Control and Design of Multi-Use Induction Machines: Traction, Generation, and Power Conversion

by

Al-Thaddeus Avestruz

S.B. in Physics, Massachusetts Institute of Technology (1994)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2006

Author .....
.....
Department of Electrical Engineering and Computer Science
25 May 2006

Certified by .....
Steven B. Leeb
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
.....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Control and Design of Multi-Use Induction Machines: Traction, Generation, and Power Conversion

by

Al-Thaddeus Avestruz

Submitted to the Department of Electrical Engineering and Computer Science
on 25 May 2006, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

An electrical machine can be made to convert electrical power while performing in its primary role of transforming electrical energy into mechanical energy. One way of doing this is to design the machine with multiple stator windings where one winding acts as a primary for drive and power, and the others as secondaries for electrical power. The challenge is to control the mechanical outputs of torque and speed while independently regulating the electrical outputs of voltage and current. This thesis analyzes and demonstrates an approach that takes advantage of topological symmetries in multiphase systems to overcome this challenge. This method is applied, but not relegated to induction machines.

Thesis Supervisor: Steven B. Leeb
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my advisor Professor Steven Leeb for his efforts, patience, encouragement, and confidence in me, without which, all of this would not be possible.

To Meelee Kim, for her unwavering love and support.

To my parents, Suzanne and Fred, and my grandparents for their prayers and love: Lola Beth, Lola Jeannette, and Lolo Nes, who started my interest in science and engineering. In loving memory of my grandfather Alfonso Palencia.

To my brother Mark, without whose many discussions, I would have a lesser perspective of my work. To my sister Camille, whose future I look forward to with great confidence.

To Robert Cox, Chris Laughman, and Warit Witchakool for the many technical discussions that made work so much more interesting. To my office-mate Brandon Pierquet, for the discussions and entertainment and for the solutions to the little annoyances in LaTex.

And to everyone else in LEES, especially Professors Kirtley and Perreault, and to Vivian Mizuno.

I am sure to have forgotten to mention many who have helped me, but I am grateful despite the short lapse in memory.

This work was supported by the Center for Materials Science and Engineering, a Materials Research Science and Engineering Center funded by the National Science Foundation under award number DMR 02-13282, and by the Grainger Foundation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Efficiency, economy and elegance are hallmarks of good design. The idea of multi-use machines is an attempt to mitigate the waste and superfluity that is needlessly epidemic in a contemporary world starved of energy.

Motor and generator drives have been and continue to be crucial to a wide range of industrial and commercial products and manufacturing processes; among these, variable speed drives (VSDs) that operate over a range of mechanical shaft speeds are invaluable. Since the beginning, folks have found ways to vary the shaft speed of a motor; many exist, but circuit design and components in power electronics have advanced to a point were it is more appealing, both in performance and economics, for motors to be combined with power electronics in commercial and industrial VSDs.[1]

Many products use VSDs, including modern air handling and ventilation systems that run fans at speeds and power that are optimal, ensuring occupant comfort while keeping energy consumption to a minimum. Other examples include computer-controlled machine tools such as mill machines and lathes that need continuously variable speeds for different materials and tasks; hybrid and electric vehicles (EVs) use variable speed drives for traction or propulsion. These systems typically use power electronics to control the flow of power to the motor in controlling the speed. In addition to needing power for their drives, these systems often include other power supplies that typically includes distribution through a network of power buses through different parts of the system.

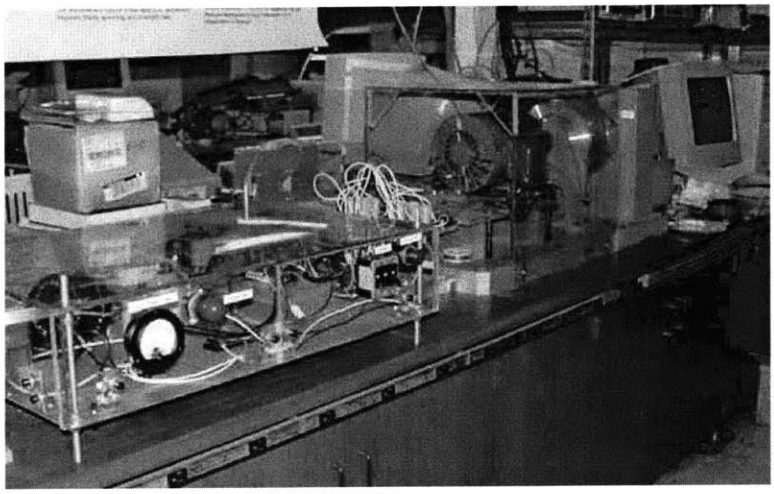


Figure 1.1: Photograph of what we have fondly called the Aardvark machine.

1.1 Machines with Multi-Use Capability

An electrical machine can be made to convert electrical power while performing in its primary role of transforming electrical energy into mechanical energy. One way of doing this is to design the machine with multiple stator windings where one winding acts as a primary for drive and power, and the others as secondaries for electrical power. The challenge is to control the mechanical outputs of torque and speed while independently regulating the electrical outputs of voltage and current.

The salient feature in these multi-use machines is the integration of magnetics for traction, generation and power conversion. While it is not eminently clear whether there is a convincing scaling law advantage in size and weight with this type of integration, it is evident that there is an economy to using the same control and power electronics for multiple purposes. Beyond rudimentary calculations for simple integrated pole face geometries, detailed studies of scaling laws for a variety of structures and magnetic circuit configurations is largely outside the scope of this work and is an obvious topic for follow-on research in machine design.

This thesis report analyzes and demonstrates an approach that takes advantage of topological symmetries in multi-phase systems to overcome this challenge. This method has been applied, but not relegated to induction machines.

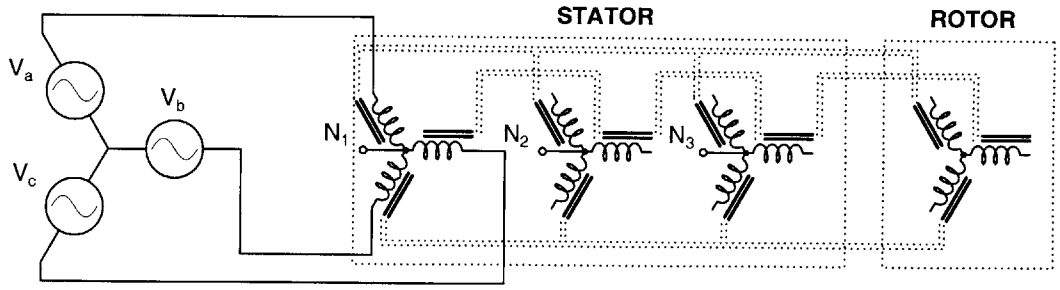


Figure 1.2: Simplified Depiction of a Multi-Stator Induction Machine.

1.2 Design Challenges and Innovations

There are a number of design challenges that must be addressed before one can assert whether this technology is viable for commercial and industrial applications. Overcoming the first challenge of proof of concept in a previous thesis [2] opened an avenue for continued research.

The first of the questions asks how one provides a wide range and monotonic control of the dc secondary output. It is answered by the use of the 3^{rd} harmonic amplitude at a phase of π radians for closed-loop control of the output; in general, the control-to-output function for dc output voltage is non-monotonic for other values of phase. Also, a look in [2] shows that the control of dc output using zero sequence phase, while effective, only provides a narrow range of variation.

The next addresses an issue that is endemic in most contraptions that we would like to use for multi-use machines: it is that of vanishingly small zero sequence reactance. A good solution, while not universal, is a variety of topologies that use a zero sequence transformer to improve the magnetizing inductance in the zero sequence circuit. Another solution, of course, is to redesign the machines, but that is the topic for another thesis.

Then, we must talk about the inverter. Not only must it be very good sinusoidal *current* source, but it must also be a very good *voltage* source of 3^{rd} or other zero sequence harmonics; it is the way you integrate the idea of zero sequence *voltage* with a field-oriented controller that is based on stator *currents*, which is the established means for variable speed drive.

This begs the question of generality. We talk for example about cars having two independent dc buses for power: the new one is 42 volts and the legacy is 12V. We can perhaps increase the number of phases in the machine so that there is more than one set of "independent" zero sequence harmonics. There are a number of ways to handle this: one can have a machine that physically has a greater number of inherent phases; the other, is to create these additional phases with linear combinations of the inherent phases. So, we start take a look at general

n-phase circuits that provide supernumerary zero sequence harmonic sets, and heterophasic transformer circuits.

How far can we go? As always, it depends on the compromises that one is willing to accept. It also matters how well we can design the machine, and this depends on at least an initial understanding of the design limitations and scaling laws. The complement to "how far" is "where else" and this is a question about machines other than induction ones. A glance at the ubiquity of the Lundell alternator adds not only to the intellectual, but also to the economic appeal.

This work provides some of the foundations and proof-of-concepts.

1.3 Previous Work

Initial work and construction of the induction machine testbed had been performed by Jack W. Holloway in a previous thesis project [2]. This work included the demonstration of dc output control of a wye-grounded winding by varying the phase of the added 3^{rd} harmonic in the inverter and the independence to zero sequence harmonics of an identical winding with the wye ungrounded.

Chapter 2

Power Conversion Control By Zero Sequence Harmonics

Zero sequence current in a multi-phase system is the portion of current that runs in the same direction through all the *connection phases* at the same time. This means that this component of the current has the same amplitude and *angular phase* in every connection phase.[1] It is the part that we can consider "common-mode" to all phase connections: the vector sum of the currents in these phases.[2]

One can also speak of a zero sequence voltage. If the circuit has a balanced terminal impedance *and* a zero sequence path, then it is that portion of the voltage at each phase that creates a zero sequence current. Often, we describe a zero sequence voltage in a way that is similar to a zero sequence current: having the same amplitude and angular phase in every connection, but it is not necessarily the case that a zero sequence voltage results in zero sequence current, as it is also the case that a positive or negative sequence voltage can result in a zero sequence current.

2.1 Induction Machine Model

A three phase induction machine can be described by the magnetic flux linkages among three stationary windings (stator) and three moving windings (rotor). Equation 2.1a describes every permutation of how the flux through every winding is linked to a current in its own and every

[1]To avoid ambiguity, we make a distinction in this paragraph between *connection phase*, which is the actual physical connection to the circuit, and *angular phase*, which is the constant parameter in the argument of a periodic function that for the moment we assume to be unmodulated.

[2]Though a Fourier transform, periodic signals can be represented as a sum of sinusoids. These sinusoids in turn can be represented as phasors.

others' winding.[3]

$$\begin{bmatrix} \boldsymbol{\lambda_S} \\ \boldsymbol{\lambda_R} \end{bmatrix} = \begin{bmatrix} \mathsf{L_S} & \mathsf{L_{SR}} \\ \mathsf{L_{SR}}^{\mathsf{T}} & \mathsf{L_R} \end{bmatrix} \begin{bmatrix} \boldsymbol{i_S} \\ \boldsymbol{i_R} \end{bmatrix} \tag{2.1a}$$

where the stator flux in the stationary reference frame

$$\boldsymbol{\lambda_S} = \begin{bmatrix} \lambda_{as} \\ \lambda_{bs} \\ \lambda_{cs} \end{bmatrix} = \begin{bmatrix} L_s & L_{ab} & L_{ab} \\ L_{ab} & L_s & L_{ab} \\ L_{ab} & L_{ab} & L_s \end{bmatrix} \begin{bmatrix} i_{as} \\ i_{bs} \\ i_{cs}, \end{bmatrix} \tag{2.1b}$$

the rotor flux

$$\boldsymbol{\lambda_R} = \begin{bmatrix} \lambda_{ar} \\ \lambda_{br} \\ \lambda_{cr} \end{bmatrix} = \begin{bmatrix} L_r & L_{ab} & L_{ab} \\ L_{ab} & L_r & L_{ab} \\ L_{ab} & L_{ab} & L_r \end{bmatrix} \begin{bmatrix} i_{ar} \\ i_{br} \\ i_{cr}, \end{bmatrix} \tag{2.1c}$$

and the mutual inductance matrix between the stator and the rotor

$$\mathsf{L_{SR}} = \begin{bmatrix} L_c \cos\theta_r & L_c \cos(\theta_r + 2\pi/3) & L_c \cos(\theta_r - 2\pi/3) \\ L_c \cos(\theta_r - 2\pi/3) & L_c \cos\theta_r & L_c \cos(\theta_r + 2\pi/3) \\ L_c \cos(\theta_r + 2\pi/3) & L_c \cos(\theta_r - 2\pi/3) & L_c \cos\theta_r \end{bmatrix}. \tag{2.1d}$$

When one applies the well-known Park's transform [4]

$$\mathsf{T} = \frac{2}{3} \begin{bmatrix} \cos\theta & \cos\left(\theta - \frac{2\pi}{3}\right) & \cos\left(\theta + \frac{2\pi}{3}\right) \\ -\sin\theta & -\sin\left(\theta - \frac{2\pi}{3}\right) & -\sin\left(\theta + \frac{2\pi}{3}\right) \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \tag{2.2a}$$

and for completeness delineating its inverse

$$\mathsf{T}^{-1} = \begin{bmatrix} \cos\theta & -\sin\theta & 1 \\ \cos\left(\theta - \frac{2\pi}{3}\right) & -\sin\left(\theta - \frac{2\pi}{3}\right) & 1 \\ \cos\left(\theta + \frac{2\pi}{3}\right) & -\sin\left(\theta + \frac{2\pi}{3}\right) & 1 \end{bmatrix}, \tag{2.2b}$$

to 2.1a, the flux linkages become block matrices, each of which are diagonal and time-invariant,

$$\begin{bmatrix} \boldsymbol{\lambda_{dqs}} \\ \boldsymbol{\lambda_{dqr}} \end{bmatrix} = \begin{bmatrix} \mathsf{L_S} & \mathsf{M} \\ \mathsf{M} & \mathsf{L_R} \end{bmatrix} \begin{bmatrix} \boldsymbol{i_{dqs}} \\ \boldsymbol{i_{dqr}} \end{bmatrix}, \tag{2.3a}$$

where

$$\mathsf{L_S} = \begin{bmatrix} L_{as} & 0 \\ 0 & L_{as} \end{bmatrix}, \tag{2.3b}$$

(2.3c)

$$\mathsf{L_R} = \begin{bmatrix} L_{ar} & 0 \\ 0 & L_{ar} \end{bmatrix}, \tag{2.3d}$$

(2.3e)

$$\mathsf{M} = \begin{bmatrix} M & 0 \\ 0 & M \end{bmatrix}, \tag{2.3f}$$

(2.3g)

$$M = \frac{3}{2} L_c. \tag{2.3h}$$

The state equations for the induction motor using flux are given by

$$-\dot{\lambda}_{ds} = r_s i_{ds} - \omega \lambda_{qs} - v_{ds} \tag{2.4a}$$

$$-\dot{\lambda}_{qs} = r_s i_{qs} + \omega \lambda_{ds} - v_{qs} \tag{2.4b}$$

$$-\dot{\lambda}_{dr} = r_r i_{dr} - \omega_s \lambda_{qr} \tag{2.4c}$$

$$-\dot{\lambda}_{qr} = r_r i_{qr} + \omega_s \lambda_{dr}, \tag{2.4d}$$

and torque of electrical origin by

$$\tau_m = \frac{3}{2} p \left(\lambda_{qr} i_{dr} - \lambda_{dr} i_{qr} \right), \tag{2.5}$$

and the equations of motion by

$$\dot{\omega_r} = \frac{1}{J} \left(\tau_m - \tau_l \right), \tag{2.6}$$

where p is the number of pole pairs, J is the moment of inertia, and τ_l is the load torque.

