connect the RS-485 transceiver to the digital isolator. This serial link can run at up to 10Mbps full duplex. Isolation is again provided by a capacitive digital isolator.

At the present, all communication is done through the Ethernet interface due to it's flexibility. Each motor control module is connected to a gigabit capable Ethernet switch which is connected to the test computer through a fiber media converter in order to ensure that the computer won't be damaged in case of a short between power and control circuitry. This connection also allows all modules to be interacted with through one connection to the host computer rather than having to wrangle several serial or USB connections. A tftp-based bootloader is used for code upgrades. A telnet-like terminal is available for debugging and control of data gathering. All control states can be dumped to a raw TCP or UDP connection at full (up to 20kHz) sample rate over this interface.

6.2 Reconfigurable Polyphase Motor

The machine used for hardware experiments in this work was originally built by Baldor as a 5hp interior permanent magnet (IPM) 4-pole industrial motor. The original nameplate ratings are 3-phases, 1800rpm at 60Hz, 230V, and 21.2 Amps. It was rewound by Dr. Andy Rockhill during his PhD research program [67]. The new winding consists of 36 full-pitch coils with all 72 coil ends broken out to a terminal box mounted on top of the machine as shown in Figure 6.2. This allows the machine to be reconfigured to present several different phase arrangements. Figure 6.2 shows the winding diagram with all 72 numbered terminals.

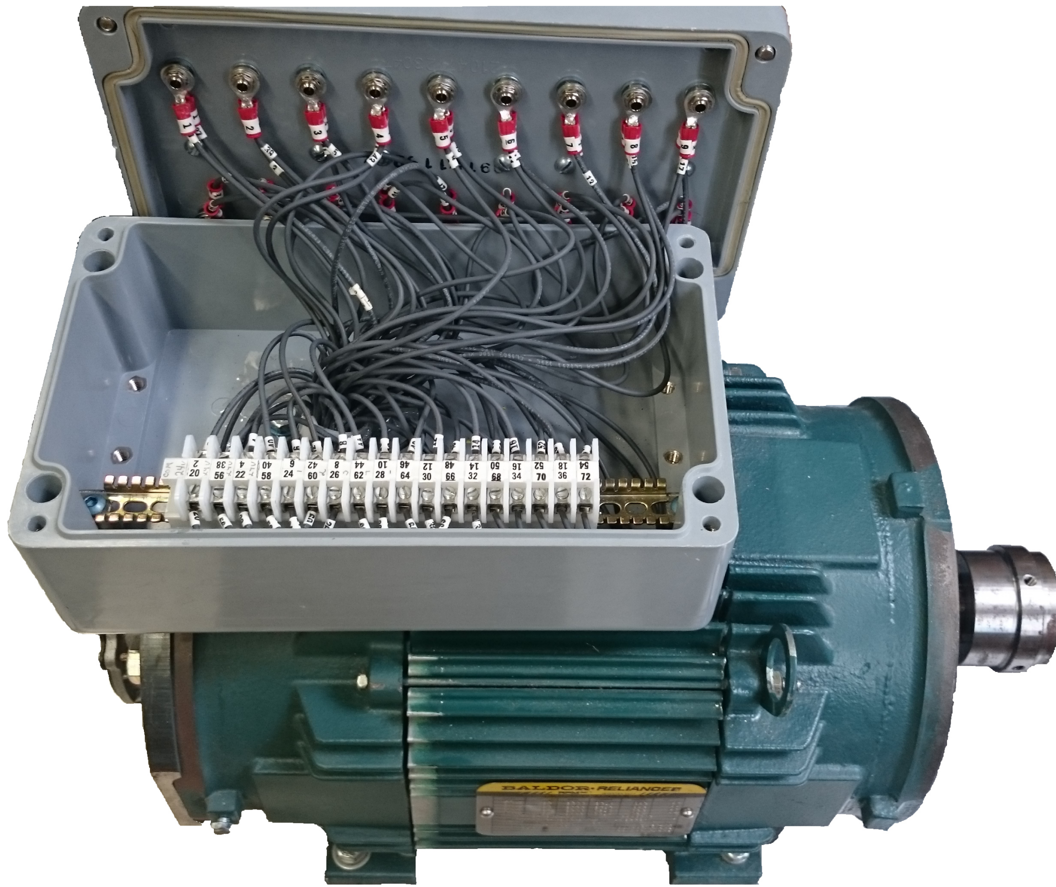


Figure 6.4 Experimental reconfigurable machine with open terminal box.

In the nine-phase configuration with two poles in series, two poles in parallel, the phase inductance matrix with the rotor aligned with phase $1-1'$ is shown in equation 6.1. This inductance matrix is the same as was reported in [67] and has been verified by measurement.

$$[\mathbf{L_{s9ph}}(\mathbf{0}^{\circ})] = \begin{bmatrix} 49.9 & 38.2 & 27.0 & 16.4 & 5.59 & -5.43 & -16.39 & -27.3 & -37.83 \\ 38.2 & 40.7 & 31.93 & 18.22 & 6.14 & -4.87 & -14.59 & -23.18 & -30.83 \\ 27.0 & 31.93 & 32.61 & 22.33 & 8.23 & -2.88 & -11.7 & -18.21 & -23.18 \\ 16.4 & 18.22 & 22.33 & 23.4 & 13.88 & 1.71 & -6.45 & -11.7 & -14.59 \\ 5.59 & 6.14 & 8.23 & 13.88 & 17.39 & 10.55 & 1.71 & -2.88 & -4.87 \\ -5.43 & -4.87 & -2.88 & 1.71 & 10.55 & 17.39 & 13.88 & 8.23 & 6.14 \\ -16.39 & -14.59 & -11.7 & -6.45 & 1.71 & 13.88 & 23.4 & 22.33 & 18.22 \\ -27.3 & -23.18 & -18.21 & -11.7 & -2.88 & 8.23 & 22.33 & 32.61 & 31.93 \\ -37.83 & -30.83 & -23.18 & -14.59 & -4.87 & 6.14 & 18.22 & 31.93 & 40.7 \end{bmatrix} \text{mH} \qquad (6.1)$$

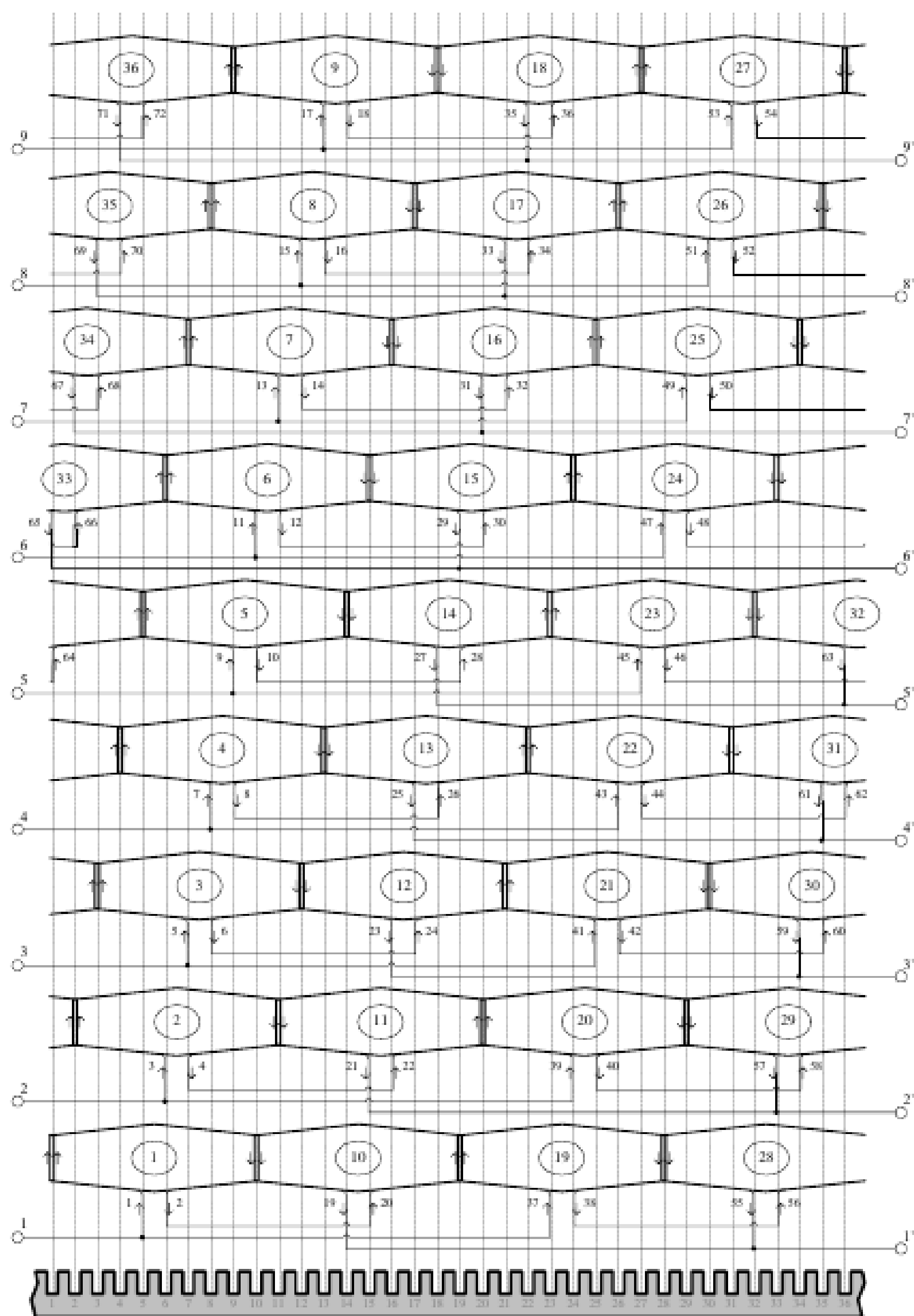


Figure 6.5 Experimental reconfigurable machine winding layout with 9-phase connection shown [67].

Table 6.2 Table showing machine terminal parameters for experimental machine wired for nine-phase operation with two poles series, two parallel.

r_s	L_{0s_1}	L_{2s_1}	L_{0s_3}	L_{2s_3}	L_{0s_5}
989mΩ	110.65mH	-66.65mH	18.85mH	1.80mH	5.20mH
L_{2s_5}	L_{0s_7}	L_{2s_7}	L_{0s_9}	L_{2s_9}	
0.60mH	1.30mH	0.15mH	0.05mH	0	

When rotated into the synchronous reference frame, this inductance matrix becomes what is shown in equation 6.2 after all near-zero elements are clamped to zero.

$$[\mathbf{L_{s9dq}}] = [\mathbf{C}]^{-1}[\mathbf{R}(\mathbf{0}^\circ)]^{-1}[\mathbf{L_{s9ph}}(\mathbf{0}^\circ)] = \begin{bmatrix} 44.00 & 0 & -1.80 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 177.30 & 0 & -1.80 & 0 & 0 & 0 & 0 & 0 \\ -1.80 & 0 & 18.85 & 0 & -0.60 & 0 & 0 & 0 & 0 \\ 0 & -1.80 & 0 & 18.85 & 0 & -0.60 & 0 & 0 & 0 \\ 0 & 0 & -0.60 & 0 & 5.20 & 0 & -0.15 & 0 & 0 \\ 0 & 0 & 0 & -0.60 & 0 & 5.20 & 0 & -0.15 & 0 \\ 0 & 0 & 0 & 0 & -0.15 & 0 & 1.30 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -0.15 & 0 & 1.30 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.10 \end{bmatrix} \text{mH} \tag{6.2}$$

This synchronous frame inductance matrix can be summarized in terms of fundamental and second-harmonic components as listed in Table 6.2.

A three-phase machine is needed to verify the simulation results. This machine can be rewired to form a three-phase machine by taking every three adjacent slot windings and connecting them in series. This results in 12 pole-phase sets. These sets have phase angles and connection lists shown in Table 6.3.

If these sets are wired in the two-series, two-parallel configuration used above, the phase inductance matrix is as shown in equation 6.3.

$$[\mathbf{L_{s3ph}}(\mathbf{0}^\circ)] = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} [\mathbf{L_{s9ph}}(\mathbf{0}^\circ)] \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 345.02 & 147.92 & -148.05 \\ 147.92 & 162.28 & -39.85 \\ -148.05 & -39.85 & 162.28 \end{bmatrix} \text{mH} \tag{6.3}$$

Table 6.3 Table showing wiring sets for three phase operation with three windings in series and all poles in parallel. Each list of numbers in this table should be shorted together.

A - 1 - 24 - 37 - 60	2 - 3	4 - 5	20 - 21	22 - 23
A′ - 6 - 19 - 42 - 55	38 - 39	40 - 41	56 - 57	58 - 59
B - 7 - 30 - 43 - 66	8 - 9	10 - 11	26 - 27	28 - 29
B′ - 12 - 25 - 48 - 61	44 - 45	46 - 47	62 - 63	64 - 65
C - 13 - 36 - 49 - 72	14 - 15	16 - 17	32 - 33	34 - 35
C′ - 18 - 31 - 54 - 67	50 - 51	52 - 53	68 - 69	70 - 71

Again, when this is translated into the synchronous reference frame, the inductances become the matrix in equation 6.4.

$$[\mathbf{L_{s3dq}}] = [\mathbf{C}]^{-1}[\mathbf{R}(\mathbf{0}^\circ)]^{-1}[\mathbf{L_{s3ph}}(\mathbf{0}^\circ)][\mathbf{R}(\mathbf{0}^\circ)][\mathbf{C}] = \begin{bmatrix} 122.43 & 0 & 0 \\ 0 & 494.70 & 0 \\ 0 & 0 & 52.447 \end{bmatrix} \mathrm{mH} \tag{6.4}$$

These inductances and especially the back-EMF are slightly high to be effectively driven by a 380V DC-link. In order to bring the voltage requirements down to a more reasonable value, the machine can be rewired to connect all poles in parallel. This divides the inductance matrix by 4 since the terminals now have half the voltage at twice the current [75]. In the synchronous reference frame, the final inductance matrix in shown equation 6.5. This inductance set is summarized in Table 6.4.

$$[\mathbf{L_{s3dq||}}] = \begin{bmatrix} 30.608 & 0 & 0 \\ 0 & 123.68 & 0 \\ 0 & 0 & 13.112 \end{bmatrix} \mathrm{mH} \tag{6.5}$$

Each coil has a back-EMF waveform with approximately 1.6V/Hz peak voltage as shown in Figure 6.2.

Table 6.4 Table showing machine terminal parameters for experimental machine wired for three phase operation with all poles in parallel.

r_s	L_d	L_q	L_0
741.8mΩ	30.608mH	123.68mH	13.112mH

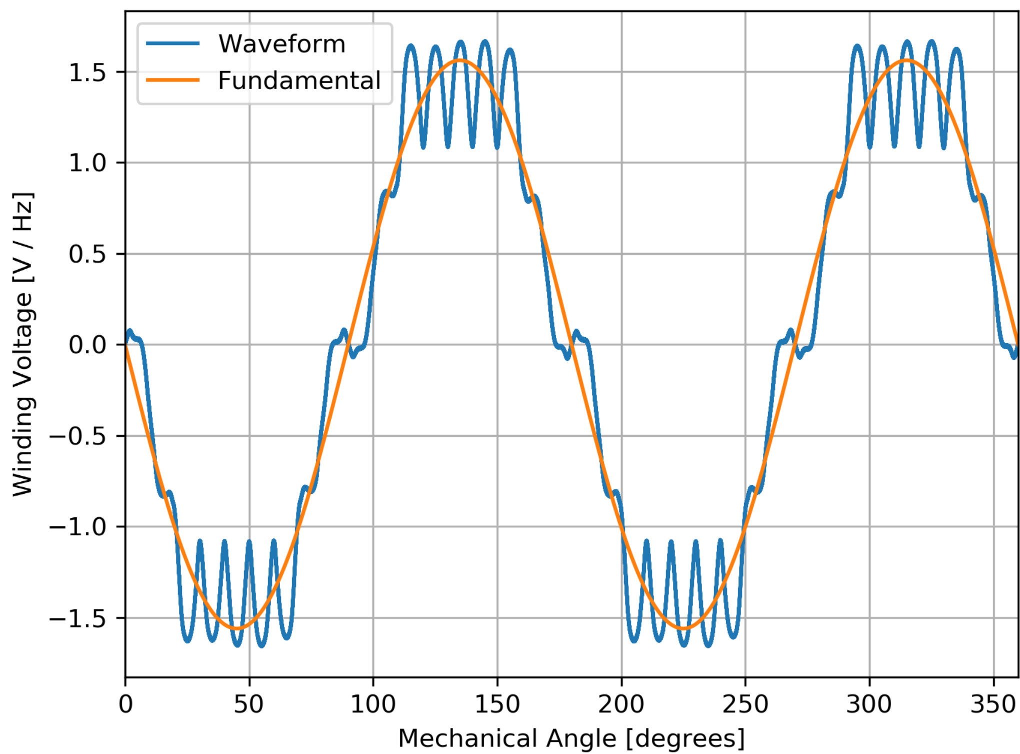


Figure 6.6 Back-EMF waveform (open-circuit voltage) for a single coil in the experimental waveform showing fundamental component.

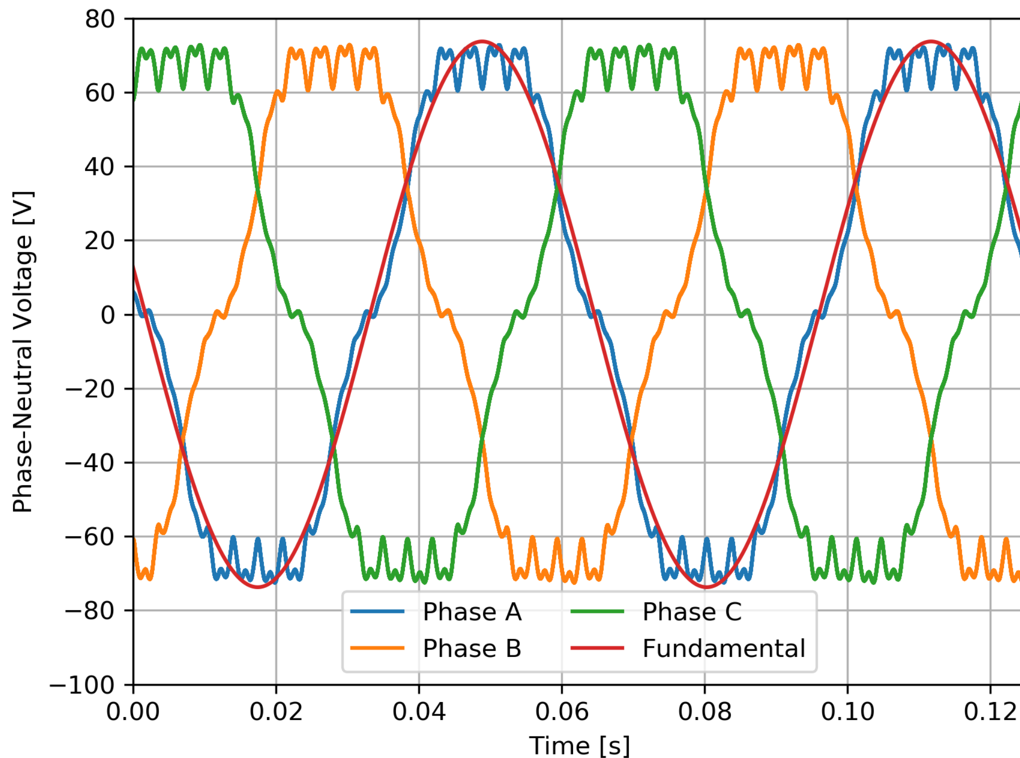


Figure 6.7 Back-EMF waveform (open-circuit voltage) for three adjacent coils in series as used in the tests described in the next chapter.

6.3 Firmware Description

While hardware design is the core of any experimental apparatus, in the case of distributed controls with significant communications, careful firmware design is essential. The firmware design for the flexible modular drives described above consists of four major parts.

6.3.1 RTOS

While many research power electronic control projects can operate effectively with only a main-loop with a few timer-triggered interrupts, the third party software requirements posed by using Ethernet caused an RTOS to be the path of least resistance in implementing a functional drive

platform. As this was a research project with limited budget and even more limited engineering staff, only open-source RTOS options were considered.

At the time firmware development started, this limited the choices to FreeRTOS and ChibiOS/RT. Both supported the STM32F407 that was chosen for this drive controller and had LwIP integration for relatively seamless Ethernet support. At the time, FreeRTOS was in the process of being bought out by ARM to be the official RTOS of their mBed platform. In the end ChibiOS/RT was chosen due to better HAL support, better licensing stability (GPL), and published performance results indicating it was capable of saturating the 100Mbit/s Ethernet connection on the STM32F407 while still having left over CPU time. FreeRTOS has a licence term that prohibits performance comparisons.

6.3.2 Bootloader

A bootloader is essential to ensuring multiple drives are all running the same firmware. In this design, the bootloader is a 64kB image residing at the start of the DSP flash which places the controller I/O in a safe mode before loading firmware over the Ethernet interface. This avoids the issue of having the power electronics go into an unsafe state when the firmware is changing and avoids having to change physical wiring in order to update firmware.

Once the bootloader loads, it first sets up the I/O, then copies itself into RAM. This allows the bootloader code to continue running while the flash is locked due to erase or programming cycles.

The bootloader firmware then starts a set of threads implementing a tftp server, a shell server, and an LED blinker. The tftp server allows firmware push uploads, nvram settings downloads and uploads, and bootloader uploads using the local filenames "core", "nvram", and "bloader".

The led blinker thread flashes the green LED while the bootloader is running. The blue led will be toggled every time the main firmware is checked.

The shell thread provides a command shell over TCP on port 3333. This shell implements peek and poke commands, a crc check, RTC setup commands, and nvram editing.

Once the server threads are started, the firmware attempts to fetch a firmware image from its gateway IP over tftp named drive.bin. It then downloads an nvram image named after its IP address "IPADDR.nvram". If the firmware image exists and the first 512 bytes match the firmware programmed in flash, the drive will make a reboot attempt quickly.

If the firmware image doesn't match, the bootloader will erase the main drive firmware flash and fetch the full firmware image again, writing it to flash. It then will wait until the 30s timer expires to check the magic number, size, and CRC32 checksum before booting the firmware. These numbers are written into the unused reserved interrupt vectors in the Cortex-M4 image. The CRC is the Ethernet polynomial and is calculated over the whole image.

If the main firmware fails these checksums the bootloader will loop once every 30 seconds until a valid firmware appears.

This automatic bootloader is essential to ensuring that all drive modules in the system are running the same version of firmware and that their settings can be easily centrally managed. Previous work on distributed controls at WEMPEC faced significant difficulty as any change to software had to be manually distributed to each modular controller which often led to one controller running outdated firmware causing hard to track down errors and in some cases damaged hardware.

6.3.3 Control loop structure

The control loop is tightly integrated with the telemetry communications interface. In the primary firmware, the control loop consists of three threads and two interrupts. Static constants are stored in the STM32's battery backed SRAM. This 4kB storage space includes the board's MAC and IP addresses, sensor scaling gains and offsets, basis vectors for translating the sensor inputs and voltage outputs into the proper reference frames, the encoder line count, dead-time compensation parameters, controller gains, magnet flux levels, machine electrical parameters, pole count, and

controller angle offset. The bootloader's automatic update action helps ensure that this information is up to date at power on.

On firmware initialization, the control vector is zeroed out and the initial control state is set to PASSIVE with no switches active. The control loop, PWM, and ADC peripherals are then set up and the command input thread is launched.

A control cycle begins when the PWM interrupt fires. It checks a completion semaphore to verify that the previous control calculation has finished. If the calculation is not done, it immediately sets the control state to passive and shuts off all the switches. If the control calculation had finished in time, the control state pointer is advanced along the buffer and the semaphore for completion is reset. The encoder timer count is also sampled in this interrupt. Finally, if enough control steps have passed to fill a packet, it signals a read-ready semaphore. If this code is used directly in the future it should be reworked to take a different action if the drive is operating in a flux weakening region as suddenly going passive will cause uncontrolled generation and may damage the drive.

The next step in the control cycle occurs when the ADC finishes its conversion of the current and voltage measurements. The conversions are started automatically through a hardware trigger slaved to the PWM carrier reset. The sample is taken at precisely the minimum of the PWM carrier though the SAR conversion process takes several microseconds. When the conversion is complete, the ADC completion interrupt resets the ADC DMA pointers to cause the next ADC conversion to fill the next control step's ADC array and then signals a semaphore that input is complete.

While these interrupts are firing, the controller thread is waiting on the input complete semaphore. When this is signaled, the control thread locks a mutex on the current control state vector, scales the ADC inputs according to the stored constants in nvram, computes a control step matching the controller simulaiton above, and then depending on the present control state either sets the voltage output to the PWM or sets a different fixed PWM ouput, signalling the completion semaphore once it's set a PWM output. This thread only interacts with the control state vector circular buffer which

both makes automatic code generation for the control law possible, it allows the actual control code to be used as a controller in the loop for verification. At present this has only been done using the peek and poke interfaces available through the shell but with some additional code could be easily integrated into simulink.

The control states include the passive state already discussed, a set of precharge states which turn on the lower-side switches in order to charge up the bootstrap gate drive power supplies, a constant V/angle state to operate the drive in open loop, and the current/torque controlled state. At the present time the torque table is computed on the host computer though that is not a limitation of the hardware or control laws.

6.3.4 Communications

A major driver of both the RTOS and control structure. The control state circular buffer is structured in order to facilitate fast, low-overhead telemetry at full control rate of all measurements and internal control states.

The main drive firmware has several communications interfaces, presently all are presented over Ethernet. The first, on TCP port 3333 is a shell interface providing peek and poke functionality along with command line getters and setters for nvram constants.

The second is a simple web server which primarily serves two JSON files which describe the datatypes, array sizes, and offsets for both the nvram structure and the control state vector. This is essential as the high speed communications send only raw binary for speed and this format is determined at compile time and may change with firmware updates.

Finally, the main control communications consist of two threads, one for input and one for output. The output thread starts life as the thread setting up PWM and ADC peripherals. Once the controller is running, it waits for a read-ready semaphore to be signaled. When this is signaled, it prepares a UDP packet containing a number of control state vectors in order to nearly fill a standard

Ethernet MTU of 1500 bytes. At the present this is four control steps resulting in a 1328 byte packet. This packet is then sent to the address and port stored in nvram which is usually the gateway address and port 8192. This packet is sent with very minimal overhead as it is presented to the Ethernet hardware as a chained DMA request with a header and a segment of the circular control buffer. At a 12kHz PWM rate, this results in a packet being sent every $333\mu s$ for a total transfer rate of about 40Mbps from a single drive module. While this is well within the capabilities of the 100Base-TX transceiver on the microcontroller, it does require a 1000Base-TX capable switch and telemetry sink if any more than two drive modules are used at once.

The input thread operates in parallel to the control and output threads. It listens on IP multicast address 232.10.11.86 on port 8192 for UDP packets. It then acquires the mutex locking the control state vector and decodes these packets as offset and float pairs followed by newlines in order to update parts of the present control state. This allows controller state changes as well as operating point commands to be input to all drive modules in the system at the same time while ensuring that the input packet will not end up overwriting any part of the control state while the control law calculation thread is active. This locking was not in the initial firmware and was a major issue causing the distributed motor control to become unstable as one drive module would miss a command and begin fighting the others.

6.3.5 PC Support Software

Presently, the PC support software is very limited consisting of a logging interface and a set of state transition buttons. Command input was accomplished by running a Wireshark replay of a hand-crafted set of command packets, where Wireshark is a set of popular network protocol software tools . Future development should introduce a live graphical view of controller parameters and a command generation system that can provide torque commands directly.

6.4 Experimental Results

The experimental verification for this work is focused on verifying the stability of distributed vector control with the neutral-voltage sensing compensation controller.

While the hardware was designed with higher-phase order experimentation in mind with fully self-contained sensing and logging, the firmware challenges in realizing these features were significant. The control techniques developed in the preceding work depends on synchronized command input to all distributed phase modules as well as each module having the same sensing and controller gains.

As networking software development is somewhat out of scope for this dissertation, a choice was made to attempt confirmation of the proposed distributed controller through pre-calculated, time-triggered tables rather than using a simultaneous command input.

To this end, the telemetry output feed was disabled in order to reduce the incidence of race-condition based overflows crashing the drive code. A pre-calculated table sequence was loaded consisting of a one second stabilization phase in V/Hz mode in order to ensure that the encoder was properly aligned and any startup-transients had settled, then a series of timed commands stepped by a single bit digital input. Each distributed controller had its rotations and phase selections hard-coded in order to reduce power-on commissioning. In order to reduce the number of places problems could arise, and in order to match the simulations from chapter 4, a three phase configuration was used.

6.4.1 3-Phase Distributed Drive with Neutral Voltage Sensing

In the absence of telemetry code, data was collected using a LeCroy oscilloscope by measuring the three phase currents and neutral voltage during test runs. This experiment was repeated many time in hope of capturing a full 5 second sequence of torque commands operating without

error. In every case, the test was cut short with one phase missing a command transition and the regulators tripping on over-current when the other phases tried to correct. The best run of the set is presented here. This run shows the one second settling time followed by a 1.6 second duration of distributed vector control operation including one change of operating point with a 100ms transition. Approximately one sixth cycle after the end of this waveform, the drive faulted passive again as the over-current limit tripped.

These tests were run at a rotor speed of approx. 500rpm in order to limit the machine back-EMF to less than 200V. The back-emf voltage was purposely kept low so that the phase drives would not be damaged in the event of a drive fault while operating under load since the control software was faulting during every test run. Since these tests were carried out using a 4-pole machine, the 500rpm operating speed corresponds to an excitation frequency of approx. 16Hz. This excitation frequency is sufficiently high to avoid any interactions with the neutral point controller which has a break frequency below 1Hz.

Currents were measured with clamp-on current probes on the three phases. As the current measurements have the machine inductance filtering them, these measurements are very legible. The neutral point voltage is much more challenging to measure because of the large harmonic noise components introduced by the inverter switching . The neutral voltage is conditioned for the controller by scaling it down using standard voltage dividers applied to the drive's phase and DC-link voltages . A low-pass filter is also introduced to suppress the high-frequency harmonic noise components that are present in the raw neutral point voltage.

In contrast, the oscilloscope directly measures the neutral point voltage using a high-bandwidth differential voltage probe. Since the expected neutral point voltage is on the order of a few volts, there is a significant instrumentation challenge associated with extracting this voltage waveform from the raw 200V PWM signal that is measured. The raw measurement is shown in Figure 6.4.1. In order to extract this tiny measurement, a windowed FFT was performed and segmented in order

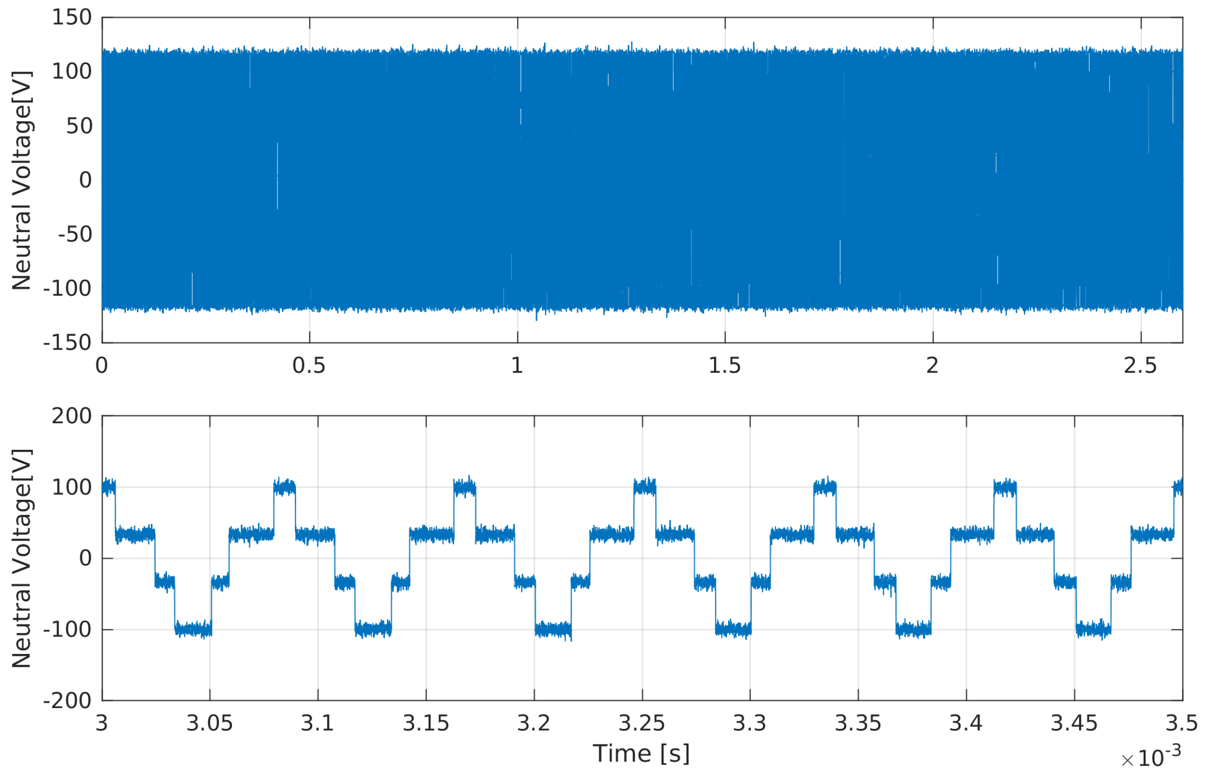


Figure 6.8 Unfiltered neutral point voltage waveform showing distributed controller operation with PWM for the full duration of the test (top). The lower waveform is a time zoomed axis showing the PWM steps. Both axes are enumerated in seconds with the upper being 2.6 seconds long and the lower being 500 microseconds long.

to exclude all higher frequency and DC components. The FFT was then inverted to reconstruct the neutral point voltage waveform in the lower part of figure 6.4.1. This process is essentially a non-causal filter approximately 200ms long with a cut-off frequency of just under 100Hz. Of interest, the PWM waveforms are nearly center-aligned during this test as shown in the lower portion of Figure 6.4.1 where the four levels of center-aligned PWM common mode are visible. This can be attributed more to luck in this case rather than due to design since the drive clocks are not synchronized but are driven by crystal oscillators that are a few ppm apart. This results in PWM synchronization that drifts in and out of alignment about once a minute.

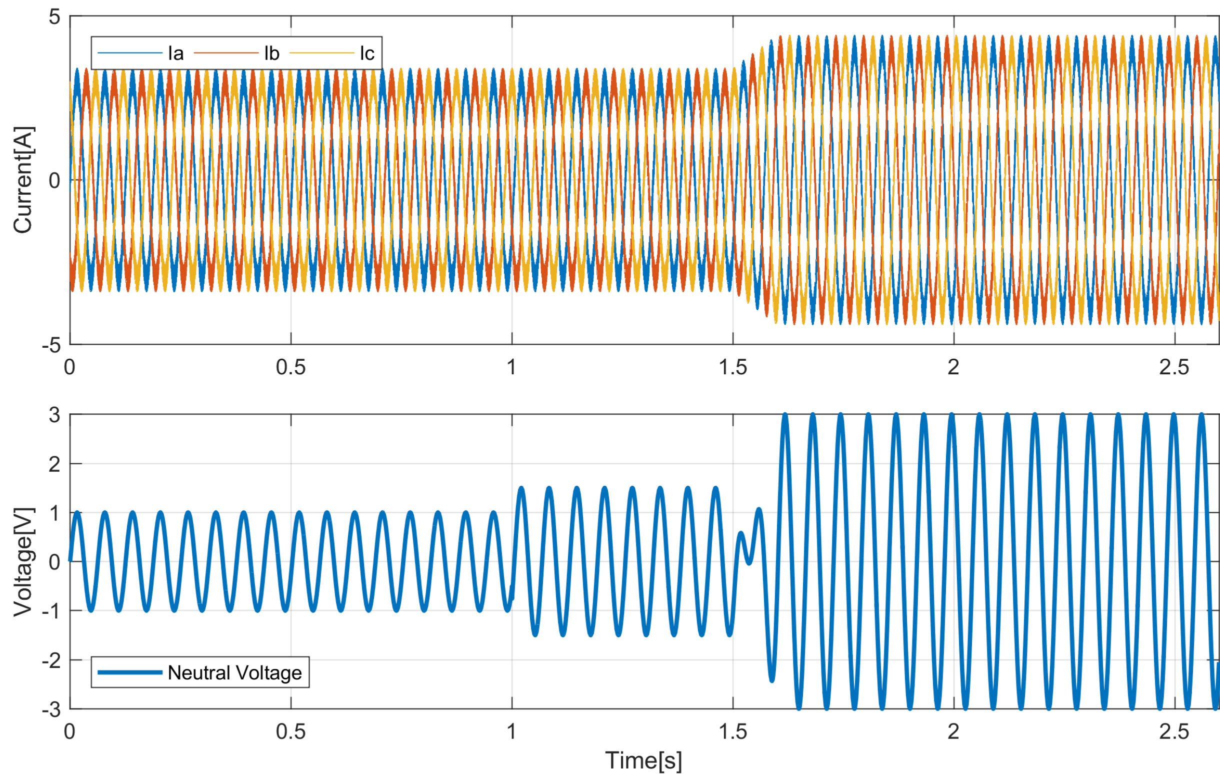


Figure 6.9 Measured phase currents and neutral point voltage waveforms (filtered) demonstrating successful distributed controller operation.

This neutral point voltage waveform shows that there is some initial fundamental frequency voltage component even during V/Hz operation which is most likely the result of imperfect dead-time compensation. After distributed control takes over, the neutral voltage increases slightly in amplitude as the proportional control begins acting to keep the independent complex vector regulators aligned. When the operating point changes after one second, the neutral voltage changes both phase and amplitude, increasing in amplitude along with the phase currents. The phase shift is different from what was observed during simulation, but that is not surprising since the simulation does not include saturation or dead-time effects in the machine or power electronics.

The currents are well behaved throughout the full test sequence until one phase misses its command transition and control of the other two phases is lost without it. The results of this test

sequence show that the distributed control method operates correctly and stably to regulate the phase currents and the neutral point voltage until the software problems cause the drive to stop operating.