2.2 A Transformer Model for a Stator with Multiple Windings

Each phase in the stator windings is shifted by 120 electrical degrees; e.g. the flux linked between phase a and phase b is given by

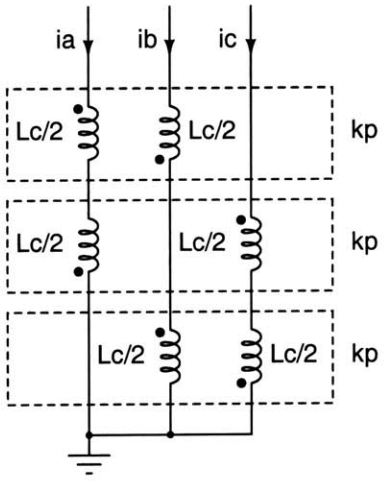
$$L_{ab} = L_c \cos(120°) = -\frac{1}{2} L_c. \tag{2.7}$$

Usually, the coupling is symmetric as well as equal among the phases, i.e., $L_{ab} = L_{ac} = L_{ba} = \ldots$.

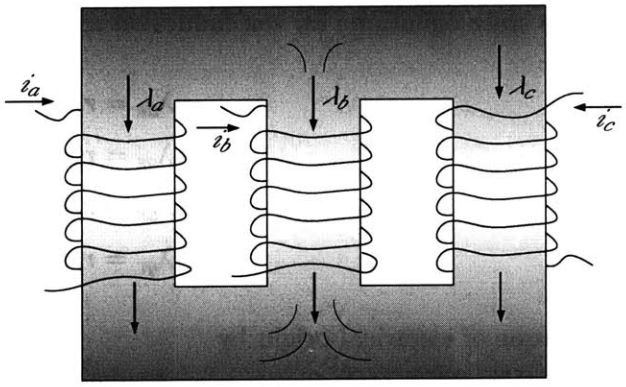
In a machine with multiple stators, additional flux linkages exist between the stator windings. Between two stators,

$$\begin{bmatrix} \lambda_{s_1} \\ \lambda_{s_2} \end{bmatrix} = \begin{bmatrix} L_{s_1} & M_{12} \\ M_{21} & L_{s_2} \end{bmatrix} \begin{bmatrix} i_{s_1} \\ i_{s_2} \end{bmatrix}. \tag{2.8}$$

In the next section, one will see how this bears more than just a casual similarity to a three-legged, three-phase transformer. In fact, the multiple stators of an induction by themselves behave the same way as the transformer illustrated in Figure 2.1, despite being wrapped around a circle.



(a)



(b)

Figure 2.1: Electrical model for the primary of a Three-Legged Transformer. k_p is the phase-to-phase coupling coefficient.

The coupling coefficient between windings of a transformer is the fraction of the flux coupled

between the primary and the secondary and is given by

$$k = \frac{M}{\sqrt{L_1 L_2}}, \tag{2.9a}$$

which generalizes to a matrix element

$$k_{mn} = \frac{M_{mn}}{\sqrt{L_m L_n}} \tag{2.9b}$$

that relates an arbitrary winding to another, where L_1 and L_2 are the primary and secondary side open-circuit inductances, respectively, and M is the mutual inductance between windings.

The inductance matrix given by Equation 2.8 has been implemented in PSPICE as illustrated in Figure 2.2.

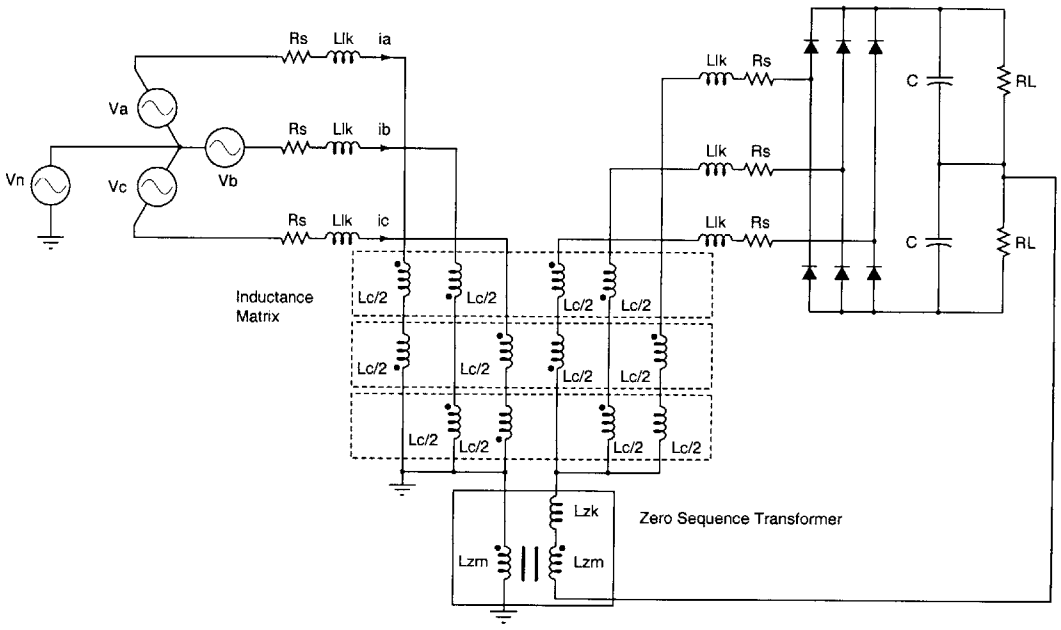


Figure 2.2: In-circuit transformer model of the flux linkage between two identical stator windings wound in-hand. A zero sequence transformer is described in §2.3.2.

This coupling between phases can be implemented in SPICE by coupled inductor statements; Appendix A contains the SPICE deck as well as the schematic file for PSPICE. Note that in Figure 2.2, the negative coupling coefficients and subsequent negative inductances are captured in the winding polarity of the coupled inductors, which helps to illustrate a better physical

sense of what the flux is doing. Figure 2.3 illustrates how the model can be used to determine parameter effects on the system design.

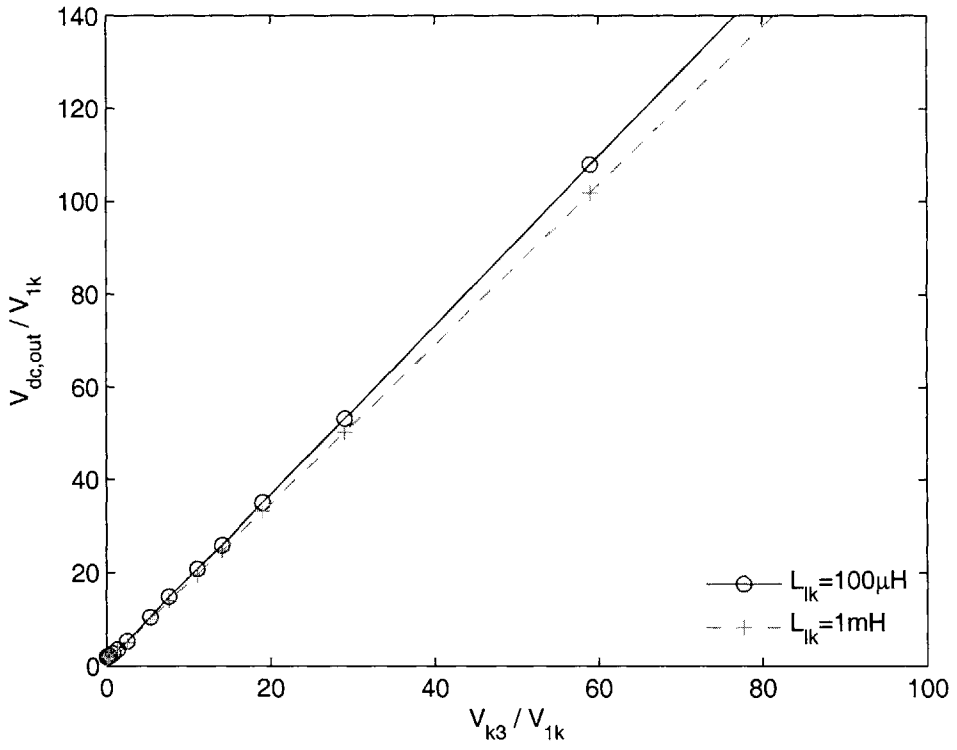


Figure 2.3: Simulation result of the effect of a change on the series inductance in the 3rd harmonic control function.

2.3 *Zero Sequence Harmonic Control in Three Phase Systems*

By introducing one or more triple-n harmonics into the drive voltage of a three-phase machine, the rectified output voltage from grounded-wye windings can be controlled and subsequently regulated. To first-order, these triple-n harmonic voltages produce triple-n harmonic currents, and introduce negligible net torque on the rotor, effectively decoupling voltage regulation from drive.

The rectifiers in Figure 1.1 are designed to operate in the discontinuous conduction mode. In this case, rectifiers in winding 2 behave as peak detectors of the line-to-neutral voltages, with

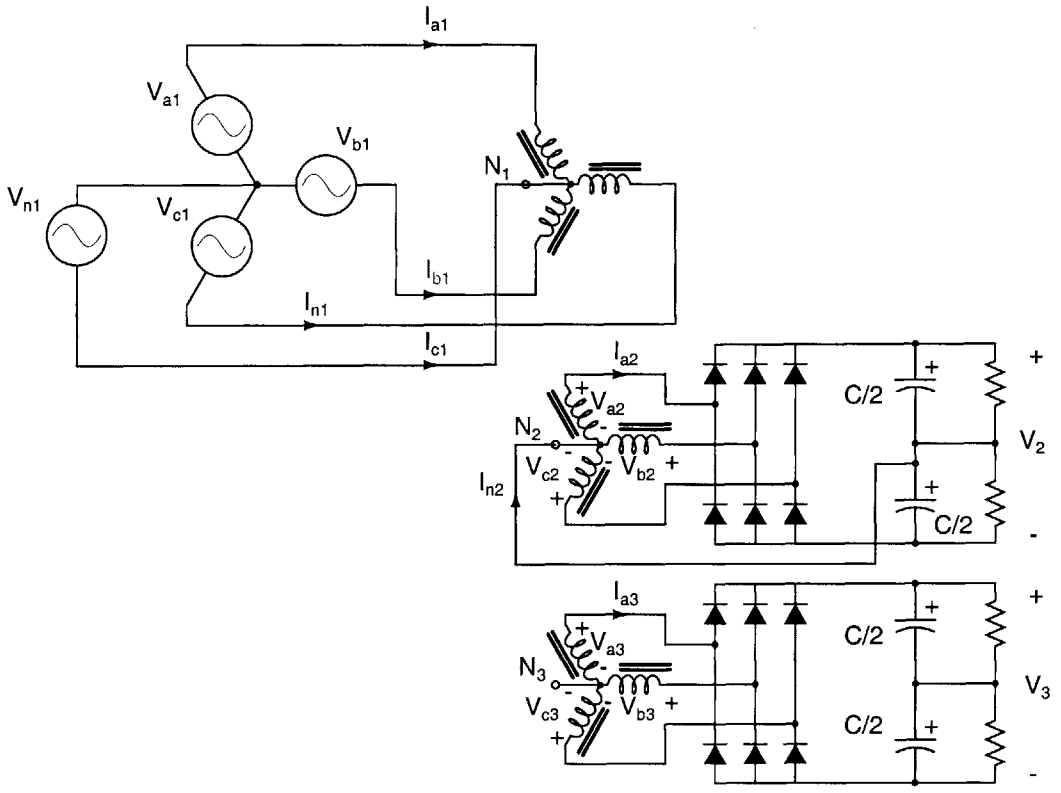


Figure 2.4: Multiple Stator Connections Driving Three-Phase Rectifiers.

the dc output voltage given by

$$V_2 = V_{k1} \sin\theta_p + V_{k3} \sin(3\theta_p + \phi_3), \tag{2.10}$$

where V_{k1} and V_{k3} are the amplitudes of the fundamental and third harmonic inverter voltages, respectively, ϕ_3 is the phase angle of the third harmonic relative to the fundamental and θ_p is the phase angle where the drive voltage is at a maximum. The angle θ_p is given by the extremum relation for the drive voltage

$$\frac{dV_2}{d\theta_p} = V_{k1} \cos\theta_p + V_{k3} \cos(3\theta_p + \phi_3) = 0, \tag{2.11}$$

which unfortunately is transcendental.

Figures 2.6 and 2.7 illustrates how the dc output voltage will vary with third harmonic amplitude and phase for a rectifier operating in discontinuous mode. One notices that for a third harmonic phase $\phi_3 = \pi$, the dc output voltage is not only monotonic, but also linear with third harmonic voltage V_{k3}; the reason for this is immediately obvious from Figure 2.5, as the peaks of the fundamental and third harmonic occur coincidentally. While not proven, it can be plausibly argued that while V_2 is monotonic with V_{k3} over various intervals for different values of ϕ_3, linearity as well as the widest range of V_{k3} for monotonicity occurs only for $\phi_3 = \pi$.

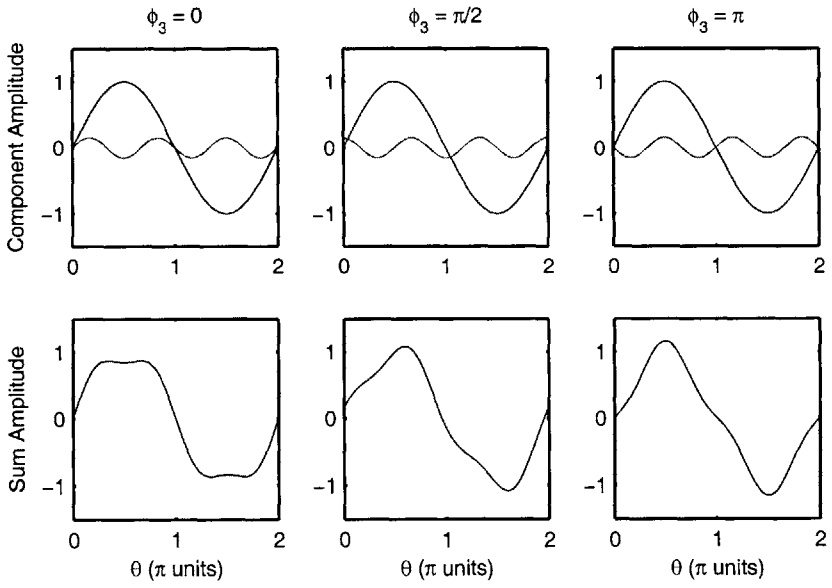


Figure 2.5: Drive Waveform with Third Harmonic

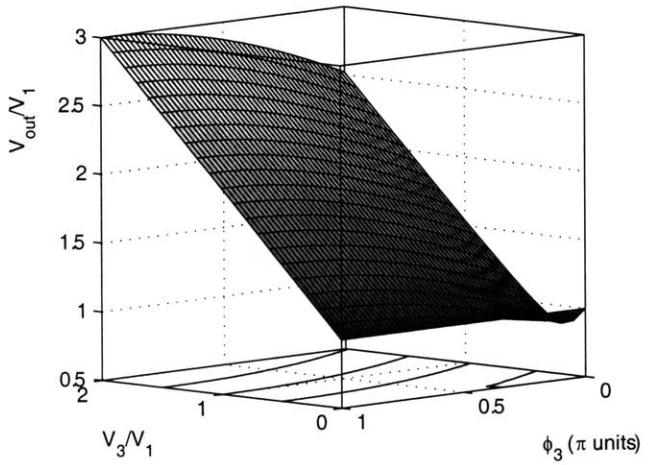
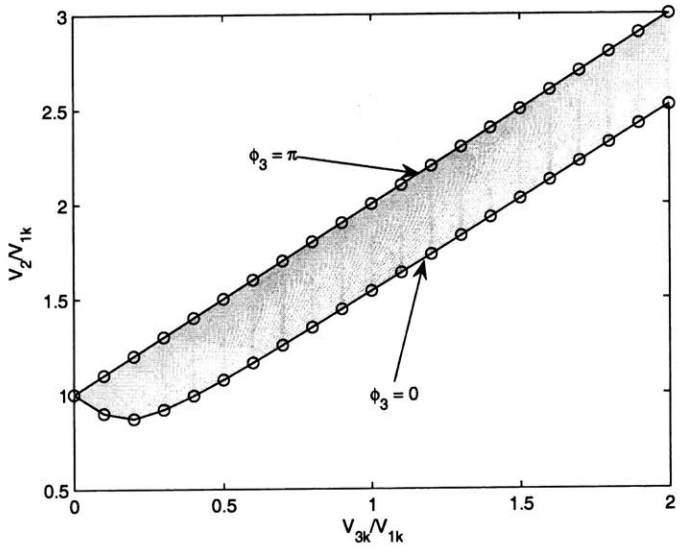


Figure 2.6: Control surface for peak amplitude control using third harmonic voltage amplitude (V_3) and phase (ϕ_3) with fundamental voltage (V_1) held constant.

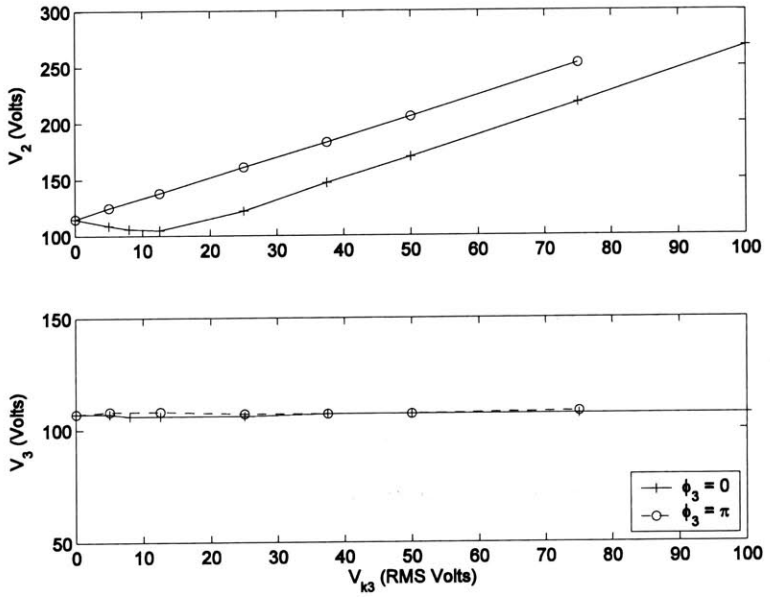
2.3.1 *Voltage Output Control by Harmonic Amplitude Variation*

The dc output V_2 of a grounded wye-connected rectifier can be controlled by varying the third harmonic voltage applied to the primary drive winding in Figure 1.1. For $\phi_3 = 0$, it can be exactly calculated (for a system that can be modeled as having no ac-side line inductance) that V_2 is monotonic with $|V_{k3}|$ for $|V_{k3}| > |V_{k1}|/6$ (at $|V_{k3}| = |V_{k1}|/6$, $|V_2| = \sqrt{3}|V_{k1}|/2$). The dependence of V_2 on V_{k3} is plotted in Figure 2.7(a) for values of ϕ_3 between 0 and π. At $\phi_3 = \pi$, V_2 is affine for $V_{k3} > 0$. At other value ϕ_3 closed-form solutions to $V_2(V_{k3})$ are likely to not exist, but numerical methods can be used to estimate where this function is monotonic. Figure 2.7(b) shows that experimental data does indeed agree with calculations.

The strategy used for π-phase harmonic control is founded on the fact that the peaks of the 3$^{\text{rd}}$ harmonic and the fundamental coincide. In this case, where we have assumed discontinuous current in the phase connections of the secondary windings and 1:1 turns ratio, the dc output voltage will be very nearly equal to the sums of the peak voltages, $V_{k1} + V_{k3}$. In the implementation, a closed-loop PI controller ensures as V_{k1} drops, which for example is the case when the speed is lowered, that the 3$^{\text{rd}}$ harmonic V_{k3} makes up the difference.

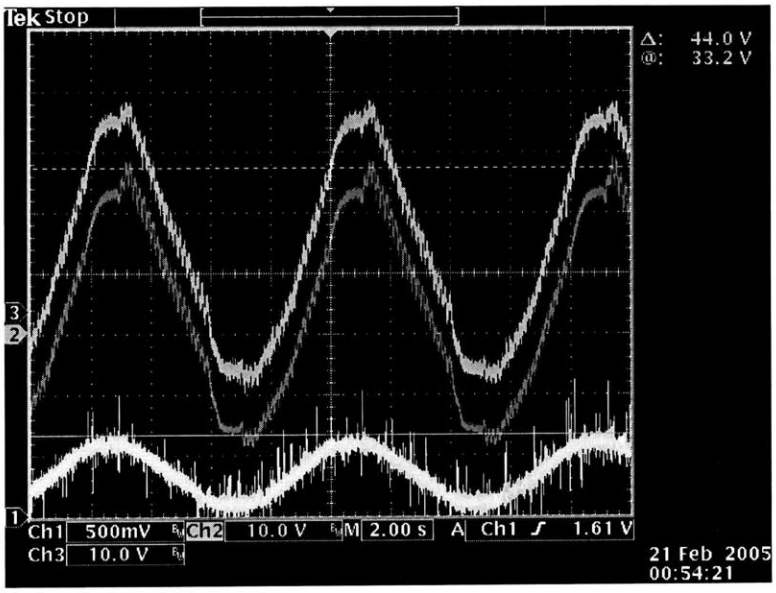


(a) MATLAB Calculations

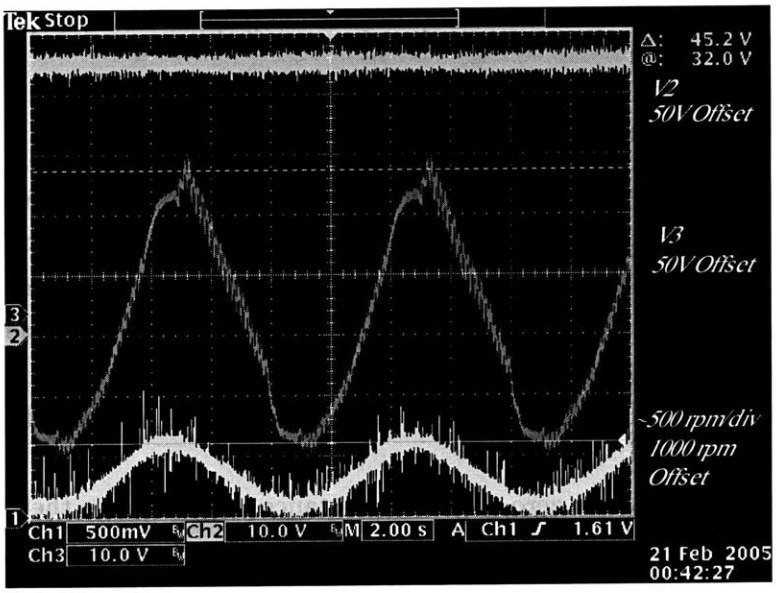


(b) Experimental Results[5]

Figure 2.7: Peak amplitude control using third harmonic voltage amplitude (V_3) over a spread of phase (ϕ_3) with fundamental voltage (V_1) held constant.



(a) No 3rd Harmonic



(b) PI Control with Third Harmonic

Figure 2.8: PI Control of Dc Output Using Third Harmonic. The top trace shows V_2, the dc output of the grounded-wye secondary winding.

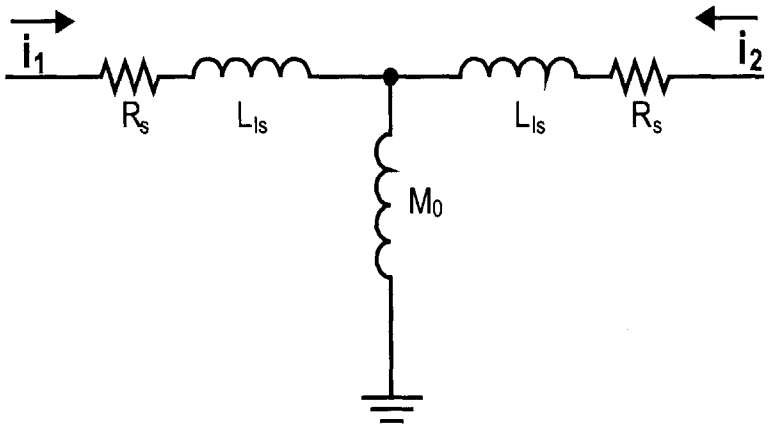


Figure 2.9: T-Model for the Zero Sequence Circuit.

2.3.2 Zero Sequence Circuit

The zero sequence circuit is the portion of a multi-phase system where current can run in the same direction at the same time; it is that the part that is connected to one might call "common" to all the phases—with nomenclatures that include *wye* or *neutral.* Not all polyphase systems have a zero sequence path, often systems that do are called wye-grounded.

Zero Sequence Reactance in a Three Phase Circuit

A key issue in driving a zero sequence current through two magnetically coupled windings with grounded wyes is the effective magnetizing inductance, which is the phase-to-phase leakage in an induction machine, hence is typically kept as small as possible tfor good machine performance [6]. Figure 2.9 shows a zero sequence circuit model for an induction machine with two identical stators. Small magnetizing inductance M_0 results in high zero sequence reactive current and represents additional loss in the stator resistance R_s, as well as additional switch stress in the power electronics.

One method to increase the zero sequence reactance is to decrease the phase-to-phase coupling. Looking for the moment at the flux from phase a of a three-legged transformer,

$$\lambda_a = L_c i_a - k_p \frac{L_c}{2} i_b - k_p \frac{L_c}{2} i_c, \tag{2.12}$$

which implies that for a balanced set of currents with k_p now less than unity, each leg must now support a higher volt-seconds, hence resulting in a larger core.

From this transformer picture of the inductance matrix, it becomes more obvious that the

amount of flux coupled to the secondary winding sorely depends on the coupling between the positive inductance windings (L_c) being large (i.e. nearly unity k_0[3]), yet with little coupling (k_p) between adjacent phases (e.g. a and b). For a stator geometry with little saliency, which is mostly the case with induction motors, the coupling between phases tends to be typically high. When viewed as the single-circuit T-model illustrated in Figure 2.9, this results in a zero sequence magnetizing inductance that is deplorably small; a zero sequence transformer in series with the neutral current path alleviates this problem with the caveat that it is now the zero sequence transformer that must transfer essentially all the zero sequence power to the secondary side.

Zero Sequence Transformer Topologies

One solution to the problem of small magnetizing inductance is shown in Figure 2.10, where a zero sequence transformer with an acceptable magnetizing inductance is used in the wye connection; a turns ratio other than unity offers an additional degree of freedom in optimizing machine design through the scaling of the zero sequence voltage and current. This zero sequence transformer can be integrated into the machine back-iron, although this is not currently implemented.

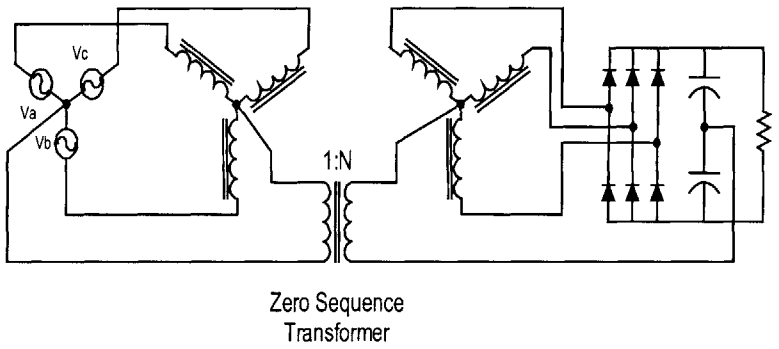


Figure 2.10: Zero sequence transformer.

Power can be derived directly from the wye point as illustrated in Figure 2.11. In this topology, fewer rectifiers are required and a spit-capacitor ground is not needed; power to the output is derived solely from the zero sequence harmonics, so rectifier currents do not contribute to torque ripples, even without special accommodations in the control. Because the rectifiers only draw zero sequence current, standard field-oriented control schemes with fewer current sensors are more easily implemented.

[3] $k_0 = M/L_c$ for identical stator windings.

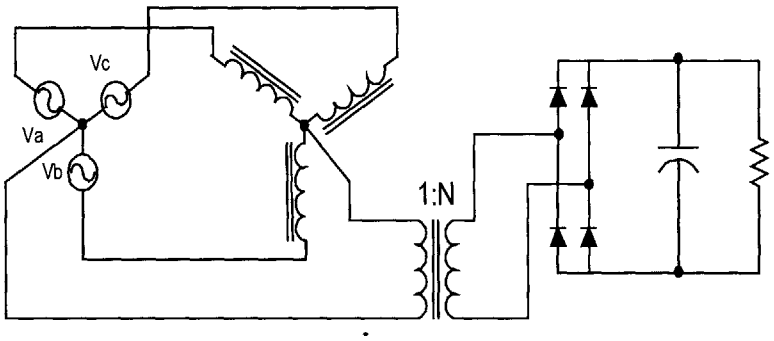


Figure 2.11: Direct zero sequence transformer.

Although using a direct zero sequence transformer offers a number of advantages, the trade-off is that all the dc output power is converted solely through a single-phase circuit, whereas the topology shown in Figure 2.10 allows, in certain regimes of operation, a portion of the power to be transferred to the output through the three phase circuit.

Under instances where the stator is not driven by an inverter, such as in an alternator, zero sequence harmonics can be exogenously driven through the wye as illustrated in Figure 2.12.

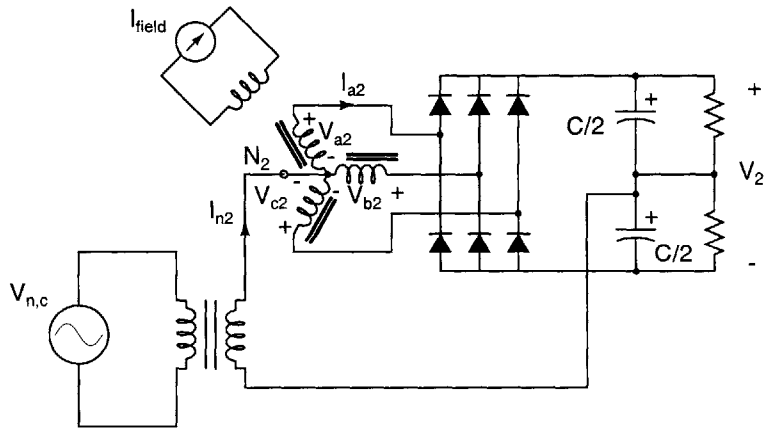


Figure 2.12: Exogenously driven zero sequence transformer.

2.4 *Toward a Machine Design*

Multi-use machines combine the challenges of both machine and transformer design. For example, in "standard" machines, insulation breakdown is a troublesome, but not a safety-critical

issue. In transformers where the primary is greater than some voltage threshold[4] and where the secondary must be *safe*, the requirements for isolation (creepage and clearance) and insulation breakdown are strict and regulated by legislation.

While this section does not pretend to be a comprehensive treatise on the design of multi-use machines, nor does it suggest a design example, it attempts to offer some perspectives on the design limitations and advantages, which provides a motivation and some foundations for future work.