A simulation run approximating the drive conditions is shown in figures 6.10 and 6.11. The simulation doesn't support the constant voltage mode used in hardware so the simulation starts in a zero-current command state. The phase interruption at operating point transition in the neutral voltage which appears at the transition is likely due to the actual hardware having more than just current sensor errors causing imbalance. The experimental run only covers the first 2.6 seconds of the simulation as the drive tripped almost immediately after the 2.6 second command change. The neutral point correction gain here is only 0.1 Siemens as the hardware is using a set of LEM sensors with accuracy in the 1 percent range. The current regulator is still tuned for 400 Hz bandwidth and the break point set slightly below the measured machine electrical time constant.

During recent debugging efforts, it was determined that the missing command transition is the result of a race condition that causes the ADC current measurements to overwrite the control command when the command update arrives between the PWM output updating and the ADC conversion completion. Since this time window corresponds to approximately 20us out of a 83us period, it is a common occurrence. This bug was discovered only after the rest of the code had been rewritten to be fully-flexible and tightly integrated with the telemetry code. Given these circumstances, back-porting the fix to re-try the fixed trajectory tests is not a trivial exercise since most of the control code would have to be rewritten again.

6.5 Conclusions

This chapter described the experimental hardware and firmware design. First, an experimental drive platform was described incorporating an integrated three-phase IGBT drive module, wide input voltage range logic power supply, Ethernet, and integrated sensing. This platform uses galvanic

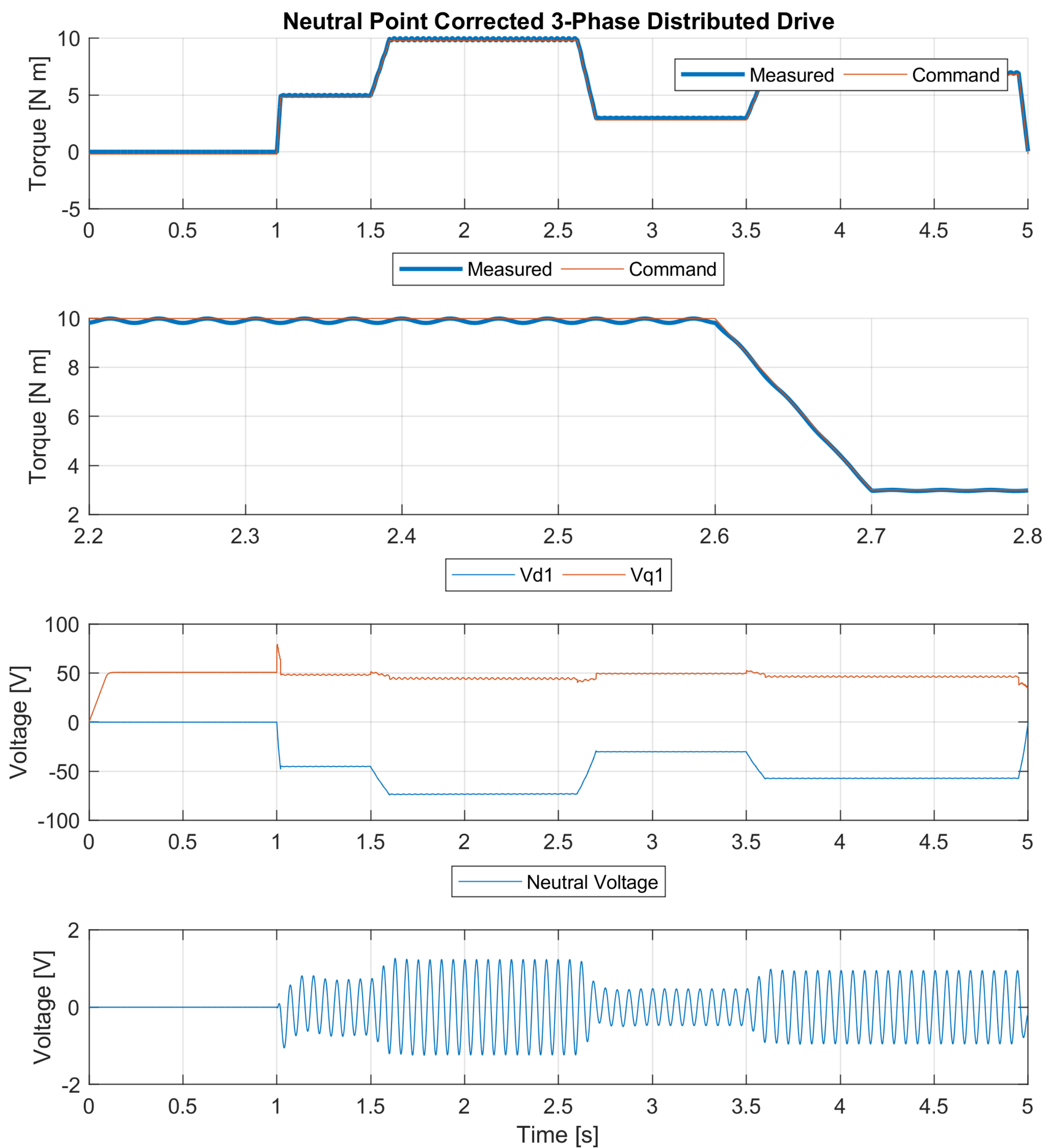


Figure 6.10 Distributed three-phase drive approximating the experimental setup with Neutral-Point based linear stabilization. Torque and machine terminal voltages are shown.

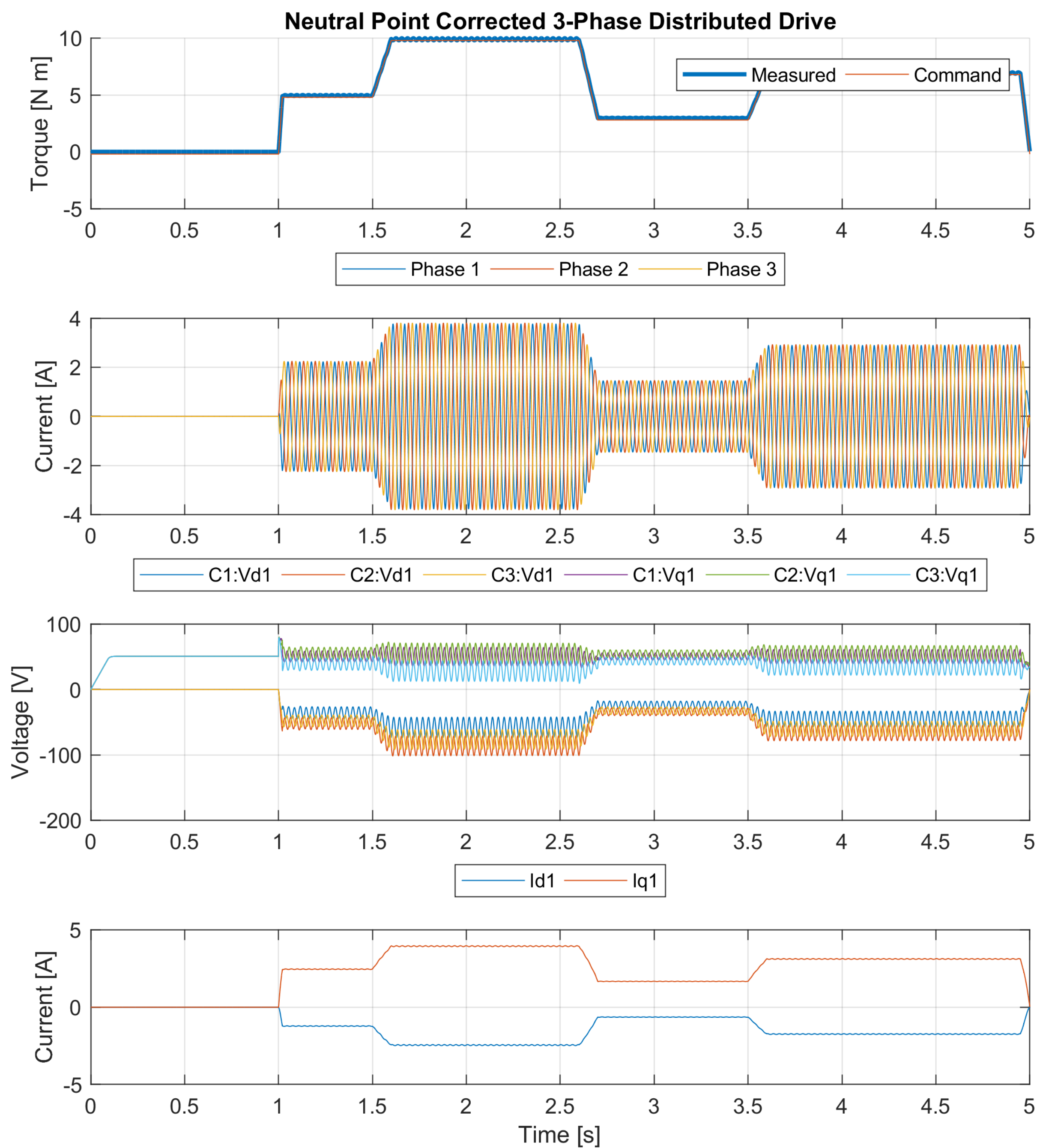


Figure 6.11 Distributed three-phase drive approximating the experimental setup with Neutral-Point based linear stabilization. Torque, terminal currents, and controller command voltages are shown.

isolation on the communications bus in order to reduce cost which is essential as the original experimental plan included tests using up to nine drives.

Next, the experimental test machine was described. The machine started life as an off-the-shelf industrial IPM synchronous motor donated by Baldor. This machine has been used in earlier research at WEMPEC and includes a flexible winding arrangement capable of being wired in a number of different phase configurations with is ideal for research in fault tolerance and arbitrary phase order machine control. The terminal parameters and winding arrangement for the three-phase tests that follow are also presented here.

Furthermore, the firmware structure for the embedded drive software is discussed including third-party libraries chosen to reduce feature development time and how the drive interacts with host computer software to receive commands and offload telemetry including all measurements and control states ate full switching rate.

Finally, an initial verification of the distributed current controller has been demonstrated using minimal firmware for a three-phase interior permanent magnet synchronous machine. This verification experiment delivered neutral voltage and phase current waveforms that exhibit key features which are highly correlated to those of the simulation results presented in Chapter 4. Machine terminal currents and the neutral point voltage were measured with an oscilloscope, and signal processing was used to strip away the high-frequency noise components in order to expose the underlying neutral point voltage waveform. The measured results demonstrate that closed-loop control of the neutral voltage is effective for preventing saturation of the distributed controllers, yielding well-regulated phase currents. The distributed current controller successfully assumed control after an initial interval of constant V/Hz excitation and continued operating until one of the phase controllers missed a command update, causing an over-current drive fault and shutdown. The cause of this missed transition was eventually discovered, but too late to incorporate it in an updated set of controller software.

Chapter 7

Contributions, Conclusions, and Future Work

7.1 Key Results Summary

This research is pursuing development, for the first time, of a distributed vector-controlled drive system for an n-phase machine (n=3 or higher) with no centralized controller and a minimum of shared sensing. Each modular drive uses a well-known current-regulated vector control algorithm and is supplied with a minimal set of current sensor feedback signals including the current in its own machine winding and at least one of the machine's $(n-1)$ other phase currents. Particular attention has been focused on the case where each controller receives only one current feedback signal from another phase in addition to its own (i.e., a total of 2 phase current signals). Using control techniques investigated previously for implementation of fault-tolerant machine controls and paralleled converters, a control method has been developed to coordinate the operation of multiple identical modular drives to implement vector control for an n-phase PM synchronous machine.

Simulation results have been developed for two 3-phase machine drives exciting PM synchronous machines with a floating-wye stator winding configuration, one with a conventional centralized (monolithic) controller, and the other using the proposed distributed motor control scheme with three modular phase controllers. Under ideal conditions with no asymmetries, the performance of both of these systems is identical. The two drive systems were then simulated again using non-ideal currents sensors as a source of asymmetry with a combination of gain and offset errors. Under these more realistic conditions, the current regulator behavior in the two drives are highly

similar in several ways. However, the drive with the distributed controllers exhibits an inherent instability in the controllers' output voltage responses arising from the presence of three current regulators attempting to control two independent current variables for a floating-wye (or delta) stator winding configuration. This instability has been suppressed by introducing a neutral-point feedback control loop in each controller that stabilizes the distributed drive system while maintaining high performance.

Analytical models have been presented that have been subsequently used to investigate how the effects of independent phase control and system unbalance present themselves. This analysis has been extended beyond three machine phases to include higher phase-order numbers (n >3). Control topologies and tuning techniques have been developed that allow a distributed drive to achieve nearly identical current and torque responses as a conventional current-regulated vector-controlled drive with a centralized controller while requiring no special inter-phase communications for normal operation.

7.2 Contributions

The key contributions achieved to date during this research are summarized as follows.

1. This research has developed and demonstrated, for the first time, a control technique that extends well-known multi-phase ac current regulation used in standard machine drives with centralized controllers to modular drives with distributed controllers in which each controller controls a single machine phase.

    (a) The effects of the presence of current sensor errors and partial sensor information in these modular drives were investigated. The results have been used to examine and explain the effects of system asymmetry in modular drives with distributed controllers and floating-wye or delta-connected phase windings on current regulator performance.

In these cases, the phase voltage outputs saturate due to open-loop integrators without feedback, causing the current regulators to lose control of their assigned currents.

(b) A closed-loop feedback technique has been developed that uses the neutral point voltage to correct for system asymmetry and provide the required compliance to the over-constrained distributed control problem, resulting in stable operation without sacrificing the performance of the complex vector current regulator.

(c) The closed-loop feedback technique using neutral voltages was analyzed as a disturbance rejecting regulator where the complex vector current regulators form the plant of the control system. Motor drive asymmetries are mapped to different disturbance injection locations.

(d) This distributed current regulator including the closed-loop instability suppression algorithm was demonstrated successfully for a three-phase machine (floating-wye) using both closed-form analysis and simulation.

(e) The distributed current regulator has been applied to open-winding machines with both H-bridge and individual phase controllers showing that the neutral voltage regulator can operate even when no physical neutral voltage exists. However, the neutral regulator gain must be increased to cope with the added zero-sequence admittance of the open winding machine.

(f) Finally, simulation was used to show that this distributed control current regulation technique can also be applied successfully to ac machines with five phases (floating-wye), demonstrating its broader promise for application to ac machines with high phase-order numbers greater than three.

2. This distributed current regulator control algorithm has demonstrated its ability to control torque production in PM synchronous machines with modular phase drives for both general-purpose V/Hz and high-performance vector control drive configurations.

   (a) It was shown that each phase control unit in a modular machine drive does not need to receive full current sensor feedback from the other $(n-1)$ phases in order to operate properly. This has been demonstrated using both closed-form analysis and simulation for both 3-phase and 5-phase machines .

   (b) It was demonstrated via simulation that PWM carrier synchronization is not needed among the modular phase drives in order to achieve well-behaved torque production provided that some penalties in common-mode EMI production, machine losses, and dV/dt stress can be tolerated, at least temporarily.

   (c) An analytical method based on modified controllability analysis was developed to evaluate alternative sensor connections and compensation arrangements for distributed machine control.

   (d) It was shown via simulation that the torque developed in both three- and five-phase ac machines can be controlled using distributed modular phase controllers with only two currents sensed per controller.

   (e) Single-current sensor operation was demonstrated in simulation by using a fictive-axis control technique designed for controlling single phase machines. However, this technique is sensitive to machine parameters and significantly limits the tuning capabilities of the drive. Furthermore, with only a a single current sensor there is no redundant sensor information available for fault detection.

3. The distributed current regulator control algorithm was extended to continue controlling torque in PM synchronous machines after open-circuit faults.

(a) By adding proportional feedforward functions to the current command generation, the distributed controller can operate post-fault with no change in tuning or structure.

(b) The neutral-voltage based controller stabilization method was extended by adding a non-zero neutral voltage command in order to match the non-zero neutral voltage of a machine operating with an incomplete set of phases.

(c) The distributed controller was shown to be compatible with well-known methods for redistributing the current commands from the remaining healthy phases in order to reduce the torque ripple in a faulted motor drive or to restore the original pre-fault average torque production.

4. Finally, a versatile and flexible open hardware platform has been developed for configuring and testing motor drive systems with distributed modular phase controllers.

   (a) The drive platform incorporates a DSP, Ethernet, and integrated three-phase power modules with integrated sensing to allow for large system data collection without the need for high-channel-count oscilloscopes.

   (b) Since the platform has been purposely designed with open hardware and software, any user has complete control over switching algorithms and sensing arrangements using either galvanically-isolated or non-isolated sensors, which is essential for the development of non-standard controls.

   (c) This hardware has been used to successfully demonstrate three-phase dc-to-ac power conversion while line-powered with only isolated Ethernet for control inputs at both 50V and 350V dc-link voltages.

   (d) This hardware platform configuration has shown itself to be well-suited for safe prototyping of control algorithms at low bus voltage levels in order to avoid equipment

damage, while still allowing full-power tests with no hardware changes after the low-power testing is completed. This capability is lacking in most off-the-shelf solutions due to hardware under-voltage lockouts in commercial systems, but it is essential for new controls development.

5. The flexible hardware platform was used to implement a three-phase distributed current regulator controlling an IPM synchronous machine.
   (a) The drive platform demonstrated the stability of the distributed control algorithm with neutral-point feedback while following a varying operating point command.
   (b) Measurements of the neutral voltage during both V/Hz operation and distributed current control document the desired feedback operation of the neutral voltage regulator.

7.3 Future Work

This work has uncovered a number of open questions which should be addressed in future research.

1. The fault-tolerance methods presented in this work have only been examined through simulation. These fault ride-through strategies should be verified in hardware.
2. This work has only considered fault ride-through control fault detection and intercalation is still an open problem and techniques must be developed that can robustly detect and determine faults even in the limited sensing environment of a distributed motor drive.
3. There is a gap in the literature with respect to modeling machines with multiple phases on the same electromagnetic axis under fault conditions. This is unfortunate as this is a common structure for fault-tolerant machines.

4. The development of the verification firmware has shown that command synchronization is essential to the operation of a distributed drive. The limits of this synchronization requirement should be studied and methods to ride-through missed command inputs should be considered.

5. This work has used an incremental shaft encoder to determine the rotor position. This encoder represents a single point of failure in a fault tolerant system. Future work should develop a position self-sensing method that is capable of operating under the limited sensing and distributed control environment.

6. The ability of the distributed drive to operate through open circuit faults provides and opportunity to preform online hot-swapping of drive modules while the machine is running. This capability would consist of "failing" a modular drive to regulate at zero current, isolation of the DC-link and drive phase, and finally transition back to healthy operation after the module has been replaced.

7. Drive module commissioning has been done manually in this work. As the drive modules are functionally identical automatic commissioning where the drive modules determine their own logical locations and the rotor position.

8. The distributed drive control method presented in this work should be implemented on an tightly integrated modular motor drive.

9. This work has focused on using the distributed current regulator to control synchronous electrical machines. Grid connected applications are a major application of polyphase power electronic converters. This distributed current regulator may be capable of operating as a grid-connected inverter or active rectifier if a distributed grid-synchronization algorithm can be devised.

Beyond this list, the existing distributed controller at the conclusion of my research is limited by additional potential failure points within the distributed control system. They are listed below in order of severity in order to provide guidance for future research efforts that seek to enhance the robustness of the distributed control hardware and software.

1. *The requirement of synchronous command input*: This work has assumed that all controllers receive simultaneous, identical command input. If this requirement is not met, the distributed drive fails quickly.

2. *Rotor position feedback:* Each modular controller in this distributed drive obtains position feedback from an incremental encoder. This creates a significant risk for a single-point failure, meaning that any fault in this feedback will likely cause drive failure.

3. *Neutral voltage sensing:* The stability of the distributed drive presented here uses a physical measurement of the neutral point voltage in order to coordinate the drive modules. While this is not a high-bandwidth or high-accuracy feedback path, loss of this sensor will cause the drive to trip due to voltage saturation almost immediately.

4. *Current feedback:* While this distributed motor control technique is no more dependent on current feedback than a standard drive with conventional centralized control, the distributed nature of the control poses some additional limitations that complicate the task of recognizing and rejecting erroneous current feedback information.

7.4 Minor Conclusions

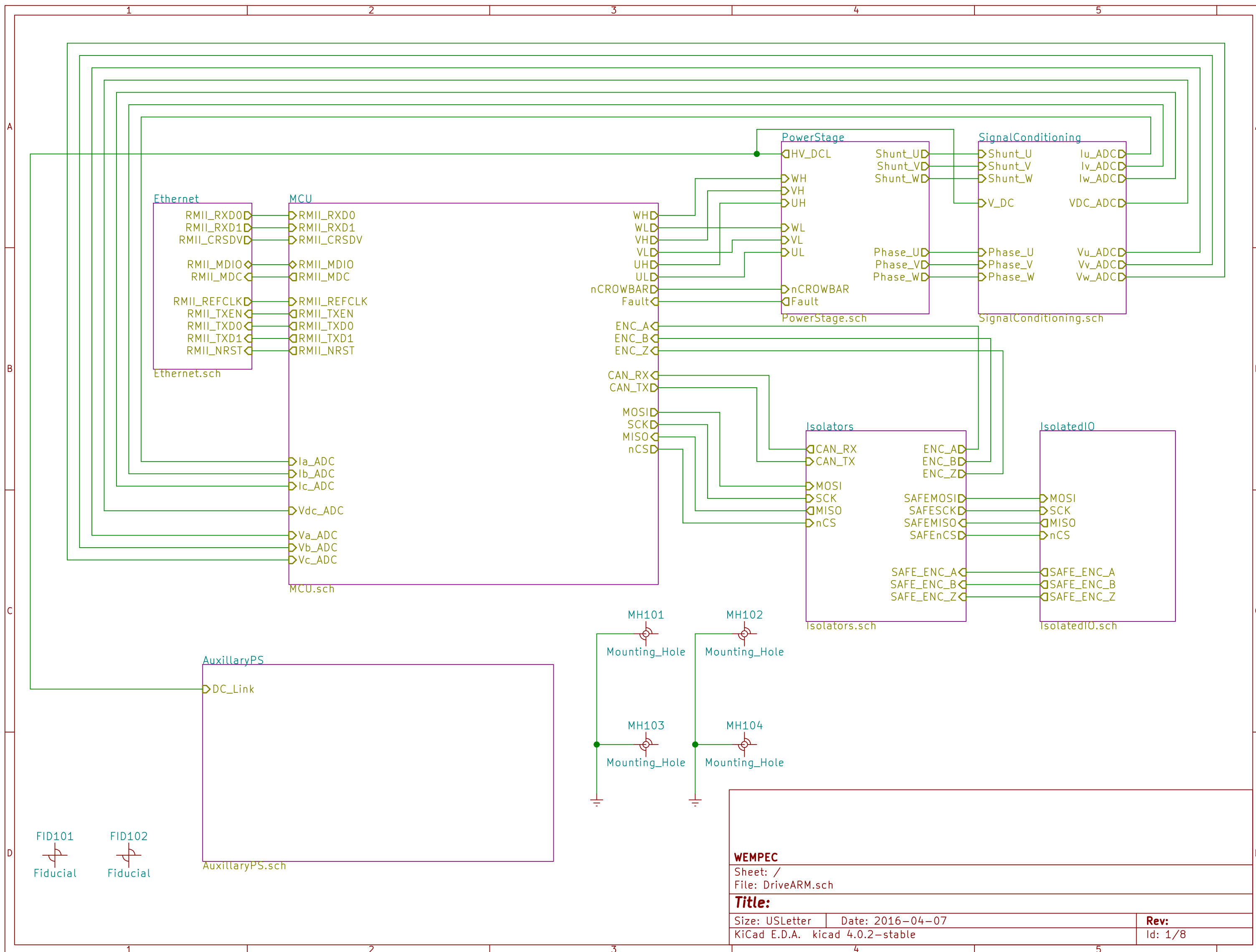
In addition to the major contributions above, there are a few minor conclusions which should be stated as they are important to further work in this area. These insights are already known, but are not common knowledge in our corner of engineering.

First among these, distributed control is difficult and while a simple approach may be possible to vectorize and distribute a control or computation problem, the coordination costs and software details merely to keep a distributed system synchronized can quickly overwhelm the functional portions of the problem.

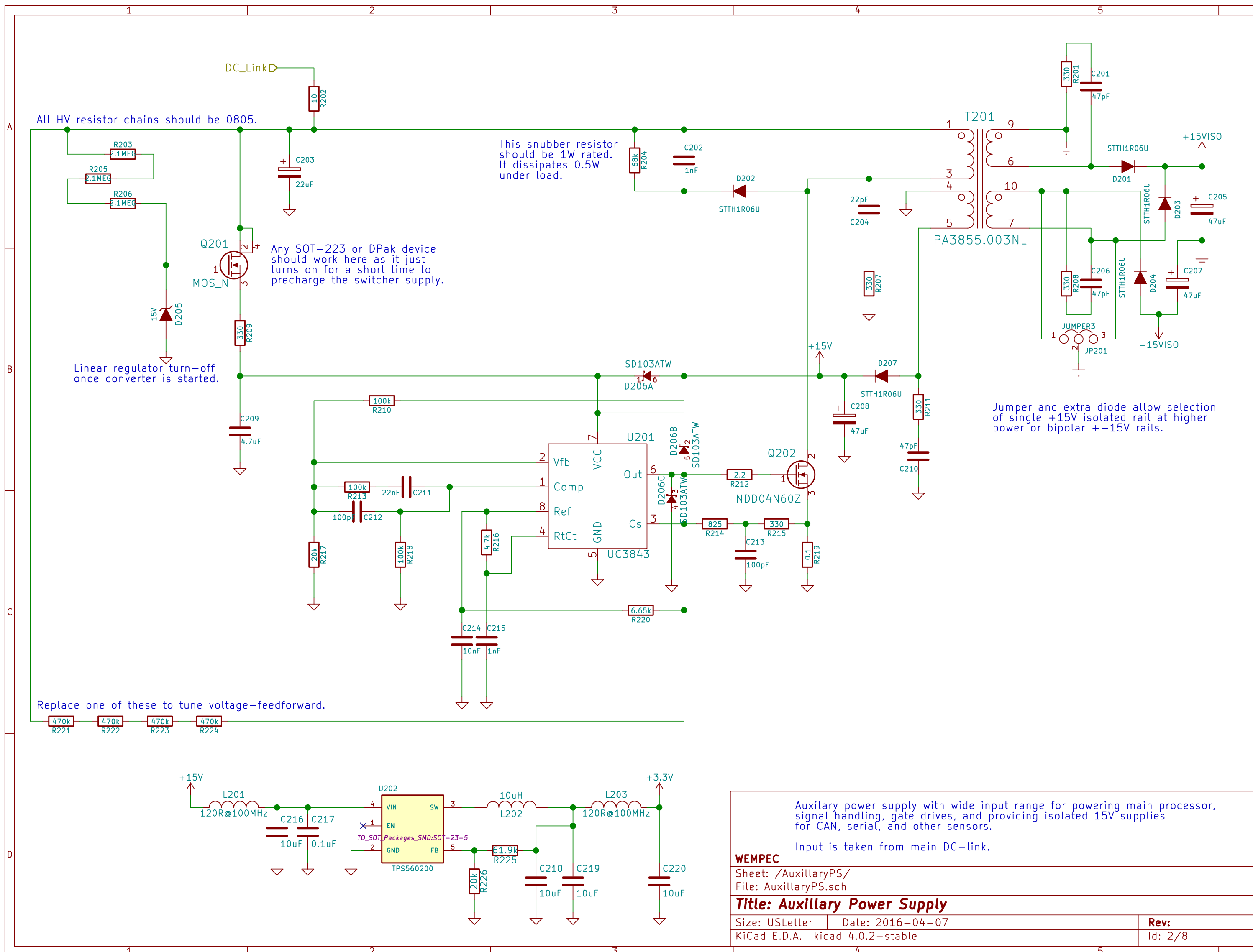
Second, tooling and protection are the most essential part of any power electronic control development. When comparing this work to previous efforts on distributed power electronics, significant time was saved due to hardware local protection features keeping damage from occurring when the control software inevitably fails. On the flip side, the limited insight available into the internal memory of DSP meant that locating control code failures was an exercise in educated guessing. In the future, more attention should be paid to software tooling and collaboration should be sought with more experienced software engineers.

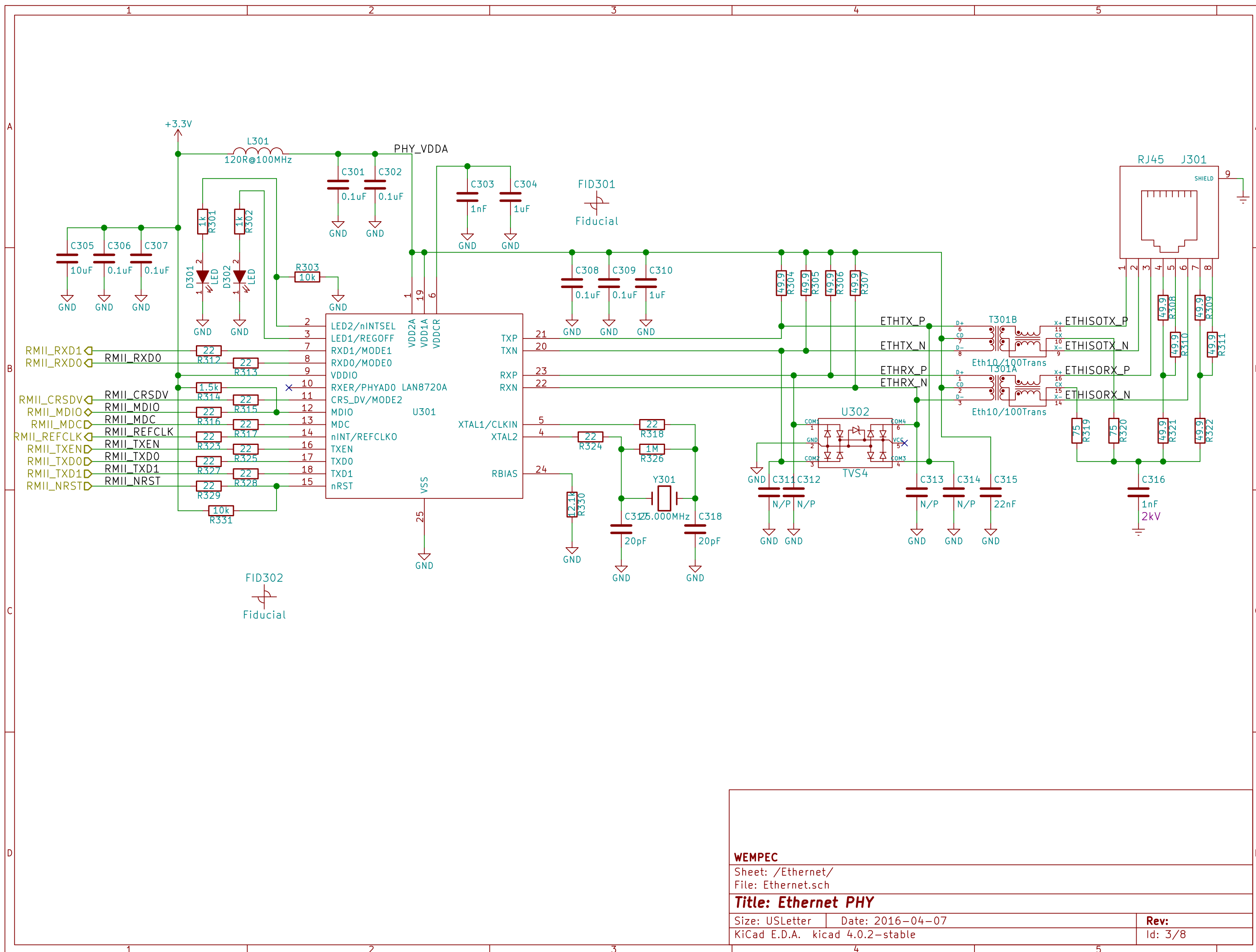
Finally, while modern power electronics has converged on a nearly optimal set of control techniques such as triangular carrier discontinuous SVPWM, these techniques increasingly depend on the high performance of modern controls. Insights from older techniques including asynchronous PWM and hysteretic controls are still quite applicable, and can achieve significant performance on modern equipment but may be more robust especially when attempting to create control systems that fail gracefully.

APPENDIX
Controller Schematics

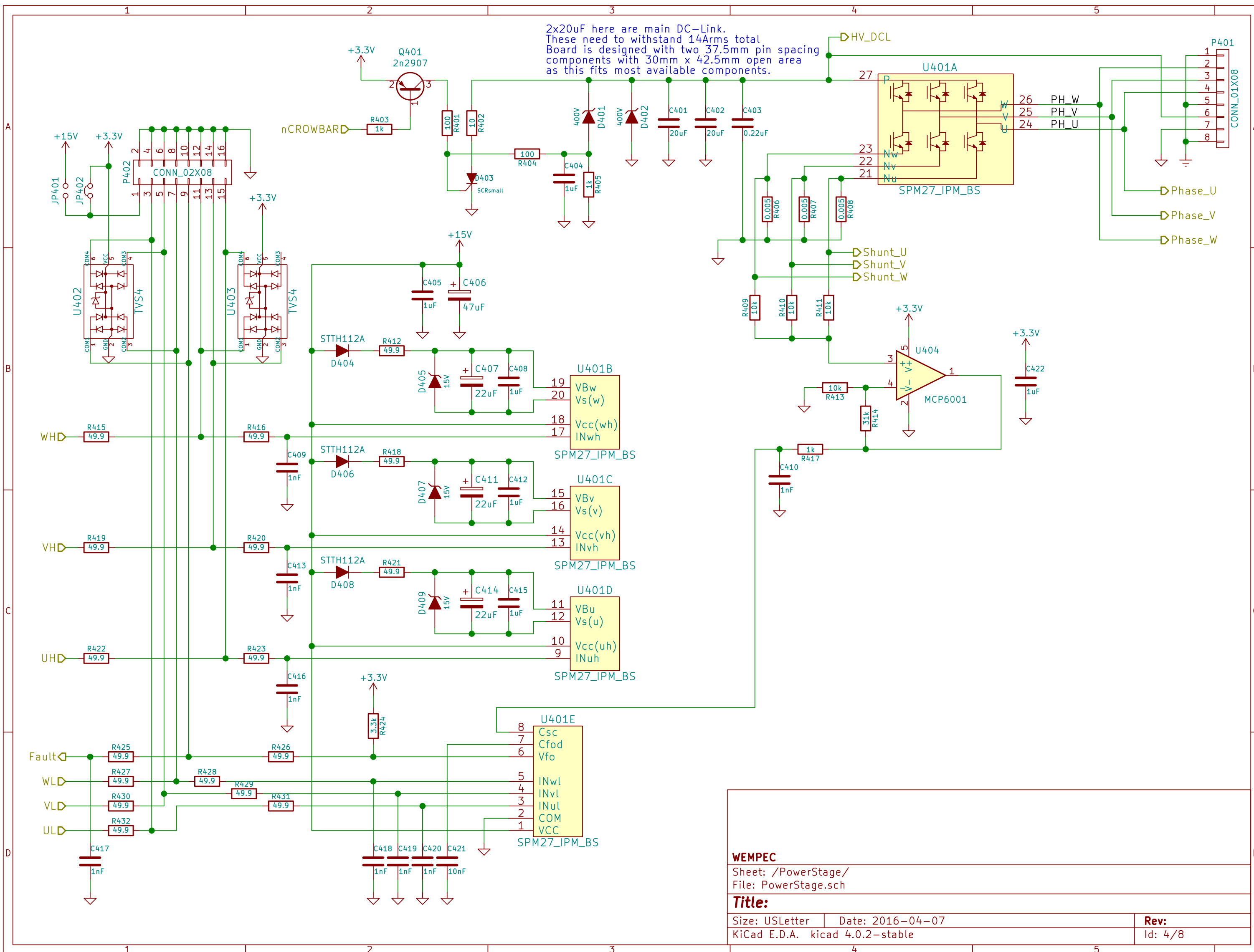


PowerStage
HV_DCL
WH
VH
UH
WL
VL
UL
nCROWBAR
Fault
Shunt_U
Shunt_V
Shunt_W
Phase_U
Phase_V
Phase_W
PowerStage.sch
SignalConditioning
Shunt_U
Shunt_V
Shunt_W
V_DC
Phase_U
Phase_V
Phase_W
Iu_ADC
Iv_ADC
Iw_ADC
VDC_ADC
Vu_ADC
Vv_ADC
Vw_ADC
SignalConditioning.sch
Ethernet
RMII_RXD0
RMII_RXD1
RMII_CRSDV
RMII_MDIO
RMII_MDC
RMII_REFCLK
RMII_TXEN
RMII_TXD0
RMII_TXD1
RMII_NRST
Ethernet.sch
MCU
RMII_RXD0
RMII_RXD1
RMII_CRSDV
RMII_MDIO
RMII_MDC
RMII_REFCLK
RMII_TXEN
RMII_TXD0
RMII_TXD1
RMII_NRST
Ia_ADC
Ib_ADC
Ic_ADC
Vdc_ADC
Va_ADC
Vb_ADC
Vc_ADC
WH
WL
VH
VL
UH
UL
nCROWBAR
Fault
ENC_A
ENC_B
ENC_Z
CAN_RX
CAN_TX
MOSI
SCK
MISO
nCS
MCU.sch
Isolators
CAN_RX
CAN_TX
MOSI
SCK
MISO
nCS
ENC_A
ENC_B
ENC_Z
SAFEMOSI
SAFESCK
SAFEMISO
SAFEnCS
SAFE_ENC_A
SAFE_ENC_B
SAFE_ENC_Z
Isolators.sch
IsolatedIO
MOSI
SCK
MISO
nCS
SAFE_ENC_A
SAFE_ENC_B
SAFE_ENC_Z
IsolatedIO.sch
MH101
Mounting_Hole
MH102
Mounting_Hole
MH103
Mounting_Hole
MH104
Mounting_Hole
AuxillaryPS
DC_Link
AuxillaryPS.sch
FID101
Fiducial
FID102
Fiducial
WEMPEC
Sheet: /
File: DriveARM.sch
Title:
Size: USLetter
Date: 2016-04-07
Rev:
KiCad E.D.A. kicad 4.0.2-stable
Id: 1/8