2.4.1 Fundamental Design Limitations

In the discussion of a machine design, there is an advantage to keeping the coupling between phases high, in very much the same way that a three-legged three phase transformer is better than three separate single phase transformers: the flux in each leg of the core has to carry only 1/2 of the flux than for each separate single phase transformer, resulting in a weight and volume savings for a given amount of apparent power.

2.4.2 Zero Sequence Control of Three-Phase, Three-Legged Transformer

2.4.3 Scaling Laws

The rationale for a multiple output transformer (i.e. $1 : N : M : \ldots$) as opposed to multiple single transformers (i.e. $1 : N$, $1 : M$, $\ldots$) appears to be two-fold. As power conversion is combined into a single piece of magnetics, both weight and volume is lower than the aggregate of the single transformers; per unit power handling capability along with better overall window utilization.

Area Product

The power handling capability of a transformer is proportional to the area product A_p which is the product of the the window area W_a and the core cross-sectional area A_c,

$$A_p = W_a A_c. \tag{2.13}$$

Both the weight and the volume of the transformer increase sub-linearly with A_p and hence also with power capability [7],

$$V \propto {A_p}^{0.75}. \tag{2.14}$$

[4]CE standards are 60V

Window Utilization

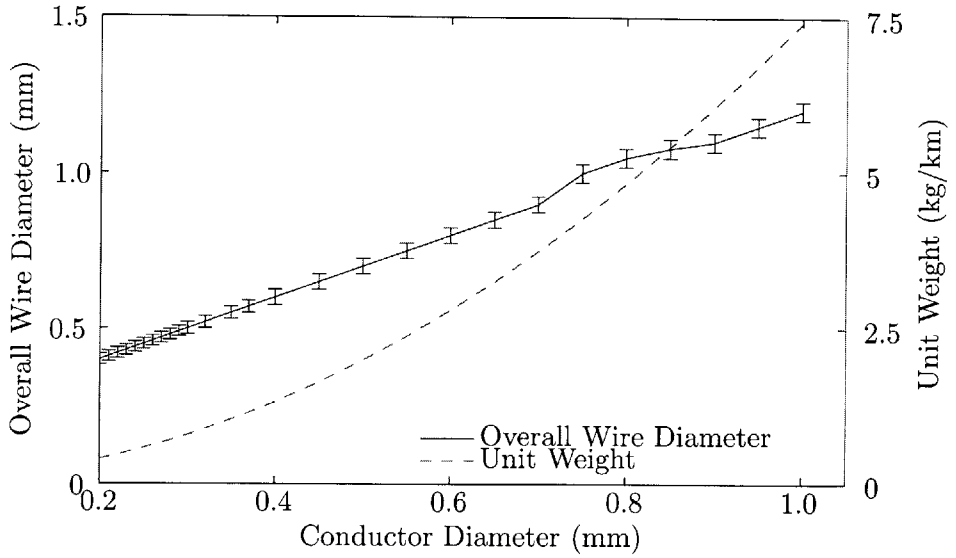
Isolation between primary and secondaries requires the use of a substantial amount of insulating tape between windings or the use of a triple-insulated wire[5] to meet safety standards (e.g. CE and UL). With a lower power winding, the insulation consists of a higher percentage of the overall cross-sectional area of the wire. This results in a lower window utilization K_u for lower power windings. A poignant example of this is illustrated in Figure 2.13, where the insulation is a significant fraction of the wire cross-section. Despite being a good fraction of the cross-section, the insulation is still much lighter than the enclosed copper so that the unit weight of the wire still increases pretty much linearly with its power handling capability.[6]

Induction Machine

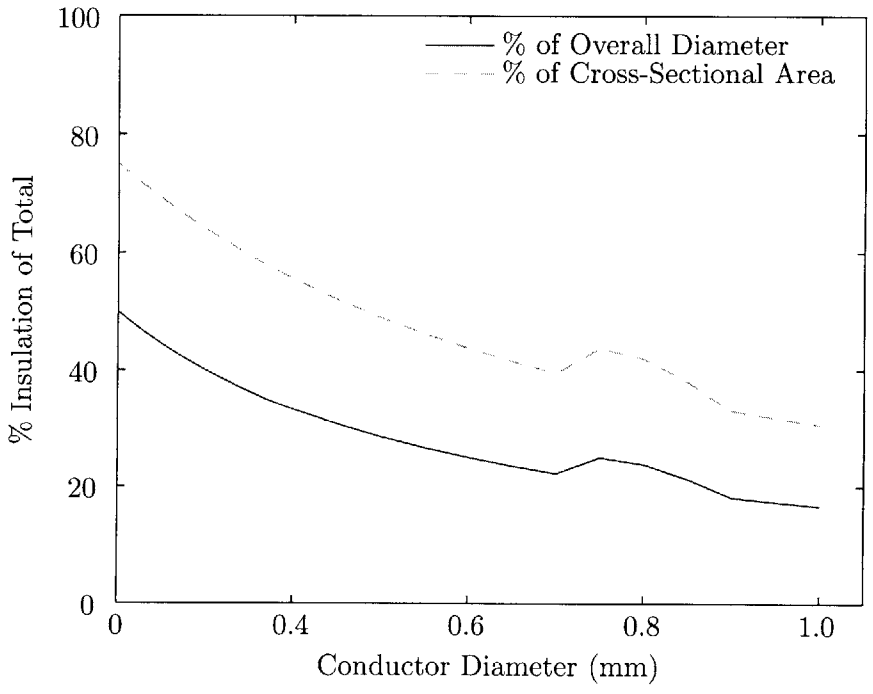
In any case, when the voltage amplitude of the third harmonic is less than that of the fundamental, there will be a fundamental component in the stator current and hence will contribute a torque ripple. However, one expects that a large fundamental drive voltage occur at high speed, where cogging is not as significant of an issue. At zero or low speed, rectifier current will be zero sequence. These have consequences in terms of per phase winding and core utilization.

[5]Furukawa Tex-E®.

[6]The power handling capability of a wire is related to its temperature rise for a given amount of current, hence is inversely proportional to unit resistance. Because of that, the amount of power that a wire can handle is proportional to cross-sectional area, or to the square of its diameter.



(a) Data plotted from Furukawa Tex-E® wire table.[8]



(b) Cross-Sectional Utilization

Figure 2.13: Better utilization of the wire cross-section by the conductor is unambiguous for larger diameters in triple insulated wire.

Chapter 3

Vector Drive Control

The general concern of a variable speed drive is accurate and fast control of speed. However facetious as this sounds, it is not so straightforward of an endeavor in an induction machine. Speed and torque have no easy relation to voltage and terminal currents as on a dc machine. One must expend more effort to implement the classic speed control loop that has a minor loop for torque.

3.1 Constant Volts per Hertz Drive

In an induction machine with constant slip, flux is inversely proportional to frequency. For operation with constant flux, the ratio of voltage to frequency is held constant. At a given flux, the maximum speed attained when the equivalent *back-EMF* equals the available inverter voltage. At higher speeds, one must operate the machine at a constant voltage, while the flux decreases with speed: this is know as *field-weakening* or constant-power operation. The maximum available torque falls roughly in proportion to the inverse square of the frequency.[4]

This type of drive is typically good for VSDs that generally operate in the steady and where the best transient performance is not required (e.g. HVACs operating under relatively constant load). In a closed-loop speed control system, response and its complement, disturbance rejection, is ultimately determine by the controllability of torque and hence flux. During a transient, neither constant slip, nor any value of slip is guaranteed with a constant V/Hz drive. In addition, at low speeds and high torques, the implied low voltage and high current means that voltage drops across the stator resistance become a serious limitation for a drive system whose control variable is terminal voltage.

While the advantage of constant V/Hz operation is that it is simple to implement and does not require current sensors except perhaps for fault detection, it is most likely not good enough

for traction and generation applications like EVs where speed and torque is both widely and strongly varying. Attempts at enhancing this method of speed controller include [9], but because of advancements in digital signal processors and subsequent price competitiveness, field oriented control methods have become more popular and accessible for high performance applications.

3.2 Indirect Field Oriented Control

Field oriented control takes advantage of the relation given by Equation 2.5

$$\tau_m = \frac{3}{2} p \left(\lambda_{qr} i_{dr} - \lambda_{dr} i_{qr} \right)$$

to control the machine torque by specifying a flux and a current. If we set $\lambda_{qr} = 0$ and hold $\lambda_{dr} = \Lambda_0$ constant, then the torque is proportional to i_{qr}, which resembles how the torque relates to terminal current in dc machine.

$$\lambda_{qr} = 0 \tag{3.1a}$$

$$\lambda_{dr} = \Lambda_0 \qquad \text{Constant,} \tag{3.1b}$$

so the torque in Equation 2.5 can be written as

$$\tau = -\frac{3}{2} p \Lambda_0 i_{qr}. \tag{3.2}$$

$\lambda_{qr} = 0$ implies that $\dot{\lambda}_{qr} = 0$ which results on the constraint in slip frequency

$$\omega_s = -\frac{r_r i_{qr}}{\lambda_{dr}} = \frac{r_r M}{L_{ar} \lambda_{dr}} i_{qs}. \tag{3.3}$$

The torque in stator coordinates is then given by

$$\tau = \frac{3}{2} p \frac{M}{L_{ar}} \Lambda_0 i_{qs}. \tag{3.4}$$

An estimator for λ_{dr} can be derived from the first order differential equation

$$\dot{\lambda}_{dr} + \frac{r_r}{L_{ar}} \lambda_{dr} = \frac{r_r M}{L_{ar} i_{ds}}, \tag{3.5}$$

where $T_r = L_{ar}/r_r$ is referred to as the rotor time constant. λ_{dr} can then be programmed by

i_{qr} through a first-order transfer function,

$$\lambda_{dr} = \frac{M}{sT_r + 1}. \tag{3.6}$$

Field-oriented control methods require the control of the current in the stator that is magnetically linked to the rotor. In a multiple stator machine, this current measurement is corrupted by the additional loads presented by the secondary stators. This can be resolved by subtracting those load components from the primary stator currents to get the drive currents, which is described in §3.3. The field-oriented system illustrated in Figure 3.3 includes an implementation of a synchronous frame regulator. [10] gives a good discussion on tuning, stability and robustness of field-oriented controllers over parameter variations.

3.3 Cartesian Feedback in the Synchronous Current Regulator

The advantage of regulating current in the synchronous frame of reference is that the stator currents are represented as dc. In RF parlance, this is the consequence of *mixing down* the sinusoidal ac currents in the stator terminals to a dc baseband. In this synchronous reference frame, zero steady state error can be achieved by placing a pole at the origin in the controller, i.e. integral control. In the stator reference frame, the currents are sinusoidal and hence cannot have zero steady error for a proportional-integral controller.[1]

By applying the conditions for field-oriented control in §3.2, the following state equations can be derived for the direct and quadrature stator currents from the state equations for $\mathrm{d}\lambda_{ds}/\mathrm{d}t$ and $\mathrm{d}\lambda_{ds}/\mathrm{d}t$:

$$-L_{as}\frac{di_{ds}}{dt} = r_s i_{ds} - \omega\left(L_{as} - \frac{M^2}{L_{ar}}\right) i_{qs} - v_{ds} \tag{3.7a}$$

$$-\left(L_{as} - \frac{M^2}{L_{ar}}\right)\frac{di_{qs}}{dt} = r_s i_{qs} + \omega L_{as} i_{ds} - v_{qs}. \tag{3.7b}$$

It is apparent from Equation 3.7a that there is a strong coupling term between the direct and quadrature axis currents that is proportional to the synchronous frequency.

A number of assumptions simplify the design of a controller for the Aardvark induction machine. A key assumption in designing the synchronous current controller is that the electrical

[1]See Roberge[11] for a discussion on the error series derivation, which is germane to the tracking error of a proportional-integral controller to a sinusoid.

time constants of the machine are much smaller than the mechanical time constants. This is important because it allows us to satisfy the condition that the bandwidth of a minor loop be higher than the outer loop crossover frequency.

3.4 Integrator Anti-Windup

There are two output limits in any real inverter: current limit and voltage saturation (i.e. compliance of the current source). The maximum current that ought to be allowed depends on the physical limits of the power devices and the load. Voltage saturation is the result of a finite dc bus voltage, hence limiting the time rate of changes in flux ($\mathrm{d}\lambda/\mathrm{d}t$). In the short-time scale, this is due to the time rate of change in current ($\mathrm{d}i/\mathrm{d}t$), and on the longer time scale (or steady state) by winding resistances (stator and rotor) and to the time rate of change of mutual inductance, which is proportional to the speed ω. Abstractly, by the product rule

$$v = \frac{\mathrm{d}\lambda}{\mathrm{d}t} = L\frac{\mathrm{d}i}{\mathrm{d}t} + i\frac{\mathrm{d}L}{\mathrm{d}t}. \tag{3.8}$$

Voltage saturation level is actually subtle: when one wants an inverter output with as few harmonics as possible, saturation occurs when the peak amplitude of the sine wave fundamental equals the one-half of the dc bus voltage for the half-bridge inverter (the full dc bus voltage for a full-bridge inverter); however, if over-modulation is allowed, the fundamental amplitude can be as high as $4/\pi$ times larger by applying 100% duty cycle for 50% of the time, i.e. a symmetric square wave.

In terms of dq-axis quantities the current limit

$${i_{s,max}}^2 \leq {i_{ds}}^2 + {i_{qs}}^2. \tag{3.9}$$

The voltage saturation limit in this dq space

$${v_{s,max}}^2 \leq {v_{ds}}^2 + {v_{qs}}^2. \tag{3.10}$$

From a control perspective, either limit presents itself as a classic actuator saturation. In a controller that integrates the error between the command (or reference) and the output, this error accumulates causing a large overshoot even when the actuator comes out of saturation and the setpoint been reached. That which is not a classical about this situation is the limitation of the magnitude of a vector of control variables (a MIMO system); a further complication is that the integrator in the controller is designed with an integrator for each variable so that there

will be zero error in the steady state.

From Equation 3.4, i_{ds} programs the flux in the machine and i_{qs} programs the torque. Typically, the flux is programmed to some optimal value below the rated motor speed; above this speed, the flux is decreased in inverse proportion to the speed, resulting in a constant power operating regime. If we assume that the speed changes at a much lower rate than the torque, the dynamics to consider in the controller design are that of the torque while that of the speed over the relevant time scale is invariant.

The condition for voltage saturation in Equation 3.10 allows for one degree of freedom, which we choose to be v_d. In the polar coordinate frame, the contour is described by

$$\theta = \cos^{-1} \frac{v_{ds}}{v_{s,max}}. \tag{3.11}$$

This allows i_{ds} to still be programmed through v_{ds} during saturation as illustrated in Figure 3.1.

In arranging the saturation conditions this way, we maintain the dc motor analogue, where the torque is constrained while the flux remains a free variable.[2]

The speed is also controlled by a PI controller, but it has SISO (single input, single output) dynamics, with an LTI function of the error commanding a torque, which we assume to be proportional to stator quadrature current i_{qs}. The field, or flux is proportional to i_{ds} which value is determined by a field-weakening function of speed that we presume to have no dynamics.[3] We would like to saturate the output of the controller for any of several reasons: a current limit given by Equation 3.9, a mechanical damage torque limit, and an electrical torque limit which depends on the flux[4]. The signaling of either these limits results in a relatively straightforward anti-windup strategy, such as limiting the speed-control integrator.

The current limit is the result of a number of factors. As already mentioned, these include a hard current limit to prevent physical damage and current source compliance due to a finite inverter voltage. Because the speed controller is much slower than the current controller, its integrator winds up at a much lower rate. We would like to signal a saturation from the current controller to the speed-control integrator only for longer time scale saturation events due to such things as demanding more torque than what is within the limits of the setting for the flux. If the speed controller with its field-weakening algorithm and torque limits were ideal, these longer time scale voltage saturation events would not occur; however, time-varying parameters

[2]In a speed controller with field weakening, flux is a non-linear function of speed.