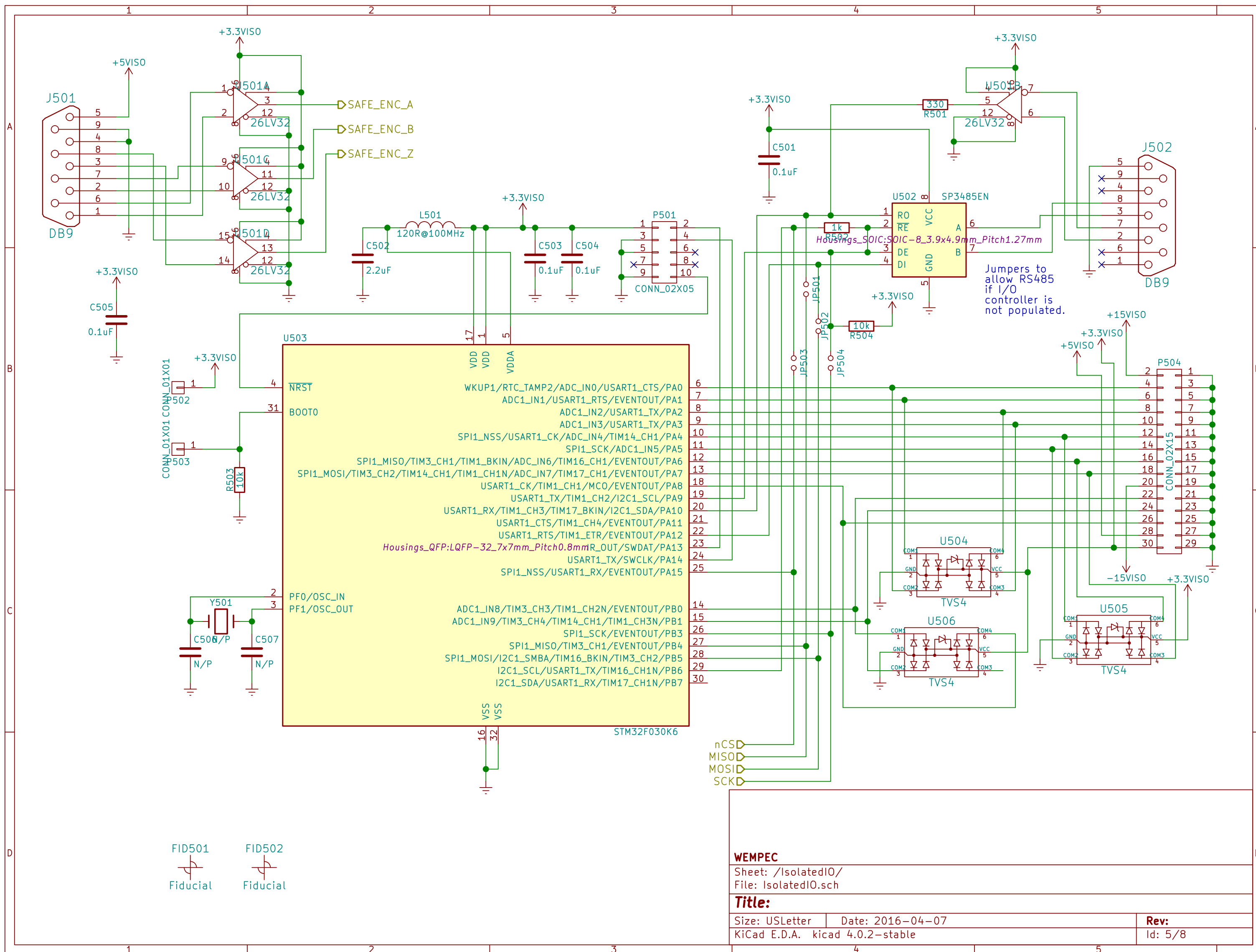

DC_Link
All HV resistor chains should be 0805.
This snubber resistor should be 1W rated. It dissipates 0.5W under load.
Any SOT-223 or DPak device should work here as it just turns on for a short time to precharge the switcher supply.
Linear regulator turn-off once converter is started.
Jumper and extra diode allow selection of single +15V isolated rail at higher power or bipolar +-15V rails.
Replace one of these to tune voltage-feedforward.
T201
PA3855.003NL
Q201
MOS_N
Q202
NDD04N60Z
U201
UC3843
Vfb
Comp
Ref
RtCt
VCC
Out
Cs
GND
U202
TPS560200
VIN
EN
GND
SW
FB
TO_SOT_Packages_SMD:SOT-23-5
+15VISO
-15VISO
+15V
+3.3V
JUMPER3
JP201
STTH1R06U
SD103ATW
L201
L202
L203
120R@100MHz
10uH
Auxilary power supply with wide input range for powering main processor, signal handling, gate drives, and providing isolated 15V supplies for CAN, serial, and other sensors.
Input is taken from main DC-link.
WEMPEC
Sheet: /AuxillaryPS/
File: AuxillaryPS.sch
Title: Auxillary Power Supply
Size: USLetter
Date: 2016-04-07
Rev:
KiCad E.D.A. kicad 4.0.2-stable
Id: 2/8



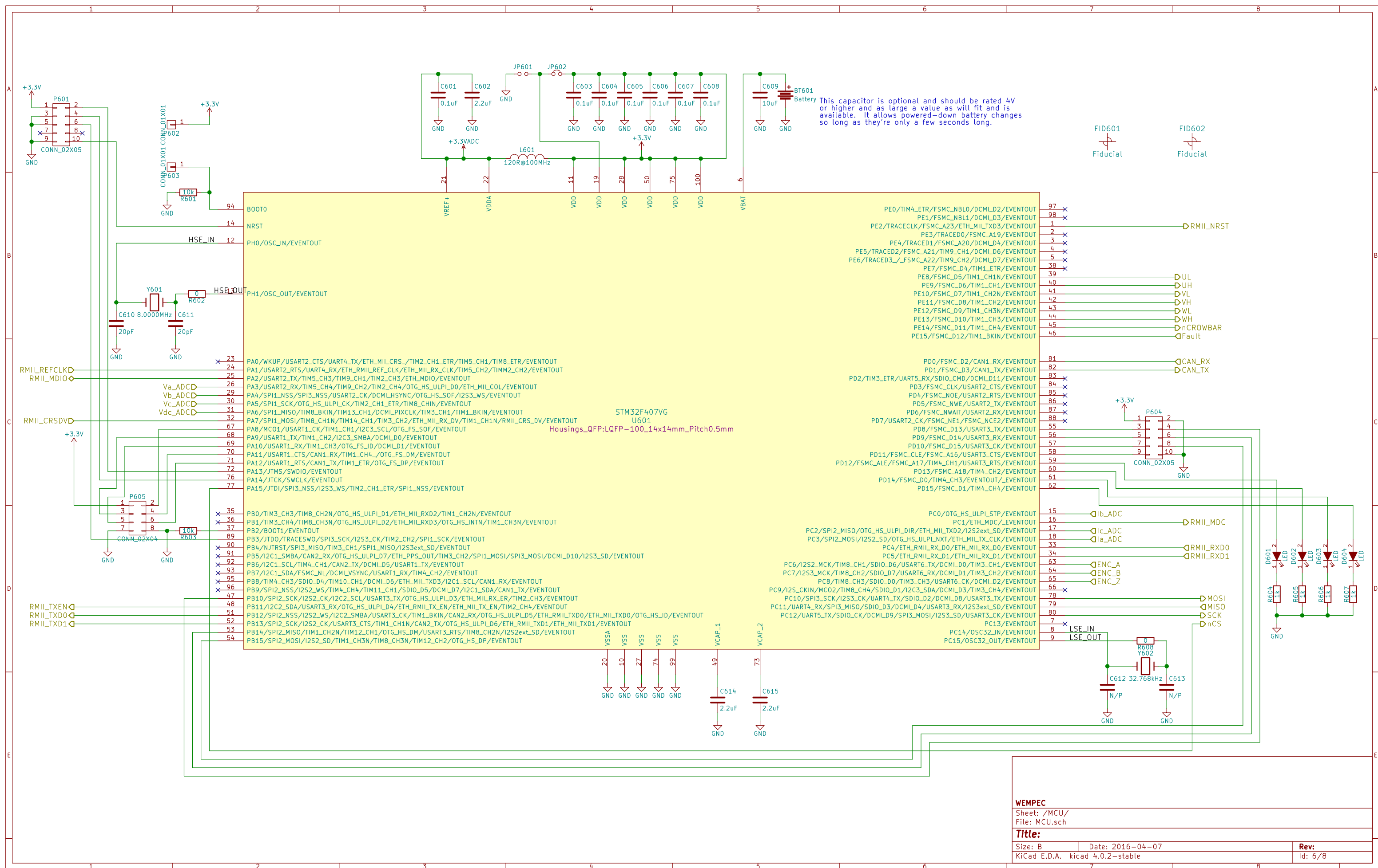
+3.3V
L301
120R@100MHz
PHY_VDDA
C301 0.1uF
C302 0.1uF
C303 1nF
C304 1uF
FID301
Fiducial
C305 10uF
C306 0.1uF
C307 0.1uF
R301 1k
R302 1k
D301 LED
D302 LED
R303 10k
C308 0.1uF
C309 0.1uF
C310 1uF
R304 49.9
R305 49.9
R306 49.9
R307 49.9
RJ45 J301
SHIELD
LED2/nINTSEL
LED1/REGOFF
RXD1/MODE1
RXD0/MODE0
VDDIO
RXER/PHYAD0
CRS_DV/MODE2
MDIO
MDC
nINT/REFCLKO
TXEN
TXD0
TXD1
nRST
VDD2A
VDD1A
VDDCR
LAN8720A
U301
TXP
TXN
RXP
RXN
XTAL1/CLKIN
XTAL2
RBIAS
VSS
RMII_RXD1
RMII_RXD0
RMII_CRSDV
RMII_MDIO
RMII_MDC
RMII_REFCLK
RMII_TXEN
RMII_TXD0
RMII_TXD1
RMII_NRST
R312 22
R313 22
R314 1.5k
R315 22
R316 22
R317 22
R323 22
R325 22
R327 22
R328 22
R329 22
R331 10k
R318 22
R324 22
R326 1M
Y301 25.000MHz
C317 20pF
C318 20pF
R330 12.1k
ETHTX_P
ETHTX_N
ETHRX_P
ETHRX_N
T301B
T301A
Eth10/100Trans
ETHISOTX_P
ETHISOTX_N
ETHISORX_P
ETHISORX_N
R308 49.9
R309 49.9
R310 49.9
R311 49.9
U302
TVS4
C311 N/P
C312 N/P
C313 N/P
C314 N/P
C315 22nF
R319 75
R320 75
R321 49.9
R322 49.9
C316 1nF 2kV
FID302
Fiducial
GND
WEMPEC
Sheet: /Ethernet/
File: Ethernet.sch
Title: Ethernet PHY
Size: USLetter
Date: 2016-04-07
Rev:
KiCad E.D.A. kicad 4.0.2-stable
Id: 3/8



2x20uF here are main DC-Link.
These need to withstand 14Arms total
Board is designed with two 37.5mm pin spacing
components with 30mm x 42.5mm open area
as this fits most available components.
HV_DCL
U401A
SPM27_IPM_BS
PH_W
PH_V
PH_U
P401
CONN_01X08
Phase_U
Phase_V
Phase_W
Shunt_U
Shunt_V
Shunt_W
Q401
2n2907
nCROWBAR
P402
CONN_02X08
JP401
JP402
U402
TVS4
U403
U401B
U401C
U401D
U401E
U404
MCP6001
WH
VH
UH
Fault
WL
VL
UL
STTH112A
WEMPEC
Sheet: /PowerStage/
File: PowerStage.sch
Title:
Size: USLetter
Date: 2016-04-07
Rev:
KiCad E.D.A. kicad 4.0.2-stable
Id: 4/8



J501
DB9
+5VISO
+3.3VISO
U501A
U501C
U501D
U501B
26LV32
SAFE_ENC_A
SAFE_ENC_B
SAFE_ENC_Z
C505
0.1uF
L501
120R@100MHz
C502
2.2uF
C503
0.1uF
C504
0.1uF
P501
CONN_02X05
C501
0.1uF
R501
330
U502
SP3485EN
R502
1k
Housings_SOIC:SOIC-8_3.9x4.9mm_Pitch1.27mm
J502
DB9
Jumpers to allow RS485 if I/O controller is not populated.
R504
10k
JP501
JP502
JP503
JP504
U503
STM32F030K6
Housings_QFP:LQFP-32_7x7mm_Pitch0.8mm
NRST
BOOT0
PF0/OSC_IN
PF1/OSC_OUT
WKUP1/RTC_TAMP2/ADC_IN0/USART1_CTS/PA0
ADC1_IN1/USART1_RTS/EVENTOUT/PA1
ADC1_IN2/USART1_TX/PA2
ADC1_IN3/USART1_TX/PA3
SPI1_NSS/USART1_CK/ADC_IN4/TIM14_CH1/PA4
SPI1_SCK/ADC1_IN5/PA5
SPI1_MISO/TIM3_CH1/TIM1_BKIN/ADC_IN6/TIM16_CH1/EVENTOUT/PA6
SPI1_MOSI/TIM3_CH2/TIM14_CH1/TIM1_CH1N/ADC_IN7/TIM17_CH1/EVENTOUT/PA7
USART1_CK/TIM1_CH1/MCO/EVENTOUT/PA8
USART1_TX/TIM1_CH2/I2C1_SCL/PA9
USART1_RX/TIM1_CH3/TIM17_BKIN/I2C1_SDA/PA10
USART1_CTS/TIM1_CH4/EVENTOUT/PA11
USART1_RTS/TIM1_ETR/EVENTOUT/PA12
IR_OUT/SWDAT/PA13
USART1_TX/SWCLK/PA14
SPI1_NSS/USART1_RX/EVENTOUT/PA15
ADC1_IN8/TIM3_CH3/TIM1_CH2N/EVENTOUT/PB0
ADC1_IN9/TIM3_CH4/TIM14_CH1/TIM1_CH3N/PB1
SPI1_SCK/USART1_RX/EVENTOUT/PB3
SPI1_MISO/TIM3_CH1/EVENTOUT/PB4
SPI1_MOSI/I2C1_SMBA/TIM16_BKIN/TIM3_CH2/PB5
I2C1_SCL/USART1_TX/TIM16_CH1N/PB6
I2C1_SDA/USART1_RX/TIM17_CH1N/PB7
VDD
VDDA
VSS
P502
P503
CONN_01X01
R503
10k
Y501
C506
C507
N/P
P504
CONN_02X15
+15VISO
-15VISO
U504
U505
U506
TVS4
nCS
MISO
MOSI
SCK
FID501
FID502
Fiducial
WEMPEC
Sheet: /IsolatedIO/
File: IsolatedIO.sch
Title:
Size: USLetter
Date: 2016-04-07
Rev:
KiCad E.D.A. kicad 4.0.2-stable
Id: 5/8


STM32F407VG
U601
Housings_QFP:LQFP-100_14x14mm_Pitch0.5mm
This capacitor is optional and should be rated 4V or higher and as large a value as will fit and is available. It allows powered-down battery changes so long as they're only a few seconds long.
FID601
Fiducial
FID602
Fiducial
WEMPEC
Sheet: /MCU/
File: MCU.sch
Title:
Size: B
Date: 2016-04-07
Rev:
KiCad E.D.A. kicad 4.0.2-stable
Id: 6/8

These current sense filters are MFB 3-pole Bessel filters with cutoff at 400kHz as a comprimise between settling time and noise reduction.

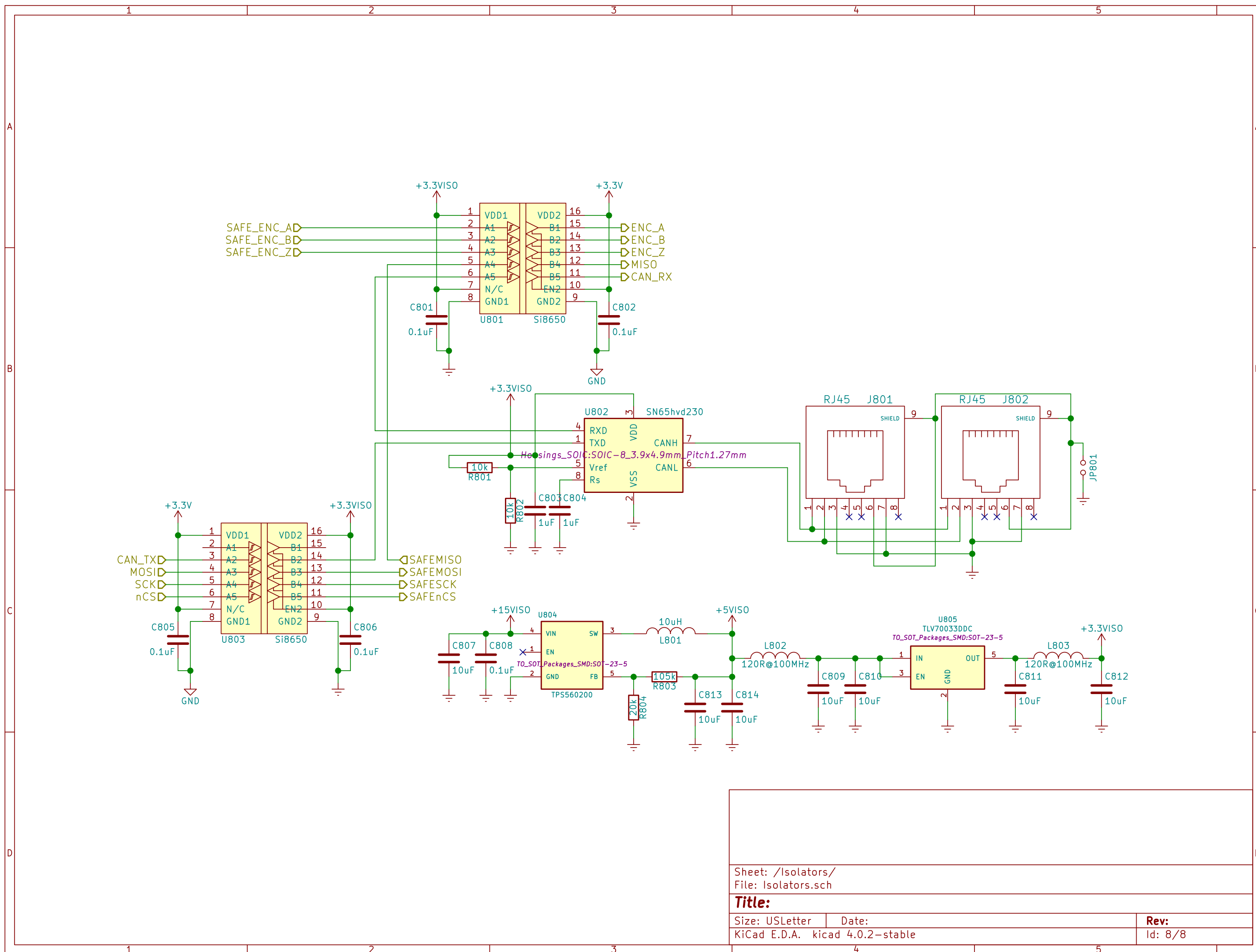
Be sure to include the amplifier dominant pole when calculating schematic values.

Each of these ground inputs should be connected kelvin-style to the associated shunt.

Phase_U, Phase_V, Phase_W, V_DC, Vu_ADC, Vv_ADC, Vw_ADC, VDC_ADC, Shunt_U, Shunt_V, Shunt_W, Iu_ADC, Iv_ADC, Iw_ADC, I_MID_REF, +3.3VADC

U701A, U701B, U701C, U701D Quad_Opamp; U702 TVS4; S701 SW_DIP_x6; P701, P703, P705, P707 CONN_01X03; P702, P704, P706 SMA

WEMPEC		
Sheet: /SignalConditioning/		
File: SignalConditioning.sch		
Title:		
Size: A	Date: 2016-04-07	Rev:
KiCad E.D.A. kicad 4.0.2-stable		Id: 7/8


+3.3VISO
+3.3V
SAFE_ENC_A
SAFE_ENC_B
SAFE_ENC_Z
VDD1
A1
A2
A3
A4
A5
N/C
GND1
VDD2
B1
B2
B3
B4
B5
EN2
GND2
ENC_A
ENC_B
ENC_Z
MISO
CAN_RX
U801
Si8650
C801
0.1uF
C802
0.1uF
GND
U802
SN65hvd230
RXD
TXD
Vref
Rs
VDD
VSS
CANH
CANL
Housings_SOIC:SOIC-8_3.9x4.9mm_Pitch1.27mm
10k
R801
R802
C803
C804
1uF
1uF
RJ45
J801
RJ45
J802
SHIELD
JP801
CAN_TX
MOSI
SCK
nCS
SAFEMISO
SAFEMOSI
SAFESCK
SAFEnCS
U803
Si8650
C805
0.1uF
C806
0.1uF
+15VISO
U804
VIN
SW
EN
GND
FB
TO_SOT_Packages_SMD:SOT-23-5
TPS560200
C807
10uF
C808
0.1uF
10uH
L801
+5VISO
L802
120R@100MHz
105k
R803
20k
R804
C813
10uF
C814
10uF
C809
10uF
C810
10uF
U805
TLV70033DDC
TO_SOT_Packages_SMD:SOT-23-5
IN
EN
OUT
GND
L803
120R@100MHz
+3.3VISO
C811
10uF
C812
10uF
Sheet: /Isolators/
File: Isolators.sch
Title:
Size: USLetter
Date:
Rev:
KiCad E.D.A. kicad 4.0.2-stable
Id: 8/8

APPENDIX
Bootloader Code

This section contains listings of the code used for the TFTP bootloader and the common files. This code needs ChibiOS-RT to be extracted into a parallel directory in order to build.

This bootloader copies itself into RAM, then starts a shell server and tftp server. The bootloader then tries to pull an nvram image and drive software image from the gateway address. If the drive software image is changed from the image in Flash it pulls the new image into flash and reboots.

The system will try to boot every 30s and will toggle the Blue LED when it checks for a healthy image.

B.1 f4boot/Makefile

```
################################################################################
# Build global options
# NOTE: Can be overridden externally.
#

# Compiler options here.
ifeq ($(USE_OPT),)
  USE_OPT = -Os -g3 -gdwarf-4 -fomit-frame-pointer -falign-functions=16 -D SHELL_CONFIG_FILE --specs=nano.specs --specs=nosys.
      ↪ specs
endif

# C specific options here (added to USE_OPT).
ifeq ($(USE_COPT),)
  USE_COPT =
endif

# C++ specific options here (added to USE_OPT).
ifeq ($(USE_CPPOPT),)
  USE_CPPOPT = -fno-rtti
endif

# Enable this if you want the linker to remove unused code and data
ifeq ($(USE_LINK_GC),)
  USE_LINK_GC = yes
endif

# Linker extra options here.
ifeq ($(USE_LDOPT),)
  USE_LDOPT =
endif

# Enable this if you want link time optimizations (LTO)
ifeq ($(USE_LTO),)
  USE_LTO = yes
endif

# If enabled, this option allows to compile the application in THUMB mode.
```

```
ifeq ($(USE_THUMB),)
  USE_THUMB = yes
endif

# Enable this if you want to see the full log while compiling.
ifeq ($(USE_VERBOSE_COMPILE),)
  USE_VERBOSE_COMPILE = no
endif

# If enabled, this option makes the build process faster by not compiling
# modules not used in the current configuration.
ifeq ($(USE_SMART_BUILD),)
  USE_SMART_BUILD = yes
endif

#
# Build global options
##############################################################################

##############################################################################
# Architecture or project specific options
#

# Stack size to be allocated to the Cortex-M process stack. This stack is
# the stack used by the main() thread.
ifeq ($(USE_PROCESS_STACKSIZE),)
  USE_PROCESS_STACKSIZE = 0x400
endif

# Stack size to the allocated to the Cortex-M main/exceptions stack. This
# stack is used for processing interrupts and exceptions.
ifeq ($(USE_EXCEPTIONS_STACKSIZE),)
  USE_EXCEPTIONS_STACKSIZE = 0x400
endif

# Enables the use of FPU (no, softfp, hard).
ifeq ($(USE_FPU),)
  USE_FPU = softfp
endif

#
# Architecture or project specific options
##############################################################################

##############################################################################
# Project, sources and paths
#

# Define project name here
PROJECT = bootloader

# Imported source files and paths
CHIBIOS = ../ChibiOS-RT
CHIBIOS_CONTRIB = $(CHIBIOS)/../ChibiOS-Contrib
# Startup files.
include $(CHIBIOS)/os/common/startup/ARMCMx/compilers/GCC/mk/startup_stm32f4xx.mk
# HAL-OSAL files (optional).
include $(CHIBIOS_CONTRIB)/os/hal/hal.mk
include $(CHIBIOS_CONTRIB)/os/hal/ports/STM32/STM32F4xx/platform.mk
#include $(CHIBIOS)/os/hal/hal.mk
#include $(CHIBIOS)/os/hal/ports/STM32/STM32F4xx/platform.mk
include ../common/board/board.mk
include $(CHIBIOS)/os/hal/osal/rt/osal.mk
# RTOS files (optional).
include $(CHIBIOS)/os/rt/rt.mk
include $(CHIBIOS)/os/common/ports/ARMCMx/compilers/GCC/mk/port_v7m.mk
# Other files (optional).
#include $(CHIBIOS)/test/rt/test.mk
include $(CHIBIOS)/os/hal/lib/streams/streams.mk
include $(CHIBIOS)/os/various/shell/shell.mk
include $(CHIBIOS)/os/various/lwip_bindings/lwip.mk

# Define linker script file here
LDSCRIPT= STM32F407xG.ld

# C sources that can be compiled in ARM or THUMB mode depending on the global
# setting.
CSRC = $(STARTUPSRC) \
       $(KERNSRC) \
       $(PORTSRC) \
       $(OSALSRC) \
       $(HALSRC) \
       $(PLATFORMSRC) \
       $(BOARDSRC) \
       $(LWSRC) \
       $(STREAMSSRC) \
```

```
        $(SHELLSRC) \
        $(CHIBIOS)/os/various/evtimer.c \
        main.c netstream.c tcp_shell.c shellconf.c \
        ../common/nvram.c ../commmon/exceptionvectors.c \
        ../common/rtclib.c memfs.c tftp.c flash.c \
        boot.c ../common/strtof.c lwipsupport.c

# C++ sources that can be compiled in ARM or THUMB mode depending on the global
# setting.
CPPSRC =

# C sources to be compiled in ARM mode regardless of the global setting.
# NOTE: Mixing ARM and THUMB mode enables the -mthumb-interwork compiler
#       option that results in lower performance and larger code size.
ACSRC =

# C++ sources to be compiled in ARM mode regardless of the global setting.
# NOTE: Mixing ARM and THUMB mode enables the -mthumb-interwork compiler
#       option that results in lower performance and larger code size.
ACPPSRC =

# C sources to be compiled in THUMB mode regardless of the global setting.
# NOTE: Mixing ARM and THUMB mode enables the -mthumb-interwork compiler
#       option that results in lower performance and larger code size.
TCSRC =

# C sources to be compiled in THUMB mode regardless of the global setting.
# NOTE: Mixing ARM and THUMB mode enables the -mthumb-interwork compiler
#       option that results in lower performance and larger code size.
TCPPSRC =

# List ASM source files here
ASMSRC =
ASMXSRC = $(STARTUPASM) $(PORTASM) $(OSALASM)

INCDIR = $(CHIBIOS)/os/license \
         $(STARTUPINC) $(KERNINC) $(PORTINC) $(OSALINC) \
         $(HALINC) $(PLATFORMINC) $(BOARDINC) \
         $(STREAMSINC) $(SHELLINC) $(LWINC) \
         $(CHIBIOS)/os/various \
         $(CHIBIOS_CONTRIB)/os/various

#
# Project, sources and paths
##############################################################################

##############################################################################
# Compiler settings
#

MCU  = cortex-m4

#TRGT = arm-elf-
TRGT = arm-none-eabi-
CC   = $(TRGT)gcc
CPPC = $(TRGT)g++
# Enable loading with g++ only if you need C++ runtime support.
# NOTE: You can use C++ even without C++ support if you are careful. C++
#       runtime support makes code size explode.
LD   = $(TRGT)gcc
#LD   = $(TRGT)g++
CP   = $(TRGT)objcopy
AS   = $(TRGT)gcc -x assembler-with-cpp
AR   = $(TRGT)ar
OD   = $(TRGT)objdump
SZ   = $(TRGT)size
HEX  = $(CP) -O ihex
BIN  = $(CP) -O binary

# ARM-specific options here
AOPT =

# THUMB-specific options here
TOPT = -mthumb -DTHUMB

# Define C warning options here
CWARN = -Wall -Wextra -Wundef -Wstrict-prototypes

# Define C++ warning options here
CPPWARN = -Wall -Wextra -Wundef

#
# Compiler settings
##############################################################################

##############################################################################
```

```
# Start of user section
#

# List all user C define here, like -D_DEBUG=1
#UDEFS =
UDEFS = -DCHPRINTF_USE_FLOAT=1 -DLWIP_DEBUG=1 -DTFTP_DEBUG=1

# Define ASM defines here
UADEFS =

# List all user directories here
UINCDIR = ../common

# List the user directory to look for the libraries here
ULIBDIR =

# List all user libraries here
ULIBS =

BUILDDIR = build

POST_MAKE_ALL_RULE_HOOK: $(BUILDDIR)/$(PROJECT).bin
	../../util/checksum.sh $(BUILDDIR)/$(PROJECT).bin
	cp $(BUILDDIR)/$(PROJECT).bin ../../tftp/

#
# End of user defines
##############################################################################

RULESPATH = $(CHIBIOS)/os/common/startup/ARMCMx/compilers/GCC
include $(RULESPATH)/rules.mk
```

B.2 f4boot/main.c

```
/*
    ChibiOS - Copyright (C) 2006..2016 Giovanni Di Sirio

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

        http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
*/

#include <stdio.h>
#include <string.h>

#include "ch.h"
#include "hal.h"
#include "vectors.h"

#include "chprintf.h"
#include "shellconf.h"
#include "shell.h"

#include "nvram.h"

#include "memfs.h"
#include "boot.h"
#include "tftp.h"

#include "lwipthread.h"
#include "tcp_shell.h"
#include "lwipsupport.h"

static thread_t *shelltp = NULL;
static THD_WORKING_AREA(wa_shell_thread, 1024) __attribute__((section(".ram3")));
static THD_WORKING_AREA(wa_tcp_shell_thread, 512) __attribute__((section(".ram3")));
static THD_WORKING_AREA(wa_boot_clock, 512) __attribute__((section(".ram3")));
static THD_WORKING_AREA(wa_tftp_server, TFTP_THREAD_STACK_SIZE);

/*===========================================================================*/
/* Command line related.                                                     */
/*===========================================================================*/

#define SHELL_WA_SIZE     THD_WORKING_AREA_SIZE(1024)
```

```
extern const ShellCommand commands[];

static char shhist[SHELL_MAX_HIST_BUFF];

static const ShellConfig shell_cfg1 = {
  (BaseSequentialStream *)&SD3,
  commands,
  shhist,
  SHELL_MAX_HIST_BUFF
};

/*
 * Shell exit event.
 */
static void ShellHandler(eventid_t id) {

  (void)id;
  if (chThdTerminatedX(shelltp)) {
    chThdWait(shelltp);                 /* Returning memory to heap.        */
    shelltp = NULL;
  }
}


/*===========================================================================*/
/* TFTP server related.                                                      */
/*===========================================================================*/
static const struct tftp_context memfiles = {
  mem_open, mem_close, mem_read, mem_write
};

/*===========================================================================*/
/* Main and generic code.                                                    */
/*===========================================================================*/
/*
 * Green LED blinker thread, times are in milliseconds.
 */
static THD_WORKING_AREA(waThread1, 64) __attribute__((section(".ram3")));
static THD_FUNCTION(Thread1, arg) {

  (void)arg;
  chRegSetThreadName("blinker");
  while (true) {
    palTogglePad(GPIOD, GPIOD_LED_GREEN);
    chThdSleepMilliseconds(shelltp ? 125 : 500);
  }
}

extern uint32_t __ram7_start__, __ram7_size__;
extern vectors_t _vectors;


/*
 * Application entry point.
 */
int main(void) {
  static const evhandler_t evhndl[] = {
    ShellHandler
  };
  event_listener_t el0;
  semaphore_t bootSem;


  /*
   * Transition system to running in RAM by copying lower 64k of flash, then
   * resetting the physical remap to start running from RAM.
   */
  memcpy(&__ram7_start__, &_vectors, (uint32_t)&__ram7_size__);
  rccEnableAPB2(RCC_APB2ENR_SYSCFGEN,0);
  SYSCFG->MEMRMP = 0x03;

  /*
   * System initializations.
   * - HAL initialization, this also initializes the configured device drivers
   *   and performs the board-specific initializations.
   * - Kernel initialization, the main() function becomes a thread and the
   *   RTOS is active.
   */
  halInit();
  chSysInit();

  /* TODO: Check the bootloader's checksum and try a backup */

  /*
   * Startup the battery backed nvram and get the ethernet running.
   */
```

```
  startNVRAM();
  lwipInit(checkEthAddress(&(nvram_storage->eth_options)));
//  lwipInit(NULL);

  /*
   * Activates the serial driver 6 and SDC driver 1 using default
   * configuration.
   */
  sdStart(&SD3, NULL);

  halLwipDiagInit();
  halLwipAcquire((BaseSequentialStream *)&SD3);

  /*
   * Shell manager initialization.
   */
  shellInit();
  shelltp = NULL;

  /*
   * Creates the blinker thread.
   */
  chThdCreateStatic(waThread1, sizeof(waThread1), NORMALPRIO, Thread1, NULL);

  /*
   * Create TFTP thread (using netconn).
   */
  chThdCreateStatic(wa_tftp_server, sizeof(wa_tftp_server), NORMALPRIO - 16,
                    tftp_server, (void *) &memfiles);

  /*
   * Create the low priority bootloader thread that waits then tries booting
   * the main flash image.
   */
  chSemObjectInit(&bootSem,0);
  chThdCreateStatic(wa_boot_clock, sizeof(wa_boot_clock), LOWPRIO + 1,
                    boot_clock, &bootSem);

  /*
   * Normal main() thread activity, handling shell start/exit.
   */

  chEvtRegister(&shell_terminated, &el0, 0);
  chThdCreateStatic(wa_tcp_shell_thread, sizeof(wa_tcp_shell_thread), NORMALPRIO,
                  tcp_shell_thread, (void *)3333U);

  chThdSleepSeconds(3);
  char nvramname[64];
  chsnprintf(nvramname, 64, "%s.nvram",
          ipaddr_ntoa((struct ip_addr *) &(nvram_storage->eth_options.address)));

  /* Try to grab a firmware image from the gateway, if it matches, boot now */
  tftp_get((struct ip_addr *) &(nvram_storage->eth_options.gateway),
           "test",
           "drive.bin",
           "r",
           &memfiles);

  chThdSleepSeconds(3);

  /* Grab some settings if we have them */
  tftp_get((struct ip_addr *) &(nvram_storage->eth_options.gateway),
           "config",
           nvramname,
           "r",
           &memfiles);

  /* Compare header to flashed firmware */
  if (*(uint32_t *)(CCMDATARAM_BASE + 0x24) == 0x57495343) {
    /* Server has a firmware */
    if (memcmp((void *)CCMDATARAM_BASE, (void *)(FLASH_BASE + BLOADER_SIZE), 512)) {
      /* Firmware is different */
      tftp_get((struct ip_addr *) &(nvram_storage->eth_options.gateway),
               "core",
               "drive.bin",
               "r",
               &memfiles);
    } else {
      /* Firmware is same, boot soon */
      chThdSleepSeconds(3);
      chSemSignal(&bootSem);
    }
  }

  while (true) {
    if (!shelltp) {
```

```
      shelltp = chThdCreateStatic(wa_shell_thread, sizeof(wa_shell_thread),
                                  NORMALPRIO + 1, shellThread,
                                  (void *)&shell_cfg1);
      chRegSetThreadNameX(shelltp, "shell");
    }
    chEvtDispatch(evhndl, chEvtWaitOneTimeout(ALL_EVENTS, MS2ST(500)));
  }
}

/* vim: set sts=2 sw=2 expandtab : */
```

B.3 f4boot/boot.h

```
#ifndef BOOT_H
#define BOOT_H

#include <ch.h>
#include <stdint.h>

/**
 * The final value for a standard CRC-32 if an embedded checkusm is correct.
 */
#define CRC_FINAL 0x38fb2284UL

bool bootCheckImage(void * start, void * end);
void bootJumpImage(void * address);
THD_FUNCTION(boot_clock, arg);

#endif /* BOOT_H */
/* vim: set tabstop=8 softtabstop=2 shiftwidth=2 expandtab : */
```

B.4 f4boot/chconf.h

```
/*
    ChibiOS - Copyright (C) 2006..2016 Giovanni Di Sirio

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

        http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
*/

/**
 * @file    templates/chconf.h
 * @brief   Configuration file template.
 * @details A copy of this file must be placed in each project directory, it
 *          contains the application specific kernel settings.
 *
 * @addtogroup config
 * @details Kernel related settings and hooks.
 * @{
 */

#ifndef CHCONF_H
#define CHCONF_H

#define _CHIBIOS_RT_CONF_

/*===========================================================================*/
/**
 * @name System timers settings
 * @{
 */
/*===========================================================================*/

/**
 * @brief   System time counter resolution.
 * @note    Allowed values are 16 or 32 bits.
 */
#define CH_CFG_ST_RESOLUTION                32

/**
 * @brief   System tick frequency.
```

```
 * @details Frequency of the system timer that drives the system ticks. This
 *          setting also defines the system tick time unit.
 */
#define CH_CFG_ST_FREQUENCY                 10000

/**
 * @brief   Time delta constant for the tick-less mode.
 * @note    If this value is zero then the system uses the classic
 *          periodic tick. This value represents the minimum number
 *          of ticks that is safe to specify in a timeout directive.
 *          The value one is not valid, timeouts are rounded up to
 *          this value.
 */
#define CH_CFG_ST_TIMEDELTA                 2

/** @} */

/*===========================================================================*/
/**
 * @name Kernel parameters and options
 * @{
 */
/*===========================================================================*/

/**
 * @brief   Round robin interval.
 * @details This constant is the number of system ticks allowed for the
 *          threads before preemption occurs. Setting this value to zero
 *          disables the preemption for threads with equal priority and the
 *          round robin becomes cooperative. Note that higher priority
 *          threads can still preempt, the kernel is always preemptive.
 * @note    Disabling the round robin preemption makes the kernel more compact
 *          and generally faster.
 * @note    The round robin preemption is not supported in tickless mode and
 *          must be set to zero in that case.
 */
#define CH_CFG_TIME_QUANTUM                 0

/**
 * @brief   Managed RAM size.
 * @details Size of the RAM area to be managed by the OS. If set to zero
 *          then the whole available RAM is used. The core memory is made
 *          available to the heap allocator and/or can be used directly through
 *          the simplified core memory allocator.
 *
 * @note    In order to let the OS manage the whole RAM the linker script must
 *          provide the @p __heap_base__ and @p __heap_end__ symbols.
 * @note    Requires @p CH_CFG_USE_MEMCORE.
 */
#define CH_CFG_MEMCORE_SIZE                 0

/**
 * @brief   Idle thread automatic spawn suppression.
 * @details When this option is activated the function @p chSysInit()
 *          does not spawn the idle thread. The application @p main()
 *          function becomes the idle thread and must implement an
 *          infinite loop.
 */
#define CH_CFG_NO_IDLE_THREAD               FALSE

/** @} */

/*===========================================================================*/
/**
 * @name Performance options
 * @{
 */
/*===========================================================================*/

/**
 * @brief   OS optimization.
 * @details If enabled then time efficient rather than space efficient code
 *          is used when two possible implementations exist.
 *
 * @note    This is not related to the compiler optimization options.
 * @note    The default is @p TRUE.
 */
#define CH_CFG_OPTIMIZE_SPEED               FALSE

/** @} */

/*===========================================================================*/
/**
 * @name Subsystem options
 * @{
 */
```

```
/*===========================================================================*/

/**
 * @brief   Time Measurement APIs.
 * @details If enabled then the time measurement APIs are included in
 *          the kernel.
 *
 * @note    The default is @p TRUE.
 */
#define CH_CFG_USE_TM                       FALSE

/**
 * @brief   Threads registry APIs.
 * @details If enabled then the registry APIs are included in the kernel.
 *
 * @note    The default is @p TRUE.
 */
#define CH_CFG_USE_REGISTRY                 TRUE

/**
 * @brief   Threads synchronization APIs.
 * @details If enabled then the @p chThdWait() function is included in
 *          the kernel.
 *
 * @note    The default is @p TRUE.
 */
#define CH_CFG_USE_WAITEXIT                 TRUE

/**
 * @brief   Semaphores APIs.
 * @details If enabled then the Semaphores APIs are included in the kernel.
 *
 * @note    The default is @p TRUE.
 */
#define CH_CFG_USE_SEMAPHORES               TRUE

/**
 * @brief   Semaphores queuing mode.
 * @details If enabled then the threads are enqueued on semaphores by
 *          priority rather than in FIFO order.
 *
 * @note    The default is @p FALSE. Enable this if you have special
 *          requirements.
 * @note    Requires @p CH_CFG_USE_SEMAPHORES.
 */
#define CH_CFG_USE_SEMAPHORES_PRIORITY      FALSE

/**
 * @brief   Mutexes APIs.
 * @details If enabled then the mutexes APIs are included in the kernel.
 *
 * @note    The default is @p TRUE.
 */
#define CH_CFG_USE_MUTEXES                  FALSE

/**
 * @brief   Enables recursive behavior on mutexes.
 * @note    Recursive mutexes are heavier and have an increased
 *          memory footprint.
 *
 * @note    The default is @p FALSE.
 * @note    Requires @p CH_CFG_USE_MUTEXES.
 */
#define CH_CFG_USE_MUTEXES_RECURSIVE        FALSE

/**
 * @brief   Conditional Variables APIs.
 * @details If enabled then the conditional variables APIs are included
 *          in the kernel.
 *
 * @note    The default is @p TRUE.
 * @note    Requires @p CH_CFG_USE_MUTEXES.
 */
#define CH_CFG_USE_CONDVARS                 FALSE

/**
 * @brief   Conditional Variables APIs with timeout.
 * @details If enabled then the conditional variables APIs with timeout
 *          specification are included in the kernel.
 *
 * @note    The default is @p TRUE.
 * @note    Requires @p CH_CFG_USE_CONDVARS.
 */
#define CH_CFG_USE_CONDVARS_TIMEOUT         FALSE

/**
```

```
 * @brief    Events Flags APIs.
 * @details If enabled then the event flags APIs are included in the kernel.
 *
 * @note     The default is @p TRUE.
 */
#define CH_CFG_USE_EVENTS                   TRUE

/**
 * @brief    Events Flags APIs with timeout.
 * @details If enabled then the events APIs with timeout specification
 *           are included in the kernel.
 *
 * @note     The default is @p TRUE.
 * @note     Requires @p CH_CFG_USE_EVENTS.
 */
#define CH_CFG_USE_EVENTS_TIMEOUT           TRUE

/**
 * @brief    Synchronous Messages APIs.
 * @details If enabled then the synchronous messages APIs are included
 *           in the kernel.
 *
 * @note     The default is @p TRUE.
 */
#define CH_CFG_USE_MESSAGES                 FALSE

/**
 * @brief    Synchronous Messages queuing mode.
 * @details If enabled then messages are served by priority rather than in
 *           FIFO order.
 *
 * @note     The default is @p FALSE. Enable this if you have special
 *           requirements.
 * @note     Requires @p CH_CFG_USE_MESSAGES.
 */
#define CH_CFG_USE_MESSAGES_PRIORITY        FALSE

/**
 * @brief    Mailboxes APIs.
 * @details If enabled then the asynchronous messages (mailboxes) APIs are
 *           included in the kernel.
 *
 * @note     The default is @p TRUE.
 * @note     Requires @p CH_CFG_USE_SEMAPHORES.
 */
#define CH_CFG_USE_MAILBOXES                TRUE

/**
 * @brief    Core Memory Manager APIs.
 * @details If enabled then the core memory manager APIs are included
 *           in the kernel.
 *
 * @note     The default is @p TRUE.
 */
#define CH_CFG_USE_MEMCORE                  TRUE

/**
 * @brief    Heap Allocator APIs.
 * @details If enabled then the memory heap allocator APIs are included
 *           in the kernel.
 *
 * @note     The default is @p TRUE.
 * @note     Requires @p CH_CFG_USE_MEMCORE and either @p CH_CFG_USE_MUTEXES or
 *           @p CH_CFG_USE_SEMAPHORES.
 * @note     Mutexes are recommended.
 */
#define CH_CFG_USE_HEAP                     TRUE

/**
 * @brief    Memory Pools Allocator APIs.
 * @details If enabled then the memory pools allocator APIs are included
 *           in the kernel.
 *
 * @note     The default is @p TRUE.
 */
#define CH_CFG_USE_MEMPOOLS                 FALSE

/**
 * @brief    Dynamic Threads APIs.
 * @details If enabled then the dynamic threads creation APIs are included
 *           in the kernel.
 *
 * @note     The default is @p TRUE.
 * @note     Requires @p CH_CFG_USE_WAITEXIT.
 * @note     Requires @p CH_CFG_USE_HEAP and/or @p CH_CFG_USE_MEMPOOLS.
 */
```

```
#define CH_CFG_USE_DYNAMIC                  TRUE

/** @} */

/*===========================================================================*/
/**
 * @name Debug options
 * @{
 */
/*===========================================================================*/

/**
 * @brief   Debug option, kernel statistics.
 *
 * @note    The default is @p FALSE.
 */
#define CH_DBG_STATISTICS                   FALSE

/**
 * @brief   Debug option, system state check.
 * @details If enabled the correct call protocol for system APIs is checked
 *          at runtime.
 *
 * @note    The default is @p FALSE.
 */
#define CH_DBG_SYSTEM_STATE_CHECK           FALSE

/**
 * @brief   Debug option, parameters checks.
 * @details If enabled then the checks on the API functions input
 *          parameters are activated.
 *
 * @note    The default is @p FALSE.
 */
#define CH_DBG_ENABLE_CHECKS                FALSE

/**
 * @brief   Debug option, consistency checks.
 * @details If enabled then all the assertions in the kernel code are
 *          activated. This includes consistency checks inside the kernel,
 *          runtime anomalies and port-defined checks.
 *
 * @note    The default is @p FALSE.
 */
#define CH_DBG_ENABLE_ASSERTS               FALSE

/**
 * @brief   Debug option, trace buffer.
 * @details If enabled then the trace buffer is activated.
 *
 * @note    The default is @p CH_DBG_TRACE_MASK_DISABLED.
 */
#define CH_DBG_TRACE_MASK                   CH_DBG_TRACE_MASK_DISABLED

/**
 * @brief   Trace buffer entries.
 * @note    The trace buffer is only allocated if @p CH_DBG_TRACE_MASK is
 *          different from @p CH_DBG_TRACE_MASK_DISABLED.
 */
#define CH_DBG_TRACE_BUFFER_SIZE            128

/**
 * @brief   Debug option, stack checks.
 * @details If enabled then a runtime stack check is performed.
 *
 * @note    The default is @p FALSE.
 * @note    The stack check is performed in a architecture/port dependent way.
 *          It may not be implemented or some ports.
 * @note    The default failure mode is to halt the system with the global
 *          @p panic_msg variable set to @p NULL.
 */
#define CH_DBG_ENABLE_STACK_CHECK           FALSE

/**
 * @brief   Debug option, stacks initialization.
 * @details If enabled then the threads working area is filled with a byte
 *          value when a thread is created. This can be useful for the
 *          runtime measurement of the used stack.
 *
 * @note    The default is @p FALSE.
 */
#define CH_DBG_FILL_THREADS                 TRUE

/**
 * @brief   Debug option, threads profiling.
 * @details If enabled then a field is added to the @p thread_t structure that
```

```
 *          counts the system ticks occurred while executing the thread.
 *
 * @note    The default is @p FALSE.
 * @note    This debug option is not currently compatible with the
 *          tickless mode.
 */
#define CH_DBG_THREADS_PROFILING            FALSE

/** @} */

/*===========================================================================*/
/**
 * @name Kernel hooks
 * @{
 */
/*===========================================================================*/

/**
 * @brief   Threads descriptor structure extension.
 * @details User fields added to the end of the @p thread_t structure.
 */
#define CH_CFG_THREAD_EXTRA_FIELDS                                          \
  /* Add threads custom fields here.*/

/**
 * @brief   Threads initialization hook.
 * @details User initialization code added to the @p chThdInit() API.
 *
 * @note    It is invoked from within @p chThdInit() and implicitly from all
 *          the threads creation APIs.
 */
#define CH_CFG_THREAD_INIT_HOOK(tp) {                                       \
  /* Add threads initialization code here.*/                                \
}

/**
 * @brief   Threads finalization hook.
 * @details User finalization code added to the @p chThdExit() API.
 */
#define CH_CFG_THREAD_EXIT_HOOK(tp) {                                       \
  /* Add threads finalization code here.*/                                  \
}

/**
 * @brief   Context switch hook.
 * @details This hook is invoked just before switching between threads.
 */
#define CH_CFG_CONTEXT_SWITCH_HOOK(ntp, otp) {                              \
  /* Context switch code here.*/                                            \
}

/**
 * @brief   ISR enter hook.
 */
#define CH_CFG_IRQ_PROLOGUE_HOOK() {                                        \
  /* IRQ prologue code here.*/                                              \
}

/**
 * @brief   ISR exit hook.
 */
#define CH_CFG_IRQ_EPILOGUE_HOOK() {                                        \
  /* IRQ epilogue code here.*/                                              \
}

/**
 * @brief   Idle thread enter hook.
 * @note    This hook is invoked within a critical zone, no OS functions
 *          should be invoked from here.
 * @note    This macro can be used to activate a power saving mode.
 */
#define CH_CFG_IDLE_ENTER_HOOK() {                                          \
  /* Idle-enter code here.*/                                                \
}

/**
 * @brief   Idle thread leave hook.
 * @note    This hook is invoked within a critical zone, no OS functions
 *          should be invoked from here.
 * @note    This macro can be used to deactivate a power saving mode.
 */
#define CH_CFG_IDLE_LEAVE_HOOK() {                                          \
  /* Idle-leave code here.*/                                                \
}

/**
```

```
 * @brief    Idle Loop hook.
 * @details This hook is continuously invoked by the idle thread loop.
 */
#define CH_CFG_IDLE_LOOP_HOOK() {                                         \
  /* Idle loop code here.*/                                               \
}

/**
 * @brief    System tick event hook.
 * @details This hook is invoked in the system tick handler immediately
 *           after processing the virtual timers queue.
 */
#define CH_CFG_SYSTEM_TICK_HOOK() {                                       \
  /* System tick event code here.*/                                       \
}

/**
 * @brief    System halt hook.
 * @details This hook is invoked in case to a system halting error before
 *           the system is halted.
 */
#define CH_CFG_SYSTEM_HALT_HOOK(reason) {                                 \
  /* System halt code here.*/                                             \
}

/**
 * @brief    Trace hook.
 * @details This hook is invoked each time a new record is written in the
 *           trace buffer.
 */
#define CH_CFG_TRACE_HOOK(tep) {                                          \
  /* Trace code here.*/                                                   \
}

/** @} */

/*===========================================================================*/
/* Port-specific settings (override port settings defaulted in chcore.h).    */
/*===========================================================================*/

#endif  /* CHCONF_H */

/** @} */
```

B.5 f4boot/mcuconf.h

```
/*
    ChibiOS - Copyright (C) 2006..2016 Giovanni Di Sirio

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

        http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
*/

#ifndef MCUCONF_H
#define MCUCONF_H

/*
 * STM32F4xx drivers configuration.
 * The following settings override the default settings present in
 * the various device driver implementation headers.
 * Note that the settings for each driver only have effect if the whole
 * driver is enabled in halconf.h.
 *
 * IRQ priorities:
 * 15...0        Lowest...Highest.
 *
 * DMA priorities:
 * 0...3         Lowest...Highest.
 */

#define STM32F4xx_MCUCONF

/*
 * HAL driver system settings.
```

```
 */
#define STM32_NO_INIT                       FALSE
#define STM32_HSI_ENABLED                   TRUE
#define STM32_LSI_ENABLED                   TRUE
#define STM32_HSE_ENABLED                   TRUE
#define STM32_LSE_ENABLED                   TRUE
#define STM32_CLOCK48_REQUIRED              TRUE
#define STM32_SW                            STM32_SW_PLL
#define STM32_PLLSRC                        STM32_PLLSRC_HSE
#define STM32_PLLM_VALUE                    8
#define STM32_PLLN_VALUE                    336
#define STM32_PLLP_VALUE                    2
#define STM32_PLLQ_VALUE                    7
#define STM32_HPRE                          STM32_HPRE_DIV1
#define STM32_PPRE1                         STM32_PPRE1_DIV4
#define STM32_PPRE2                         STM32_PPRE2_DIV2
#define STM32_RTCSEL                        STM32_RTCSEL_LSE
#define STM32_RTCPRE_VALUE                  8
#define STM32_MCO1SEL                       STM32_MCO1SEL_HSI
#define STM32_MCO1PRE                       STM32_MCO1PRE_DIV1
#define STM32_MCO2SEL                       STM32_MCO2SEL_SYSCLK
#define STM32_MCO2PRE                       STM32_MCO2PRE_DIV5
#define STM32_I2SSRC                        STM32_I2SSRC_CKIN
#define STM32_PLLI2SN_VALUE                 192
#define STM32_PLLI2SR_VALUE                 5
#define STM32_PVD_ENABLE                    FALSE
#define STM32_PLS                           STM32_PLS_LEV0
#define STM32_BKPRAM_ENABLE                 TRUE

/*
 * ADC driver system settings.
 */
#define STM32_ADC_ADCPRE                    ADC_CCR_ADCPRE_DIV4
#define STM32_ADC_USE_ADC1                  FALSE
#define STM32_ADC_USE_ADC2                  FALSE
#define STM32_ADC_USE_ADC3                  FALSE
#define STM32_ADC_ADC1_DMA_STREAM           STM32_DMA_STREAM_ID(2, 4)
#define STM32_ADC_ADC2_DMA_STREAM           STM32_DMA_STREAM_ID(2, 2)
#define STM32_ADC_ADC3_DMA_STREAM           STM32_DMA_STREAM_ID(2, 1)
#define STM32_ADC_ADC1_DMA_PRIORITY         2
#define STM32_ADC_ADC2_DMA_PRIORITY         2
#define STM32_ADC_ADC3_DMA_PRIORITY         2
#define STM32_ADC_IRQ_PRIORITY              6
#define STM32_ADC_ADC1_DMA_IRQ_PRIORITY     6
#define STM32_ADC_ADC2_DMA_IRQ_PRIORITY     6
#define STM32_ADC_ADC3_DMA_IRQ_PRIORITY     6

/*
 * CAN driver system settings.
 */
#define STM32_CAN_USE_CAN1                  TRUE
#define STM32_CAN_USE_CAN2                  FALSE
#define STM32_CAN_CAN1_IRQ_PRIORITY         11
#define STM32_CAN_CAN2_IRQ_PRIORITY         11

/*
 * DAC driver system settings.
 */
#define STM32_DAC_DUAL_MODE                 FALSE
#define STM32_DAC_USE_DAC1_CH1              FALSE
#define STM32_DAC_USE_DAC1_CH2              FALSE
#define STM32_DAC_DAC1_CH1_IRQ_PRIORITY     10
#define STM32_DAC_DAC1_CH2_IRQ_PRIORITY     10
#define STM32_DAC_DAC1_CH1_DMA_PRIORITY     2
#define STM32_DAC_DAC1_CH2_DMA_PRIORITY     2
#define STM32_DAC_DAC1_CH1_DMA_STREAM       STM32_DMA_STREAM_ID(1, 5)
#define STM32_DAC_DAC1_CH2_DMA_STREAM       STM32_DMA_STREAM_ID(1, 6)

/*
 * EXT driver system settings.
 */
#define STM32_EXT_EXTI0_IRQ_PRIORITY        6
#define STM32_EXT_EXTI1_IRQ_PRIORITY        6
#define STM32_EXT_EXTI2_IRQ_PRIORITY        6
#define STM32_EXT_EXTI3_IRQ_PRIORITY        6
#define STM32_EXT_EXTI4_IRQ_PRIORITY        6
#define STM32_EXT_EXTI5_9_IRQ_PRIORITY      6
#define STM32_EXT_EXTI10_15_IRQ_PRIORITY    6
#define STM32_EXT_EXTI16_IRQ_PRIORITY       6
#define STM32_EXT_EXTI17_IRQ_PRIORITY       15
#define STM32_EXT_EXTI18_IRQ_PRIORITY       6
#define STM32_EXT_EXTI19_IRQ_PRIORITY       6
#define STM32_EXT_EXTI20_IRQ_PRIORITY       6
#define STM32_EXT_EXTI21_IRQ_PRIORITY       15
#define STM32_EXT_EXTI22_IRQ_PRIORITY       15
```

```
/*
 * GPT driver system settings.
 */
#define STM32_GPT_USE_TIM1                  FALSE
#define STM32_GPT_USE_TIM2                  FALSE
#define STM32_GPT_USE_TIM3                  FALSE
#define STM32_GPT_USE_TIM4                  FALSE
#define STM32_GPT_USE_TIM5                  FALSE
#define STM32_GPT_USE_TIM6                  FALSE
#define STM32_GPT_USE_TIM7                  FALSE
#define STM32_GPT_USE_TIM8                  FALSE
#define STM32_GPT_USE_TIM9                  FALSE
#define STM32_GPT_USE_TIM11                 FALSE
#define STM32_GPT_USE_TIM12                 FALSE
#define STM32_GPT_USE_TIM14                 FALSE
#define STM32_GPT_TIM1_IRQ_PRIORITY         7
#define STM32_GPT_TIM2_IRQ_PRIORITY         7
#define STM32_GPT_TIM3_IRQ_PRIORITY         7
#define STM32_GPT_TIM4_IRQ_PRIORITY         7
#define STM32_GPT_TIM5_IRQ_PRIORITY         7
#define STM32_GPT_TIM6_IRQ_PRIORITY         7
#define STM32_GPT_TIM7_IRQ_PRIORITY         7
#define STM32_GPT_TIM8_IRQ_PRIORITY         7
#define STM32_GPT_TIM9_IRQ_PRIORITY         7
#define STM32_GPT_TIM11_IRQ_PRIORITY        7
#define STM32_GPT_TIM12_IRQ_PRIORITY        7
#define STM32_GPT_TIM14_IRQ_PRIORITY        7

/*
 * I2C driver system settings.
 */
#define STM32_I2C_USE_I2C1                  FALSE
#define STM32_I2C_USE_I2C2                  FALSE
#define STM32_I2C_USE_I2C3                  FALSE
#define STM32_I2C_BUSY_TIMEOUT              50
#define STM32_I2C_I2C1_RX_DMA_STREAM        STM32_DMA_STREAM_ID(1, 0)
#define STM32_I2C_I2C1_TX_DMA_STREAM        STM32_DMA_STREAM_ID(1, 6)
#define STM32_I2C_I2C2_RX_DMA_STREAM        STM32_DMA_STREAM_ID(1, 2)
#define STM32_I2C_I2C2_TX_DMA_STREAM        STM32_DMA_STREAM_ID(1, 7)
#define STM32_I2C_I2C3_RX_DMA_STREAM        STM32_DMA_STREAM_ID(1, 2)
#define STM32_I2C_I2C3_TX_DMA_STREAM        STM32_DMA_STREAM_ID(1, 4)
#define STM32_I2C_I2C1_IRQ_PRIORITY         5
#define STM32_I2C_I2C2_IRQ_PRIORITY         5
#define STM32_I2C_I2C3_IRQ_PRIORITY         5
#define STM32_I2C_I2C1_DMA_PRIORITY         3
#define STM32_I2C_I2C2_DMA_PRIORITY         3
#define STM32_I2C_I2C3_DMA_PRIORITY         3
#define STM32_I2C_DMA_ERROR_HOOK(i2cp)      osalSysHalt("DMA_failure")

/*
 * I2S driver system settings.
 */
#define STM32_I2S_USE_SPI2                  FALSE
#define STM32_I2S_USE_SPI3                  FALSE
#define STM32_I2S_SPI2_IRQ_PRIORITY         10
#define STM32_I2S_SPI3_IRQ_PRIORITY         10
#define STM32_I2S_SPI2_DMA_PRIORITY         1
#define STM32_I2S_SPI3_DMA_PRIORITY         1
#define STM32_I2S_SPI2_RX_DMA_STREAM        STM32_DMA_STREAM_ID(1, 3)
#define STM32_I2S_SPI2_TX_DMA_STREAM        STM32_DMA_STREAM_ID(1, 4)
#define STM32_I2S_SPI3_RX_DMA_STREAM        STM32_DMA_STREAM_ID(1, 0)
#define STM32_I2S_SPI3_TX_DMA_STREAM        STM32_DMA_STREAM_ID(1, 7)
#define STM32_I2S_DMA_ERROR_HOOK(i2sp)      osalSysHalt("DMA_failure")

/*
 * ICU driver system settings.
 */
#define STM32_ICU_USE_TIM1                  FALSE
#define STM32_ICU_USE_TIM2                  FALSE
#define STM32_ICU_USE_TIM3                  FALSE
#define STM32_ICU_USE_TIM4                  FALSE
#define STM32_ICU_USE_TIM5                  FALSE
#define STM32_ICU_USE_TIM8                  FALSE
#define STM32_ICU_USE_TIM9                  FALSE
#define STM32_ICU_TIM1_IRQ_PRIORITY         7
#define STM32_ICU_TIM2_IRQ_PRIORITY         7
#define STM32_ICU_TIM3_IRQ_PRIORITY         7
#define STM32_ICU_TIM4_IRQ_PRIORITY         7
#define STM32_ICU_TIM5_IRQ_PRIORITY         7
#define STM32_ICU_TIM8_IRQ_PRIORITY         7
#define STM32_ICU_TIM9_IRQ_PRIORITY         7

/*
 * MAC driver system settings.
 */
#define STM32_MAC_TRANSMIT_BUFFERS          2
```

```
#define STM32_MAC_RECEIVE_BUFFERS              4
#define STM32_MAC_BUFFERS_SIZE                 1522
#define STM32_MAC_PHY_TIMEOUT                  100
#define STM32_MAC_ETH1_CHANGE_PHY_STATE        TRUE
#define STM32_MAC_ETH1_IRQ_PRIORITY            13
#define STM32_MAC_IP_CHECKSUM_OFFLOAD          1

/*
 * PWM driver system settings.
 */
#define STM32_PWM_USE_ADVANCED                 FALSE
#define STM32_PWM_USE_TIM1                     FALSE
#define STM32_PWM_USE_TIM2                     FALSE
#define STM32_PWM_USE_TIM3                     FALSE
#define STM32_PWM_USE_TIM4                     FALSE
#define STM32_PWM_USE_TIM5                     FALSE
#define STM32_PWM_USE_TIM8                     FALSE
#define STM32_PWM_USE_TIM9                     FALSE
#define STM32_PWM_TIM1_IRQ_PRIORITY            7
#define STM32_PWM_TIM2_IRQ_PRIORITY            7
#define STM32_PWM_TIM3_IRQ_PRIORITY            7
#define STM32_PWM_TIM4_IRQ_PRIORITY            7
#define STM32_PWM_TIM5_IRQ_PRIORITY            7
#define STM32_PWM_TIM8_IRQ_PRIORITY            7
#define STM32_PWM_TIM9_IRQ_PRIORITY            7

/*
 * SDC driver system settings.
 */
#define STM32_SDC_SDIO_DMA_PRIORITY            3
#define STM32_SDC_SDIO_IRQ_PRIORITY            9
#define STM32_SDC_WRITE_TIMEOUT_MS             1000
#define STM32_SDC_READ_TIMEOUT_MS              1000
#define STM32_SDC_CLOCK_ACTIVATION_DELAY       10
#define STM32_SDC_SDIO_UNALIGNED_SUPPORT       TRUE
#define STM32_SDC_SDIO_DMA_STREAM              STM32_DMA_STREAM_ID(2, 3)

/*
 * SERIAL driver system settings.
 */
#define STM32_SERIAL_USE_USART1                FALSE
#define STM32_SERIAL_USE_USART2                FALSE
#define STM32_SERIAL_USE_USART3                TRUE
#define STM32_SERIAL_USE_UART4                 FALSE
#define STM32_SERIAL_USE_UART5                 FALSE
#define STM32_SERIAL_USE_USART6                FALSE
#define STM32_SERIAL_USART1_PRIORITY           12
#define STM32_SERIAL_USART2_PRIORITY           12
#define STM32_SERIAL_USART3_PRIORITY           12
#define STM32_SERIAL_UART4_PRIORITY            12
#define STM32_SERIAL_UART5_PRIORITY            12
#define STM32_SERIAL_USART6_PRIORITY           12

/*
 * SPI driver system settings.
 */
#define STM32_SPI_USE_SPI1                     FALSE
#define STM32_SPI_USE_SPI2                     FALSE
#define STM32_SPI_USE_SPI3                     FALSE
#define STM32_SPI_SPI1_RX_DMA_STREAM           STM32_DMA_STREAM_ID(2, 0)
#define STM32_SPI_SPI1_TX_DMA_STREAM           STM32_DMA_STREAM_ID(2, 3)
#define STM32_SPI_SPI2_RX_DMA_STREAM           STM32_DMA_STREAM_ID(1, 3)
#define STM32_SPI_SPI2_TX_DMA_STREAM           STM32_DMA_STREAM_ID(1, 4)
#define STM32_SPI_SPI3_RX_DMA_STREAM           STM32_DMA_STREAM_ID(1, 0)
#define STM32_SPI_SPI3_TX_DMA_STREAM           STM32_DMA_STREAM_ID(1, 7)
#define STM32_SPI_SPI1_DMA_PRIORITY            1
#define STM32_SPI_SPI2_DMA_PRIORITY            1
#define STM32_SPI_SPI3_DMA_PRIORITY            1
#define STM32_SPI_SPI1_IRQ_PRIORITY            10
#define STM32_SPI_SPI2_IRQ_PRIORITY            10
#define STM32_SPI_SPI3_IRQ_PRIORITY            10
#define STM32_SPI_DMA_ERROR_HOOK(spip)         osalSysHalt("DMA failure")

/*
 * ST driver system settings.
 */
#define STM32_ST_IRQ_PRIORITY                  8
#define STM32_ST_USE_TIMER                     2

/*
 * UART driver system settings.
 */
#define STM32_UART_USE_USART1                  FALSE
#define STM32_UART_USE_USART2                  FALSE
#define STM32_UART_USE_USART3                  FALSE
#define STM32_UART_USE_UART4                   FALSE
```

```
#define STM32_UART_USE_UART5                FALSE
#define STM32_UART_USE_USART6               FALSE
#define STM32_UART_USART1_RX_DMA_STREAM     STM32_DMA_STREAM_ID(2, 5)
#define STM32_UART_USART1_TX_DMA_STREAM     STM32_DMA_STREAM_ID(2, 7)
#define STM32_UART_USART2_RX_DMA_STREAM     STM32_DMA_STREAM_ID(1, 5)
#define STM32_UART_USART2_TX_DMA_STREAM     STM32_DMA_STREAM_ID(1, 6)
#define STM32_UART_USART3_RX_DMA_STREAM     STM32_DMA_STREAM_ID(1, 1)
#define STM32_UART_USART3_TX_DMA_STREAM     STM32_DMA_STREAM_ID(1, 3)
#define STM32_UART_UART4_RX_DMA_STREAM      STM32_DMA_STREAM_ID(1, 2)
#define STM32_UART_UART4_TX_DMA_STREAM      STM32_DMA_STREAM_ID(1, 4)
#define STM32_UART_UART5_RX_DMA_STREAM      STM32_DMA_STREAM_ID(1, 0)
#define STM32_UART_UART5_TX_DMA_STREAM      STM32_DMA_STREAM_ID(1, 7)
#define STM32_UART_USART6_RX_DMA_STREAM     STM32_DMA_STREAM_ID(2, 2)
#define STM32_UART_USART6_TX_DMA_STREAM     STM32_DMA_STREAM_ID(2, 7)
#define STM32_UART_USART1_IRQ_PRIORITY      12
#define STM32_UART_USART2_IRQ_PRIORITY      12
#define STM32_UART_USART3_IRQ_PRIORITY      12
#define STM32_UART_UART4_IRQ_PRIORITY       12
#define STM32_UART_UART5_IRQ_PRIORITY       12
#define STM32_UART_USART6_IRQ_PRIORITY      12
#define STM32_UART_USART1_DMA_PRIORITY      0
#define STM32_UART_USART2_DMA_PRIORITY      0
#define STM32_UART_USART3_DMA_PRIORITY      0
#define STM32_UART_UART4_DMA_PRIORITY       0
#define STM32_UART_UART5_DMA_PRIORITY       0
#define STM32_UART_USART6_DMA_PRIORITY      0
#define STM32_UART_DMA_ERROR_HOOK(uartp)    osalSysHalt("DMA failure")

/*
 * USB driver system settings.
 */
#define STM32_USB_USE_OTG1                  FALSE
#define STM32_USB_USE_OTG2                  FALSE
#define STM32_USB_OTG1_IRQ_PRIORITY         14
#define STM32_USB_OTG2_IRQ_PRIORITY         14
#define STM32_USB_OTG1_RX_FIFO_SIZE         512
#define STM32_USB_OTG2_RX_FIFO_SIZE         1024
#define STM32_USB_OTG_THREAD_PRIO           LOWPRIO
#define STM32_USB_OTG_THREAD_STACK_SIZE     128
#define STM32_USB_OTGFIFO_FILL_BASEPRI      0

/*
 * WDG driver system settings.
 */
#define STM32_WDG_USE_IWDG                  FALSE

/*
 * header for community drivers.
 */
#include "mcuconf_community.h"

#endif /* MCUCONF_H */
```

B.6 f4boot/halconf.h

```
/*
    ChibiOS - Copyright (C) 2006..2016 Giovanni Di Sirio

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

        http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
*/

/**
 * @file    templates/halconf.h
 * @brief   HAL configuration header.
 * @details HAL configuration file, this file allows to enable or disable the
 *          various device drivers from your application. You may also use
 *          this file in order to override the device drivers default settings.
 *
 * @addtogroup HAL_CONF
 * @{
 */

#ifndef HALCONF_H
```

```
#define HALCONF_H

#include "mcuconf.h"

/**
 * @brief   Enables the TM subsystem.
 */
#if !defined(HAL_USE_TM) || defined(__DOXYGEN__)
#define HAL_USE_TM                  TRUE
#endif

/**
 * @brief   Enables the PAL subsystem.
 */
#if !defined(HAL_USE_PAL) || defined(__DOXYGEN__)
#define HAL_USE_PAL                 TRUE
#endif

/**
 * @brief   Enables the ADC subsystem.
 */
#if !defined(HAL_USE_ADC) || defined(__DOXYGEN__)
#define HAL_USE_ADC                 FALSE
#endif

/**
 * @brief   Enables the CAN subsystem.
 */
#if !defined(HAL_USE_CAN) || defined(__DOXYGEN__)
#define HAL_USE_CAN                 FALSE
#endif

/**
 * @brief   Enables the DAC subsystem.
 */
#if !defined(HAL_USE_DAC) || defined(__DOXYGEN__)
#define HAL_USE_DAC                 FALSE
#endif

/**
 * @brief   Enables the EXT subsystem.
 */
#if !defined(HAL_USE_EXT) || defined(__DOXYGEN__)
#define HAL_USE_EXT                 FALSE
#endif

/**
 * @brief   Enables the GPT subsystem.
 */
#if !defined(HAL_USE_GPT) || defined(__DOXYGEN__)
#define HAL_USE_GPT                 FALSE
#endif

/**
 * @brief   Enables the I2C subsystem.
 */
#if !defined(HAL_USE_I2C) || defined(__DOXYGEN__)
#define HAL_USE_I2C                 FALSE
#endif

/**
 * @brief   Enables the I2S subsystem.
 */
#if !defined(HAL_USE_I2S) || defined(__DOXYGEN__)
#define HAL_USE_I2S                 FALSE
#endif

/**
 * @brief   Enables the ICU subsystem.
 */
#if !defined(HAL_USE_ICU) || defined(__DOXYGEN__)
#define HAL_USE_ICU                 FALSE
#endif

/**
 * @brief   Enables the MAC subsystem.
 */
#if !defined(HAL_USE_MAC) || defined(__DOXYGEN__)
#define HAL_USE_MAC                 TRUE
#endif

/**
 * @brief   Enables the MMC_SPI subsystem.
 */
#if !defined(HAL_USE_MMC_SPI) || defined(__DOXYGEN__)
#define HAL_USE_MMC_SPI             FALSE
```

```
#endif

/**
 * @brief   Enables the PWM subsystem.
 */
#if !defined(HAL_USE_PWM) || defined(__DOXYGEN__)
#define HAL_USE_PWM                 FALSE
#endif

/**
 * @brief   Enables the QSPI subsystem.
 */
#if !defined(HAL_USE_QSPI) || defined(__DOXYGEN__)
#define HAL_USE_QSPI                FALSE
#endif

/**
 * @brief   Enables the RTC subsystem.
 */
#if !defined(HAL_USE_RTC) || defined(__DOXYGEN__)
#define HAL_USE_RTC                 TRUE
#endif

/**
 * @brief   Enables the SDC subsystem.
 */
#if !defined(HAL_USE_SDC) || defined(__DOXYGEN__)
#define HAL_USE_SDC                 FALSE
#endif

/**
 * @brief   Enables the SERIAL subsystem.
 */
#if !defined(HAL_USE_SERIAL) || defined(__DOXYGEN__)
#define HAL_USE_SERIAL              TRUE
#endif

/**
 * @brief   Enables the SERIAL over USB subsystem.
 */
#if !defined(HAL_USE_SERIAL_USB) || defined(__DOXYGEN__)
#define HAL_USE_SERIAL_USB          FALSE
#endif

/**
 * @brief   Enables the SPI subsystem.
 */
#if !defined(HAL_USE_SPI) || defined(__DOXYGEN__)
#define HAL_USE_SPI                 FALSE
#endif

/**
 * @brief   Enables the UART subsystem.
 */
#if !defined(HAL_USE_UART) || defined(__DOXYGEN__)
#define HAL_USE_UART                FALSE
#endif

/**
 * @brief   Enables the USB subsystem.
 */
#if !defined(HAL_USE_USB) || defined(__DOXYGEN__)
#define HAL_USE_USB                 FALSE
#endif

/**
 * @brief   Enables the WDG subsystem.
 */
#if !defined(HAL_USE_WDG) || defined(__DOXYGEN__)
#define HAL_USE_WDG                 FALSE
#endif

/*===========================================================================*/
/* ADC driver related settings.                                              */
/*===========================================================================*/

/**
 * @brief   Enables synchronous APIs.
 * @note    Disabling this option saves both code and data space.
 */
#if !defined(ADC_USE_WAIT) || defined(__DOXYGEN__)
#define ADC_USE_WAIT                TRUE
#endif

/**
 * @brief   Enables the @p adcAcquireBus() and @p adcReleaseBus() APIs.
```

```
 * @note      Disabling this option saves both code and data space.
 */
#if !defined(ADC_USE_MUTUAL_EXCLUSION) || defined(__DOXYGEN__)
#define ADC_USE_MUTUAL_EXCLUSION      TRUE
#endif

/*===========================================================================*/
/* CAN driver related settings.                                              */
/*===========================================================================*/

/**
 * @brief   Sleep mode related APIs inclusion switch.
 */
#if !defined(CAN_USE_SLEEP_MODE) || defined(__DOXYGEN__)
#define CAN_USE_SLEEP_MODE            TRUE
#endif

/*===========================================================================*/
/* I2C driver related settings.                                              */
/*===========================================================================*/

/**
 * @brief   Enables the mutual exclusion APIs on the I2C bus.
 */
#if !defined(I2C_USE_MUTUAL_EXCLUSION) || defined(__DOXYGEN__)
#define I2C_USE_MUTUAL_EXCLUSION      TRUE
#endif

/*===========================================================================*/
/* MAC driver related settings.                                              */
/*===========================================================================*/

/**
 * @brief   Enables an event sources for incoming packets.
 */
#if !defined(MAC_USE_ZERO_COPY) || defined(__DOXYGEN__)
#define MAC_USE_ZERO_COPY             TRUE
#endif

/**
 * @brief   Enables an event sources for incoming packets.
 */
#if !defined(MAC_USE_EVENTS) || defined(__DOXYGEN__)
#define MAC_USE_EVENTS                TRUE
#endif

/*===========================================================================*/
/* MMC_SPI driver related settings.                                          */
/*===========================================================================*/

/**
 * @brief   Delays insertions.
 * @details If enabled this options inserts delays into the MMC waiting
 *          routines releasing some extra CPU time for the threads with
 *          lower priority, this may slow down the driver a bit however.
 *          This option is recommended also if the SPI driver does not
 *          use a DMA channel and heavily loads the CPU.
 */
#if !defined(MMC_NICE_WAITING) || defined(__DOXYGEN__)
#define MMC_NICE_WAITING              TRUE
#endif

/*===========================================================================*/
/* SDC driver related settings.                                              */
/*===========================================================================*/

/**
 * @brief   Number of initialization attempts before rejecting the card.
 * @note    Attempts are performed at 10mS intervals.
 */
#if !defined(SDC_INIT_RETRY) || defined(__DOXYGEN__)
#define SDC_INIT_RETRY                100
#endif

/**
 * @brief   Include support for MMC cards.
 * @note    MMC support is not yet implemented so this option must be kept
 *          at @p FALSE.
 */
#if !defined(SDC_MMC_SUPPORT) || defined(__DOXYGEN__)
#define SDC_MMC_SUPPORT               FALSE
#endif

/**
 * @brief   Delays insertions.
 * @details If enabled this options inserts delays into the MMC waiting
```

```
 *            routines releasing some extra CPU time for the threads with
 *            lower priority, this may slow down the driver a bit however.
 */
#if !defined(SDC_NICE_WAITING) || defined(__DOXYGEN__)
#define SDC_NICE_WAITING                TRUE
#endif

/*===========================================================================*/
/* SERIAL driver related settings.                                           */
/*===========================================================================*/

/**
 * @brief   Default bit rate.
 * @details Configuration parameter, this is the baud rate selected for the
 *          default configuration.
 */
#if !defined(SERIAL_DEFAULT_BITRATE) || defined(__DOXYGEN__)
#define SERIAL_DEFAULT_BITRATE          38400
#endif

/**
 * @brief   Serial buffers size.
 * @details Configuration parameter, you can change the depth of the queue
 *          buffers depending on the requirements of your application.
 * @note    The default is 16 bytes for both the transmission and receive
 *          buffers.
 */
#if !defined(SERIAL_BUFFERS_SIZE) || defined(__DOXYGEN__)
#define SERIAL_BUFFERS_SIZE             16
#endif

/*===========================================================================*/
/* SERIAL_USB driver related setting.                                        */
/*===========================================================================*/

/**
 * @brief   Serial over USB buffers size.
 * @details Configuration parameter, the buffer size must be a multiple of
 *          the USB data endpoint maximum packet size.
 * @note    The default is 256 bytes for both the transmission and receive
 *          buffers.
 */
#if !defined(SERIAL_USB_BUFFERS_SIZE) || defined(__DOXYGEN__)
#define SERIAL_USB_BUFFERS_SIZE         256
#endif

/**
 * @brief   Serial over USB number of buffers.
 * @note    The default is 2 buffers.
 */
#if !defined(SERIAL_USB_BUFFERS_NUMBER) || defined(__DOXYGEN__)
#define SERIAL_USB_BUFFERS_NUMBER       2
#endif

/*===========================================================================*/
/* SPI driver related settings.                                              */
/*===========================================================================*/

/**
 * @brief   Enables synchronous APIs.
 * @note    Disabling this option saves both code and data space.
 */
#if !defined(SPI_USE_WAIT) || defined(__DOXYGEN__)
#define SPI_USE_WAIT                    TRUE
#endif

/**
 * @brief   Enables the @p spiAcquireBus() and @p spiReleaseBus() APIs.
 * @note    Disabling this option saves both code and data space.
 */
#if !defined(SPI_USE_MUTUAL_EXCLUSION) || defined(__DOXYGEN__)
#define SPI_USE_MUTUAL_EXCLUSION        TRUE
#endif

/*===========================================================================*/
/* UART driver related settings.                                             */
/*===========================================================================*/

/**
 * @brief   Enables synchronous APIs.
 * @note    Disabling this option saves both code and data space.
 */
#if !defined(UART_USE_WAIT) || defined(__DOXYGEN__)
#define UART_USE_WAIT                   FALSE
#endif
```

```
/**
 * @brief   Enables the @p uartAcquireBus() and @p uartReleaseBus() APIs.
 * @note    Disabling this option saves both code and data space.
 */
#if !defined(UART_USE_MUTUAL_EXCLUSION) || defined(__DOXYGEN__)
#define UART_USE_MUTUAL_EXCLUSION     FALSE
#endif

/*===========================================================================*/
/* USB driver related settings.                                              */
/*===========================================================================*/

/**
 * @brief   Enables synchronous APIs.
 * @note    Disabling this option saves both code and data space.
 */
#if !defined(USB_USE_WAIT) || defined(__DOXYGEN__)
#define USB_USE_WAIT                  FALSE
#endif


/*===========================================================================*/
/* Community drivers's includes                                              */
/*===========================================================================*/

#include "halconf_community.h"

#endif /* HALCONF_H */

/** @} */
```