[3]Only perhaps presumptuous in that the flux dynamics have a time scale in the neighborhood of the rotor time constant (T_r given in Equation 3.5), which we assume to be much smaller than the mechanical time constants.

[4]The electrical torque limits can be precalculated from the flux.

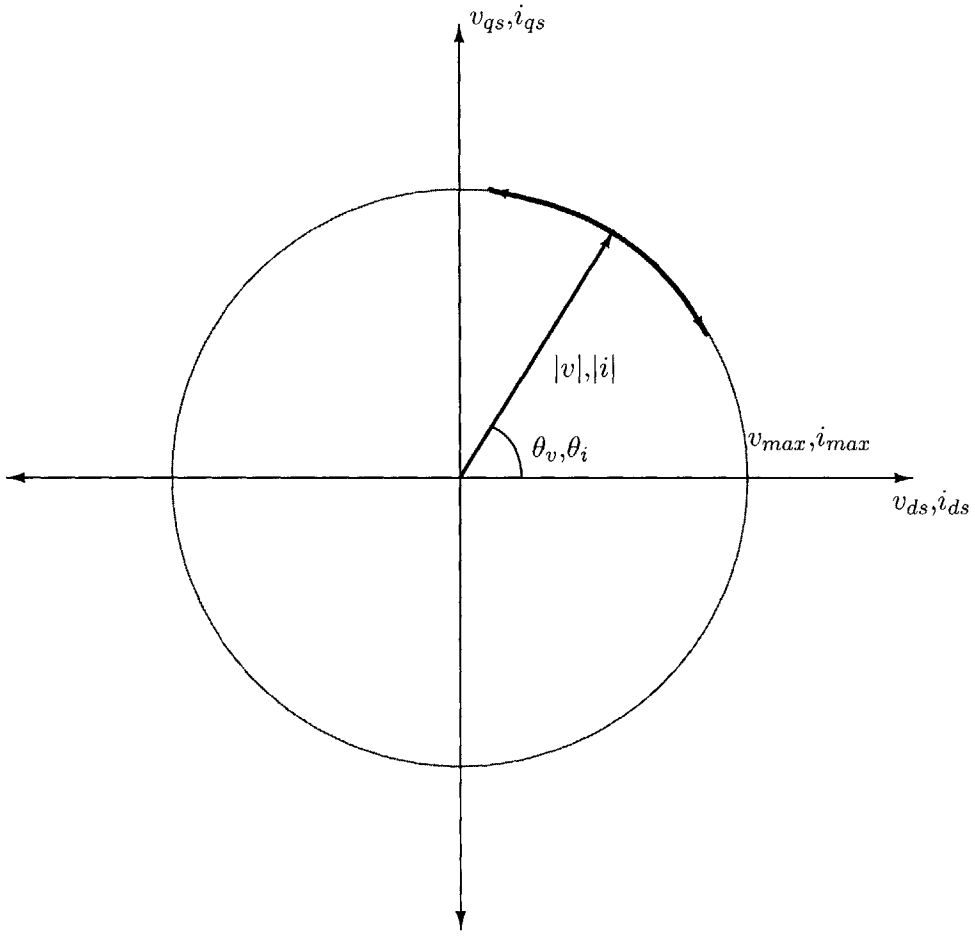


Figure 3.1: Contour of the saturation limit for the synchronous current regulator and the allowed trajectory in dq-space. There is a maximum torque limit that can also be described on the dq axis, but ought to be considered as part of the speed controller and not included in this diagram.

such as stator and rotor resistances, as well as nonlinearities in the iron permeability may well cause the speed controller to ask for more than the current regulator can provide.

The question is how does one determine the cause of the voltage saturation. Recall Equation 3.8. One way to do that is to keep track of the magnitude of the time derivative of the current $\mathrm{d}i/\mathrm{d}t$ during the voltage saturation. If

$$\left\| \frac{\mathrm{d}}{\mathrm{d}t} \begin{bmatrix} i_{ds} \\ i_{qs} \end{bmatrix} \right\|_2 \leq \epsilon \tag{3.12}$$

where ϵ is some threshold while the voltage is still saturated, then current controller signals a saturation event to the speed controller.

3.5 Simulation

Table 3.1: Aardvark Machine Parameters[2]

Stator Resistance	R_a	2.0Ω
Rotor Resistance	R_2	1.5Ω
Stator Reactance[†]	X_1	2.8Ω
Rotor Reactance[†]	X_2	2.8Ω
Mutual Reactance[†]	X_M	42.09Ω
Free Moment of Inertia	J_0	$0.0168\ \mathrm{kg} \cdot \mathrm{m}^2$

The electrical parameters for the Aardvark induction machine are listed in Table 3.1. These parameters were derived by J. Holloway in [2] from a non-linear least square fit to the start up transient of the induction machine using IEEE blocked-rotor and no-load tests as well as impedance measurements for values for the initial guess.

The parameters in Table 3.1 along with an indirect field oriented controller and the strategies for anti-windup form the basis for a Simulink® model and simulation. Figure 3.2 predicts good transient performance under step speed reversals and steps in torque load. A diagram of the simulation can be found in Appendix C.

[4]Reactances are customarily referenced to 60 Hz.

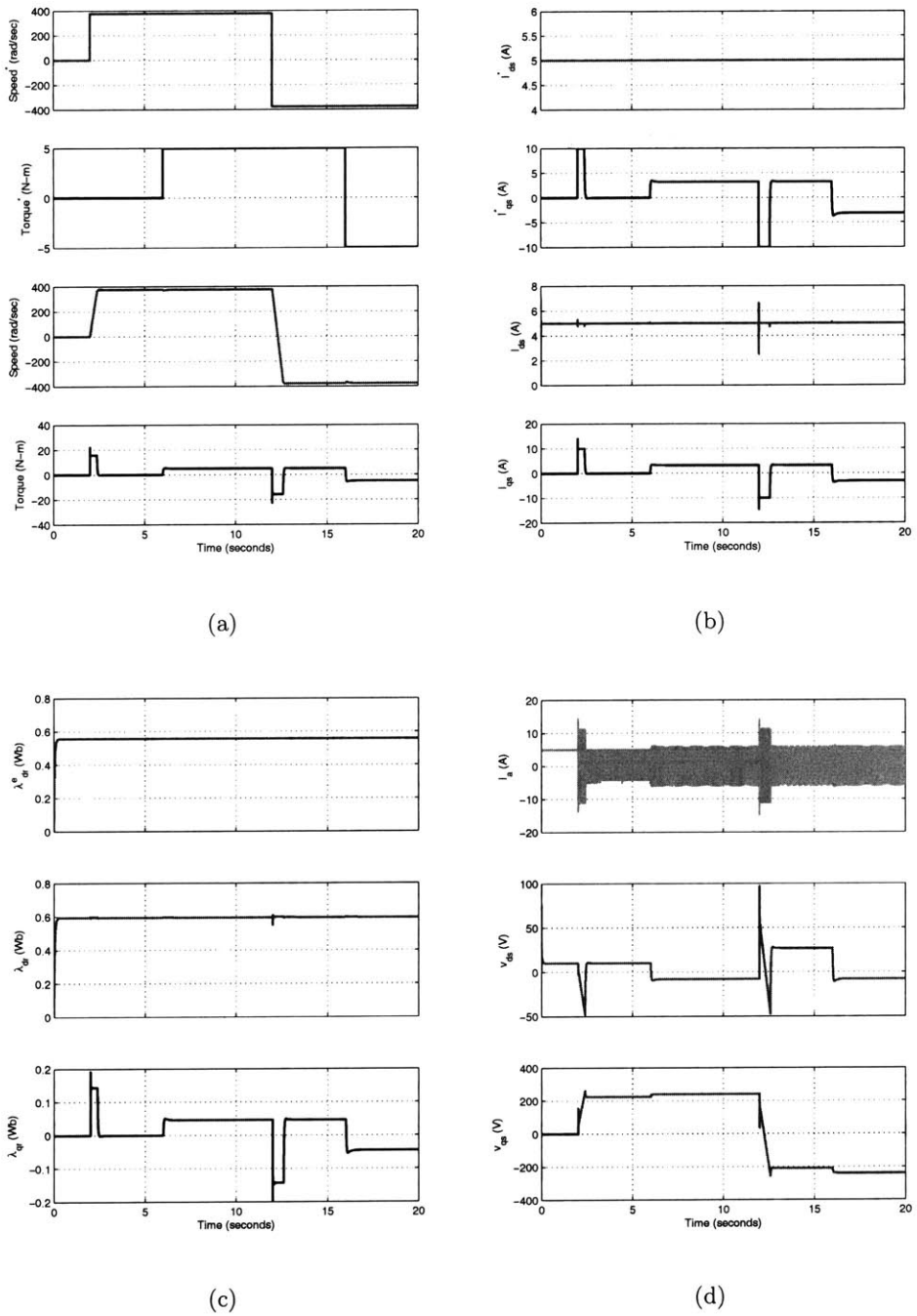


Figure 3.2: Simulation of induction motor under field-oriented control with speed and torque load steps.

3.6 Implementation

The controller illustrated in Figure 3.3 is currently being implemented with *minimal current sensing* as described in §4.4.2. There were a number of issues that precluded the inclusion of results in this thesis. These included a number of hardware issues that included slow and erratic behavior in the opto-coupler circuit for the speed sensor and incorrect gains in the current sensing circuits. In the firmware, timing miscues in the field oriented control routine caused incorrect updates to the PWM routine.

A new opto-coupler circuit, as well as current sensing circuitry have been designed and tested, but not yet integrated into the motor controller. Field-oriented control firmware is currently being debugged and results are forthcoming.

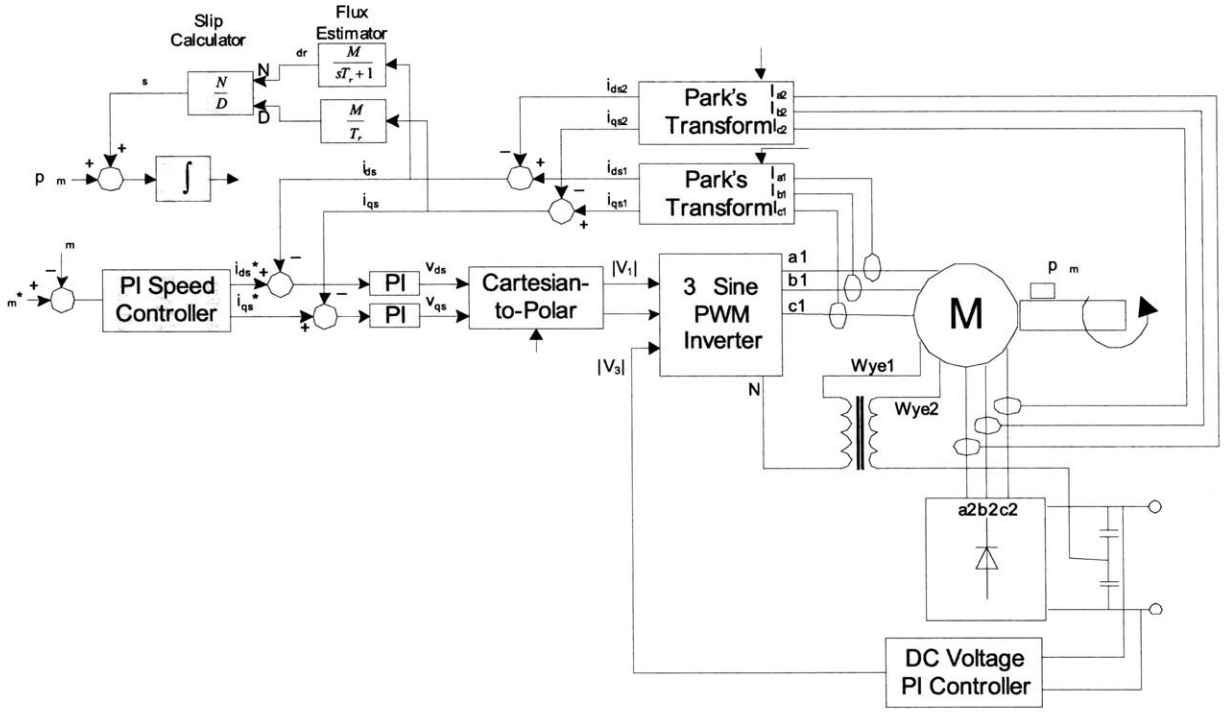


Figure 3.3: Implementation of a field-Oriented controller for a multi-use induction machine. The secondary dc output is regulated by varying the 3^{rd} or other zero sequence harmonic in the PWM inverter. In this diagram, the turns ratios between the stator windings and the ratios in the zero sequence transformer are assumed to be 1:1. Other turns ratios are possible by proper scaling when subtracting the transformed stator winding currents.

Chapter 4

Inverter Design

4.1 Power Module and Digital Signal Processor

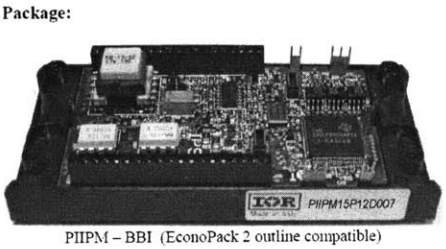
A half-bridge inverter was designed around International Rectifier's PIIPM15P12D007 programmable isolated integrated power module. The power module contains the power electronics (e.g. IGBTs, gate drives, etc.), ac mains rectifiers, as well as a TI TMS320LF2406A DSP for digital control of the digital control of the motor drive and for any zero sequence control of dc output voltages.

The combined power electronic and control platform used in this thesis is shown in Figure 4.1. The PIIPM15P12D007 from International Rectifier (IR) combines all the necessary power electronics (i.e. IGBTs, gate drives, and protection) with a TMS320LF2406A control-optimized DSP from Texas Instruemnts, along with associated sensors, peripherals, auxiliary supplies, and communications (e.g. RS485, JTAG, CAN). Although this platform has been discontinued by IR, it is close to an ideal model for the development of digital motor control systems.

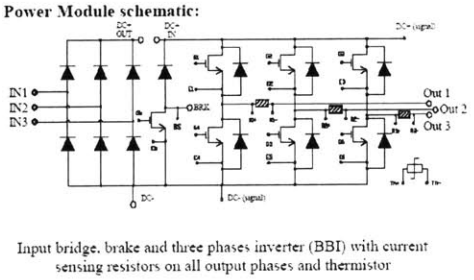
4.2 Sine Wave Generation

4.2.1 Synchronous PWM

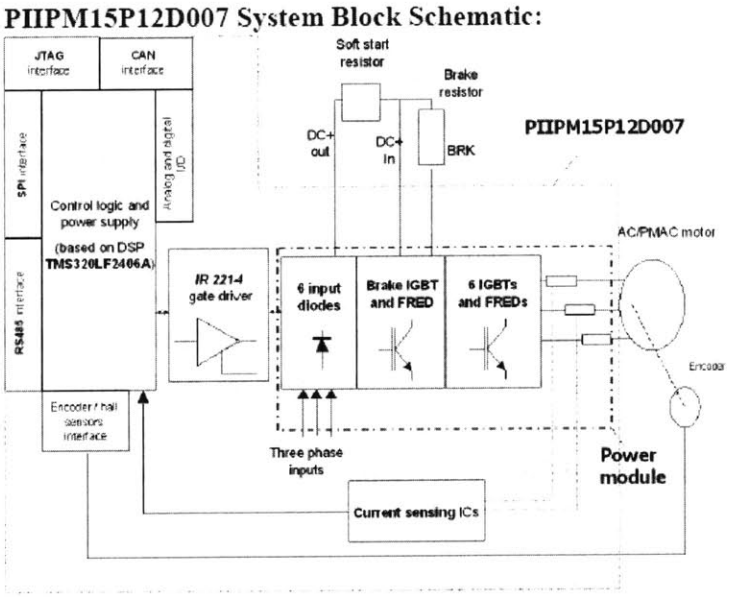
In variable speed drive, the inverter fundamental frequency varies with speed; in a field-oriented controller, it also varies with torque. When using a constant PWM frequency, non-integer ratios between the PWM frequency and the fundamental result in subharmonic content as a result of the "beating". *Synchronous PWM* was achieved by vaying the PWM switching frequency about a nominal (e.g. 10 kHz) so that the switching frequency is always a multiple n of the generated sine wave; an algorithm for hysteresis about the transition points of n was included



(a) PIIPM Integrated Power Module



(b) PIIPM Schematic



(c) PIIPM Block Diagram

Figure 4.1: International Rectifier's Integrated Power Module[12]

to eliminate switching frequency jitter and oscillation. A good discussion of synchronous PWM can be found in [13].