B.7 f4boot/tftp.h

```
/*******************************************************************
 *
 * @file tftp_server.c
 *
 * @author   Logan Gunthorpe <logang@deltatee.com>
 *
 * @brief    Trivial File Transfer Protocol (RFC 1350)
 *
 * Copyright (c) Deltatee Enterprises Ltd. 2013
 * All rights reserved.
 *
 *******************************************************************/

/*
 * Redistribution and use in source and binary forms, with or without
 * modification,are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 *    this list of conditions and the following disclaimer in the documentation
 *    and/or other materials provided with the distribution.
 * 3. The name of the author may not be used to endorse or promote products
 *    derived from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
 * EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
 * TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * Author: Logan Gunthorpe <logang@deltatee.com>
 *
 */

#include "tftp.h"

#include <lwip/opt.h>
#include <lwip/debug.h>
#include <lwip/api.h>

#ifndef TFTP_DEBUG
#define TFTP_DEBUG LWIP_DBG_ON
#endif
```

```
#ifndef TFTP_PORT
#define TFTP_PORT 69
#endif

#ifndef TFTP_TIMEOUT_MSECS
#define TFTP_TIMEOUT_MSECS     10000
#endif

#ifndef TFTP_MAX_RETRIES
#define TFTP_MAX_RETRIES       5
#endif

#define RRQ    1
#define WRQ    2
#define DATA   3
#define ACK    4
#define ERROR  5

#define ERROR_FILE_NOT_FOUND      1
#define ERROR_ACCESS_VIOLATION    2
#define ERROR_DISK_FULL           3
#define ERROR_ILLEGAL_OPERATION   4
#define ERROR_UNKNOWN_TRFR_ID     5
#define ERROR_FILE_EXISTS         6
#define ERROR_NO_SUCH_USER        7

#include <string.h>


struct tftp_state {
  const struct tftp_context *ctx;
  struct tftp_handle *handle;
  int blknum;
  struct netbuf *last_data;
  int last_pkt;
  int retries;
  struct netconn *conn;
};


static void close_session(struct tftp_state *ts)
{
  if (ts->conn) netconn_disconnect(ts->conn);
  if (ts->last_data != NULL) {
    netbuf_delete(ts->last_data);
    ts->last_data = NULL;
  }

  if (ts->handle) {
    ts->ctx->close(ts->handle);
    ts->handle = NULL;
    LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE, ("tftp: closing\n"));
  }
}

static void send_error(struct tftp_state *ts, int code, const char *str) {
  int str_length = strlen(str);

  struct netbuf *pkt = netbuf_new();
  u16_t *payload = (u16_t *) netbuf_alloc(pkt, 4 + str_length + 1);

  payload[0] = htons(ERROR);
  payload[1] = htons(code);
  memcpy(&payload[2], str, str_length + 1);

  netconn_send(ts->conn, pkt);
  netbuf_delete(pkt);
}

static void send_wrq(struct tftp_state *ts, const char *remote, const char *mode) {
  size_t remote_len = strlen(remote);
  size_t mode_len = strlen(mode);

  struct netbuf *pkt = netbuf_new();
  uint16_t *payload = (uint16_t *) netbuf_alloc(pkt, 4 + remote_len + mode_len);

  payload[0] = htons(WRQ);
  char * strings_start = (char *)&payload[1];
  strcpy(strings_start, remote);
  strcpy(strings_start + remote_len + 1, mode);

  netconn_send(ts->conn, pkt);
  netbuf_delete(pkt);
}
```

```
static void send_rrq(struct tftp_state *ts, const char *remote, const char *mode) {
  size_t remote_len = strlen(remote);
  size_t mode_len = strlen(mode);

  struct netbuf *pkt = netbuf_new();
  uint16_t *payload = (uint16_t *) netbuf_alloc(pkt, 4 + remote_len + mode_len);

  payload[0] = htons(RRQ);
  char * strings_start = (char *)&payload[1];
  strcpy(strings_start, remote);
  strcpy(strings_start + remote_len + 1, mode);

  netconn_send(ts->conn, pkt);
  netbuf_delete(pkt);
}

static void send_ack(struct tftp_state *ts, int blknum) {
  struct netbuf *pkt = netbuf_new();
  u16_t *payload = (u16_t *) netbuf_alloc(pkt, 4);

  payload[0] = htons(ACK);
  payload[1] = htons(blknum);
  netconn_send(ts->conn, pkt);
  netbuf_delete(pkt);
}

static void resend_data(struct tftp_state *ts) {
  netconn_send(ts->conn, ts->last_data);
}

static void send_data(struct tftp_state *ts) {
  ts->last_data = netbuf_new();

  u16_t *payload = (u16_t *) netbuf_alloc(ts->last_data, 4+512);

  payload[0] = htons(DATA);
  payload[1] = htons(ts->blknum);

  int len = ts->ctx->read(ts->handle, &payload[2], 512);

  if (len < 0) {
    send_error(ts, ERROR_ACCESS_VIOLATION,
               "Error occured while reading the file.");
    netbuf_free(ts->last_data);
    ts->last_data = NULL;
    close_session(ts);
    return;
  }

  if (len != 512) {
    ts->ctx->close(ts->handle);
    ts->handle = NULL;
  }

  pbuf_realloc(ts->last_data->p, 4 + len);
  resend_data(ts);
}

/**
 * Process an incoming UDP packet. This is pretty much zero-copy for the
 * incoming packet. Doesn't touch anything outside of it's packet and state
 * structure so it should work in several threads at once or in a thread pool if wanted.
 *
 * @param conn     the netconn where the we should send packets back.
 * @param pktbuf   the netbuf containing the incoming packet.
 * @param ts       pointer to the current tftp state.
 */
static void tftp_recv_packet(struct netbuf **pktbuf, struct netconn *conn,
    struct tftp_state *ts) {
  char *buf;
  uint16_t *sbuf;
  uint16_t len;
  int blknum;
  struct ip_addr addr;
  uint16_t port;

  /*
   * Get pointers into the packet to get opcode and length
   * TODO: deal with ptr error
   * TODO: deal with len<2
   */
  netbuf_data(*pktbuf, (void **) &buf, &len);
  sbuf = (uint16_t *) buf;      /* Aliasing memory !!! */

  int opcode = ntohs(sbuf[0]);
```

```
buf[len] = 0;
ts->retries = 0;

switch (opcode) {
case RRQ:
case WRQ:
  /* If we just started a session, don't try a new one until timout or some data */
  if (ts->blknum == 1) break;

  /* Terminate the last session and make a new one */
  close_session(ts);
  ts->blknum = 1;


  ts->conn = conn;
  if(ERR_OK != netconn_connect(ts->conn, netbuf_fromaddr(*pktbuf),
                               netbuf_fromport(*pktbuf))) {
    LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE,
                ("Cannot connect return path"));
  }

  char *filename = &buf[2];
  char *mode = &buf[2];

  /* Run through the buffer to get the next element after the name */
  while (*mode)
    mode++;
  mode++;

  /* Terminate the packet handling if we've walked off the buffer */
  /* TODO: Manage having filename/mode split between pbufs */
  if ((mode - buf) >= len) break;

  ts->handle = ts->ctx->open(filename, mode, opcode == WRQ);

  if (!ts->handle) {
    send_error(ts, ERROR_FILE_NOT_FOUND,
               "Unable to open requested file.");
    close_session(ts);
    break;
  }

  LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE,
              ("tftp: %s request from ",
               (opcode == WRQ) ? "write" : "read"));
  netconn_peer(conn, &addr, &port);
  ip_addr_debug_print(TFTP_DEBUG | LWIP_DBG_STATE, &addr);
  LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE,
              (" for '%s' mode '%s'\n", filename, mode));

  if (opcode == WRQ) {
    send_ack(ts, 0);
  }
  else {
    send_data(ts);
  }
  break;

case DATA:
  /* Bail out if we don't have an active session going */
  if (ts->handle == NULL) break;

  /* Check that this is a new block so we don't write retransmits */
  blknum = ntohs(sbuf[1]);

  if ((1 == blknum) && (NULL == ts->conn)) {
    ts->conn = conn;
    if(ERR_OK != netconn_connect(ts->conn, netbuf_fromaddr(*pktbuf),
                                 netbuf_fromport(*pktbuf))) {
      LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE,
                  ("Cannot connect return path"));
    }
  }

  if (blknum != ts->blknum) {
    /* If we get the previous block again, it means our ack was lost */
    if (blknum == ts->blknum - 1) {
      send_ack(ts, blknum);
    }
    break;
  }

  int ret = ts->ctx->write(ts->handle, buf + 4, len-4);

  if (ret < 0) {
    send_error(ts, ERROR_ACCESS_VIOLATION,
```

```
                "error writing file");
      close_session(ts);
    }
    else {
      ts->blknum++;
      send_ack(ts, blknum);
    }

    if (len < 512+4)
      close_session(ts);
    break;

  case ACK:
    /* Skip the packet if we get an ack for the wrong block */
    blknum = ntohs(sbuf[1]);
    if (blknum != ts->blknum) break;

    if (0 == blknum) {
      if(ERR_OK != netconn_connect(ts->conn, netbuf_fromaddr(*pktbuf),
                                   netbuf_fromport(*pktbuf))) {
        LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE,
                    ("Cannot connect return path"));
      }
    }

    if (ts->last_data != NULL) {
      netbuf_delete(ts->last_data);
      ts->last_data = NULL;
    }

    if (ts->handle) {
      ts->blknum++;
      send_data(ts);
    }
    else {
      close_session(ts);
    }

    break;
  }
  netbuf_delete(*pktbuf);
}

/*
 * Exported Functions
 */


/**
 * Serve local files and/or memory over tftp. This can only manage one
 * connection at a time. Ideally it should work without threading issues.
 *
 * @param p file handling structure.
 */
THD_FUNCTION(tftp_server, p)
{
  struct netconn *conn;
  struct netbuf *pktbuf;
  err_t err;
  struct tftp_state tftp_state;

  chRegSetThreadName("tftp_server");

  /* Create new UDP connection handle */
  conn = netconn_new(NETCONN_UDP);
  LWIP_ERROR("tftp_server: invalid conn", (conn != NULL),
             chThdExit(MSG_RESET);
           );

  /* Bind connection to the TFTP port with default IP */
  netconn_bind(conn, NULL, TFTP_PORT);

  /* Setup the TFTP server state */
  tftp_state.handle = NULL;
  tftp_state.ctx = ((struct tftp_context *) p);
  tftp_state.last_data = NULL;
  tftp_state.conn = NULL;

  /* Set the reciever timeout so we can run retries properly. */
  netconn_set_recvtimeout(conn, TFTP_TIMEOUT_MSECS);

  chThdSetPriority(TFTP_THREAD_PRIORITY);
  while (!chThdShouldTerminateX()) {
    err = netconn_recv(conn, &pktbuf);

    /* Handle the new packet through the state machine. */
```

```
    switch (err) {
    case ERR_TIMEOUT:
      if (tftp_state.last_data != NULL &&
          tftp_state.retries < TFTP_MAX_RETRIES) {
        LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE,
                    ("tftp: timeout, retrying\n"));
        resend_data(&tftp_state);
        tftp_state.retries++;
      }
      else {
        LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE, ("tftp: timeout\n"));
        close_session(&tftp_state);
      }
      break;
    case ERR_OK:
      tftp_recv_packet(&pktbuf, conn, &tftp_state);
      break;
    default:
      continue;
    }

  }

  chThdExit(MSG_OK);
}

void tftp_put( struct ip_addr *ip,
    const char *local,
    const char *remote,
    const char *mode,
    const struct tftp_context *ctx
  ) {

  struct netconn *conn;
  struct netbuf *pktbuf;
  err_t err;
  struct tftp_state client_state;

  /* Create new UDP connection handle */
  conn = netconn_new(NETCONN_UDP);
  LWIP_ERROR("tftp_client: invalid conn", (conn != NULL),
             return;
            );

  err = netconn_connect(conn, ip, TFTP_PORT);
  LWIP_ERROR("tftp_client: cannot connect to server", (err == ERR_OK),
             return;
            );

  /* Setup the TFTP server state */
  client_state.handle = ctx->open(local, "r", 1);
  LWIP_ERROR("tftp_client: Cannot open file", (client_state.handle != NULL),
             return;
            );
  client_state.retries = 0;
  client_state.ctx = ((struct tftp_context *) ctx);
  client_state.last_data = NULL;
  client_state.conn = conn;
  client_state.blknum = 0;

  /* Set the reciever timeout so we can run retries properly. */
  netconn_set_recvtimeout(conn, TFTP_TIMEOUT_MSECS);

  send_wrq(&client_state, remote, mode);
  /* Disconnect the netconn because we don't know the remote port */
  netconn_disconnect(conn);

  while(client_state.handle) {
    err = netconn_recv(conn, &pktbuf);

    /* Handle the new packet through the state machine. */
    switch (err) {
    case ERR_TIMEOUT:
      if (client_state.retries < TFTP_MAX_RETRIES) {
        if (client_state.last_data != NULL) {
          LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE,
                      ("tftp: timeout, retrying\n"));
          resend_data(&client_state);
          client_state.retries++;
        } else if (client_state.blknum == 0) {
          LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE,
                      ("tftp: timeout, retrying\n"));
          client_state.conn = conn;
          err = netconn_connect(conn, ip, TFTP_PORT);
          LWIP_ERROR("tftp_client: cannot connect to server",
              (err == ERR_OK), return; );
```

```
          send_wrq(&client_state, remote, mode);
          client_state.retries++;
          netconn_disconnect(conn);
          client_state.conn = NULL;
          client_state.retries++;
        }
      } else {
        LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE, ("tftp: timeout\n"));
        close_session(&client_state);
      }
      break;
    case ERR_OK:
      tftp_recv_packet(&pktbuf, conn, &client_state);
      break;
    default:
      continue;
    }
  }
}

void tftp_get( struct ip_addr *ip,
    const char *local,
    const char *remote,
    const char *mode,
    const struct tftp_context *ctx
  ) {

  struct netconn *conn;
  struct netbuf *pktbuf;
  err_t err;
  struct tftp_state client_state;

  /* Create new UDP connection handle */
  conn = netconn_new(NETCONN_UDP);
  LWIP_ERROR("tftp_client: invalid conn", (conn != NULL),
             return;
            );

  err = netconn_connect(conn, ip, TFTP_PORT);
  LWIP_ERROR("tftp_client: cannot connect to server", (err == ERR_OK),
             return;
            );

  /* Setup the TFTP server state */
  client_state.handle = ctx->open(local, "w", 1);
  LWIP_ERROR("tftp_client: Cannot open file", (client_state.handle != NULL),
             return;
            );
  client_state.retries = 0;
  client_state.ctx = ((struct tftp_context *) ctx);
  client_state.last_data = NULL;
  client_state.conn = conn;
  client_state.blknum = 1;

  /* Set the reciever timeout so we can run retries properly. */
  netconn_set_recvtimeout(conn, TFTP_TIMEOUT_MSECS);

  send_rrq(&client_state, remote, mode);
  /* Disconnect the netconn because we don't know the remote port */
  netconn_disconnect(conn);
  client_state.conn = NULL;

  while(client_state.handle) {
    err = netconn_recv(conn, &pktbuf);

    /* Handle the new packet through the state machine. */
    switch (err) {
    case ERR_TIMEOUT:
      if (client_state.retries < TFTP_MAX_RETRIES) {
        if (client_state.last_data != NULL) {
          LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE,
                      ("tftp: timeout, retrying\n"));
          resend_data(&client_state);
          client_state.retries++;
        } else if (client_state.blknum == 1) {
          LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE,
                      ("tftp: timeout, retrying\n"));
          client_state.conn = conn;
          err = netconn_connect(conn, ip, TFTP_PORT);
          LWIP_ERROR("tftp_client: cannot connect to server",
              (err == ERR_OK), return; );
          send_rrq(&client_state, remote, mode);
          client_state.retries++;
          netconn_disconnect(conn);
          client_state.conn = NULL;
        }
```

```
      } else {
        LWIP_DEBUGF(TFTP_DEBUG | LWIP_DBG_STATE, ("tftp:␣timeout\n"));
        close_session(&client_state);
      }
      break;
    case ERR_OK:
      tftp_recv_packet(&pktbuf, conn, &client_state);
      break;
    default:
      continue;
    }
  }
}

/* vim: set tabstop=8 softtabstop=2 shiftwidth=2 expandtab : */
```

B.8 f4boot/tcp_shell.h

```
#ifndef TCP_SHELL_SERVER_H
#define TCP_SHELL_SERVER_H
#ifdef __cplusplus
extern "C" {
#endif
THD_FUNCTION(tcp_shell_thread, arg);
#ifdef __cplusplus
}
#endif

#endif
```

B.9 f4boot/tcp_shell.c

```
#include "ch.h"
#include "hal.h"
#include "shellconf.h"
#include "shell.h"

#include "lwip/opt.h"
#include "lwip/arch.h"
#include "lwip/api.h"

#include "netstream.h"

#include "tcp_shell.h"

extern const ShellCommand commands[];
static char shhist[SHELL_MAX_HIST_BUFF];

static THD_WORKING_AREA(waTCPShell, 512);

/*
 * TCP server thread.
 */

THD_FUNCTION(tcp_shell_thread, arg) {
    uint16_t port = (uint32_t)arg;
    struct netconn *conn, *newconn;
    err_t err;
    NetStream nsp;
    ShellConfig shell_cfgp;
    thread_t *shelltp = NULL;

    chRegSetThreadName("termserver");

    /* Create a new TCP connection handle */
    conn = netconn_new(NETCONN_TCP);
    LWIP_ERROR("TCP␣server:␣invalid␣conn", (conn != NULL), chThdExit(MSG_RESET););

    /* Bind to a port. */
    netconn_bind(conn, NULL, port);

    /* Listen for connections. */
    netconn_listen(conn);

    /* Clear out the shell config */
    for(size_t i = 0; i < sizeof(shell_cfgp); i++) ((char *)&shell_cfgp)[i] = 0;

    while (!chThdShouldTerminateX()) {
        err = netconn_accept(conn, &newconn);
```

```
        if (err != ERR_OK)
            continue;

        palSetPad(GPIOD, GPIOD_LED_ORANGE);
        nsObjectInit(&nsp);
        netconn_set_recvtimeout(newconn, 600000);
        nsStart(&nsp, newconn);

        shell_cfgp.sc_channel = (BaseSequentialStream *) &nsp;
        shell_cfgp.sc_commands = commands;
        shell_cfgp.sc_histbuf = shhist;
        /* Drop the const-cast so we can initialize this shell definition */
        *((int *)(&(shell_cfgp.sc_histsize))) = SHELL_MAX_HIST_BUFF;

        shelltp = chThdCreateStatic(waTCPShell, sizeof(waTCPShell),
                    NORMALPRIO + 1, shellThread, (void *) &shell_cfgp);
        chRegSetThreadNameX(shelltp, "tcpshell");

        chThdWait(shelltp);
        palClearPad(GPIOD, GPIOD_LED_ORANGE);

        netconn_close(newconn);
        netconn_delete(newconn);
    }

    netconn_close(conn);
    netconn_delete(conn);
    chThdExit(MSG_OK);
}
```

B.10 f4boot/memfs.h

```
#ifndef __MEMFS_H__
#define __MEMFS_H__

#include <stddef.h>

#define BLOADER_SIZE 1024*64

struct file_handle;

struct file_handle *mem_open(const char *fname, const char *mode,
                             int write);
void mem_close(struct file_handle *handle);
int mem_read(struct file_handle *handle, void *buf, size_t bytes);
int mem_write(struct file_handle *handle, void *buf, size_t bytes);

#endif /* __MEMFS_H__ */
```

B.11 f4boot/memfs.c

```
#include <stddef.h>
#include <string.h>
#include "memfs.h"
#include "ch.h"
#include "flash.h"
#include "boot.h"

#define MEMFS_MAX_FILES 8

typedef struct file_handle {
  void    *base_addr;
  size_t  count;
  size_t  len;
  void    (*close)(struct file_handle *h);
  size_t  (*read)(struct file_handle *h, void *buf, size_t bytes);
  size_t  (*write)(struct file_handle *h, const void *buf, size_t bytes);
} file_handle;

static file_handle file_handles[MEMFS_MAX_FILES];

static void ram_check(struct file_handle *h);
static size_t ram_read(struct file_handle *h, void *buf, size_t bytes) ;
static size_t ram_write(struct file_handle *h, const void *buf, size_t bytes) ;
static size_t flash_write(struct file_handle *h, const void *buf, size_t bytes) ;
enum filetype {tRAM, tBACKUP, tFLASH, tBOOT};

static const struct file_entry {
  void *          base_addr;
  size_t          len;
```

```
  enum   filetype   type;
  void             (*close)(struct file_handle *h);
  size_t           (*read)(struct file_handle *h, void *buf, size_t bytes);
  size_t           (*write)(struct file_handle *h, const void *buf,
                            size_t bytes);
  const char       *name;
} files[] = {
  {(void *)BKPSRAM_BASE, 4096, tBACKUP, NULL, &ram_read, &ram_write, "config"},
  {(void *)(FLASH_BASE + BLOADER_SIZE), FLASH_END - FLASH_BASE - BLOADER_SIZE,
    tFLASH, NULL, &ram_read, &flash_write, "core"},
  {(void *)CCMDATARAM_BASE, BLOADER_SIZE,
    tRAM, &ram_check, &ram_read, &ram_write, "bloader"},
  {(void *)CCMDATARAM_BASE, 512, tRAM, NULL, &ram_read, &ram_write, "test"},
  {NULL, 0, tBOOT, NULL, NULL, NULL, "boot"},
  {NULL, 0, 0, NULL, NULL, NULL, NULL}
};

/*
 * Check <64k image in RAM and copy it to flashbase if healthy
 */
static void ram_check(struct file_handle *h) {
  size_t safecount = (h->count > BLOADER_SIZE) ? BLOADER_SIZE : h->count;

  if (bootCheckImage(h->base_addr, (h->base_addr + safecount))) {
    flashErase((uintptr_t) FLASH_BASE, h->count);
    flashWrite((uintptr_t) FLASH_BASE, h->base_addr, safecount);
  }
}

/*
 * Local acessor functions for ram and flash memory spaces.
 */

static size_t ram_read(struct file_handle *h, void *buf, size_t bytes) {

  /* Check that the write will fit in the file's area */
  if (h->count + bytes > h->len) {
    bytes = h->len - h->count;
  }
  memcpy(buf, h->base_addr + h->count, bytes);
  h->count += bytes;
  return bytes;
}

static size_t ram_write(struct file_handle *h, const void *buf, size_t bytes) {

  /* Check that the write will fit in the file's area */
  if (h->count + bytes > h->len) {
    bytes = h->len - h->count;
  }

  /* Do the copy */
  memcpy(h->base_addr + h->count, buf, bytes);
  h->count += bytes;
  return bytes;
}

static size_t flash_write(struct file_handle *h, const void *buf, size_t bytes) {

  /* Check that the write will fit in the file's area */
  if (h->count + bytes > h->len) {
    bytes = h->len - h->count;
  }

  flashWrite((uintptr_t) (h->base_addr + h->count), buf, bytes);
  h->count += bytes;

  return bytes;
}

/*
 * Exported functions
 */

struct file_handle *mem_open(const char *fname, const char *mode,
                             int write){
  struct file_handle *file = file_handles;
  const struct file_entry *f_index = files;

  (void) mode;
  (void) write;

  while (*file->read && (file - file_handles < MEMFS_MAX_FILES)) {
    file++;

    /* If we're at the end of the list, we're out of file handles. */
```

```
    if ((file - file_handles) >= MEMFS_MAX_FILES) {
      /* TODO: Add debug callout in this case */
      return NULL;
    }
  }

  while (f_index->name != NULL) {
    if (strncmp(f_index->name, fname, strlen(f_index->name)) == 0) {
      file->base_addr = f_index->base_addr;
      file->count = 0;
      file->len = f_index->len;
      file->read = f_index->read;
      file->write = f_index->write;
      file->close = f_index->close;
      break;
    }
    f_index++;
  }

  if (f_index->name == NULL) {
    return NULL;
  }

  /* If we have a flash file, erase it now */
  if (f_index->type == tFLASH) {
    flashErase((uintptr_t) (file->base_addr), file->len);
  }

  return file;
}

void mem_close(struct file_handle *handle) {
  /* Call the file's close function */
  if (handle->close) {
    handle->close(handle);
  }

  /* Release the file entry from the table */
  *handle = (file_handle) {NULL, 0, 0, NULL, NULL, NULL};
}

int mem_read(struct file_handle *handle, void *buf, size_t bytes) {
  if (handle->read) {
    return handle->read(handle, buf, bytes);
  }
  return 0;
}

int mem_write(struct file_handle *handle, void *buf, size_t bytes) {
  if (handle->write) {
    return handle->write(handle, buf, bytes);
  }
  return 0;
}

/* vim: set sts=2 sw=2 expandtab : */
```