4.2.2 Table-Based Implementation

A sine wave drive with low spurious harmonic content is important for rectifier output voltage control. The three-phase inverter is based on a 108-element sine reference table that drives a symmetric PWM whose output is illustrated by the MATLAB plot in Figure 4.2. The size of the table was chosen to be both a multiple of 3 (aligned three-phase system) and 4 (quarter-wave symmetry). This results in a THD (total harmonic distortion) of 1.71% and a maximum error of 2.90%.

Table 4.1: THD and Maximum For a DLT Sine Reference Using Different Interpolating Functions

Interpolating Function	**THD (%)**	**Maximum Error (%)**
ceiling( )	3.37	5.81
floor( )	3.37	5.81
round( )	1.71	2.90

Although not implemented, an equivalent 27-element quarter-wave table could be used. The generation of the third harmonic is achieved by accessing every third table entry during each PWM update. A key to the generation of a sine wave with low harmonic content is the alignment of the PWM switching instances with the table element entries, which ensures synchronous PWM. A discussion of table-based implementations are presented in [14]. Figure 4.4 shows no harmonic content to at least 1.25 kHz with a 60 Hz fundamental and a third harmonic amplitude that is 50% of the fundamental. The algorithm is simple computationally because it performs only a direct table lookup and does not require interpolation. With this algorithm, the resolution for third harmonic phase modulation is determined by the size of the table. If a better resolution is required without the penalty of a large table size, interpolation for only the third harmonic lookup is required.

4.2.3 Parabolic Approximations

Real-time, on-the-fly second order approximations are a good alternative to look-up table based implementations. Angular resolution is limited only by the working precision of the desired number type. A second-order approximation requires at most three multiplications and three

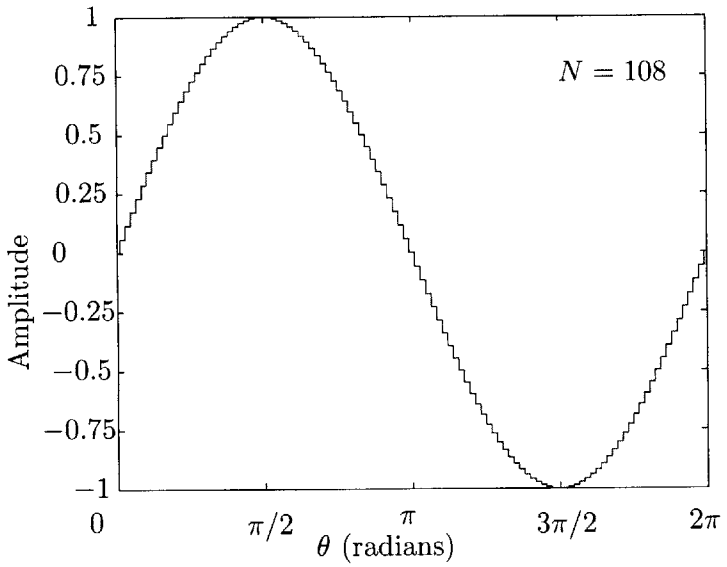


Figure 4.2: Sine reference created from a 108-element direct-lookup table.

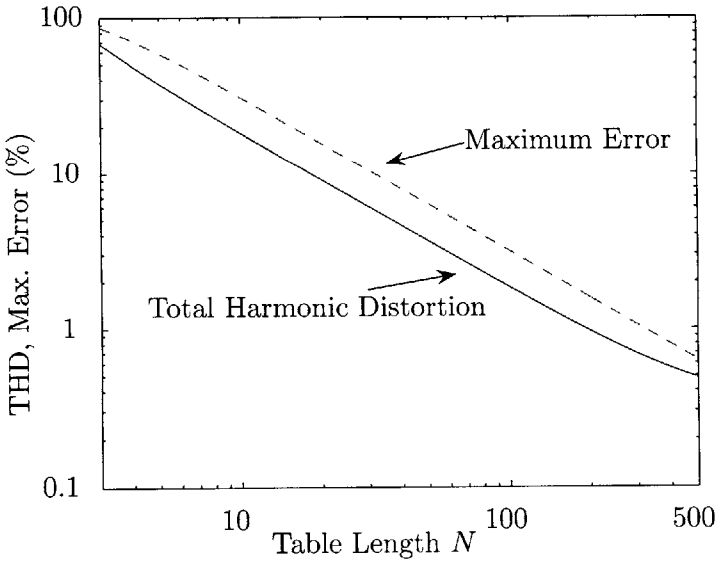


Figure 4.3: THD and maximum error versus table length N for a sine reference using a direct lookup table.

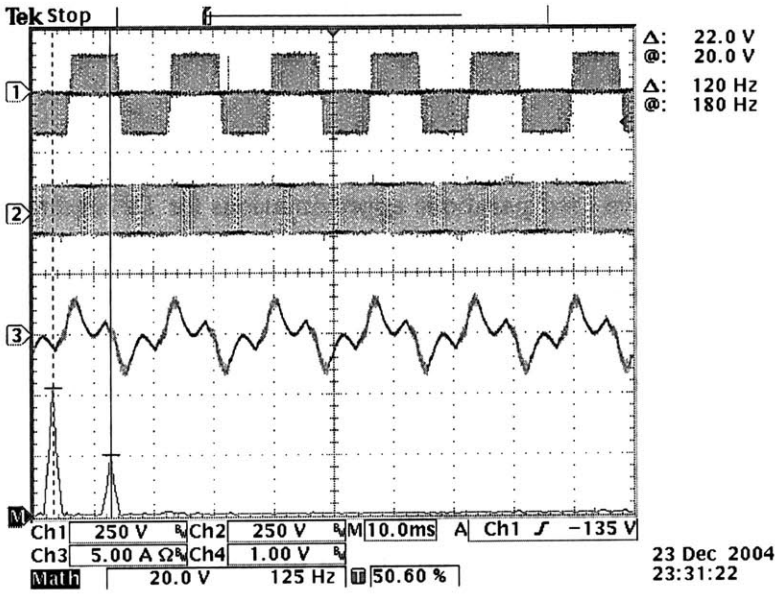


Figure 4.4: Inverter output voltage.

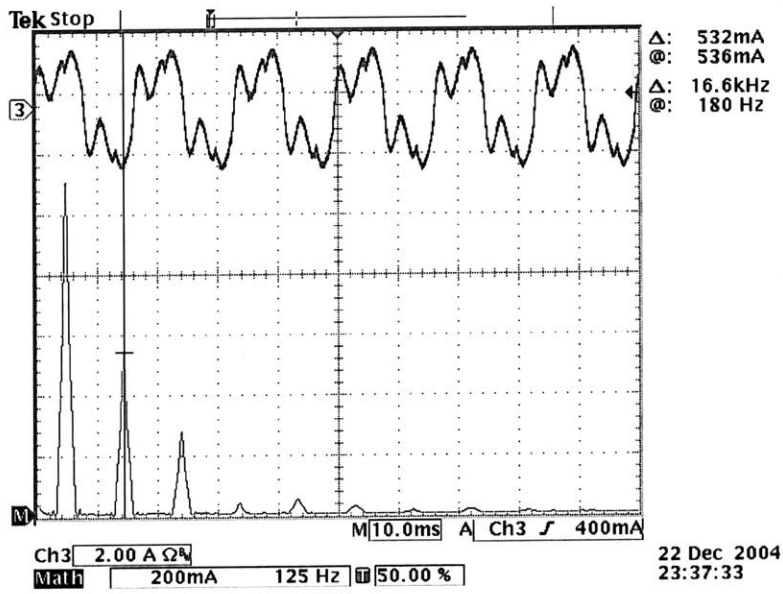


Figure 4.5: Inverter Current for Fundamental + 25% Third Harmonic

additions; a half-wave approximation requires one compare while a quarter-wave approximation requires at most three compares, if one assumes the argument to the approximating function is already limited to principal values, i.e. $[0, 2\pi]$. Over the approximating interval (e.g. $[0, \pi]$ for a half wave),

$$\mathcal{S}(\theta) = x_2(\theta - x_1)^2 + x_0 \tag{4.1}$$

A number of authors have used parabolic approximations for *DFSS*(direct-digital frequency synthesis) in communications, but have used either a three-point fit (zero crossing and peak with no errors) [15], least-squares fitting [16], or Taylor series approximations[16, 17]. In addition, 4th order approximations based on doubly iterated parabolic approximations have been proposed [15]. In the proceeding sections, we will see that choosing the appropriate metric for the fitting optimization gives a more appropriate result for an intended application. Only a second order approximation is shown, but a higher order iterated parabolic approximation with the appropriate metric can easily be implemented.

The coefficients of the approximation are chosen in some optimal way:

- Maximizing the fundamental.
- Minimizing harmonic distortion.
- Minimizing percentage error.
- Zero error at the endpoints.
- Zero error at the peak.
- Zero error at the zero crossings.
- $\mathcal{L}^1$ optimal.
- $\mathcal{L}^2$ optimal, which minimizes the mean square error.
- $\mathcal{L}^\infty$ optimal, which minimizes peak error.
- And so forth ...

In general, these conditions do not result in the same coefficients and are sometimes conflicting. As a second-order approximation, only three degrees are freedom are available.

As a sine reference for an inverter, values of the approximating function must match at the endpoints of the sub-intervals; this does not necessarily mean that there must be zero error at

these points. However, enforcing zero error at the zero crossings minimizes crossover distortion and prevents an ambiguity that could lead to a systematic dc offset.

In the case of a half-wave approximation, enforcing zero error at the zero crossings leaves only one degree of freedom for any other optimization, whereas in the quarter-wave approximation, two degrees of freedom are still available, which in reviewing Tables 4.2 and 4.3, result in better optimization figures of merit.

In the calculation of the Park's transform, it seems reasonable that the sine approximation be $\mathcal{L}^\infty$ optimal, which results in the smallest peak error for the calculation of d-axis and q-axis quantities, while the sine reference for the inverter may use coefficients that reduce total harmonic distortion. Ultimately, it is a multi-parameter design optimization in the design of the inverter where proper weighting of such things as torque ripple and efficiency in the machine, controller stability, among many others, must be taken into consideration.

Minimizing total harmonic distortion is not equivalent to maximizing the fundamental, which is equivalent to minimizing the objective function

$$\mathscr{G}(x_0, x_1, x_2) = \left| \int_0^{2\pi/n} \sin^2\theta' \,\mathrm{d}\theta' - \int_0^{2\pi/n} \left[x_2(\theta' - x_1)^2 + x_0\right] \sin\theta' \,\mathrm{d}\theta' \right| \tag{4.2}$$

$$= \left| \int_0^{2\pi/n} \mathscr{E}(\theta') \sin(\theta') \,\mathrm{d}\theta' \right| \tag{4.3}$$

where n is the number of sub-intervals and the error

$$\mathscr{E}(\theta) = \mathcal{S}(\theta) - \sin\theta. \tag{4.4}$$

Total harmonic distortion (THD) when the dc term is zero, is defined as the ratio of rms value of the harmonics in the waveform to the rms value of the fundamental,

$$\mathrm{THD} = \sqrt{2\left(\frac{\frac{1}{2\pi/n}\int_0^{2\pi/n} \mathcal{S}^2(\theta')\,\mathrm{d}\theta'}{{a_1}^2} - 1\right)} \tag{4.5}$$

where a_1 is the fundamental Fourier coefficient,

$$a_1 = \frac{n}{\pi}\int_0^{2\pi/n} \mathcal{S}(\theta')\sin\theta' \,\mathrm{d}\theta'.$$

Optimizing for total harmonic distortion is equivalent to minimizing

$$\mathscr{G}(x_0, x_1, x_2) = \frac{\int_0^{2\pi/n} \mathcal{S}^2(\theta')\,\mathrm{d}\theta'}{{a_1}^2}, \tag{4.6}$$

which is proportional to the reciprocal of the square of the distortion factor. The distortion factor is the ratio of the rms of the fundamental to the rms of the waveform.

The $\mathcal{L}^\infty$ norm to minimize becomes

$$\mathscr{G}(x_0, x_1, x_2) = \|\cdot\|_\infty = \sup\{|\,\mathscr{E}(\theta)| : \theta \in [0, 2\pi/n]\}, \tag{4.7}$$

and the $\mathcal{L}^2$ norm, or least-squares objective

$$\mathscr{G}(x_0, x_1, x_2) = \|\cdot\|_2{}^2 = \int_0^{2\pi/n} \mathscr{E}^2(\theta)\,\mathrm{d}\theta'. \tag{4.8}$$

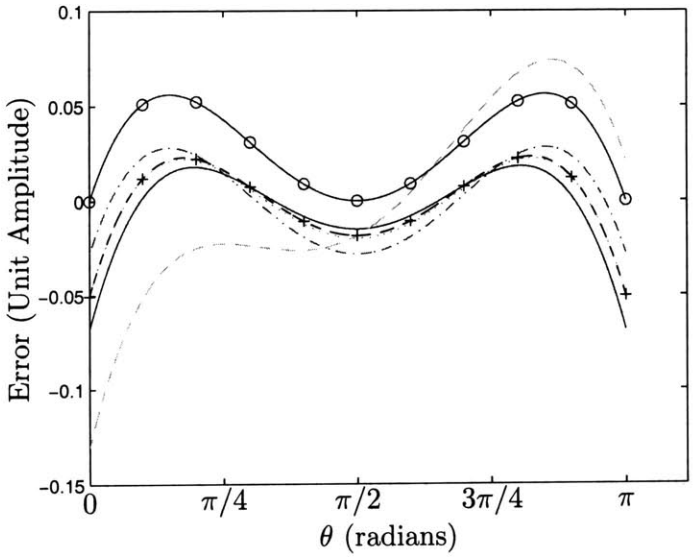
Half-Wave Approximation

A half-wave approximation is an optimal second-order fit to each of two sub-intervals: $[0, \pi]$ and $[\pi, 2\pi]$. The error over the approximating interval are illustrated in Figure 4.6 for a fit to a unit amplitude sine wave. Table 4.2 was calculated in MATLAB using a uniform grid of 10,000 points using both constrained and unconstrained non-linear optimizations. The optimization for THD appeared sensitive to the initial conditions, which indicates that the minimum might be relatively flat, or that multiple local minima exist. For these calculations, the initial conditions for all the optimizations are the case for zero error at the peak and at the zero crossings.

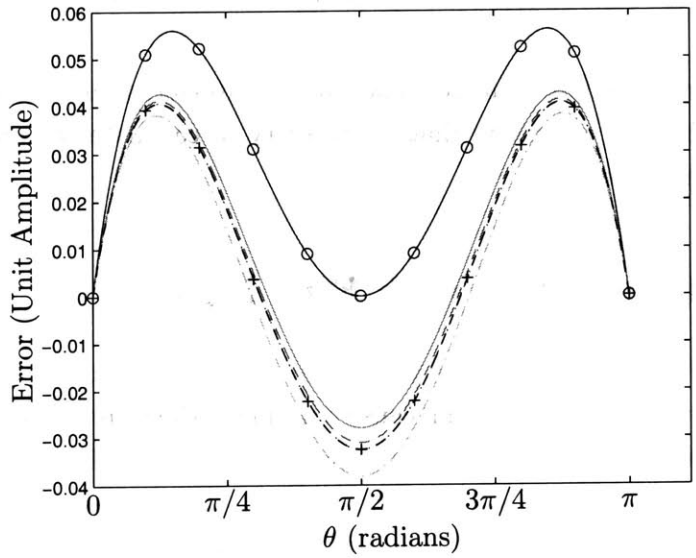
Quarter-Wave Approximation

The quarter-wave approximation is an optimal second order fit to a sine wave over the interval $[0, \pi/2]$. Table 4.3 and Figure 4.7 shows that this approximation results in better values in comparison to the half-wave case for both THD and error, respectively. These calculations were performed with 100,000 points over the quarter-wave interval. In the unconstrained case, it seems that a lower THD was achieved by fitting using least squares than for fitting by directly optimizing THD, possibly similar reasons that the THD fit was sensitive to initial conditions in the half-wave case.

The quarter-wave approximation with an $\mathcal{L}^\infty$-optimal fit results in slightly better THD and much better maximum error figures (1.67% and 1.65%, respectively) than in the 108-element



(a) Unconstrained Optimization



(b) Zero Error at Zero Crossings

Figure 4.6: Error from parabolic half-wave approximations to a sine function with unit amplitude. $\mathcal{L}^1(-)$; $\mathcal{L}^2(\cdots)$; $\mathcal{L}^\infty$ $(-\cdot)$; Maximal Fundamental $(--)$; Minimum THD$(-+-)$; Zero Error for Peak and Zero Crossing $(-\circ-)$.

Table 4.2: Coefficients for Half-Wave Sine Approximations (10,000 point discretization for all calculations).

Goal	x_0	x_1	x_2	$\mathscr{G}$	THD (%)
$\mathcal{L}^1$	0.9851	1.5708	−0.4270	143	2.84
$\mathcal{L}^2$	0.9802	1.5708	−0.4177	2.9829	2.64
$\mathcal{L}^\infty$	0.9719	1.5708	−0.4051	0.0281	2.95
Max. Fundamental	0.9831	1.6286	−0.4199	8.26×10^{-5}	6.77
Min. Distortion	0.9924	1.5708	−0.4229	0.0238	2.64
Zero Crossing Error					
Zero Error Peak	1.	$\pi/2$	$-4/\pi^2$		3.93
$\mathcal{L}^1$	0.9721	1.5708	−0.3940	240.30	3.93
$\mathcal{L}^2$	0.9675	1.5708	−0.3921	7.2249	3.93
$\mathcal{L}^\infty$	0.9618	1.5708	−0.3898	0.0382	3.93
Max. Fundamental	0.9689	1.5708	−0.3927	0.0131	3.93
Min. Distortion	1.0000	1.5708	−0.4053	0.0641	3.93

direct table lookup case (1.71% and 2.91%), in the case where there is no error at the zero crossings.

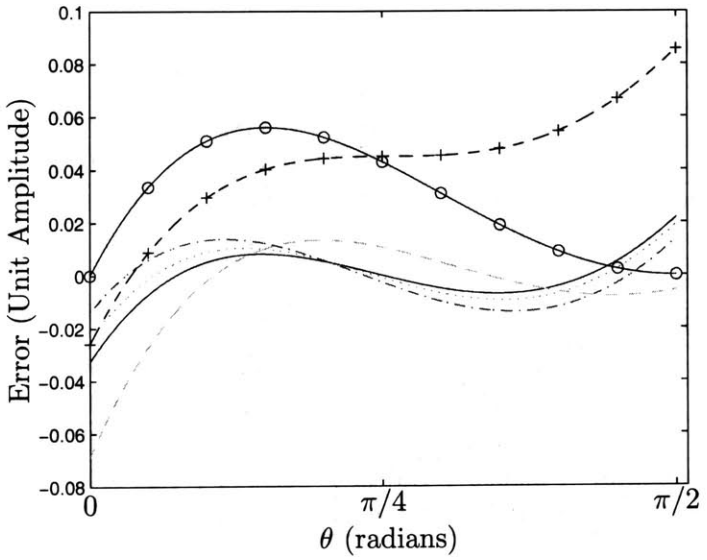
In addition to being useful as a sine reference for the inverter, a quarter wave approximation forms the basis for the $\cos^{-1}$ function that is necessary to calculate the phase output[1] of the synchronous current regulator.

4.3 Three Phase Filters with Four Legs

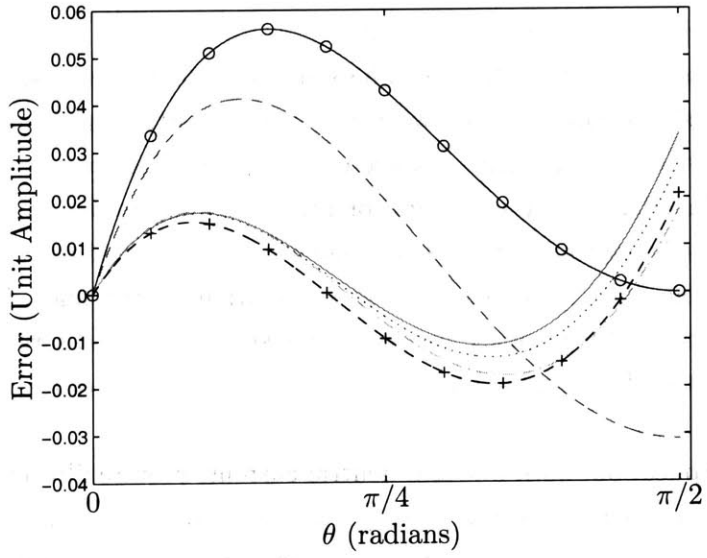
A number of issues arise from currents and voltages at the PWM frequency (10-30 kHz). Among these include capacitive currents which can cause wear in bearings and cause inadvertent ground loops that cause additional inverter loading and create additional difficultics in sensor measurement. The PWM waveform is rich with harmonics, which makes electromagnetic compatibility another issue even with relatively PWM switching frequencies. A number of authors have have tried to analyze [18] and mitigate [19] these issues.

Three-phase filters are used to reduce the PWM frequency content from the inverter in driving the stator. Although these filters represent additional cost and component count, it reduces losses in both the stator windings and the core in addition to eliminating resonances

[1]Equation 3.11.



(a) Unconstrained Optimization



(b) Zero Error at Zero Crossings

Figure 4.7: Error from parabolic quarter-wave approximations to a sine function with unit amplitude. $\mathcal{L}^1(-)$; $\mathcal{L}^2(\cdots)$; $\mathcal{L}^\infty$ $(-\cdot)$; Maximal Fundamental $(-\,-)$; Minimum THD$(-+-)$; Zero Error for Peak and Zero Crossing $(-\circ-)$.

Table 4.3: Coefficients for Quarter-Wave Sine Approximations. (10,000 point uniform discretization for all calculations).

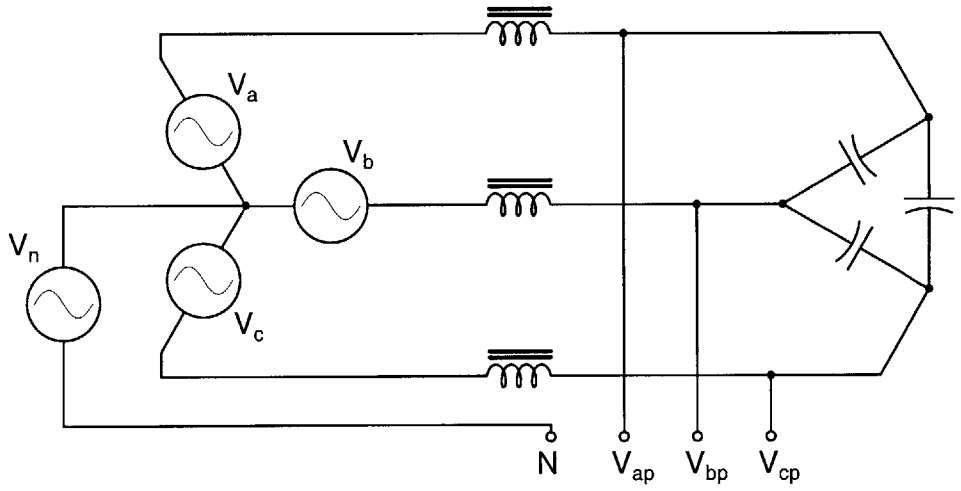
Goal	x_0	x_1	x_2	$\mathscr{G}$	THD (%)
$\mathcal{L}^1$	1.0341	1.7596	−0.3445	34.63	1.27
$\mathcal{L}^2$	1.0325	1.7676	−0.3382	0.3518	1.19
$\mathcal{L}^\infty$	1.0273	1.7723	−0.3315	0.0139	1.32
Max. Fundamental	0.9948	1.7676	−0.4133	8.76×10^{-5}	1.32
Min. Distortion	1.0453	1.5708	−0.3424	0.0492	1.32
Zero Crossing Error					
Zero Error Peak	1.	$\pi/2$	$-4/\pi^2$		3.8
$\mathcal{L}^1$	1.0612	1.8730	−0.3025	50.95	1.66
$\mathcal{L}^2$	1.0524	1.8563	−0.3054	0.6804	1.65
$\mathcal{L}^\infty$	1.0378	1.8261	−0.3112	0.0175	1.67
Max. Fundamental	0.9699	1.5744	−0.3913	4.28×10^{-5}	3.81
Min. Distortion	1.0455	1.8564	−0.3034	0.0272	1.65

due to parasitics and inductance nonlinearities from high frequency effects, which can couple changes in zero sequence current to dq-axis flux.

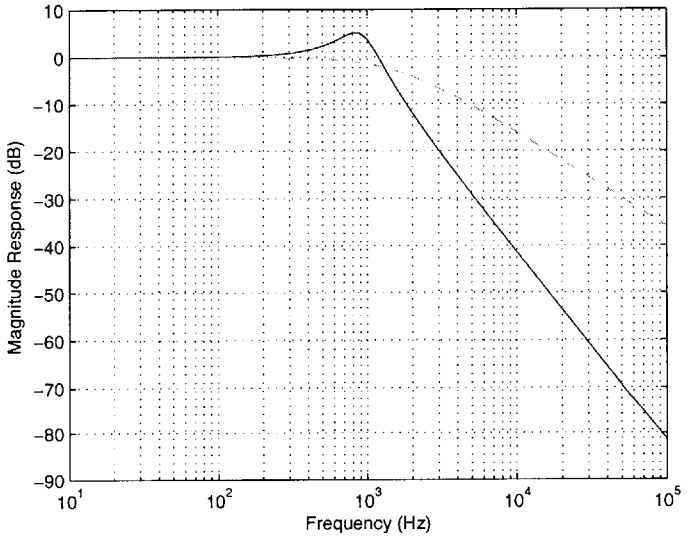
There are a number of ways to do three phase filters; among the most convenient is to use series inductors and shunt capacitors between the inverter output and the motor. Figure 4.8 shows such a filter with the capacitors connected across the line in a Δ configuration which similar to that used in [20, 21]. This configuration filters only line-to-line currents and is useful in circuits where the wye is ungrounded, or where one wants the neutral to have a low impedance. It is apparent from the filter response that the attenuation on the line with frequency is 2^{nd} order, but only a 1^{st} order LR response between the resistive load and series inductance for the the zero sequence path. The single-phase, off-line analogy is using differential inductors with *X-capacitors* (line-to-line).

In a multi-use application, both line and neutral currents require filtering. The filter in Figure 4.9 attenuates high frequencies from both phase and zero sequence components identically. This topology suffers if there are mismatches in the line-to-neutral capacitors by unbalancing the response, hence converting some of the differential current to common mode, which is reminiscent of what happens with *Y-capacitors* in single-phase, off-line applications.

The improved filter illustrated in Figure 4.10 is more complicated because it uses a coupled inductor along with series inductors and shunt capacitors. This circuit is a novel three-phase, four-wire derivation of the coupled inductor filters in [22, 23, 24] In this topology, the zero

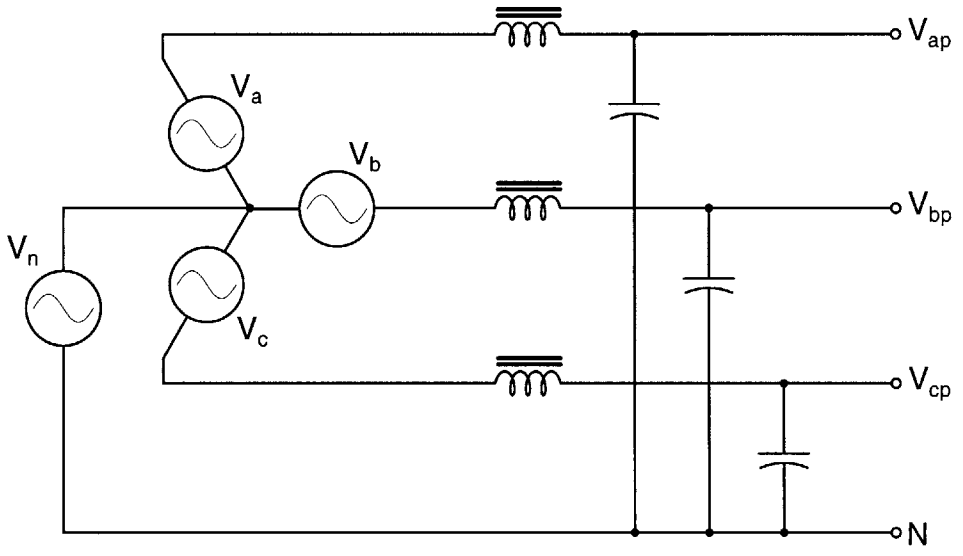


(a)

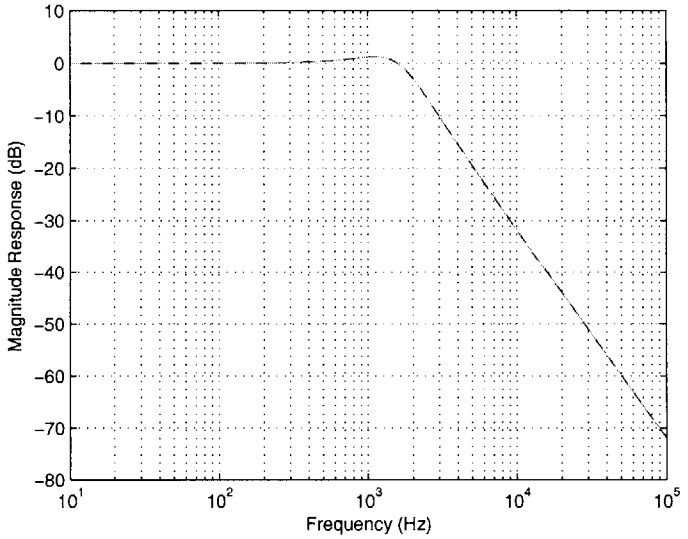


(b) Filter Response

Figure 4.8: Line-to-line filter with delta-connected capacitors do not filter zero sequence components, but provide additional inductance in the zero sequence path. Filter response with Y-connected 10Ω resistive load. (−) Line Response; (– –) Zero Sequence Response.(LTSpice)



(a)



(b) Filter Response

Figure 4.9: Typical three phase filter for both phase-to-phase and zero sequence components. Zero sequence may be introduced by mismatches in filter components. Filter response with Y-connected 10Ω resistive load. (−) Line Response; (- - -) Zero Sequence Response.(LTSpice)

sequence components are filtered separately from the positive and negative sequence components. Because the coupled inductor is connected similarly to a common-mode choke, the net flux in the core is only that of the zero sequence current if the leakages between each leg is small enough. This topology is necessarily better than using just a common-mode choke because power is a significant amount of power is expected to be carried on the neutral (i.e. zero sequence), which benefits by having a filter with a 2nd order response; this is quite different from the single-phase, off-line case where one tries to minimize the current in the ground wire.