B.12 f4boot/lwipopts.h

```
/**
 * @file
 *
 * lwIP Options Configuration
 */

/*
 * Copyright (c) 2001-2004 Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without modification,
 * are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 *    this list of conditions and the following disclaimer in the documentation
 *    and/or other materials provided with the distribution.
 * 3. The name of the author may not be used to endorse or promote products
 *    derived from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT
```

```
 * SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
 * OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
 * IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
 * OF SUCH DAMAGE.
 *
 * This file is part of the lwIP TCP/IP stack.
 *
 * Author: Adam Dunkels <adam@sics.se>
 *
 */
#ifndef __LWIPOPT_H__
#define __LWIPOPT_H__


/*
   ----------------------------------------------
   ---------- Platform specific locking ----------
   ----------------------------------------------
*/

/**
 * SYS_LIGHTWEIGHT_PROT==1: if you want inter-task protection for certain
 * critical regions during buffer allocation, deallocation and memory
 * allocation and deallocation.
 */
#ifndef SYS_LIGHTWEIGHT_PROT
#define SYS_LIGHTWEIGHT_PROT            0
#endif

/**
 * NO_SYS==1: Provides VERY minimal functionality. Otherwise,
 * use lwIP facilities.
 */
#ifndef NO_SYS
#define NO_SYS                          0
#endif

/**
 * NO_SYS_NO_TIMERS==1: Drop support for sys_timeout when NO_SYS==1
 * Mainly for compatibility to old versions.
 */
#ifndef NO_SYS_NO_TIMERS
#define NO_SYS_NO_TIMERS                0
#endif

/**
 * MEMCPY: override this if you have a faster implementation at hand than the
 * one included in your C library
 */
#ifndef MEMCPY
#define MEMCPY(dst,src,len)             memcpy(dst,src,len)
#endif

/**
 * SMEMCPY: override this with care! Some compilers (e.g. gcc) can inline a
 * call to memcpy() if the length is known at compile time and is small.
 */
#ifndef SMEMCPY
#define SMEMCPY(dst,src,len)            memcpy(dst,src,len)
#endif

/*
   ------------------------------------
   ---------- Memory options ----------
   ------------------------------------
*/
/**
 * MEM_LIBC_MALLOC==1: Use malloc/free/realloc provided by your C-library
 * instead of the lwip internal allocator. Can save code size if you
 * already use it.
 */
#ifndef MEM_LIBC_MALLOC
#define MEM_LIBC_MALLOC                 0
#endif

/**
* MEMP_MEM_MALLOC==1: Use mem_malloc/mem_free instead of the lwip pool allocator.
* Especially useful with MEM_LIBC_MALLOC but handle with care regarding execution
* speed and usage from interrupts!
*/
#ifndef MEMP_MEM_MALLOC
#define MEMP_MEM_MALLOC                 0
#endif
```

```
/**
 * MEM_ALIGNMENT: should be set to the alignment of the CPU
 *    4 byte alignment -> #define MEM_ALIGNMENT 4
 *    2 byte alignment -> #define MEM_ALIGNMENT 2
 */
#ifndef MEM_ALIGNMENT
#define MEM_ALIGNMENT                   4
#endif

/**
 * MEM_SIZE: the size of the heap memory. If the application will send
 * a lot of data that needs to be copied, this should be set high.
 */
#ifndef MEM_SIZE
#define MEM_SIZE                        3200
#endif

/**
 * MEMP_SEPARATE_POOLS: if defined to 1, each pool is placed in its own array.
 * This can be used to individually change the location of each pool.
 * Default is one big array for all pools
 */
#ifndef MEMP_SEPARATE_POOLS
#define MEMP_SEPARATE_POOLS             0
#endif

/**
 * MEMP_OVERFLOW_CHECK: memp overflow protection reserves a configurable
 * amount of bytes before and after each memp element in every pool and fills
 * it with a prominent default value.
 *    MEMP_OVERFLOW_CHECK == 0 no checking
 *    MEMP_OVERFLOW_CHECK == 1 checks each element when it is freed
 *    MEMP_OVERFLOW_CHECK >= 2 checks each element in every pool every time
 *      memp_malloc() or memp_free() is called (useful but slow!)
 */
#ifndef MEMP_OVERFLOW_CHECK
#define MEMP_OVERFLOW_CHECK             0
#endif

/**
 * MEMP_SANITY_CHECK==1: run a sanity check after each memp_free() to make
 * sure that there are no cycles in the linked lists.
 */
#ifndef MEMP_SANITY_CHECK
#define MEMP_SANITY_CHECK               0
#endif

/**
 * MEM_USE_POOLS==1: Use an alternative to malloc() by allocating from a set
 * of memory pools of various sizes. When mem_malloc is called, an element of
 * the smallest pool that can provide the length needed is returned.
 * To use this, MEMP_USE_CUSTOM_POOLS also has to be enabled.
 */
#ifndef MEM_USE_POOLS
#define MEM_USE_POOLS                   0
#endif

/**
 * MEM_USE_POOLS_TRY_BIGGER_POOL==1: if one malloc-pool is empty, try the next
 * bigger pool - WARNING: THIS MIGHT WASTE MEMORY but it can make a system more
 * reliable. */
#ifndef MEM_USE_POOLS_TRY_BIGGER_POOL
#define MEM_USE_POOLS_TRY_BIGGER_POOL   0
#endif

/**
 * MEMP_USE_CUSTOM_POOLS==1: whether to include a user file lwippools.h
 * that defines additional pools beyond the "standard" ones required
 * by lwIP. If you set this to 1, you must have lwippools.h in your
 * inlude path somewhere.
 */
#ifndef MEMP_USE_CUSTOM_POOLS
#define MEMP_USE_CUSTOM_POOLS           0
#endif

/**
 * Set this to 1 if you want to free PBUF_RAM pbufs (or call mem_free()) from
 * interrupt context (or another context that doesn't allow waiting for a
 * semaphore).
 * If set to 1, mem_malloc will be protected by a semaphore and SYS_ARCH_PROTECT,
 * while mem_free will only use SYS_ARCH_PROTECT. mem_malloc SYS_ARCH_UNPROTECTs
 * with each loop so that mem_free can run.
 *
 * ATTENTION: As you can see from the above description, this leads to dis-/
 * enabling interrupts often, which can be slow! Also, on low memory, mem_malloc
```

```
 * can need longer.
 *
 * If you don't want that, at least for NO_SYS=0, you can still use the following
 * functions to enqueue a deallocation call which then runs in the tcpip_thread
 * context:
 * - pbuf_free_callback(p);
 * - mem_free_callback(m);
 */
#ifndef LWIP_ALLOW_MEM_FREE_FROM_OTHER_CONTEXT
#define LWIP_ALLOW_MEM_FREE_FROM_OTHER_CONTEXT 0
#endif

/*
   ------------------------------------------------
   ---------- Internal Memory Pool Sizes ----------
   ------------------------------------------------
*/
/**
 * MEMP_NUM_PBUF: the number of memp struct pbufs (used for PBUF_ROM and PBUF_REF).
 * If the application sends a lot of data out of ROM (or other static memory),
 * this should be set high.
 */
#ifndef MEMP_NUM_PBUF
#define MEMP_NUM_PBUF                   16
#endif

/**
 * MEMP_NUM_RAW_PCB: Number of raw connection PCBs
 * (requires the LWIP_RAW option)
 */
#ifndef MEMP_NUM_RAW_PCB
#define MEMP_NUM_RAW_PCB                4
#endif

/**
 * MEMP_NUM_UDP_PCB: the number of UDP protocol control blocks. One
 * per active UDP "connection".
 * (requires the LWIP_UDP option)
 */
#ifndef MEMP_NUM_UDP_PCB
#define MEMP_NUM_UDP_PCB                4
#endif

/**
 * MEMP_NUM_TCP_PCB: the number of simulatenously active TCP connections.
 * (requires the LWIP_TCP option)
 */
#ifndef MEMP_NUM_TCP_PCB
#define MEMP_NUM_TCP_PCB                5
#endif

/**
 * MEMP_NUM_TCP_PCB_LISTEN: the number of listening TCP connections.
 * (requires the LWIP_TCP option)
 */
#ifndef MEMP_NUM_TCP_PCB_LISTEN
#define MEMP_NUM_TCP_PCB_LISTEN         8
#endif

/**
 * MEMP_NUM_TCP_SEG: the number of simultaneously queued TCP segments.
 * (requires the LWIP_TCP option)
 */
#ifndef MEMP_NUM_TCP_SEG
#define MEMP_NUM_TCP_SEG                16
#endif

/**
 * MEMP_NUM_REASSDATA: the number of IP packets simultaneously queued for
 * reassembly (whole packets, not fragments!)
 */
#ifndef MEMP_NUM_REASSDATA
#define MEMP_NUM_REASSDATA              5
#endif

/**
 * MEMP_NUM_FRAG_PBUF: the number of IP fragments simultaneously sent
 * (fragments, not whole packets!).
 * This is only used with IP_FRAG_USES_STATIC_BUF==0 and
 * LWIP_NETIF_TX_SINGLE_PBUF==0 and only has to be > 1 with DMA-enabled MACs
 * where the packet is not yet sent when netif->output returns.
 */
#ifndef MEMP_NUM_FRAG_PBUF
#define MEMP_NUM_FRAG_PBUF              15
#endif
```

```
/**
 * MEMP_NUM_ARP_QUEUE: the number of simulateously queued outgoing
 * packets (pbufs) that are waiting for an ARP request (to resolve
 * their destination address) to finish.
 * (requires the ARP_QUEUEING option)
 */
#ifndef MEMP_NUM_ARP_QUEUE
#define MEMP_NUM_ARP_QUEUE              30
#endif

/**
 * MEMP_NUM_IGMP_GROUP: The number of multicast groups whose network interfaces
 * can be members et the same time (one per netif - allsystems group -, plus one
 * per netif membership).
 * (requires the LWIP_IGMP option)
 */
#ifndef MEMP_NUM_IGMP_GROUP
#define MEMP_NUM_IGMP_GROUP             8
#endif

/**
 * MEMP_NUM_SYS_TIMEOUT: the number of simulateously active timeouts.
 * (requires NO_SYS==0)
 * The default number of timeouts is calculated here for all enabled modules.
 * The formula expects settings to be either '0' or '1'.
 */
#ifndef MEMP_NUM_SYS_TIMEOUT
#define MEMP_NUM_SYS_TIMEOUT            (LWIP_TCP + IP_REASSEMBLY + LWIP_ARP + (2*LWIP_DHCP) + LWIP_AUTOIP + LWIP_IGMP + LWIP_DNS
    ↪ + PPP_SUPPORT)
#endif

/**
 * MEMP_NUM_NETBUF: the number of struct netbufs.
 * (only needed if you use the sequential API, like api_lib.c)
 */
#ifndef MEMP_NUM_NETBUF
#define MEMP_NUM_NETBUF                 8
#endif

/**
 * MEMP_NUM_NETCONN: the number of struct netconns.
 * (only needed if you use the sequential API, like api_lib.c)
 */
#ifndef MEMP_NUM_NETCONN
#define MEMP_NUM_NETCONN                8
#endif

/**
 * MEMP_NUM_TCPIP_MSG_API: the number of struct tcpip_msg, which are used
 * for callback/timeout API communication.
 * (only needed if you use tcpip.c)
 */
#ifndef MEMP_NUM_TCPIP_MSG_API
#define MEMP_NUM_TCPIP_MSG_API          8
#endif

/**
 * MEMP_NUM_TCPIP_MSG_INPKT: the number of struct tcpip_msg, which are used
 * for incoming packets.
 * (only needed if you use tcpip.c)
 */
#ifndef MEMP_NUM_TCPIP_MSG_INPKT
#define MEMP_NUM_TCPIP_MSG_INPKT        8
#endif

/**
 * MEMP_NUM_SNMP_NODE: the number of leafs in the SNMP tree.
 */
#ifndef MEMP_NUM_SNMP_NODE
#define MEMP_NUM_SNMP_NODE              50
#endif

/**
 * MEMP_NUM_SNMP_ROOTNODE: the number of branches in the SNMP tree.
 * Every branch has one leaf (MEMP_NUM_SNMP_NODE) at least!
 */
#ifndef MEMP_NUM_SNMP_ROOTNODE
#define MEMP_NUM_SNMP_ROOTNODE          30
#endif

/**
 * MEMP_NUM_SNMP_VARBIND: the number of concurrent requests (does not have to
 * be changed normally) - 2 of these are used per request (1 for input,
 * 1 for output)
 */
#ifndef MEMP_NUM_SNMP_VARBIND
```

```
#define MEMP_NUM_SNMP_VARBIND           2
#endif

/**
 * MEMP_NUM_SNMP_VALUE: the number of OID or values concurrently used
 * (does not have to be changed normally) - 3 of these are used per request
 * (1 for the value read and 2 for OIDs - input and output)
 */
#ifndef MEMP_NUM_SNMP_VALUE
#define MEMP_NUM_SNMP_VALUE             3
#endif

/**
 * MEMP_NUM_NETDB: the number of concurrently running lwip_addrinfo() calls
 * (before freeing the corresponding memory using lwip_freeaddrinfo()).
 */
#ifndef MEMP_NUM_NETDB
#define MEMP_NUM_NETDB                  1
#endif

/**
 * MEMP_NUM_LOCALHOSTLIST: the number of host entries in the local host list
 * if DNS_LOCAL_HOSTLIST_IS_DYNAMIC==1.
 */
#ifndef MEMP_NUM_LOCALHOSTLIST
#define MEMP_NUM_LOCALHOSTLIST          1
#endif

/**
 * MEMP_NUM_PPPOE_INTERFACES: the number of concurrently active PPPoE
 * interfaces (only used with PPPOE_SUPPORT==1)
 */
#ifndef MEMP_NUM_PPPOE_INTERFACES
#define MEMP_NUM_PPPOE_INTERFACES       1
#endif

/**
 * PBUF_POOL_SIZE: the number of buffers in the pbuf pool.
 */
#ifndef PBUF_POOL_SIZE
#define PBUF_POOL_SIZE                  16
#endif

/*
   ------------------------------------
   ---------- ARP options -------------
   ------------------------------------
*/
/**
 * LWIP_ARP==1: Enable ARP functionality.
 */
#ifndef LWIP_ARP
#define LWIP_ARP                        1
#endif

/**
 * ARP_TABLE_SIZE: Number of active MAC-IP address pairs cached.
 */
#ifndef ARP_TABLE_SIZE
#define ARP_TABLE_SIZE                  10
#endif

/**
 * ARP_QUEUEING==1: Multiple outgoing packets are queued during hardware address
 * resolution. By default, only the most recent packet is queued per IP address.
 * This is sufficient for most protocols and mainly reduces TCP connection
 * startup time. Set this to 1 if you know your application sends more than one
 * packet in a row to an IP address that is not in the ARP cache.
 */
#ifndef ARP_QUEUEING
#define ARP_QUEUEING                    0
#endif

/**
 * ETHARP_TRUST_IP_MAC==1: Incoming IP packets cause the ARP table to be
 * updated with the source MAC and IP addresses supplied in the packet.
 * You may want to disable this if you do not trust LAN peers to have the
 * correct addresses, or as a limited approach to attempt to handle
 * spoofing. If disabled, lwIP will need to make a new ARP request if
 * the peer is not already in the ARP table, adding a little latency.
 * The peer *is* in the ARP table if it requested our address before.
 * Also notice that this slows down input processing of every IP packet!
 */
#ifndef ETHARP_TRUST_IP_MAC
#define ETHARP_TRUST_IP_MAC             0
#endif
```

```
/**
 * ETHARP_SUPPORT_VLAN==1: support receiving ethernet packets with VLAN header.
 * Additionally, you can define ETHARP_VLAN_CHECK to an u16_t VLAN ID to check.
 * If ETHARP_VLAN_CHECK is defined, only VLAN-traffic for this VLAN is accepted.
 * If ETHARP_VLAN_CHECK is not defined, all traffic is accepted.
 * Alternatively, define a function/define ETHARP_VLAN_CHECK_FN(eth_hdr, vlan)
 * that returns 1 to accept a packet or 0 to drop a packet.
 */
#ifndef ETHARP_SUPPORT_VLAN
#define ETHARP_SUPPORT_VLAN              0
#endif

/** LWIP_ETHERNET==1: enable ethernet support for PPPoE even though ARP
 * might be disabled
 */
#ifndef LWIP_ETHERNET
#define LWIP_ETHERNET                    (LWIP_ARP || PPPOE_SUPPORT)
#endif

/** ETH_PAD_SIZE: number of bytes added before the ethernet header to ensure
 * alignment of payload after that header. Since the header is 14 bytes long,
 * without this padding e.g. addresses in the IP header will not be aligned
 * on a 32-bit boundary, so setting this to 2 can speed up 32-bit-platforms.
 */
#ifndef ETH_PAD_SIZE
#define ETH_PAD_SIZE                     0
#endif

/** ETHARP_SUPPORT_STATIC_ENTRIES==1: enable code to support static ARP table
 * entries (using etharp_add_static_entry/etharp_remove_static_entry).
 */
#ifndef ETHARP_SUPPORT_STATIC_ENTRIES
#define ETHARP_SUPPORT_STATIC_ENTRIES    0
#endif


/*
   --------------------------------
   ---------- IP options ----------
   --------------------------------
*/
/**
 * IP_FORWARD==1: Enables the ability to forward IP packets across network
 * interfaces. If you are going to run lwIP on a device with only one network
 * interface, define this to 0.
 */
#ifndef IP_FORWARD
#define IP_FORWARD                       0
#endif

/**
 * IP_OPTIONS_ALLOWED: Defines the behavior for IP options.
 *      IP_OPTIONS_ALLOWED==0: All packets with IP options are dropped.
 *      IP_OPTIONS_ALLOWED==1: IP options are allowed (but not parsed).
 */
#ifndef IP_OPTIONS_ALLOWED
#define IP_OPTIONS_ALLOWED               1
#endif

/**
 * IP_REASSEMBLY==1: Reassemble incoming fragmented IP packets. Note that
 * this option does not affect outgoing packet sizes, which can be controlled
 * via IP_FRAG.
 */
#ifndef IP_REASSEMBLY
#define IP_REASSEMBLY                    1
#endif

/**
 * IP_FRAG==1: Fragment outgoing IP packets if their size exceeds MTU. Note
 * that this option does not affect incoming packet sizes, which can be
 * controlled via IP_REASSEMBLY.
 */
#ifndef IP_FRAG
#define IP_FRAG                          1
#endif

/**
 * IP_REASS_MAXAGE: Maximum time (in multiples of IP_TMR_INTERVAL - so seconds, normally)
 * a fragmented IP packet waits for all fragments to arrive. If not all fragments arrived
 * in this time, the whole packet is discarded.
 */
#ifndef IP_REASS_MAXAGE
#define IP_REASS_MAXAGE                  3
#endif
```

```
/**
 * IP_REASS_MAX_PBUFS: Total maximum amount of pbufs waiting to be reassembled.
 * Since the received pbufs are enqueued, be sure to configure
 * PBUF_POOL_SIZE > IP_REASS_MAX_PBUFS so that the stack is still able to receive
 * packets even if the maximum amount of fragments is enqueued for reassembly!
 */
#ifndef IP_REASS_MAX_PBUFS
#define IP_REASS_MAX_PBUFS              10
#endif

/**
 * IP_FRAG_USES_STATIC_BUF==1: Use a static MTU-sized buffer for IP
 * fragmentation. Otherwise pbufs are allocated and reference the original
 * packet data to be fragmented (or with LWIP_NETIF_TX_SINGLE_PBUF==1,
 * new PBUF_RAM pbufs are used for fragments).
 * ATTENTION: IP_FRAG_USES_STATIC_BUF==1 may not be used for DMA-enabled MACs!
 */
#ifndef IP_FRAG_USES_STATIC_BUF
#define IP_FRAG_USES_STATIC_BUF         0
#endif

/**
 * IP_FRAG_MAX_MTU: Assumed max MTU on any interface for IP frag buffer
 * (requires IP_FRAG_USES_STATIC_BUF==1)
 */
#if IP_FRAG_USES_STATIC_BUF && !defined(IP_FRAG_MAX_MTU)
#define IP_FRAG_MAX_MTU                 1500
#endif

/**
 * IP_DEFAULT_TTL: Default value for Time-To-Live used by transport layers.
 */
#ifndef IP_DEFAULT_TTL
#define IP_DEFAULT_TTL                  255
#endif

/**
 * IP_SOF_BROADCAST=1: Use the SOF_BROADCAST field to enable broadcast
 * filter per pcb on udp and raw send operations. To enable broadcast filter
 * on recv operations, you also have to set IP_SOF_BROADCAST_RECV=1.
 */
#ifndef IP_SOF_BROADCAST
#define IP_SOF_BROADCAST                0
#endif

/**
 * IP_SOF_BROADCAST_RECV (requires IP_SOF_BROADCAST=1) enable the broadcast
 * filter on recv operations.
 */
#ifndef IP_SOF_BROADCAST_RECV
#define IP_SOF_BROADCAST_RECV           0
#endif

/**
 * IP_FORWARD_ALLOW_TX_ON_RX_NETIF==1: allow ip_forward() to send packets back
 * out on the netif where it was received. This should only be used for
 * wireless networks.
 * ATTENTION: When this is 1, make sure your netif driver correctly marks incoming
 * link-layer-broadcast/multicast packets as such using the corresponding pbuf flags!
 */
#ifndef IP_FORWARD_ALLOW_TX_ON_RX_NETIF
#define IP_FORWARD_ALLOW_TX_ON_RX_NETIF 0
#endif

/**
 * LWIP_RANDOMIZE_INITIAL_LOCAL_PORTS==1: randomize the local port for the first
 * local TCP/UDP pcb (default==0). This can prevent creating predictable port
 * numbers after booting a device.
 */
#ifndef LWIP_RANDOMIZE_INITIAL_LOCAL_PORTS
#define LWIP_RANDOMIZE_INITIAL_LOCAL_PORTS 0
#endif

/*
   ----------------------------------
   ---------- ICMP options ----------
   ----------------------------------
*/
/**
 * LWIP_ICMP==1: Enable ICMP module inside the IP stack.
 * Be careful, disable that make your product non-compliant to RFC1122
 */
#ifndef LWIP_ICMP
#define LWIP_ICMP                       1
#endif
```

```
/**
 * ICMP_TTL: Default value for Time-To-Live used by ICMP packets.
 */
#ifndef ICMP_TTL
#define ICMP_TTL                        (IP_DEFAULT_TTL)
#endif

/**
 * LWIP_BROADCAST_PING==1: respond to broadcast pings (default is unicast only)
 */
#ifndef LWIP_BROADCAST_PING
#define LWIP_BROADCAST_PING             0
#endif

/**
 * LWIP_MULTICAST_PING==1: respond to multicast pings (default is unicast only)
 */
#ifndef LWIP_MULTICAST_PING
#define LWIP_MULTICAST_PING             0
#endif

/*
   ----------------------------------
   ---------- RAW options ----------
   ----------------------------------
*/
/**
 * LWIP_RAW==1: Enable application layer to hook into the IP layer itself.
 */
#ifndef LWIP_RAW
#define LWIP_RAW                        0
#endif

/**
 * LWIP_RAW==1: Enable application layer to hook into the IP layer itself.
 */
#ifndef RAW_TTL
#define RAW_TTL                         (IP_DEFAULT_TTL)
#endif

/*
   ----------------------------------
   ---------- DHCP options ----------
   ----------------------------------
*/
/**
 * LWIP_DHCP==1: Enable DHCP module.
 */
#ifndef LWIP_DHCP
#define LWIP_DHCP                       0
#endif

/**
 * DHCP_DOES_ARP_CHECK==1: Do an ARP check on the offered address.
 */
#ifndef DHCP_DOES_ARP_CHECK
#define DHCP_DOES_ARP_CHECK             ((LWIP_DHCP) && (LWIP_ARP))
#endif

/**
 * LWIP_DHCP_BOOTP_FILE==1: Store offered_si_addr and boot_file_name.
 */
#ifndef LWIP_DHCP_BOOTP_FILE
#define LWIP_DHCP_BOOTP_FILE            0
#endif

/*
   ------------------------------------
   ---------- AUTOIP options ----------
   ------------------------------------
*/
/**
 * LWIP_AUTOIP==1: Enable AUTOIP module.
 */
#ifndef LWIP_AUTOIP
#define LWIP_AUTOIP                     0
#endif

/**
 * LWIP_DHCP_AUTOIP_COOP==1: Allow DHCP and AUTOIP to be both enabled on
 * the same interface at the same time.
 */
#ifndef LWIP_DHCP_AUTOIP_COOP
#define LWIP_DHCP_AUTOIP_COOP           0
#endif
```

```
/**
 * LWIP_DHCP_AUTOIP_COOP_TRIES: Set to the number of DHCP DISCOVER probes
 * that should be sent before falling back on AUTOIP. This can be set
 * as low as 1 to get an AutoIP address very quickly, but you should
 * be prepared to handle a changing IP address when DHCP overrides
 * AutoIP.
 */
#ifndef LWIP_DHCP_AUTOIP_COOP_TRIES
#define LWIP_DHCP_AUTOIP_COOP_TRIES        9
#endif

/*
   ----------------------------------
   ---------- SNMP options ----------
   ----------------------------------
*/
/**
 * LWIP_SNMP==1: Turn on SNMP module. UDP must be available for SNMP
 * transport.
 */
#ifndef LWIP_SNMP
#define LWIP_SNMP                          0
#endif

/**
 * SNMP_CONCURRENT_REQUESTS: Number of concurrent requests the module will
 * allow. At least one request buffer is required.
 * Does not have to be changed unless external MIBs answer request asynchronously
 */
#ifndef SNMP_CONCURRENT_REQUESTS
#define SNMP_CONCURRENT_REQUESTS           1
#endif

/**
 * SNMP_TRAP_DESTINATIONS: Number of trap destinations. At least one trap
 * destination is required
 */
#ifndef SNMP_TRAP_DESTINATIONS
#define SNMP_TRAP_DESTINATIONS             1
#endif

/**
 * SNMP_PRIVATE_MIB:
 * When using a private MIB, you have to create a file 'private_mib.h' that contains
 * a 'struct mib_array_node mib_private' which contains your MIB.
 */
#ifndef SNMP_PRIVATE_MIB
#define SNMP_PRIVATE_MIB                   0
#endif

/**
 * Only allow SNMP write actions that are 'safe' (e.g. disabeling netifs is not
 * a safe action and disabled when SNMP_SAFE_REQUESTS = 1).
 * Unsafe requests are disabled by default!
 */
#ifndef SNMP_SAFE_REQUESTS
#define SNMP_SAFE_REQUESTS                 1
#endif

/**
 * The maximum length of strings used. This affects the size of
 * MEMP_SNMP_VALUE elements.
 */
#ifndef SNMP_MAX_OCTET_STRING_LEN
#define SNMP_MAX_OCTET_STRING_LEN          127
#endif

/**
 * The maximum depth of the SNMP tree.
 * With private MIBs enabled, this depends on your MIB!
 * This affects the size of MEMP_SNMP_VALUE elements.
 */
#ifndef SNMP_MAX_TREE_DEPTH
#define SNMP_MAX_TREE_DEPTH                15
#endif

/**
 * The size of the MEMP_SNMP_VALUE elements, normally calculated from
 * SNMP_MAX_OCTET_STRING_LEN and SNMP_MAX_TREE_DEPTH.
 */
#ifndef SNMP_MAX_VALUE_SIZE
#define SNMP_MAX_VALUE_SIZE                LWIP_MAX((SNMP_MAX_OCTET_STRING_LEN)+1, sizeof(s32_t)*(SNMP_MAX_TREE_DEPTH))
#endif

/*
```

```
   ----------------------------------
   ---------- IGMP options ----------
   ----------------------------------
*/
/**
 * LWIP_IGMP==1: Turn on IGMP module.
 */
#ifndef LWIP_IGMP
#define LWIP_IGMP                       0
#endif

/*
   ----------------------------------
   ---------- DNS options -----------
   ----------------------------------
*/
/**
 * LWIP_DNS==1: Turn on DNS module. UDP must be available for DNS
 * transport.
 */
#ifndef LWIP_DNS
#define LWIP_DNS                        0
#endif

/** DNS maximum number of entries to maintain locally. */
#ifndef DNS_TABLE_SIZE
#define DNS_TABLE_SIZE                  4
#endif

/** DNS maximum host name length supported in the name table. */
#ifndef DNS_MAX_NAME_LENGTH
#define DNS_MAX_NAME_LENGTH             256
#endif

/** The maximum of DNS servers */
#ifndef DNS_MAX_SERVERS
#define DNS_MAX_SERVERS                 2
#endif

/** DNS do a name checking between the query and the response. */
#ifndef DNS_DOES_NAME_CHECK
#define DNS_DOES_NAME_CHECK             1
#endif

/** DNS message max. size. Default value is RFC compliant. */
#ifndef DNS_MSG_SIZE
#define DNS_MSG_SIZE                    512
#endif

/** DNS_LOCAL_HOSTLIST: Implements a local host-to-address list. If enabled,
 *  you have to define
 *    #define DNS_LOCAL_HOSTLIST_INIT {{"host1", 0x123}, {"host2", 0x234}}
 *  (an array of structs name/address, where address is an u32_t in network
 *  byte order).
 *
 *  Instead, you can also use an external function:
 *  #define DNS_LOOKUP_LOCAL_EXTERN(x) extern u32_t my_lookup_function(const char *name)
 *  that returns the IP address or INADDR_NONE if not found.
 */
#ifndef DNS_LOCAL_HOSTLIST
#define DNS_LOCAL_HOSTLIST              0
#endif /* DNS_LOCAL_HOSTLIST */

/** If this is turned on, the local host-list can be dynamically changed
 *  at runtime. */
#ifndef DNS_LOCAL_HOSTLIST_IS_DYNAMIC
#define DNS_LOCAL_HOSTLIST_IS_DYNAMIC   0
#endif /* DNS_LOCAL_HOSTLIST_IS_DYNAMIC */

/*
   ----------------------------------
   ---------- UDP options ----------
   ----------------------------------
*/
/**
 * LWIP_UDP==1: Turn on UDP.
 */
#ifndef LWIP_UDP
#define LWIP_UDP                        1
#endif

/**
 * LWIP_UDPLITE==1: Turn on UDP-Lite. (Requires LWIP_UDP)
 */
#ifndef LWIP_UDPLITE
#define LWIP_UDPLITE                    0
```

```
#endif

/**
 * UDP_TTL: Default Time-To-Live value.
 */
#ifndef UDP_TTL
#define UDP_TTL                         (IP_DEFAULT_TTL)
#endif

/**
 * LWIP_NETBUF_RECVINFO==1: append destination addr and port to every netbuf.
 */
#ifndef LWIP_NETBUF_RECVINFO
#define LWIP_NETBUF_RECVINFO            0
#endif

/*
   ----------------------------------
   ---------- TCP options ----------
   ----------------------------------
*/
/**
 * LWIP_TCP==1: Turn on TCP.
 */
#ifndef LWIP_TCP
#define LWIP_TCP                        1
#endif

/**
 * TCP_TTL: Default Time-To-Live value.
 */
#ifndef TCP_TTL
#define TCP_TTL                         (IP_DEFAULT_TTL)
#endif

/**
 * TCP_WND: The size of a TCP window.  This must be at least
 * (2 * TCP_MSS) for things to work well
 */
#ifndef TCP_WND
#define TCP_WND                         (4 * TCP_MSS)
#endif

/**
 * TCP_MAXRTX: Maximum number of retransmissions of data segments.
 */
#ifndef TCP_MAXRTX
#define TCP_MAXRTX                      12
#endif

/**
 * TCP_SYNMAXRTX: Maximum number of retransmissions of SYN segments.
 */
#ifndef TCP_SYNMAXRTX
#define TCP_SYNMAXRTX                   6
#endif

/**
 * TCP_QUEUE_OOSEQ==1: TCP will queue segments that arrive out of order.
 * Define to 0 if your device is low on memory.
 */
#ifndef TCP_QUEUE_OOSEQ
#define TCP_QUEUE_OOSEQ                 (LWIP_TCP)
#endif

/**
 * TCP_MSS: TCP Maximum segment size. (default is 536, a conservative default,
 * you might want to increase this.)
 * For the receive side, this MSS is advertised to the remote side
 * when opening a connection. For the transmit size, this MSS sets
 * an upper limit on the MSS advertised by the remote host.
 */
#ifndef TCP_MSS
#define TCP_MSS                         536
#endif

/**
 * TCP_CALCULATE_EFF_SEND_MSS: "The maximum size of a segment that TCP really
 * sends, the 'effective send MSS,' MUST be the smaller of the send MSS (which
 * reflects the available reassembly buffer size at the remote host) and the
 * largest size permitted by the IP layer" (RFC 1122)
 * Setting this to 1 enables code that checks TCP_MSS against the MTU of the
 * netif used for a connection and limits the MSS if it would be too big otherwise.
 */
#ifndef TCP_CALCULATE_EFF_SEND_MSS
#define TCP_CALCULATE_EFF_SEND_MSS      1
```

```
#endif


/**
 * TCP_SND_BUF: TCP sender buffer space (bytes).
 * To achieve good performance, this should be at least 2 * TCP_MSS.
 */
#ifndef TCP_SND_BUF
#define TCP_SND_BUF                 (2 * TCP_MSS)
#endif

/**
 * TCP_SND_QUEUELEN: TCP sender buffer space (pbufs). This must be at least
 * as much as (2 * TCP_SND_BUF/TCP_MSS) for things to work.
 */
#ifndef TCP_SND_QUEUELEN
#define TCP_SND_QUEUELEN            ((4 * (TCP_SND_BUF) + (TCP_MSS - 1))/(TCP_MSS))
#endif

/**
 * TCP_SNDLOWAT: TCP writable space (bytes). This must be less than
 * TCP_SND_BUF. It is the amount of space which must be available in the
 * TCP snd_buf for select to return writable (combined with TCP_SNDQUEUELOWAT).
 */
#ifndef TCP_SNDLOWAT
#define TCP_SNDLOWAT                LWIP_MIN(LWIP_MAX(((TCP_SND_BUF)/2), (2 * TCP_MSS) + 1), (TCP_SND_BUF) - 1)
#endif

/**
 * TCP_SNDQUEUELOWAT: TCP writable bufs (pbuf count). This must be less
 * than TCP_SND_QUEUELEN. If the number of pbufs queued on a pcb drops below
 * this number, select returns writable (combined with TCP_SNDLOWAT).
 */
#ifndef TCP_SNDQUEUELOWAT
#define TCP_SNDQUEUELOWAT           LWIP_MAX(((TCP_SND_QUEUELEN)/2), 5)
#endif

/**
 * TCP_OOSEQ_MAX_BYTES: The maximum number of bytes queued on ooseq per pcb.
 * Default is 0 (no limit). Only valid for TCP_QUEUE_OOSEQ==0.
 */
#ifndef TCP_OOSEQ_MAX_BYTES
#define TCP_OOSEQ_MAX_BYTES         0
#endif

/**
 * TCP_OOSEQ_MAX_PBUFS: The maximum number of pbufs queued on ooseq per pcb.
 * Default is 0 (no limit). Only valid for TCP_QUEUE_OOSEQ==0.
 */
#ifndef TCP_OOSEQ_MAX_PBUFS
#define TCP_OOSEQ_MAX_PBUFS         0
#endif

/**
 * TCP_LISTEN_BACKLOG: Enable the backlog option for tcp listen pcb.
 */
#ifndef TCP_LISTEN_BACKLOG
#define TCP_LISTEN_BACKLOG          0
#endif

/**
 * The maximum allowed backlog for TCP listen netconns.
 * This backlog is used unless another is explicitly specified.
 * 0xff is the maximum (u8_t).
 */
#ifndef TCP_DEFAULT_LISTEN_BACKLOG
#define TCP_DEFAULT_LISTEN_BACKLOG  0xff
#endif

/**
 * TCP_OVERSIZE: The maximum number of bytes that tcp_write may
 * allocate ahead of time in an attempt to create shorter pbuf chains
 * for transmission. The meaningful range is 0 to TCP_MSS. Some
 * suggested values are:
 *
 * 0:         Disable oversized allocation. Each tcp_write() allocates a new
              pbuf (old behaviour).
 * 1:         Allocate size-aligned pbufs with minimal excess. Use this if your
 *            scatter-gather DMA requires aligned fragments.
 * 128:       Limit the pbuf/memory overhead to 20%.
 * TCP_MSS:   Try to create unfragmented TCP packets.
 * TCP_MSS/4: Try to create 4 fragments or less per TCP packet.
 */
#ifndef TCP_OVERSIZE
#define TCP_OVERSIZE                TCP_MSS
#endif
```

```
/**
 * LWIP_TCP_TIMESTAMPS==1: support the TCP timestamp option.
 */
#ifndef LWIP_TCP_TIMESTAMPS
#define LWIP_TCP_TIMESTAMPS             0
#endif

/**
 * TCP_WND_UPDATE_THRESHOLD: difference in window to trigger an
 * explicit window update
 */
#ifndef TCP_WND_UPDATE_THRESHOLD
#define TCP_WND_UPDATE_THRESHOLD    (TCP_WND / 4)
#endif

/**
 * LWIP_EVENT_API and LWIP_CALLBACK_API: Only one of these should be set to 1.
 *     LWIP_EVENT_API==1: The user defines lwip_tcp_event() to receive all
 *         events (accept, sent, etc) that happen in the system.
 *     LWIP_CALLBACK_API==1: The PCB callback function is called directly
 *         for the event. This is the default.
 */
#if !defined(LWIP_EVENT_API) && !defined(LWIP_CALLBACK_API)
#define LWIP_EVENT_API                  0
#define LWIP_CALLBACK_API               1
#endif


/*
   ----------------------------------
   ---------- Pbuf options ----------
   ----------------------------------
*/
/**
 * PBUF_LINK_HLEN: the number of bytes that should be allocated for a
 * link level header. The default is 14, the standard value for
 * Ethernet.
 */
#ifndef PBUF_LINK_HLEN
#define PBUF_LINK_HLEN                  (14 + ETH_PAD_SIZE)
#endif

/**
 * PBUF_POOL_BUFSIZE: the size of each pbuf in the pbuf pool. The default is
 * designed to accomodate single full size TCP frame in one pbuf, including
 * TCP_MSS, IP header, and link header.
 */
#ifndef PBUF_POOL_BUFSIZE
#define PBUF_POOL_BUFSIZE               LWIP_MEM_ALIGN_SIZE(TCP_MSS+40+PBUF_LINK_HLEN)
#endif

/*
   ------------------------------------------------
   ---------- Network Interfaces options ----------
   ------------------------------------------------
*/
/**
 * LWIP_NETIF_HOSTNAME==1: use DHCP_OPTION_HOSTNAME with netif's hostname
 * field.
 */
#ifndef LWIP_NETIF_HOSTNAME
#define LWIP_NETIF_HOSTNAME             0
#endif

/**
 * LWIP_NETIF_API==1: Support netif api (in netifapi.c)
 */
#ifndef LWIP_NETIF_API
#define LWIP_NETIF_API                  0
#endif

/**
 * LWIP_NETIF_STATUS_CALLBACK==1: Support a callback function whenever an interface
 * changes its up/down status (i.e., due to DHCP IP acquistion)
 */
#ifndef LWIP_NETIF_STATUS_CALLBACK
#define LWIP_NETIF_STATUS_CALLBACK      0
#endif

/**
 * LWIP_NETIF_LINK_CALLBACK==1: Support a callback function from an interface
 * whenever the link changes (i.e., link down)
 */
#ifndef LWIP_NETIF_LINK_CALLBACK
#define LWIP_NETIF_LINK_CALLBACK        0
```

```
#endif

/**
 * LWIP_NETIF_REMOVE_CALLBACK==1: Support a callback function that is called
 * when a netif has been removed
 */
#ifndef LWIP_NETIF_REMOVE_CALLBACK
#define LWIP_NETIF_REMOVE_CALLBACK          0
#endif

/**
 * LWIP_NETIF_HWADDRHINT==1: Cache link-layer-address hints (e.g. table
 * indices) in struct netif. TCP and UDP can make use of this to prevent
 * scanning the ARP table for every sent packet. While this is faster for big
 * ARP tables or many concurrent connections, it might be counterproductive
 * if you have a tiny ARP table or if there never are concurrent connections.
 */
#ifndef LWIP_NETIF_HWADDRHINT
#define LWIP_NETIF_HWADDRHINT               0
#endif

/**
 * LWIP_NETIF_LOOPBACK==1: Support sending packets with a destination IP
 * address equal to the netif IP address, looping them back up the stack.
 */
#ifndef LWIP_NETIF_LOOPBACK
#define LWIP_NETIF_LOOPBACK                 0
#endif

/**
 * LWIP_LOOPBACK_MAX_PBUFS: Maximum number of pbufs on queue for loopback
 * sending for each netif (0 = disabled)
 */
#ifndef LWIP_LOOPBACK_MAX_PBUFS
#define LWIP_LOOPBACK_MAX_PBUFS             0
#endif

/**
 * LWIP_NETIF_LOOPBACK_MULTITHREADING: Indicates whether threading is enabled in
 * the system, as netifs must change how they behave depending on this setting
 * for the LWIP_NETIF_LOOPBACK option to work.
 * Setting this is needed to avoid reentering non-reentrant functions like
 * tcp_input().
 *    LWIP_NETIF_LOOPBACK_MULTITHREADING==1: Indicates that the user is using a
 *       multithreaded environment like tcpip.c. In this case, netif->input()
 *       is called directly.
 *    LWIP_NETIF_LOOPBACK_MULTITHREADING==0: Indicates a polling (or NO_SYS) setup.
 *       The packets are put on a list and netif_poll() must be called in
 *       the main application loop.
 */
#ifndef LWIP_NETIF_LOOPBACK_MULTITHREADING
#define LWIP_NETIF_LOOPBACK_MULTITHREADING    (!NO_SYS)
#endif

/**
 * LWIP_NETIF_TX_SINGLE_PBUF: if this is set to 1, lwIP tries to put all data
 * to be sent into one single pbuf. This is for compatibility with DMA-enabled
 * MACs that do not support scatter-gather.
 * Beware that this might involve CPU-memcpy before transmitting that would not
 * be needed without this flag! Use this only if you need to!
 *
 * @todo: TCP and IP-frag do not work with this, yet:
 */
#ifndef LWIP_NETIF_TX_SINGLE_PBUF
#define LWIP_NETIF_TX_SINGLE_PBUF             0
#endif /* LWIP_NETIF_TX_SINGLE_PBUF */

/*
   ------------------------------------
   ---------- LOOPIF options ----------
   ------------------------------------
*/
/**
 * LWIP_HAVE_LOOPIF==1: Support loop interface (127.0.0.1) and loopif.c
 */
#ifndef LWIP_HAVE_LOOPIF
#define LWIP_HAVE_LOOPIF                    0
#endif

/*
   ------------------------------------
   ---------- SLIPIF options ----------
   ------------------------------------
*/
/**
 * LWIP_HAVE_SLIPIF==1: Support slip interface and slipif.c
```

```
 */
#ifndef LWIP_HAVE_SLIPIF
#define LWIP_HAVE_SLIPIF                0
#endif

/*
   ------------------------------------
   ---------- Thread options ----------
   ------------------------------------
*/
/**
 * TCPIP_THREAD_NAME: The name assigned to the main tcpip thread.
 */
#ifndef TCPIP_THREAD_NAME
#define TCPIP_THREAD_NAME              "tcpip_thread"
#endif

/**
 * TCPIP_THREAD_STACKSIZE: The stack size used by the main tcpip thread.
 * The stack size value itself is platform-dependent, but is passed to
 * sys_thread_new() when the thread is created.
 */
#ifndef TCPIP_THREAD_STACKSIZE
#define TCPIP_THREAD_STACKSIZE          1024
#endif

/**
 * TCPIP_THREAD_PRIO: The priority assigned to the main tcpip thread.
 * The priority value itself is platform-dependent, but is passed to
 * sys_thread_new() when the thread is created.
 */
#ifndef TCPIP_THREAD_PRIO
#define TCPIP_THREAD_PRIO               (LOWPRIO + 1)
#endif

/**
 * TCPIP_MBOX_SIZE: The mailbox size for the tcpip thread messages
 * The queue size value itself is platform-dependent, but is passed to
 * sys_mbox_new() when tcpip_init is called.
 */
#ifndef TCPIP_MBOX_SIZE
#define TCPIP_MBOX_SIZE                 MEMP_NUM_PBUF
#endif

/**
 * SLIPIF_THREAD_NAME: The name assigned to the slipif_loop thread.
 */
#ifndef SLIPIF_THREAD_NAME
#define SLIPIF_THREAD_NAME             "slipif_loop"
#endif

/**
 * SLIP_THREAD_STACKSIZE: The stack size used by the slipif_loop thread.
 * The stack size value itself is platform-dependent, but is passed to
 * sys_thread_new() when the thread is created.
 */
#ifndef SLIPIF_THREAD_STACKSIZE
#define SLIPIF_THREAD_STACKSIZE         1024
#endif

/**
 * SLIPIF_THREAD_PRIO: The priority assigned to the slipif_loop thread.
 * The priority value itself is platform-dependent, but is passed to
 * sys_thread_new() when the thread is created.
 */
#ifndef SLIPIF_THREAD_PRIO
#define SLIPIF_THREAD_PRIO              (LOWPRIO + 1)
#endif

/**
 * PPP_THREAD_NAME: The name assigned to the pppInputThread.
 */
#ifndef PPP_THREAD_NAME
#define PPP_THREAD_NAME                "pppInputThread"
#endif

/**
 * PPP_THREAD_STACKSIZE: The stack size used by the pppInputThread.
 * The stack size value itself is platform-dependent, but is passed to
 * sys_thread_new() when the thread is created.
 */
#ifndef PPP_THREAD_STACKSIZE
#define PPP_THREAD_STACKSIZE            1024
#endif

/**
```

```
 * PPP_THREAD_PRIO: The priority assigned to the pppInputThread.
 * The priority value itself is platform-dependent, but is passed to
 * sys_thread_new() when the thread is created.
 */
#ifndef PPP_THREAD_PRIO
#define PPP_THREAD_PRIO                 (LOWPRIO + 1)
#endif

/**
 * DEFAULT_THREAD_NAME: The name assigned to any other lwIP thread.
 */
#ifndef DEFAULT_THREAD_NAME
#define DEFAULT_THREAD_NAME             "lwIP"
#endif

/**
 * DEFAULT_THREAD_STACKSIZE: The stack size used by any other lwIP thread.
 * The stack size value itself is platform-dependent, but is passed to
 * sys_thread_new() when the thread is created.
 */
#ifndef DEFAULT_THREAD_STACKSIZE
#define DEFAULT_THREAD_STACKSIZE        1024
#endif

/**
 * DEFAULT_THREAD_PRIO: The priority assigned to any other lwIP thread.
 * The priority value itself is platform-dependent, but is passed to
 * sys_thread_new() when the thread is created.
 */
#ifndef DEFAULT_THREAD_PRIO
#define DEFAULT_THREAD_PRIO             (LOWPRIO + 1)
#endif

/**
 * DEFAULT_RAW_RECVMBOX_SIZE: The mailbox size for the incoming packets on a
 * NETCONN_RAW. The queue size value itself is platform-dependent, but is passed
 * to sys_mbox_new() when the recvmbox is created.
 */
#ifndef DEFAULT_RAW_RECVMBOX_SIZE
#define DEFAULT_RAW_RECVMBOX_SIZE       4
#endif

/**
 * DEFAULT_UDP_RECVMBOX_SIZE: The mailbox size for the incoming packets on a
 * NETCONN_UDP. The queue size value itself is platform-dependent, but is passed
 * to sys_mbox_new() when the recvmbox is created.
 */
#ifndef DEFAULT_UDP_RECVMBOX_SIZE
#define DEFAULT_UDP_RECVMBOX_SIZE       4
#endif

/**
 * DEFAULT_TCP_RECVMBOX_SIZE: The mailbox size for the incoming packets on a
 * NETCONN_TCP. The queue size value itself is platform-dependent, but is passed
 * to sys_mbox_new() when the recvmbox is created.
 */
#ifndef DEFAULT_TCP_RECVMBOX_SIZE
#define DEFAULT_TCP_RECVMBOX_SIZE       40
#endif

/**
 * DEFAULT_ACCEPTMBOX_SIZE: The mailbox size for the incoming connections.
 * The queue size value itself is platform-dependent, but is passed to
 * sys_mbox_new() when the acceptmbox is created.
 */
#ifndef DEFAULT_ACCEPTMBOX_SIZE
#define DEFAULT_ACCEPTMBOX_SIZE         4
#endif

/*
   ----------------------------------------------
   ---------- Sequential layer options ----------
   ----------------------------------------------
*/
/**
 * LWIP_TCPIP_CORE_LOCKING: (EXPERIMENTAL!)
 * Don't use it if you're not an active lwIP project member
 */
#ifndef LWIP_TCPIP_CORE_LOCKING
#define LWIP_TCPIP_CORE_LOCKING         0
#endif

/**
 * LWIP_TCPIP_CORE_LOCKING_INPUT: (EXPERIMENTAL!)
 * Don't use it if you're not an active lwIP project member
 */
```

```
#ifndef LWIP_TCPIP_CORE_LOCKING_INPUT
#define LWIP_TCPIP_CORE_LOCKING_INPUT     0
#endif

/**
 * LWIP_NETCONN==1: Enable Netconn API (require to use api_lib.c)
 */
#ifndef LWIP_NETCONN
#define LWIP_NETCONN                      1
#endif

/** LWIP_TCPIP_TIMEOUT==1: Enable tcpip_timeout/tcpip_untimeout tod create
 * timers running in tcpip_thread from another thread.
 */
#ifndef LWIP_TCPIP_TIMEOUT
#define LWIP_TCPIP_TIMEOUT                1
#endif

/*
   ----------------------------------
   ---------- Socket options ----------
   ----------------------------------
*/
/**
 * LWIP_SOCKET==1: Enable Socket API (require to use sockets.c)
 */
#ifndef LWIP_SOCKET
#define LWIP_SOCKET                       0
#endif

/**
 * LWIP_COMPAT_SOCKETS==1: Enable BSD-style sockets functions names.
 * (only used if you use sockets.c)
 */
#ifndef LWIP_COMPAT_SOCKETS
#define LWIP_COMPAT_SOCKETS               0
#endif

/**
 * LWIP_POSIX_SOCKETS_IO_NAMES==1: Enable POSIX-style sockets functions names.
 * Disable this option if you use a POSIX operating system that uses the same
 * names (read, write & close). (only used if you use sockets.c)
 */
#ifndef LWIP_POSIX_SOCKETS_IO_NAMES
#define LWIP_POSIX_SOCKETS_IO_NAMES       0
#endif

/**
 * LWIP_TCP_KEEPALIVE==1: Enable TCP_KEEPIDLE, TCP_KEEPINTVL and TCP_KEEPCNT
 * options processing. Note that TCP_KEEPIDLE and TCP_KEEPINTVL have to be set
 * in seconds. (does not require sockets.c, and will affect tcp.c)
 */
#ifndef LWIP_TCP_KEEPALIVE
#define LWIP_TCP_KEEPALIVE                0
#endif

/**
 * LWIP_SO_SNDTIMEO==1: Enable send timeout for sockets/netconns and
 * SO_SNDTIMEO processing.
 */
#ifndef LWIP_SO_SNDTIMEO
#define LWIP_SO_SNDTIMEO                  1
#endif

/**
 * LWIP_SO_RCVTIMEO==1: Enable receive timeout for sockets/netconns and
 * SO_RCVTIMEO processing.
 */
#ifndef LWIP_SO_RCVTIMEO
#define LWIP_SO_RCVTIMEO                  1
#endif

/**
 * LWIP_SO_RCVBUF==1: Enable SO_RCVBUF processing.
 */
#ifndef LWIP_SO_RCVBUF
#define LWIP_SO_RCVBUF                    0
#endif

/**
 * If LWIP_SO_RCVBUF is used, this is the default value for recv_bufsize.
 */
#ifndef RECV_BUFSIZE_DEFAULT
#define RECV_BUFSIZE_DEFAULT              INT_MAX
#endif
```

```
/**
 * SO_REUSE==1: Enable SO_REUSEADDR option.
 */
#ifndef SO_REUSE
#define SO_REUSE                        0
#endif

/**
 * SO_REUSE_RXTOALL==1: Pass a copy of incoming broadcast/multicast packets
 * to all local matches if SO_REUSEADDR is turned on.
 * WARNING: Adds a memcpy for every packet if passing to more than one pcb!
 */
#ifndef SO_REUSE_RXTOALL
#define SO_REUSE_RXTOALL                0
#endif

/*
   ----------------------------------------
   ---------- Statistics options ----------
   ----------------------------------------
*/
/**
 * LWIP_STATS==1: Enable statistics collection in lwip_stats.
 */
#ifndef LWIP_STATS
#define LWIP_STATS                      0
#endif

#if LWIP_STATS

/**
 * LWIP_STATS_DISPLAY==1: Compile in the statistics output functions.
 */
#ifndef LWIP_STATS_DISPLAY
#define LWIP_STATS_DISPLAY              0
#endif

/**
 * LINK_STATS==1: Enable link stats.
 */
#ifndef LINK_STATS
#define LINK_STATS                      0
#endif

/**
 * ETHARP_STATS==1: Enable etharp stats.
 */
#ifndef ETHARP_STATS
#define ETHARP_STATS                    (LWIP_ARP)
#endif

/**
 * IP_STATS==1: Enable IP stats.
 */
#ifndef IP_STATS
#define IP_STATS                        0
#endif

/**
 * IPFRAG_STATS==1: Enable IP fragmentation stats. Default is
 * on if using either frag or reass.
 */
#ifndef IPFRAG_STATS
#define IPFRAG_STATS                    (IP_REASSEMBLY || IP_FRAG)
#endif

/**
 * ICMP_STATS==1: Enable ICMP stats.
 */
#ifndef ICMP_STATS
#define ICMP_STATS                      0
#endif

/**
 * IGMP_STATS==1: Enable IGMP stats.
 */
#ifndef IGMP_STATS
#define IGMP_STATS                      (LWIP_IGMP)
#endif

/**
 * UDP_STATS==1: Enable UDP stats. Default is on if
 * UDP enabled, otherwise off.
 */
#ifndef UDP_STATS
#define UDP_STATS                       (LWIP_UDP)
```

```
#endif

/**
 * TCP_STATS==1: Enable TCP stats. Default is on if TCP
 * enabled, otherwise off.
 */
#ifndef TCP_STATS
#define TCP_STATS                       (LWIP_TCP)
#endif

/**
 * MEM_STATS==1: Enable mem.c stats.
 */
#ifndef MEM_STATS
#define MEM_STATS                       ((MEM_LIBC_MALLOC == 0) && (MEM_USE_POOLS == 0))
#endif

/**
 * MEMP_STATS==1: Enable memp.c pool stats.
 */
#ifndef MEMP_STATS
#define MEMP_STATS                      (MEMP_MEM_MALLOC == 0)
#endif

/**
 * SYS_STATS==1: Enable system stats (sem and mbox counts, etc).
 */
#ifndef SYS_STATS
#define SYS_STATS                       (NO_SYS == 0)
#endif

#else

#define LINK_STATS                      0
#define ETHARP_STATS                    0
#define IP_STATS                        0
#define IPFRAG_STATS                    0
#define ICMP_STATS                      0
#define IGMP_STATS                      0
#define UDP_STATS                       0
#define TCP_STATS                       0
#define MEM_STATS                       0
#define MEMP_STATS                      0
#define SYS_STATS                       0
#define LWIP_STATS_DISPLAY              0

#endif /* LWIP_STATS */

/*
   ---------------------------------------
   ---------- PPP options ----------
   ---------------------------------------
*/
/**
 * PPP_SUPPORT==1: Enable PPP.
 */
#ifndef PPP_SUPPORT
#define PPP_SUPPORT                     0
#endif

/**
 * PPPOE_SUPPORT==1: Enable PPP Over Ethernet
 */
#ifndef PPPOE_SUPPORT
#define PPPOE_SUPPORT                   0
#endif

/**
 * PPPOS_SUPPORT==1: Enable PPP Over Serial
 */
#ifndef PPPOS_SUPPORT
#define PPPOS_SUPPORT                   PPP_SUPPORT
#endif

#if PPP_SUPPORT

/**
 * NUM_PPP: Max PPP sessions.
 */
#ifndef NUM_PPP
#define NUM_PPP                         1
#endif

/**
 * PAP_SUPPORT==1: Support PAP.
 */
```

```
#ifndef PAP_SUPPORT
#define PAP_SUPPORT                     0
#endif

/**
 * CHAP_SUPPORT==1: Support CHAP.
 */
#ifndef CHAP_SUPPORT
#define CHAP_SUPPORT                    0
#endif

/**
 * MSCHAP_SUPPORT==1: Support MSCHAP. CURRENTLY NOT SUPPORTED! DO NOT SET!
 */
#ifndef MSCHAP_SUPPORT
#define MSCHAP_SUPPORT                  0
#endif

/**
 * CBCP_SUPPORT==1: Support CBCP. CURRENTLY NOT SUPPORTED! DO NOT SET!
 */
#ifndef CBCP_SUPPORT
#define CBCP_SUPPORT                    0
#endif

/**
 * CCP_SUPPORT==1: Support CCP. CURRENTLY NOT SUPPORTED! DO NOT SET!
 */
#ifndef CCP_SUPPORT
#define CCP_SUPPORT                     0
#endif

/**
 * VJ_SUPPORT==1: Support VJ header compression.
 */
#ifndef VJ_SUPPORT
#define VJ_SUPPORT                      0
#endif

/**
 * MD5_SUPPORT==1: Support MD5 (see also CHAP).
 */
#ifndef MD5_SUPPORT
#define MD5_SUPPORT                     0
#endif

/*
 * Timeouts
 */
#ifndef FSM_DEFTIMEOUT
#define FSM_DEFTIMEOUT                  6       /* Timeout time in seconds */
#endif

#ifndef FSM_DEFMAXTERMREQS
#define FSM_DEFMAXTERMREQS              2       /* Maximum Terminate-Request transmissions */
#endif

#ifndef FSM_DEFMAXCONFREQS
#define FSM_DEFMAXCONFREQS              10      /* Maximum Configure-Request transmissions */
#endif

#ifndef FSM_DEFMAXNAKLOOPS
#define FSM_DEFMAXNAKLOOPS              5       /* Maximum number of nak loops */
#endif

#ifndef UPAP_DEFTIMEOUT
#define UPAP_DEFTIMEOUT                 6       /* Timeout (seconds) for retransmitting req */
#endif

#ifndef UPAP_DEFREQTIME
#define UPAP_DEFREQTIME                 30      /* Time to wait for auth-req from peer */
#endif

#ifndef CHAP_DEFTIMEOUT
#define CHAP_DEFTIMEOUT                 6       /* Timeout time in seconds */
#endif

#ifndef CHAP_DEFTRANSMITS
#define CHAP_DEFTRANSMITS               10      /* max # times to send challenge */
#endif

/* Interval in seconds between keepalive echo requests, 0 to disable. */
#ifndef LCP_ECHOINTERVAL
#define LCP_ECHOINTERVAL                0
#endif
```

```
/* Number of unanswered echo requests before failure. */
#ifndef LCP_MAXECHOFAILS
#define LCP_MAXECHOFAILS                3
#endif

/* Max Xmit idle time (in jiffies) before resend flag char. */
#ifndef PPP_MAXIDLEFLAG
#define PPP_MAXIDLEFLAG                 100
#endif

/*
 * Packet sizes
 *
 * Note - lcp shouldn't be allowed to negotiate stuff outside these
 *    limits.  See lcp.h in the pppd directory.
 * (XXX - these constants should simply be shared by lcp.c instead
 *    of living in lcp.h)
 */
#define PPP_MTU                         1500     /* Default MTU (size of Info field) */
#ifndef PPP_MAXMTU
/* #define PPP_MAXMTU  65535 - (PPP_HDRLEN + PPP_FCSLEN) */
#define PPP_MAXMTU                      1500 /* Largest MTU we allow */
#endif
#define PPP_MINMTU                      64
#define PPP_MRU                         1500     /* default MRU = max length of info field */
#define PPP_MAXMRU                      1500     /* Largest MRU we allow */
#ifndef PPP_DEFMRU
#define PPP_DEFMRU                      296             /* Try for this */
#endif
#define PPP_MINMRU                      128             /* No MRUs below this */

#ifndef MAXNAMELEN
#define MAXNAMELEN                      256     /* max length of hostname or name for auth */
#endif
#ifndef MAXSECRETLEN
#define MAXSECRETLEN                    256     /* max length of password or secret */
#endif

#endif /* PPP_SUPPORT */

/*
   ----------------------------------------
   ---------- Checksum options ----------
   ----------------------------------------
*/
/**
 * CHECKSUM_GEN_IP==1: Generate checksums in software for outgoing IP packets.
 */
#ifndef CHECKSUM_GEN_IP
#define CHECKSUM_GEN_IP                 1
#endif

/**
 * CHECKSUM_GEN_UDP==1: Generate checksums in software for outgoing UDP packets.
 */
#ifndef CHECKSUM_GEN_UDP
#define CHECKSUM_GEN_UDP                1
#endif

/**
 * CHECKSUM_GEN_TCP==1: Generate checksums in software for outgoing TCP packets.
 */
#ifndef CHECKSUM_GEN_TCP
#define CHECKSUM_GEN_TCP                1
#endif

/**
 * CHECKSUM_GEN_ICMP==1: Generate checksums in software for outgoing ICMP packets.
 */
#ifndef CHECKSUM_GEN_ICMP
#define CHECKSUM_GEN_ICMP               1
#endif

/**
 * CHECKSUM_CHECK_IP==1: Check checksums in software for incoming IP packets.
 */
#ifndef CHECKSUM_CHECK_IP
#define CHECKSUM_CHECK_IP               1
#endif

/**
 * CHECKSUM_CHECK_UDP==1: Check checksums in software for incoming UDP packets.
 */
#ifndef CHECKSUM_CHECK_UDP
#define CHECKSUM_CHECK_UDP              1
#endif
```

```
/**
 * CHECKSUM_CHECK_TCP==1: Check checksums in software for incoming TCP packets.
 */
#ifndef CHECKSUM_CHECK_TCP
#define CHECKSUM_CHECK_TCP              1
#endif

/**
 * LWIP_CHECKSUM_ON_COPY==1: Calculate checksum when copying data from
 * application buffers to pbufs.
 */
#ifndef LWIP_CHECKSUM_ON_COPY
#define LWIP_CHECKSUM_ON_COPY           0
#endif

/*
   ------------------------------------
   ---------- Hook options ---------------
   ------------------------------------
*/

/* Hooks are undefined by default, define them to a function if you need them. */

/**
 * LWIP_HOOK_IP4_INPUT(pbuf, input_netif):
 * - called from ip_input() (IPv4)
 * - pbuf: received struct pbuf passed to ip_input()
 * - input_netif: struct netif on which the packet has been received
 * Return values:
 * - 0: Hook has not consumed the packet, packet is processed as normal
 * - != 0: Hook has consumed the packet.
 * If the hook consumed the packet, 'pbuf' is in the responsibility of the hook
 * (i.e. free it when done).
 */

/**
 * LWIP_HOOK_IP4_ROUTE(dest):
 * - called from ip_route() (IPv4)
 * - dest: destination IPv4 address
 * Returns the destination netif or NULL if no destination netif is found. In
 * that case, ip_route() continues as normal.
 */

/*
   ------------------------------------
   ---------- Debugging options ----------
   ------------------------------------
*/
/**
 * LWIP_DBG_MIN_LEVEL: After masking, the value of the debug is
 * compared against this value. If it is smaller, then debugging
 * messages are written.
 */
#ifndef LWIP_DBG_MIN_LEVEL
#define LWIP_DBG_MIN_LEVEL              LWIP_DBG_LEVEL_ALL
#endif

/**
 * LWIP_DBG_TYPES_ON: A mask that can be used to globally enable/disable
 * debug messages of certain types.
 */
#ifndef LWIP_DBG_TYPES_ON
#define LWIP_DBG_TYPES_ON               LWIP_DBG_ON
#endif

/**
 * ETHARP_DEBUG: Enable debugging in etharp.c.
 */
#ifndef ETHARP_DEBUG
#define ETHARP_DEBUG                    LWIP_DBG_OFF
#endif

/**
 * NETIF_DEBUG: Enable debugging in netif.c.
 */
#ifndef NETIF_DEBUG
#define NETIF_DEBUG                     LWIP_DBG_OFF
#endif

/**
 * PBUF_DEBUG: Enable debugging in pbuf.c.
 */
#ifndef PBUF_DEBUG
#define PBUF_DEBUG                      LWIP_DBG_OFF
#endif
```

```
/**
 * API_LIB_DEBUG: Enable debugging in api_lib.c.
 */
#ifndef API_LIB_DEBUG
#define API_LIB_DEBUG                    LWIP_DBG_OFF
#endif

/**
 * API_MSG_DEBUG: Enable debugging in api_msg.c.
 */
#ifndef API_MSG_DEBUG
#define API_MSG_DEBUG                    LWIP_DBG_OFF
#endif

/**
 * SOCKETS_DEBUG: Enable debugging in sockets.c.
 */
#ifndef SOCKETS_DEBUG
#define SOCKETS_DEBUG                    LWIP_DBG_OFF
#endif

/**
 * ICMP_DEBUG: Enable debugging in icmp.c.
 */
#ifndef ICMP_DEBUG
#define ICMP_DEBUG                       LWIP_DBG_OFF
#endif

/**
 * IGMP_DEBUG: Enable debugging in igmp.c.
 */
#ifndef IGMP_DEBUG
#define IGMP_DEBUG                       LWIP_DBG_OFF
#endif

/**
 * INET_DEBUG: Enable debugging in inet.c.
 */
#ifndef INET_DEBUG
#define INET_DEBUG                       LWIP_DBG_OFF
#endif

/**
 * IP_DEBUG: Enable debugging for IP.
 */
#ifndef IP_DEBUG
#define IP_DEBUG                         LWIP_DBG_OFF
#endif

/**
 * IP_REASS_DEBUG: Enable debugging in ip_frag.c for both frag & reass.
 */
#ifndef IP_REASS_DEBUG
#define IP_REASS_DEBUG                   LWIP_DBG_OFF
#endif

/**
 * RAW_DEBUG: Enable debugging in raw.c.
 */
#ifndef RAW_DEBUG
#define RAW_DEBUG                        LWIP_DBG_OFF
#endif

/**
 * MEM_DEBUG: Enable debugging in mem.c.
 */
#ifndef MEM_DEBUG
#define MEM_DEBUG                        LWIP_DBG_OFF
#endif

/**
 * MEMP_DEBUG: Enable debugging in memp.c.
 */
#ifndef MEMP_DEBUG
#define MEMP_DEBUG                       LWIP_DBG_OFF
#endif

/**
 * SYS_DEBUG: Enable debugging in sys.c.
 */
#ifndef SYS_DEBUG
#define SYS_DEBUG                        LWIP_DBG_OFF
#endif

/**
```

```
 * TIMERS_DEBUG: Enable debugging in timers.c.
 */
#ifndef TIMERS_DEBUG
#define TIMERS_DEBUG                    LWIP_DBG_OFF
#endif

/**
 * TCP_DEBUG: Enable debugging for TCP.
 */
#ifndef TCP_DEBUG
#define TCP_DEBUG                       LWIP_DBG_OFF
#endif

/**
 * TCP_INPUT_DEBUG: Enable debugging in tcp_in.c for incoming debug.
 */
#ifndef TCP_INPUT_DEBUG
#define TCP_INPUT_DEBUG                 LWIP_DBG_OFF
#endif

/**
 * TCP_FR_DEBUG: Enable debugging in tcp_in.c for fast retransmit.
 */
#ifndef TCP_FR_DEBUG
#define TCP_FR_DEBUG                    LWIP_DBG_OFF
#endif

/**
 * TCP_RTO_DEBUG: Enable debugging in TCP for retransmit
 * timeout.
 */
#ifndef TCP_RTO_DEBUG
#define TCP_RTO_DEBUG                   LWIP_DBG_OFF
#endif

/**
 * TCP_CWND_DEBUG: Enable debugging for TCP congestion window.
 */
#ifndef TCP_CWND_DEBUG
#define TCP_CWND_DEBUG                  LWIP_DBG_OFF
#endif

/**
 * TCP_WND_DEBUG: Enable debugging in tcp_in.c for window updating.
 */
#ifndef TCP_WND_DEBUG
#define TCP_WND_DEBUG                   LWIP_DBG_OFF
#endif

/**
 * TCP_OUTPUT_DEBUG: Enable debugging in tcp_out.c output functions.
 */
#ifndef TCP_OUTPUT_DEBUG
#define TCP_OUTPUT_DEBUG                LWIP_DBG_OFF
#endif

/**
 * TCP_RST_DEBUG: Enable debugging for TCP with the RST message.
 */
#ifndef TCP_RST_DEBUG
#define TCP_RST_DEBUG                   LWIP_DBG_OFF
#endif

/**
 * TCP_QLEN_DEBUG: Enable debugging for TCP queue lengths.
 */
#ifndef TCP_QLEN_DEBUG
#define TCP_QLEN_DEBUG                  LWIP_DBG_OFF
#endif

/**
 * UDP_DEBUG: Enable debugging in UDP.
 */
#ifndef UDP_DEBUG
#define UDP_DEBUG                       LWIP_DBG_OFF
#endif

/**
 * TCPIP_DEBUG: Enable debugging in tcpip.c.
 */
#ifndef TCPIP_DEBUG
#define TCPIP_DEBUG                     LWIP_DBG_OFF
#endif

/**
 * PPP_DEBUG: Enable debugging for PPP.
```

```
 */
#ifndef PPP_DEBUG
#define PPP_DEBUG                       LWIP_DBG_OFF
#endif

/**
 * SLIP_DEBUG: Enable debugging in slipif.c.
 */
#ifndef SLIP_DEBUG
#define SLIP_DEBUG                      LWIP_DBG_OFF
#endif

/**
 * DHCP_DEBUG: Enable debugging in dhcp.c.
 */
#ifndef DHCP_DEBUG
#define DHCP_DEBUG                      LWIP_DBG_OFF
#endif

/**
 * AUTOIP_DEBUG: Enable debugging in autoip.c.
 */
#ifndef AUTOIP_DEBUG
#define AUTOIP_DEBUG                    LWIP_DBG_OFF
#endif

/**
 * SNMP_MSG_DEBUG: Enable debugging for SNMP messages.
 */
#ifndef SNMP_MSG_DEBUG
#define SNMP_MSG_DEBUG                  LWIP_DBG_OFF
#endif

/**
 * SNMP_MIB_DEBUG: Enable debugging for SNMP MIBs.
 */
#ifndef SNMP_MIB_DEBUG
#define SNMP_MIB_DEBUG                  LWIP_DBG_OFF
#endif

/**
 * DNS_DEBUG: Enable debugging for DNS.
 */
#ifndef DNS_DEBUG
#define DNS_DEBUG                       LWIP_DBG_OFF
#endif

#endif /* __LWIPOPT_H__ */
```

B.13 f4boot/netstream.h

```
#ifndef NETSTREAM_H_
#define NETSTREAM_H_

#include "ch.h"
#include "lwip/api.h"
#include "hal.h"

#define _net_stream_data \
   _base_sequential_stream_data \
   struct netconn * conn; \
   struct netbuf * inbuf; \
   size_t in_offset;

struct NetStreamVMT {
   _base_sequential_stream_methods
};

/**
* @extends BaseSequentialStream
*/
typedef struct {
   const struct NetStreamVMT *vmt;_net_stream_data
} NetStream;

#ifdef __cplusplus
extern "C" {
#endif
void nsObjectInit(NetStream *sp);
void nsStart(NetStream *sp, struct netconn * conn);
#ifdef __cplusplus
}
#endif
```

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

B.14 f4boot/netstream.c

```
#include "netstream.h"
#include "lwip/api.h"

#include "ch.h"
#include "hal.h"

static size_t write(void *ip, const uint8_t *bp, size_t n) {
   NetStream *sp = ip;

   return netconn_write_partly(sp->conn, bp, n, NETCONN_COPY, NULL);
}

static size_t read(void *ip, uint8_t *bp, size_t n) {
   NetStream *sp = ip;
   err_t err;

   /* If last input buffer was completely consumed, wait for a new packet. */
   while (sp->inbuf == NULL) {
      /* Wait for new packet. */
      err = netconn_recv(sp->conn, &sp->inbuf);
      if (err != ERR_OK) {
         /* Connection closed (or any other errors). */
         return 0;
      }
   }

   /* TODO: make this use the return value of netbuf_copy_partial directly. */
   netbuf_copy_partial(sp->inbuf, bp, n, sp->in_offset);
   sp->in_offset += n;

   /* Check if there is more data to read. */
   if (sp->in_offset >= netbuf_len(sp->inbuf)) {
      n -= (sp->in_offset - netbuf_len(sp->inbuf));
      netbuf_delete(sp->inbuf);
      sp->in_offset = 0;
      sp->inbuf = NULL;
   }

   return n;
}

static msg_t put(void *ip, uint8_t b) {
   return (write(ip, &b, 1) == 1 ? Q_OK : Q_RESET);
}

static msg_t get(void *ip) {
   uint8_t b;

   return (read(ip, &b, 1) == 1 ? b : Q_RESET);
}

static const struct NetStreamVMT vmt = { write, read, put, get };

void nsObjectInit(NetStream *sp) {

   sp->vmt = &vmt;
   sp->inbuf = NULL;
   sp->in_offset = 0;
}

void nsStart(NetStream *sp, struct netconn * conn) {

   sp->conn = conn;
}
```

B.15 f4boot/STM32F407xG.ld

```
/*
    ChibiOS - Copyright (C) 2006..2016 Giovanni Di Sirio

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

        http://www.apache.org/licenses/LICENSE-2.0
```

```
    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
*/

/*
 * STM32F407xG memory setup.
 * Note: Use of ram1 and ram2 is mutually exclusive with use of ram0.
 */
MEMORY
{
    flash0  : org = 0x08000000, len = 64k
    flash1  : org = 0x08010000, len = 1M - 64k
    flash2  : org = 0x00000000, len = 64k
    flash3  : org = 0x00000000, len = 0
    flash4  : org = 0x00000000, len = 0
    flash5  : org = 0x00000000, len = 0
    flash6  : org = 0x00000000, len = 0
    flash7  : org = 0x00000000, len = 0
    ram0    : org = 0x20000000, len = 128k    /* SRAM1 + SRAM2 */
    ram1    : org = 0x20000000, len = 112k    /* SRAM1 */
    ram2    : org = 0x2001C000, len = 16k     /* SRAM2 */
    ram3    : org = 0x20010000, len = 64k     /* SRAM1 + SRAM2 - 64k */
    ram4    : org = 0x10000000, len = 64k     /* CCM SRAM */
    ram5    : org = 0x40024000, len = 4k      /* BCKP SRAM */
    ram6    : org = 0x00000000, len = 0
    ram7    : org = 0x20000000, len = 64k
}

/* For each data/text section two region are defined, a virtual region
   and a load region (_LMA suffix).*/

/* Flash region to be used for exception vectors.*/
REGION_ALIAS("VECTORS_FLASH", flash2);
REGION_ALIAS("VECTORS_FLASH_LMA", flash0);

/* Flash region to be used for constructors and destructors.*/
REGION_ALIAS("XTORS_FLASH", flash2);
REGION_ALIAS("XTORS_FLASH_LMA", flash0);

/* Flash region to be used for code text.*/
REGION_ALIAS("TEXT_FLASH", flash2);
REGION_ALIAS("TEXT_FLASH_LMA", flash0);

/* Flash region to be used for read only data.*/
REGION_ALIAS("RODATA_FLASH", flash2);
REGION_ALIAS("RODATA_FLASH_LMA", flash0);

/* Flash region to be used for various.*/
REGION_ALIAS("VARIOUS_FLASH", flash2);
REGION_ALIAS("VARIOUS_FLASH_LMA", flash0);

/* Flash region to be used for RAM(n) initialization data.*/
REGION_ALIAS("RAM_INIT_FLASH_LMA", flash0);

/* RAM region to be used for Main stack. This stack accommodates the processing
   of all exceptions and interrupts.*/
REGION_ALIAS("MAIN_STACK_RAM", ram3);

/* RAM region to be used for the process stack. This is the stack used by
   the main() function.*/
REGION_ALIAS("PROCESS_STACK_RAM", ram3);

/* RAM region to be used for data segment.*/
REGION_ALIAS("DATA_RAM", ram3);
REGION_ALIAS("DATA_RAM_LMA", flash0);

/* RAM region to be used for BSS segment.*/
REGION_ALIAS("BSS_RAM", ram3);

/* RAM region to be used for the default heap.*/
REGION_ALIAS("HEAP_RAM", ram3);

/* Generic rules inclusion.*/
INCLUDE rules.ld
```

B.16 f4boot/shellconf.h

```
#ifndef SHELLCONF_H
#define SHELLCONF_H
```

```
/**
 * @brief   Shell maximum input line length.
 */
#define SHELL_MAX_LINE_LENGTH       96

/**
 * @brief   Shell maximum arguments per command.
 */
#define SHELL_MAX_ARGUMENTS         12

/**
 * @brief   Shell maximum command history.
 */
#define SHELL_MAX_HIST_BUFF         8 * SHELL_MAX_LINE_LENGTH

/**
 * @brief   Enable shell command history
 */
#define SHELL_USE_HISTORY           TRUE

/**
 * @brief   Enable shell command completion
 */
#define SHELL_USE_COMPLETION        FALSE

/**
 * @brief   Shell Maximum Completions (Set to max commands with common prefix)
 */
#define SHELL_MAX_COMPLETIONS       8

/**
 * @brief   Enable shell escape sequence processing
 */
#define SHELL_USE_ESC_SEQ           TRUE

/*===========================================================================*/
/* Shell builtin command settings.                                           */
/*===========================================================================*/

#define SHELL_CMD_EXIT_ENABLED              TRUE

#define SHELL_CMD_INFO_ENABLED              TRUE

#define SHELL_CMD_ECHO_ENABLED              TRUE

#define SHELL_CMD_SYSTIME_ENABLED           TRUE

#define SHELL_CMD_MEM_ENABLED               TRUE

#define SHELL_CMD_THREADS_ENABLED           TRUE

#define SHELL_CMD_TEST_ENABLED              FALSE

#define SHELL_CMD_TEST_WA_SIZE              THD_WORKING_AREA_SIZE(256)

#endif /* SHELLCONF_H */
```

B.17 f4boot/shellconf.c

```
/*
 * Shell commands and helper functions.
 *
 * Copyright (C) 2015..2017 Adam Shea
 *
    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

        http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
 */

#include <stdlib.h>

#include "ch.h"
#include "hal.h"
#include "shell.h"
```

```
#include "shell_cmd.h"
#include "chprintf.h"

#include "rtclib.h"
#include "nvram.h"

/*
 * Old-school peek function.
 *
 * Call as
 * peek <address>
 *
 * <address> is the address of the 32-bit int to be read. The lower 2 bits
 * will be masked off in order to ensure alignment.
 *
 * Returns the contents of *<address> as a hex-formatted unsigned integer
 */
static void shell_peek(BaseSequentialStream *chp, int argc, char *argv[]) {
  uint32_t * p;
  uint32_t n = 1;

  if ((argc != 1) && (argc != 2)) {
    shellUsage(chp, "peek␣<address>␣[count]");
    return;
  }
  p = (uint32_t *)(strtoul(argv[0], NULL, 0) & ~0x3);

  if (argc == 2) {
    n = strtoul(argv[1], NULL, 0);
  }

  while (n){
    chprintf(chp, "0x%08x:␣", p);
    switch (n) {
      default:
        chprintf(chp, "0x%08x␣", *(p++));
        n--;
        /* FALLTHROUGH */
      case 3:
        chprintf(chp, "0x%08x␣", *(p++));
        n--;
        /* FALLTHROUGH */
      case 2:
        chprintf(chp, "0x%08x␣", *(p++));
        n--;
        /* FALLTHROUGH */
      case 1:
        chprintf(chp, "0x%08x"SHELL_NEWLINE_STR, *(p++));
        n--;
    }
  }
  return;
}

/*
 * Old-School poke function.
 *
 * Call as
 * poke <address> <data>
 *
 * <address> is the address of the 32-bit int to be written. The lower 2 bits
 * will be masked off in order to ensure alignment.
 *
 * <data> is 32 bits of data to be written to <address>.
 */
static void shell_poke(BaseSequentialStream *chp, int argc, char *argv[]) {
  uint32_t * p;

  if (argc < 2) {
    shellUsage(chp, "poke␣<address>␣<data>␣[data]␣[data]␣...");
    return;
  }
  p = (uint32_t *)(strtoul(*(argv++), NULL, 0) & ~0x3);

  while (--argc) {
    *(p++) = strtoul(*(argv++), NULL, 0);
  }
  return;
}

static void shell_date(BaseSequentialStream *chp, int argc, char *argv[]) {
  RTCDateTime timenow;
  uint32_t h,m,s,ms;

  if (argc > 6) {
    shellUsage(chp, "date␣[year]␣[month]␣[day]␣[hour]␣[minute]␣[second]");
```

```
    return;
  }

  rtcGetTime(&RTCD1, &timenow);

  ms = timenow.millisecond;
  h = ms / (1000 * 60 * 60);
  ms -= h * 1000 * 60 * 60;
  m = ms / (1000 * 60);
  ms -= m * 1000 * 60;
  s = ms / 1000;
  ms -= s * 1000;

  /* Setting year */
  switch (argc) {
    case 6: /* Set seconds */
      s = strtoul(argv[5], NULL, 0);
      ms = 0;
      /* FALLTHROUGH */
    case 5: /* Set minute */
      m = strtoul(argv[4], NULL, 0);
      /* FALLTHROUGH */
    case 4: /* Set hours */
      h = strtoul(argv[3], NULL, 0);
      /* all above will have fallen through to here.  We don't reset
       * time of day if only setting date */
      timenow.millisecond = (((h * 60) + m) * 60 + s) * 1000 + ms;
      /* FALLTHROUGH */
    case 3: /* Set day */
      timenow.day = strtoul(argv[2], NULL, 0);
      /* FALLTHROUGH */
    case 2: /* set month */
      timenow.month = strtoul(argv[1], NULL, 0);
      /* FALLTHROUGH */
    case 1: /* set year */
      timenow.year = strtol(argv[0], NULL, 0) - RTC_BASE_YEAR;
      /* We've changed time so we need to set the clock */
      rtcSetTime(&RTCD1, &timenow);
  }

  if (!rtcIsInitialized(&RTCD1)) {
    chprintf(chp, "RTC not initialized!"SHELL_NEWLINE_STR);
  }

  chprintf(chp, "%U-%U-%U %02U:%02U:%02U.%03U"SHELL_NEWLINE_STR, \
      RTC_BASE_YEAR+timenow.year, \
      timenow.month, \
      timenow.day, \
      h, m, s, ms);
  return;
}

static void shell_rtctrim(BaseSequentialStream *chp, int argc, char *argv[]) {

  if (argc > 1) {
    shellUsage(chp, "rtctrim [new_trim_ppb]");
    return;
  }

  if (argc == 1){
    rtcSetTrim(&RTCD1, strtol(argv[0], NULL, 0));
  }

  chprintf(chp, "RTC trim is %d ppb"SHELL_NEWLINE_STR, rtcGetTrim(&RTCD1));
}

/*
 * CRC32 configuration
 */
static const CRCConfig crc32_config = {
  .poly_size         = 32,
  .poly              = 0x04C11DB7,
  .initial_val       = 0xFFFFFFFF,
  .final_val         = 0xFFFFFFFF,
  .reflect_data      = 1,
  .reflect_remainder = 1
};

static void shell_docrc(BaseSequentialStream *chp, int argc, char *argv[]) {
  uint32_t result;

  if (argc != 2) {
    shellUsage(chp, "docrc <start_address> <words>");
    return;
  }
```

```
  crcAcquireUnit(&CRCD1);            /* Acquire ownership of the bus.     */
  crcStart(&CRCD1, &crc32_config);   /* Activate CRC driver               */
  crcReset(&CRCD1);

  result = crcCalc(&CRCD1, strtol(argv[1], NULL, 0) * 4, \
                   (void *) strtol(argv[0],NULL, 0));

  crcStop(&CRCD1);                   /* Deactive CRC driver               */
  crcReleaseUnit(&CRCD1);            /* Acquire ownership of the bus.     */

  chprintf(chp, "%0x"SHELL_NEWLINE_STR, result);
  return;
}

static void shell_reboot(BaseSequentialStream *chp, int argc, char *argv[]) {

  (void) argc;
  (void) argv;

  chprintf(chp, "Rebooting...\n\x04");
  chThdSleepMilliseconds(2000);

  /* Reset all interrupts to default */
  chSysDisable();

  /* Clear pending interrupts just to be on the save side */
  SCB->ICSR = SCB_ICSR_PENDSVCLR_Msk;

  /* Disable all interrupts */
  for(int i = 0; i < 8; ++i)
    NVIC->ICER[i] = NVIC->IABR[i];

  /* Reset boot point to flash */
  SYSCFG->MEMRMP = 0x00;
  SCB->VTOR = FLASH_BASE;

  NVIC_SystemReset();

}

const ShellCommand commands[] = {
  {"peek", shell_peek},
  {"poke", shell_poke},
  {"date", shell_date},
  {"rtctrim", shell_rtctrim},
  {"docrc", shell_docrc},
  {"reboot", shell_reboot},
  {"getnvram", shell_getnvram},
  {"setnvram", shell_setnvram},
  {NULL, NULL}
};

/* vim: set sw=2 sts=2 expandtab : */
```

B.18 f4boot/flash.h

```
#ifndef FLASH_H
#define FLASH_H

#include <ch.h>
#include <hal.h>
#include <stdint.h>

/**
 * @brief Number of sectors in the flash memory.
 */
#if !defined(FLASH_SECTOR_COUNT) || defined(__DOXYGEN__)
#define FLASH_SECTOR_COUNT 12
#endif

/* Error codes */

/** @brief Flash operation successful */
#define FLASH_RETURN_SUCCESS HAL_SUCCESS

/** @brief Flash operation error because of denied access, corrupted memory.*/
#define FLASH_RETURN_NO_PERMISSION -1

/** @brief Flash operation error because of bad flash, corrupted memory */
#define FLASH_RETURN_BAD_FLASH -11

#ifdef __cplusplus
```

```
extern "C" {
#endif

/**
 * @brief Maximum program/erase parallelism
 *
 * FLASH_CR_PSIZE_MASK is the mask to configure the parallelism value.
 * FLASH_CR_PSIZE_VALUE is the parallelism value suitable for the voltage range.
 *
 * PSIZE(1:0) is defined as:
 * 00 to program 8 bits per step
 * 01 to program 16 bits per step
 * 10 to program 32 bits per step
 * 11 to program 64 bits per step
 */
// Warning, flashdata_t must be unsigned!!!
#if defined(STM32F4XX) || defined(__DOXYGEN__)
#define FLASH_CR_PSIZE_MASK            FLASH_CR_PSIZE_0 | FLASH_CR_PSIZE_1
#if ((STM32_VDD >= 270) && (STM32_VDD <= 360)) || defined(__DOXYGEN__)
#define FLASH_CR_PSIZE_VALUE           FLASH_CR_PSIZE_1
typedef uint32_t flashdata_t;
#elif (STM32_VDD >= 240) && (STM32_VDD < 270)
#define FLASH_CR_PSIZE_VALUE           FLASH_CR_PSIZE_0
typedef uint16_t flashdata_t;
#elif (STM32_VDD >= 210) && (STM32_VDD < 240)
#define FLASH_CR_PSIZE_VALUE           FLASH_CR_PSIZE_0
typedef uint16_t flashdata_t;
#elif (STM32_VDD >= 180) && (STM32_VDD < 210)
#define FLASH_CR_PSIZE_VALUE           ((uint32_t)0x00000000)
typedef uint8_t flashdata_t;
#else
#error "invalid VDD voltage specified"
#endif
#endif /* defined(STM32F4XX) */

/** @brief Address in the flash memory */
typedef uintptr_t flashaddr_t;

/** @brief Index of a sector */
typedef uint8_t flashsector_t;

#define flashLocked() (FLASH->CR & FLASH_CR_LOCK)

/**
 * @brief Get the size of @p sector.
 * @return @p sector size in bytes.
 */
size_t flashSectorSize(flashsector_t sector);

/**
 * @brief Get the beginning address of @p sector.
 * @param sector Sector to retrieve the beginning address of.
 * @return First address (inclusive) of @p sector.
 */
flashaddr_t flashSectorBegin(flashsector_t sector);

/**
 * @brief Get the end address of @p sector.
 * @param sector Sector to retrieve the end address of.
 * @return End address (exclusive) of @p sector (i.e. beginning address of the next sector).
 */
flashaddr_t flashSectorEnd(flashsector_t sector);

/**
 * @brief Get the sector containing @p address.
 * @warning @p address must be in the flash addresses range.
 * @param address Address to be searched for.
 * @return Sector containing @p address.
 */
flashsector_t flashSectorAt(flashaddr_t address);

/**
 * @brief Erase the flash @p sector.
 * @details The sector is checked for errors after erase.
 * @note The sector is deleted regardless of its current state.
 *
 * @param sector Sector which is going to be erased.
 * @return FLASH_RETURN_SUCCESS          No error erasing the sector.
 * @return FLASH_RETURN_BAD_FLASH        Flash cell error.
 * @return FLASH_RETURN_NO_PERMISSION    Access denied.
 */
int flashSectorErase(flashsector_t sector);

/**
 * @brief Erase the sectors containing the span of @p size bytes starting at @p address.
 *
```

```
 * @warning If @p address doesn't match the beginning of a sector, the
 * data contained between the beginning of the sector and @p address will
 * be erased too. The same applies for data contained at @p address + @p size
 * up to the end of the sector.
 *
 * @param address Starting address of the span in flash memory.
 * @param size Size of the span in bytes.
 * @return FLASH_RETURN_SUCCESS          No error erasing the flash memory.
 * @return FLASH_RETURN_BAD_FLASH        Flash cell error.
 * @return FLASH_RETURN_NO_PERMISSION    Access denied.
 */
int flashErase(flashaddr_t address, size_t size);

/**
 * @brief Check if the @p size bytes of flash memory starting at @p address are erased.
 * @note If the memory is erased, one can write data into it safely.
 * @param address First address in flash memory to be checked.
 * @param size Size of the memory space to be checked in bytes.
 * @return TRUE Memory is already erased.
 * @return FALSE Memory is not erased.
 */
bool flashIsErased(flashaddr_t address, size_t size);

/**
 * @brief Check if the data in @p buffer are identical to the one in flash memory.
 * @param address First address in flash memory to be checked.
 * @param buffer Buffer containing the data to compare.
 * @param size Size of @p buffer in bytes.
 * @return TRUE if the flash memory and the buffer contain identical data.
 * @return FALSE if the flash memory and the buffer don't contain identical data.
 */
bool flashCompare(flashaddr_t address, const char* buffer, size_t size);

/**
 * @brief Copy data from the flash memory to a @p buffer.
 * @warning The @p buffer must be at least @p size bytes long.
 * @param address First address of the flash memory to be copied.
 * @param buffer Buffer to copy to.
 * @param size Size of the data to be copied in bytes.
 * @return FLASH_RETURN_SUCCESS if successfully copied.
 */
int flashRead(flashaddr_t address, char* buffer, size_t size);

/**
 * @brief Copy data from a @p buffer to the flash memory.
 * @warning The flash memory area receiving the data must be erased.
 * @warning The @p buffer must be at least @p size bytes long.
 * @param address First address in the flash memory where to copy the data to.
 * @param buffer Buffer containing the data to copy.
 * @param size Size of the data to be copied in bytes.
 * @return FLASH_RETURN_SUCCESS          No error.
 * @return FLASH_RETURN_NO_PERMISSION    Access denied.
 */
int flashWrite(flashaddr_t address, const char* buffer, size_t size);

#ifdef __cplusplus
}
#endif

#endif /* FLASH_H */
```

B.19 f4boot/flash.c

```
#include "flash.h"
#include <string.h>

size_t flashSectorSize(flashsector_t sector)
{
    if (sector <= 3)
        return 16 * 1024;
    else if (sector == 4)
        return 64 * 1024;
    else if (sector >= 5 && sector <= 11)
        return 128 * 1024;
    return 0;
}

flashaddr_t flashSectorBegin(flashsector_t sector)
{
    flashaddr_t address = FLASH_BASE;
    while (sector > 0)
    {
```

```
        --sector;
        address += flashSectorSize(sector);
    }
    return address;
}

flashaddr_t flashSectorEnd(flashsector_t sector)
{
    return flashSectorBegin(sector + 1);
}

flashsector_t flashSectorAt(flashaddr_t address)
{
    flashsector_t sector = 0;
    while (address >= flashSectorEnd(sector))
        ++sector;
    return sector;
}

/**
 * @brief Wait for the flash operation to finish.
 */
#define flashWaitWhileBusy() { while (FLASH->SR & FLASH_SR_BSY) {} }

/**
 * @brief Unlock the flash memory for write access.
 * @return HAL_SUCCESS  Unlock was successful.
 * @return HAL_FAILED     Unlock failed.
 */
static bool flashUnlock(void)
{
    /* Check if unlock is really needed */
    if (!flashLocked())
        return HAL_SUCCESS;

    /* Write magic unlock sequence */
    FLASH->KEYR = 0x45670123;
    FLASH->KEYR = 0xCDEF89AB;

    /* Check if unlock was successful */
    if (FLASH->CR & FLASH_CR_LOCK)
        return HAL_FAILED;
    return HAL_SUCCESS;
}

/**
 * @brief Lock the flash memory for write access.
 */
#define flashLock() { FLASH->CR |= FLASH_CR_LOCK; }

int flashSectorErase(flashsector_t sector)
{
    /* Unlock flash for write access */
    if(flashUnlock() == HAL_FAILED)
        return FLASH_RETURN_NO_PERMISSION;

    /* Wait for any busy flags. */
    flashWaitWhileBusy();

    /* Setup parallelism before any program/erase */
    FLASH->CR &= ~FLASH_CR_PSIZE_MASK;
    FLASH->CR |= FLASH_CR_PSIZE_VALUE;

    /* Start deletion of sector.
     * SNB(3:1) is defined as:
     * 0000 sector 0
     * 0001 sector 1
     * ...
     * 1011 sector 11
     * others not allowed */
    FLASH->CR &= ~(FLASH_CR_SNB_0 | FLASH_CR_SNB_1 | FLASH_CR_SNB_2 | FLASH_CR_SNB_3);
    if (sector & 0x1) FLASH->CR |= FLASH_CR_SNB_0;
    if (sector & 0x2) FLASH->CR |= FLASH_CR_SNB_1;
    if (sector & 0x4) FLASH->CR |= FLASH_CR_SNB_2;
    if (sector & 0x8) FLASH->CR |= FLASH_CR_SNB_3;
    FLASH->CR |= FLASH_CR_SER;
    FLASH->CR |= FLASH_CR_STRT;

    /* Wait until it's finished. */
    flashWaitWhileBusy();

    /* Sector erase flag does not clear automatically. */
    FLASH->CR &= ~FLASH_CR_SER;

    /* Lock flash again */
    flashLock();
```

```
    /* Check deleted sector for errors */
    if (flashIsErased(flashSectorBegin(sector), flashSectorSize(sector)) == FALSE)
        return FLASH_RETURN_BAD_FLASH;  /* Sector is not empty despite the erase cycle! */

    /* Successfully deleted sector */
    return FLASH_RETURN_SUCCESS;
}

int flashErase(flashaddr_t address, size_t size)
{
    while (size > 0)
    {
        flashsector_t sector = flashSectorAt(address);
        int err = flashSectorErase(sector);
        if (err != FLASH_RETURN_SUCCESS)
            return err;
        address = flashSectorEnd(sector);
        size_t sector_size = flashSectorSize(sector);
        if (sector_size >= size)
            break;
        else
            size -= sector_size;
    }

    return FLASH_RETURN_SUCCESS;
}

bool flashIsErased(flashaddr_t address, size_t size)
{
    /* Check for default set bits in the flash memory
     * For efficiency, compare flashdata_t values as much as possible,
     * then, fallback to byte per byte comparison. */
    while (size >= sizeof(flashdata_t))
    {
        if (*(volatile flashdata_t*)address != (flashdata_t)(-1)) // flashdata_t being unsigned, -1 is 0xFF..FF
            return FALSE;
        address += sizeof(flashdata_t);
        size -= sizeof(flashdata_t);
    }
    while (size > 0)
    {
        if (*(char*)address != 0xff)
            return FALSE;
        ++address;
        --size;
    }

    return TRUE;
}

bool flashCompare(flashaddr_t address, const char* buffer, size_t size)
{
    /* For efficiency, compare flashdata_t values as much as possible,
     * then, fallback to byte per byte comparison. */
    while (size >= sizeof(flashdata_t))
    {
        if (*(volatile flashdata_t*)address != *(flashdata_t*)buffer)
            return FALSE;
        address += sizeof(flashdata_t);
        buffer += sizeof(flashdata_t);
        size -= sizeof(flashdata_t);
    }
    while (size > 0)
    {
        if (*(volatile char*)address != *buffer)
            return FALSE;
        ++address;
        ++buffer;
        --size;
    }

    return TRUE;
}

int flashRead(flashaddr_t address, char* buffer, size_t size)
{
    memcpy(buffer, (char*)address, size);
    return FLASH_RETURN_SUCCESS;
}

static void flashWriteData(flashaddr_t address, const flashdata_t data)
{
    /* Enter flash programming mode */
    FLASH->CR |= FLASH_CR_PG;
```

```c
    /* Write the data */
    *(flashdata_t*)address = data;

    /* Wait for completion */
    flashWaitWhileBusy();

    /* Exit flash programming mode */
    FLASH->CR &= ~FLASH_CR_PG;
}

int flashWrite(flashaddr_t address, const char* buffer, size_t size)
{
    /* Unlock flash for write access */
    if(flashUnlock() == HAL_FAILED)
        return FLASH_RETURN_NO_PERMISSION;

    /* Wait for any busy flags */
    flashWaitWhileBusy();

    /* Setup parallelism before any program/erase */
    FLASH->CR &= ~FLASH_CR_PSIZE_MASK;
    FLASH->CR |= FLASH_CR_PSIZE_VALUE;

    /* Check if the flash address is correctly aligned */
    size_t alignOffset = address % sizeof(flashdata_t);
    if (alignOffset != 0)
    {
        /* Not aligned, thus we have to read the data in flash already present
         * and update them with buffer's data */

        /* Align the flash address correctly */
        flashaddr_t alignedFlashAddress = address - alignOffset;

        /* Read already present data */
        flashdata_t tmp = *(volatile flashdata_t*)alignedFlashAddress;

        /* Compute how much bytes one must update in the data read */
        size_t chunkSize = sizeof(flashdata_t) - alignOffset;
        if (chunkSize > size)
            chunkSize = size; // this happens when both address and address + size are not aligned

        /* Update the read data with buffer's data */
        memcpy((char*)&tmp + alignOffset, buffer, chunkSize);

        /* Write the new data in flash */
        flashWriteData(alignedFlashAddress, tmp);

        /* Advance */
        address += chunkSize;
        buffer += chunkSize;
        size -= chunkSize;
    }

    /* Now, address is correctly aligned. One can copy data directly from
     * buffer's data to flash memory until the size of the data remaining to be
     * copied requires special treatment. */
    while (size >= sizeof(flashdata_t))
    {
        flashWriteData(address, *(const flashdata_t*)buffer);
        address += sizeof(flashdata_t);
        buffer += sizeof(flashdata_t);
        size -= sizeof(flashdata_t);
    }

    /* Now, address is correctly aligned, but the remaining data are to
     * small to fill a entier flashdata_t. Thus, one must read data already
     * in flash and update them with buffer's data before writing an entire
     * flashdata_t to flash memory. */
    if (size > 0)
    {
        flashdata_t tmp = *(volatile flashdata_t*)address;
        memcpy(&tmp, buffer, size);
        flashWriteData(address, tmp);
    }

    /* Lock flash again */
    flashLock();

    return FLASH_RETURN_SUCCESS;
}
```

APPENDIX
Recommended Controller Software Improvement Tasks

In order to complete the control firmware, the following tasks must be accomplished:

1. Finish the full-order controller thread code. It is currently missing state transition code, precharge monitoring, bumpless controller state transitions, and may have a few sign errors.

2. Upgrade the PC-side interface software to include live monitoring of the feedback, enumeration of attached drives, and command generation.

3. Develop software for the isolated side MCU to allow the isolated ADC inputs and serial port to be used for control inputs.

4. Develop a browser accessible interface for basic controller commands for use when comissioning a drive module.

5. Develop tooling to create *nvram* images for various drive configurations. Presently the *nvram* settings have been hand-crafted for each drive module. The controllers publish a JSON description of the file format, so a simple GUI configuration program should be straightforward to write.

Bibliography

[1] B. M. Conlon, T. Blohm, M. Harpster, A. Holmes, M. Palardy, S. Tarnowsky, and L. Zhou, “The next generation ‘voltec’ extended range ev propulsion system,” *SAE International Journal of Alternative Powertrains*, vol. 4, no. 2, Apr. 2015. DOI: 10.4271/2015-01-1152.

[2] *Silicon carbide power mosfet 1200 v, 12 a, 520 mΩ*, Rev 2, SCT10N120, STMicroelectronics, May 2016. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/datasheet/group3/5e/ac/d2/a0/5d/48/4e/a3/DM00170798/files/DM00170798.pdf/jcr:content/translations/en.DM00170798.pdf.

[3] M. Villani, M. Tursini, G. Fabri, and L. Castellini, “Fault-tolerant pm brushless DC drive for aerospace application,” in *Electrical Machines (ICEM), 2010 XIX International Conference on*, Sep. 2010, pp. 1–7. DOI: 10.1109/ICELMACH.2010.5608295.

[4] R. Argile, B. Mecrow, D. Atkinson, A. Jack, and P. Sangha, “Reliability analysis of fault tolerant drive topologies,” in *Power Electronics, Machines and Drives, 2008. PEMD 2008. 4th IET Conference on*, Apr. 2008, pp. 11–15.

[5] B. J. Sykora, “Development of a demonstrator model of an integrated modular motor drive,” Master’s thesis, University of Wisconsin - Madison, 2008.

[6] N. R. Brown, T. Jahns, and R. Lorenz, “Power converter design for an integrated modular motor drive,” in *Industry Applications Conference, 2007. 42nd IAS Annual Meeting. Conference Record of the 2007 IEEE*, Sep. 2007, pp. 1322–1328. DOI: 10.1109/07IAS.2007.205.

[7] J. Wang, Y. Li, and Y. Han, “Integrated modular motor drive design with gan power fets,” *IEEE Transactions on Industry Applications*, vol. 51, no. 4, pp. 3198–3207, Jul. 2015, ISSN: 0093-9994. DOI: 10.1109/TIA.2015.2413380.

[8] M. Hennen, M. Niessen, C. Heyers, H. Brauer, and R. De Doncker, “Development and control of an integrated and distributed inverter for a fault tolerant five-phase switched reluctance traction drive,” *Power Electronics, IEEE Transactions on*, vol. 27, no. 2, pp. 547–554, Feb. 2012, ISSN: 0885-8993. DOI: 10.1109/TPEL.2011.2132763.

[9] T. M. Jahns, “Improved reliability in solid-state drives for large asynchronous ac machines by means of multiple independent phase-drive units.,” PhD thesis, Massachusetts Institute of Technology, 1978. [Online]. Available: http://hdl.handle.net/1721.1/16184.

[10] N. R. Brown, “Power converter design for an integrated modular motor drive,” Master’s thesis, University of Wisconsin - Madison, 2007.

[11] A. Shea and T. Jahns, “Hardware integration for an integrated modular motor drive including distributed control,” in *Energy Conversion Congress and Exposition (ECCE), 2014 IEEE*, Sep. 2014, pp. 4881–4887. DOI: 10.1109/ECCE.2014.6954070.

[12] C. Ditmanson and S. Kolb, “A distributed and fault-tolerant control system for a new modular wind turbine converter,” in *Power Electronics and Applications (EPE’14-ECCE Europe), 2014 16th European Conference on*, Aug. 2014, pp. 1–8. DOI: 10.1109/EPE.2014.6910973.

[13] P. Wikstrom, L. Terens, and H. Kobi, "Reliability, availability, and maintainability of high-power variable-speed drive systems," *Industry Applications, IEEE Transactions on*, vol. 36, no. 1, pp. 231–241, Jan. 2000, ISSN: 0093-9994. DOI: 10.1109/28.821821.

[14] M. Villani, M. Tursini, G. Fabri, and L. Castellini, "High reliability permanent magnet brushless motor drive for aircraft application," *Industrial Electronics, IEEE Transactions on*, vol. 59, no. 5, pp. 2073–2081, May 2012, ISSN: 0278-0046. DOI: 10.1109/TIE.2011.2160514.

[15] N. Bianchi and S. Bolognani, "Fault -tolerant pm motors in automotive applications," in *Vehicle Power and Propulsion, 2005 IEEE Conference*, Sep. 2005, pp. 747–755. DOI: 10.1109/VPPC.2005.1554642.

[16] B. Bahrani, A. Rufer, S. Kenzelmann, and L. Lopes, "Vector control of single-phase voltage-source converters based on fictive-axis emulation," *Industry Applications, IEEE Transactions on*, vol. 47, no. 2, pp. 831–840, Mar. 2011, ISSN: 0093-9994. DOI: 10.1109/TIA.2010.2101992.

[17] X. Tieyan and L. Yaohua, "The zero sequence circulating current suppression based on virtual impedance," in *Power Engineering and Automation Conference (PEAM), 2012 IEEE*, Sep. 2012, pp. 1–7. DOI: 10.1109/PEAM.2012.6612520.

[18] G. Bisheimer, C. De Angelo, J. Solsona, and G. Garcia, "Sensorless pmsm drive with tolerance to current sensor faults," in *Industrial Electronics, 2008. IECON 2008. 34th Annual Conference of IEEE*, Nov. 2008, pp. 1379–1384. DOI: 10.1109/IECON.2008.4758155.

[19] R. Cardenas, M. Diaz, F. Rojas, and J. Clare, "Fast convergence delayed signal cancellation method for sequence component separation," *Power Delivery, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014, ISSN: 0885-8977. DOI: 10.1109/TPWRD.2014.2373038.

[20] F. Grouz, L. Sbita, and M. Boussak, "Current sensors faults detection, isolation and control reconfiguration for pmsm drives," in *Electrical Engineering and Software Applications (ICEESA), 2013 International Conference on*, Mar. 2013, pp. 1–6. DOI: 10.1109/ICEESA.2013.6578414.

[21] T. Yamaguchi, Y. Tadano, and N. Hoshi, "Compensation of the current measurement error with periodic disturbance observer for motor drive," in *Power Electronics Conference (IPEC-Hiroshima 2014 - ECCE-ASIA), 2014 International*, May 2014, pp. 1242–1246. DOI: 10.1109/IPEC.2014.6869745.

[22] M. Harke, "Fundamental sensing issues in motor control," PhD thesis, University of Wisconsin, 2006.

[23] D.-W. Chung, S.-K. Sul, and D.-C. Lee, "Analysis and compensation of current measurement error in vector controlled AC motor drives," in *Industry Applications Conference, 1996. Thirty-First IAS Annual Meeting, IAS '96., Conference Record of the 1996 IEEE*, vol. 1, Oct. 1996, 388–393 vol.1. DOI: 10.1109/IAS.1996.557053.

[24] S.-H. Hwang, H.-J. Kim, J.-M. Kim, L. Liu, and H. Li, "Compensation of amplitude imbalance and imperfect quadrature in resolver signals for pmsm drives," *Industry Applications, IEEE Transactions on*, vol. 47, no. 1, pp. 134–143, Jan. 2011, ISSN: 0093-9994. DOI: 10.1109/TIA.2010.2091477.

[25] C. Secrest, J. Pointer, M. Buehner, and R. Lorenz, "Improving position sensor accuracy through spatial harmonic decoupling, and sensor scaling, offset, and orthogonality correction using self-commissioning mras-methods," in *Energy Conversion Congress and Exposition (ECCE), 2014 IEEE*, Sep. 2014, pp. 3794–3801. DOI: 10.1109/ECCE.2014.6953917.

[26] P. Schneider, M. Horio, and R. Lorenz, "Integrating giant magneto-resistive (gmr) field detectors for high bandwidth current sensing in power electronic modules," in *Energy Conversion Congress and Exposition (ECCE), 2010 IEEE*, Sep. 2010, pp. 1260–1267. DOI: 10.1109/ECCE.2010.5617820.

[27] P. Schneider and R. Lorenz, "Evaluation of point field sensing in IGBT modules for high bandwidth current measurement," in *Energy Conversion Congress and Exposition (ECCE), 2011 IEEE*, Sep. 2011, pp. 1950–1957. DOI: 10.1109/ECCE.2011.6064025.

[28] H. Che, M. Duran, E. Levi, M. Jones, W. Hew, and N. Rahim, "Post-fault operation of an asymmetrical six-phase induction machine with single and two isolated neutral points," in *Energy Conversion Congress and Exposition (ECCE), 2013 IEEE*, Sep. 2013, pp. 1131–1138. DOI: 10.1109/ECCE.2013.6646832.

[29] P. Schulting, C. van der Broeck, and R. De Doncker, "A generalized control design approach for a repetitive controller on current harmonics," in *Control and Modeling for Power Electronics (COMPEL), 2015 IEEE 16th Workshop on*, Jul. 2015, pp. 1–8. DOI: 10.1109/COMPEL.2015.7236475.

[30] S. Green, D. J. Atkinson, A. G. Jack, B. C. Mecrow, and A. King, "Sensorless operation of a fault tolerant pm drive," *IEE Proceedings - Electric Power Applications*, vol. 150, no. 2, pp. 117–125, Mar. 2003, ISSN: 1350-2352. DOI: 10.1049/ip-epa:20030153.

[31] D. Raca, P. Garcia, D. D. Reigosa, F. Briz, and R. D. Lorenz, "Carrier-signal selection for sensorless control of pm synchronous machines at zero and very low speeds," *IEEE Transactions on Industry Applications*, vol. 46, no. 1, pp. 167–178, Jan. 2010, ISSN: 0093-9994. DOI: 10.1109/TIA.2009.2036551.

[32] L. Wang and R. D. Lorenz, "Rotor position estimation for permanent magnet synchronous motor using saliency-tracking self-sensing method," in *Conference Record of the 2000 IEEE*

Industry Applications Conference. Thirty-Fifth IAS Annual Meeting and World Conference on Industrial Applications of Electrical Energy (Cat. No.00CH37129), vol. 1, 2000, 445–450 vol.1. DOI: 10.1109/IAS.2000.881148.

[33] S. C. Yang, “Saliency-based position estimation of permanent-magnet synchronous machines using square-wave voltage injection with a single current sensor,” *IEEE Transactions on Industry Applications*, vol. 51, no. 2, pp. 1561–1571, Mar. 2015, ISSN: 0093-9994. DOI: 10.1109/TIA.2014.2358796.

[34] L. Chen, G. Götting, S. Dietrich, and I. Hahn, “Self-sensing control of permanent-magnet synchronous machines with multiple saliencies using pulse-voltage-injection,” *IEEE Transactions on Industry Applications*, vol. 52, no. 4, pp. 3480–3491, Jul. 2016, ISSN: 0093-9994. DOI: 10.1109/TIA.2016.2557299.

[35] M. Roetzer, U. Vollmer, and R. M. Kennel, “Demodulation approach for slowly sampled sensorless field-oriented control systems enabling multiple-frequency injections,” *IEEE Transactions on Industry Applications*, vol. 54, no. 1, pp. 732–744, Jan. 2018, ISSN: 0093-9994. DOI: 10.1109/TIA.2017.2757458.

[36] G. A. Wauchope, “A recent development in squirrel-cage induction-motor starters,” *Journal of the Institution of Electrical Engineers-Part I: General*, vol. 91, no. 45, pp. 361–364, Sep. 1944. DOI: 10.1049/ji-1.1944.0090.

[37] B. A. Welchko and J. M. Nagashima, “The influence of topology selection on the design of ev/hev propulsion systems,” *IEEE Power Electronics Letters*, vol. 1, no. 2, pp. 36–40, Jun. 2003, ISSN: 1540-7985. DOI: 10.1109/LPEL.2003.821033.

[38] H. Stemmler and P. Guggenbach, “Configurations of high-power voltage source inverter drives,” in *Proc. Fifth European Conf. Power Electronics and Applications*, Sep. 1993, 7–14

vol.5. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=265146&isnumber=6646.

[39] V. T. Somasekhar, S. Srinivas, and K. K. Kumar, "Effect of zero-vector placement in a dual-inverter fed open-end winding induction-motor drive with a decoupled space-vector PWM strategy," *IEEE Transactions on Industrial Electronics*, vol. 55, no. 6, pp. 2497–2505, Jun. 2008, ISSN: 0278-0046. DOI: 10.1109/TIE.2008.918644.

[40] ——, "Effect of zero-vector placement in a dual-inverter fed open-end winding induction motor drive with alternate sub-hexagonal center PWM switching scheme," *IEEE Transactions on Power Electronics*, vol. 23, no. 3, pp. 1584–1591, May 2008, ISSN: 0885-8993. DOI: 10.1109/TPEL.2008.921170.

[41] B. A. Welchko, "A double-ended inverter system for the combined propulsion and energy management functions in hybrid vehicles with energy storage," in *31st Annual Conference of IEEE Industrial Electronics Society, 2005. IECON 2005.*, Nov. 2005, p. 6. DOI: 10.1109/IECON.2005.1569110.

[42] D. Pan, K. K. Huh, and T. A. Lipo, "Efficiency improvement and evaluation of floating capacitor open-winding pm motor drive for EV application," in *Proc. IEEE Energy Conversion Congress and Exposition (ECCE)*, Sep. 2014, pp. 837–844. DOI: 10.1109/ECCE.2014.6953484.

[43] D. Pan, F. Liang, Y. Wang, and T. A. Lipo, "Extension of the operating region of an ipm motor utilizing series compensation," *IEEE Transactions on Industry Applications*, vol. 50, no. 1, pp. 539–548, Jan. 2014, ISSN: 0093-9994. DOI: 10.1109/TIA.2013.2270223.

[44] J. Hong, H. Lee, and K. Nam, "Charging method for the secondary battery in dual-inverter drive systems for electric vehicles," *Power Electronics, IEEE Transactions on*, vol. 30, no. 2, pp. 909–921, Feb. 2015, ISSN: 0885-8993. DOI: 10.1109/TPEL.2014.2312194.

[45] Y. Wang, T. A. Lipo, and D. Pan, "Robust operation of double-output AC machine drive," in *Proc. 8th Int. Conf. Power Electronics - ECCE Asia*, May 2011, pp. 140–144. DOI: 10.1109/ICPE.2011.5944562.

[46] T. Husain, Y. Sozer, I. Husain, and E. Muljadi, "Design of a modular e-core flux concentrating axial flux machine," in *Energy Conversion Congress and Exposition (ECCE), 2015 IEEE*, Sep. 2015, pp. 5203–5210. DOI: 10.1109/ECCE.2015.7310392.

[47] S. Sanchez, D. Risaletto, F. Richardeau, and G. Gateau, "Comparison and design of intercell transformer structures in fault-operation for parallel multicell converters," in *Energy Conversion Congress and Exposition (ECCE), 2014 IEEE*, Sep. 2014, pp. 3089–3096. DOI: 10.1109/ECCE.2014.6953820.

[48] Y. Hayashi, H. Toyoda, T. Ise, and A. Matsumoto, "Contactless DC connector based on GaN LLC converter for next-generation data centers," *Industry Applications, IEEE Transactions on*, vol. 51, no. 4, pp. 3244–3253, Jul. 2015, ISSN: 0093-9994. DOI: 10.1109/TIA.2014.2387481.

[49] T. Klonowski, R. Andlauer, T. Leblanc, F. Faure, R. Meyer, and P. Teste, "High intensity contact opening under DC voltage," in *Electrical Contacts, 2004. Proceedings of the 50th IEEE Holm Conference on Electrical Contacts and the 22nd International Conference on Electrical Contacts*, Sep. 2004, pp. 459–466. DOI: 10.1109/HOLM.2004.1353157.

[50] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. DOI: 10.1145/359545.359563.

[51] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *J. ACM*, vol. 32, no. 1, pp. 52–78, Jan. 1985, ISSN: 0004-5411. DOI: 10.1145/2455.2457. [Online]. Available: http://doi.acm.org/10.1145/2455.2457.

[52] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. D. Natale, "Implementing synchronous models on loosely time triggered architectures," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1300–1314, Oct. 2008, ISSN: 0018-9340. DOI: 10.1109/TC.2008.81.

[53] H. Pfeifer, D. Schwier, and F. W. von Henke, "Formal verification for time-triggered clock synchronization," in *Proc. Dependable Computing for Critical Applications 7*, Jan. 1999, pp. 207–226. DOI: 10.1109/DCFTS.1999.814297.

[54] I. Broster and A. Burns, "Timely use of the can protocol in critical hard real-time systems with faults," in *Proc. 13th Euromicro Conf. Real-Time Systems*, 2001, pp. 95–102. DOI: 10.1109/EMRTS.2001.934009.

[55] L. Rodrigues, M. Guimaraes, and J. Rufino, "Fault-tolerant clock synchronization in CAN," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, IEEE Comput. Soc, Dec. 1998. DOI: 10.1109/real.1998.739775.

[56] *IEEE standard for a precision clock synchronization protocol for networked measurement and control systems*. DOI: 10.1109/ieeestd.2008.4579760.

[57] P. Fezzardi, M. Lipiński, A. Rubini, and A. Colosimo, "Ppsi - a free software ptp implementation," in *Proc. and Communication (ISPCS) 2014 IEEE Int. Symp. Precision Clock Synchronization for Measurement, Control*, Sep. 2014, pp. 71–76. DOI: 10.1109/ISPCS.2014.6948694.

[58] B. A. Welchko, T. M. Jahns, and S. Hiti, "Ipm synchronous machine drive response to a single-phase open circuit fault," *IEEE Transactions on Power Electronics*, vol. 17, no. 5, pp. 764–771, Sep. 2002, ISSN: 0885-8993. DOI: 10.1109/TPEL.2002.802180.

[59] H. S. Che, M. J. Duran, E. Levi, M. Jones, W. P. Hew, and N. A. Rahim, "Postfault operation of an asymmetrical six-phase induction machine with single and two isolated neutral points," *IEEE Transactions on Power Electronics*, vol. 29, no. 10, pp. 5406–5416, Oct. 2014, ISSN: 0885-8993. DOI: 10.1109/TPEL.2013.2293195.

[60] A. Tani, M. Mengoni, L. Zarri, G. Serra, and D. Casadei, "Control of multiphase induction motors with an odd number of phases under open-circuit phase faults," *IEEE Transactions on Power Electronics*, vol. 27, no. 2, pp. 565–577, Feb. 2012, ISSN: 0885-8993. DOI: 10.1109/TPEL.2011.2140334.

[61] B. A. Welchko, T. M. Jahns, W. L. Soong, and J. M. Nagashima, "Ipm synchronous machine drive response to symmetrical and asymmetrical short circuit faults," *IEEE Transactions on Energy Conversion*, vol. 18, no. 2, pp. 291–298, Jun. 2003, ISSN: 0885-8969. DOI: 10.1109/TEC.2003.811746.

[62] S. Dwari, L. Parsa, and T. Lipo, "Optimum control of a five-phase integrated modular permanent magnet motor under normal and open-circuit fault conditions," in *Power Electronics Specialists Conference, 2007. PESC 2007. IEEE*, Jun. 2007, pp. 1639–1644. DOI: 10.1109/PESC.2007.4342242.

[63] J.-R. Fu and T. A. Lipo, "Disturbance-free operation of a multiphase current-regulated motor drive with an opened phase," *IEEE Transactions on Industry Applications*, vol. 30, no. 5, pp. 1267–1274, Sep. 1994, ISSN: 0093-9994. DOI: 10.1109/28.315238.

[64] D. W. Novotny and T. A. Lipo, *Vector Control and Dynamics of AC Drives*. Oxford University press, Sep. 11, 1996, 456 pp., ISBN: 0198564392.

[65] T. A. Lipo, *Analysis of Synchronous Machines*. Wisconsin Power Electronics Research Center, 2008.

[66] W. Duesterhoeft, M. W. Schulz, and E. Clarke, "Determination of instantaneous currents and voltages by means of alpha, beta, and zero components," *American Institute of Electrical Engineers, Transactions of the*, vol. 70, no. 2, pp. 1248–1255, Jul. 1951, ISSN: 0096-3860. DOI: 10.1109/T-AIEE.1951.5060554.

[67] A. Rockhill, "On the modeling and control of high phase order synchronous machines," PhD thesis, University of Wisconsin - Madison, Mar. 2012.

[68] F. Briz, M. Degner, A. Diez, and R. Lorenz, "General of linear control tools for complex vectors," 1, vol. 35, Elsevier BV, 2002, pp. 241–246. DOI: 10.3182/20020721-6-es-1901.00121. [Online]. Available: https://doi.org/10.3182%2F20020721-6-es-1901.00121.

[69] H. Kim and R. Lorenz, "Synchronous frame pi current regulators in a virtually translated system," in *Industry Applications Conference, 2004. 39th IAS Annual Meeting. Conference Record of the 2004 IEEE*, vol. 2, Oct. 2004, 856–863 vol.2. DOI: 10.1109/IAS.2004.1348513.

[70] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "Sympy: Symbolic computing in python," *PeerJ Computer Science*, vol. 3, e103, Jan. 2017, ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103.

[71] D. Holmes and T. Lipo, *Pulse Width Modulation for Power Converters: Principles and Practice*, ser. IEEE Press Series on Power Engineering. John Wiley & Sons, 2003, ISBN: 9780471208143. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5264450.

[72] L. Zarri, M. Mengoni, A. Tani, G. Serra, and D. Casadei, “Minimization of the power losses in igbt multiphase inverters with carrier-based pulsewidth modulation,” *IEEE Transactions on Industrial Electronics*, vol. 57, no. 11, pp. 3695–3706, Nov. 2010, ISSN: 0278-0046. DOI: 10.1109/TIE.2010.2041737.

[73] M. C. D. Piazza and M. Pucci, “Efficiency issues in induction motor drives: Modelling and losses minimization techniques,” in *2015 IEEE Workshop on Electrical Machines Design, Control and Diagnosis (WEMDCD)*, Mar. 2015, pp. 171–177. DOI: 10.1109/WEMDCD.2015.7194526.

[74] A. Shea and T. M. Jahns, “Lag-free terminal voltage sensing in low-pass filtered PWM converters,” in *Proc. IEEE Energy Conversion Congress and Exposition (ECCE)*, Sep. 2016, pp. 1–8. DOI: 10.1109/ECCE.2016.7855222.

[75] T. A. Lipo, *Introduction to AC machine design*. Wisconsin Power Electronics Research Center, University of Wisconsin, 2004.