There are a number of considerations in the design of these three-phase filters from the perspective of doing current control (§3.3): ignoring the filter by placing the filter breakpoints well above the current-loop crossover frequency, or including the filter in the plant dynamics. In either case, the fact that filter damping depends on the resistance seen at the stator drive terminal (i.e. rotor effective resistance and secondary stator rectifier load) becomes an issue at light loads. Typically, the series resistance of the inductor and the stator provide a bound on the damping. If additional damping is required, small series resistances can be added to the capacitor, and if the degradation in the filter response by doing this is not acceptable, explicit dampening legs can be placed in parallel to the capacitors.

4.4 Minimal Implementations for Phase Current Measurement

For field-oriented drive control, only the positive and negative sequence (or equivalently, the d-axis and q-axis) currents need to be measured because zero sequence current does not contribute to torque. Zero sequence current measurement, however, may be useful for detecting overcurrent conditions in the power conversion circuit.

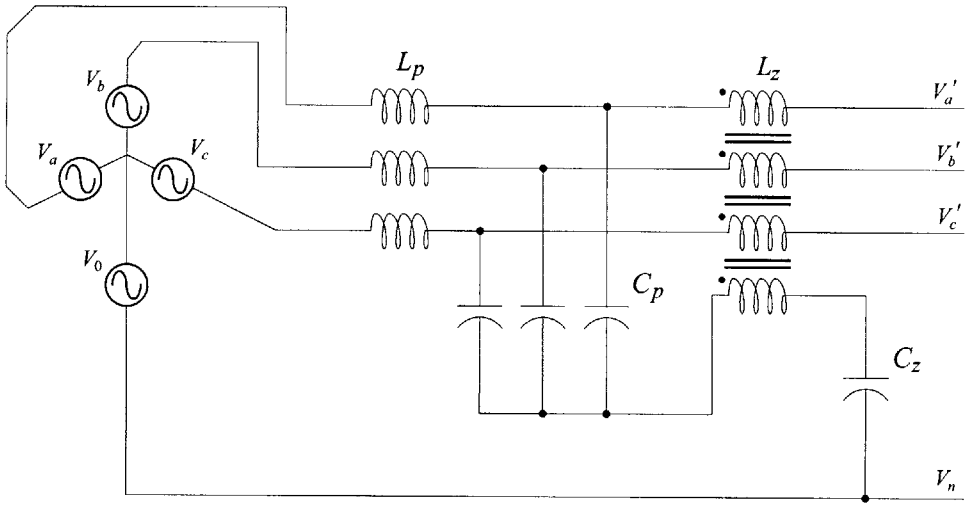
4.4.1 Balanced Three Phase, Wye Grounded and Ungrounded

In a balanced three-phase line,

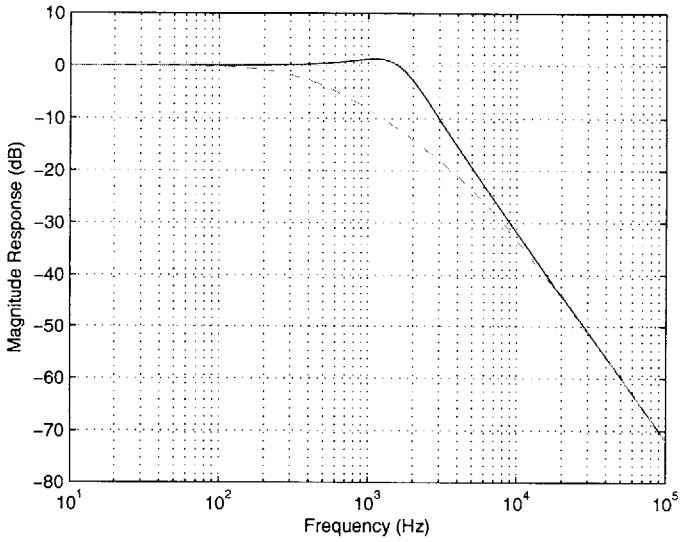
$$i_a + i_b + i_c = 0. \tag{4.9}$$

In this case, there are only two independent variables (e.g. i_a, i_b) to be measured. The third can be calculated, $i_c = -(i_a + i_b)$.

When the the wye is ungrounded, there can be no zero sequence current, which is the



(a)



(b) Filter Response

Figure 4.10: Three phase filter with additional coupled inductor stage for zero sequence components. Filter response with Y-connected 10Ω resistive load. (−) Line Response; (- - -) Zero Sequence Response.(LTSpice)

simplest balanced three-phase case for current measurement.

$$\begin{aligned} i_a &= I\cos(\omega t) \\ i_b &= I\cos(\omega t - 2\pi/3) \\ i_c &= I\cos(\omega t + 2\pi/3) \end{aligned}$$

When the wye is grounded, but the currents are still balanced even though there is a zero sequence current, i.e.

$$\begin{aligned} i'_a &= \frac{i_0}{3} + i_a = \frac{i_0}{3} + I\cos(\omega t) \\ i'_b &= \frac{i_0}{3} + i_b = \frac{i_0}{3} + I\cos(\omega t + 2\pi/3) \\ i'_c &= \frac{i_0}{3} + i_c = \frac{i_0}{3} + I\cos(\omega t - 2\pi/3), \end{aligned}$$

Equation 4.9 still holds, and only two current sensors are needed to recover the sequence currents. This can be accomplished by subtracting the 1/3 of the zero sequence current from each of the phases on each of the sensors.

4.4.2 Multiple Stator

In a machine with multiple stators, a field oriented controller requires access to the component of the phase currents in the stator that links flux to the rotor. While a detailed model of the leakage inductances and coupling coefficients is required for the exact currents, these parameters are generally time-varying and nonlinear because of the effects of temperature and magnetic saturation.

Several approximations are appropriate in the Aardvark induction machine. Because the stator primary and secondaries are wound in-hand, one can assume that these windings are well-coupled. This means that the current in the magnetizing inductances is small relative to the overall phase current. The remaining phase current then consists of the currents from the rotor and the secondaries reflected back to the primary.

In a machine with unity turns ratio between the primary and secondaries and with good coupling in the stator-stator windings, as it is in the Aardvark machine, the phase currents reflect back to the primary. In this way, the correct linear combination of current measurements from the different stator windings recovers the rotor-linked flux current as Figures 4.11 and 4.12 illustrates. The recovery of the linked current in these figures is calculated based on the assumption that there is good coupling of the zero sequence, which is the case since a zero

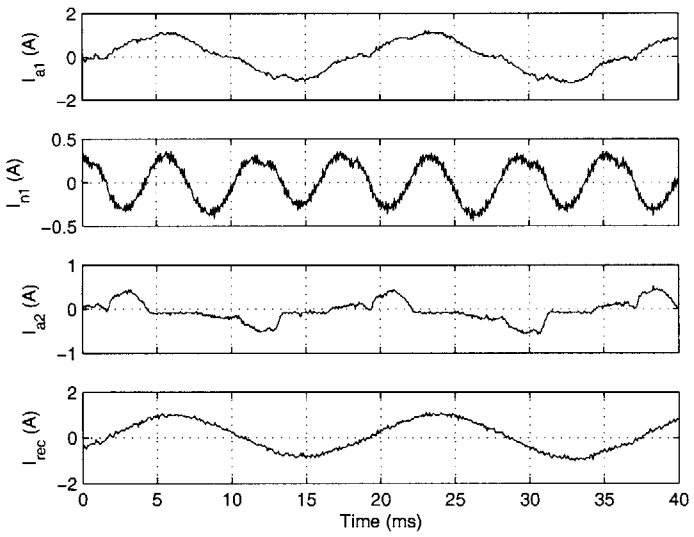


Figure 4.11: Illustration of how the phase current that links flux to the rotor (I_{rec}) can be recovered from a linear combination of current measurements from different stator windings. I_{a1} is the current phase a of the primary, I_{n1} is from the neutral of the primary, and I_{a2} is that of phase a of the secondary.

sequence transformer was part of the circuit[2]. The recovered linked current is

$$I_{rec} = I_{a1} - I_{a2} - \frac{2}{3} I_{n1}.$$

4.4.3 *Estimation from Inverter Current Out of the DC Bus*

A common objection to the idea of using an induction motor as a multi-use machine is that such a large number of current sensors are required. In a "normal" machine, the phase currents are balanced and there is no zero sequence current, hence only two current sensors are required for field-oriented control. With a multi-use machine, one expects that at best two current sensors are required for each addition stator winding.

The idea of using a single current sensor on the dc link along with the already available knowledge of the switching states has been demonstrated in [25, 26]. By extending this idea from the inverter dc link to the output dc buses, only one additional current sensor is required

[2]See Figure 2.10.

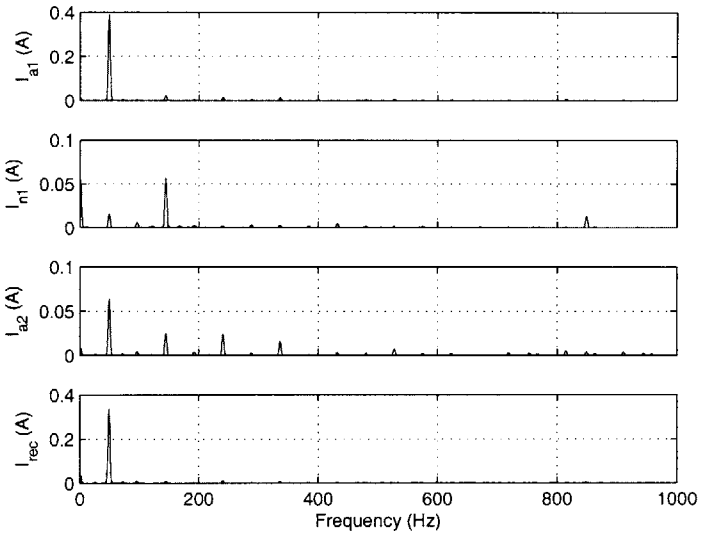


Figure 4.12: The FFT of the currents used in the recovery of current that links flux to the rotor. The dc components were subtracted and a Hanning window was used.

for each secondary stator winding, which will probably be necessary anyway for short-circuit and overload sensing and protection.

Chapter 5

Firmware Design

The DSP programme is multi-tasking and real-time. The typical approach is to use a real-time operating system (RTOS). In this design, I have used a program architecture that does not use an RTOS, but is multi-tasking and real-time.

The architecture uses a round-robin approach within a superloop with preemptive tasks handled by interrupts. Although events are handled within the super loop, tasks do not run to completion, but rather, are time-sliced. In addition, a number of persistent data structures are shared publicly among the tasks and are not explicitly protected by an operating system. Semaphores are included as part of these data structures a means for mutual exclusion, but each task is responsible for checking and setting these semaphores.

This architecture is advantageous for small programs where both the intellectual and programmatic overhead of an RTOS is undesired. A certain coding discipline is required as is understanding the timing requirements of particular routines. In any case, multi-tasking, real-time firmware is not for the unwary.

5.1 Time Slicing Algorithm

The task functions in the main loop are time triggered from a hardware counter. As each function is called in the main loop, it checks the counter to determine whether it should be in an active state or a wait state. When the function, or more precisely the task, becomes active, one iteration is performed. Actually, the function may actually iterate several times, it depends on just how the task is defined. This task iteration runs to completion and only can be preempted by an interrupt. For this scheme to work, the task iteration must complete within the timing limits set by the timing budget, which must be decided at design-time. In addition, interrupts add to the iteration time, hence a timing margin for each task is needed

for the worst-case interrupt scenario, which includes context saving and loading along with the actual service routine. It goes without saying that there can only be few interrupts and that the interrupt service routines (ISR) must be short. There are a few ways to ensure this, for example the ISR can either complete the task quickly or set a flag to wake a normal task. In this architecture, it is important to note that tasks are created at design-time and cannot be spawned.

Table 5.1 shows the execution times and approximate periods for each of the tasks tasks. *update_PWM()* and *controlV3()* are normal task functions while *periodic_ISR()* is an ISR that completes its task quickly.

One may notice that the periodicity for the example functions in Table 5.1 to be rather slow. Let's examine the slowest, *control_V3()*. Because the output voltage is controlled on the peaks of a three-phase waveform, in which 60 Hz is the highest frequency, the fastest that one can actuate a control is three times 60 Hz or 180 Hz, which also true for π-phase 3^{rd} harmonic control. The most periodic is *periodic_ISR()*, which updates the PWM duty cycle from a 108-element sine reference table. At the fastest which is 60 Hz, the update rate is $1/108^{th}$ of a 60 Hz period which is 154 μs.

Table 5.1: Examples of Task Timing

Task	Execution	Period
update_PWM()	111 μs	500 μs
periodic_ISR	8 μs	154 μs
controlV3()	117 μs	1 ms

5.2 Data Structures

As currently implemented, several persistent data structures are shared among different tasks. For example *sin_pwm* contains information about the switching frequency, fundamental and harmonic frequencies and amplitudes, among others; *VFRamping* is used for volts per hertz ramping and contains information about the voltage and frequency steps of the drive, as well as the ramp rate.

While it may be considered memory intensive to have too many persistent data structures, in the currently limited use case for this application, all the tasks remain active, so it seems appropriate in many cases to use these persistent data structures rather than maintain a *mailbox*, or some other means of data passing.

Hardware registers and other peripherals are generally abstracted from the general tasks by a data structure such as *sin_pwm*. Each of these data structures maintained by a separate task such as *update_PWM()* that updates the peripherals and lower level drivers and ensures that there are no collisions during these updates.

5.3 Output Voltage Regulation Module

The algorithm for the regulation of dc rectifier output using the π-phase 3^{rd} harmonic is illustrated in the UML[1] activity diagram in Figure 5.1. This diagram describes what happens during a single time slice.

The maximum amplitude for the 3^{rd} harmonic voltage is determined by the available voltage headroom, i.e. 100% PWM duty cycle at the peak of the inverter output, which is a combination of 1^{st} and 3^{rd} harmonics.

5.4 Synchronous PWM Module

Updates to the PWM module occur all at once. During an update all tasks are mutually excluded from the PWM data structure. *update_PWM()* has the job of brokering the transfer of PMW parameters from the higher level tasks to the interrupt service routine as well as making all the low-level calculations. Every task gets a chance to update *sin_pwm* because of the end position in the round-robin queue of *update_PWM*. *update_PWM()* checks a semaphore to see if the *periodic_interrupt_isr()* is being handled, and if not, excludes the this interrupt service routine and updates the parameters.

Synchronous PWM is implemented so that the switching frequency is an integral multiple of the highest zero sequence harmonic and hence also the fundamental. The actual multiple used to determine the switching frequency changes as upper and lower boundaries. To prevent oscillations near the boundaries, hysteresis is included in the algorithm.

5.5 Synchronous Current Controller

The main function for the synchronous current controller is illustrated in the activity diagram in Figure 5.2. The synchronous current controller keeps track of the voltage and current limits and calls appropriate handlers during voltage saturation and soft overcurrents.

[1](UML—Universal Modeling Language, although it isn't claimed that these diagrams are compliant to the most recent standards.

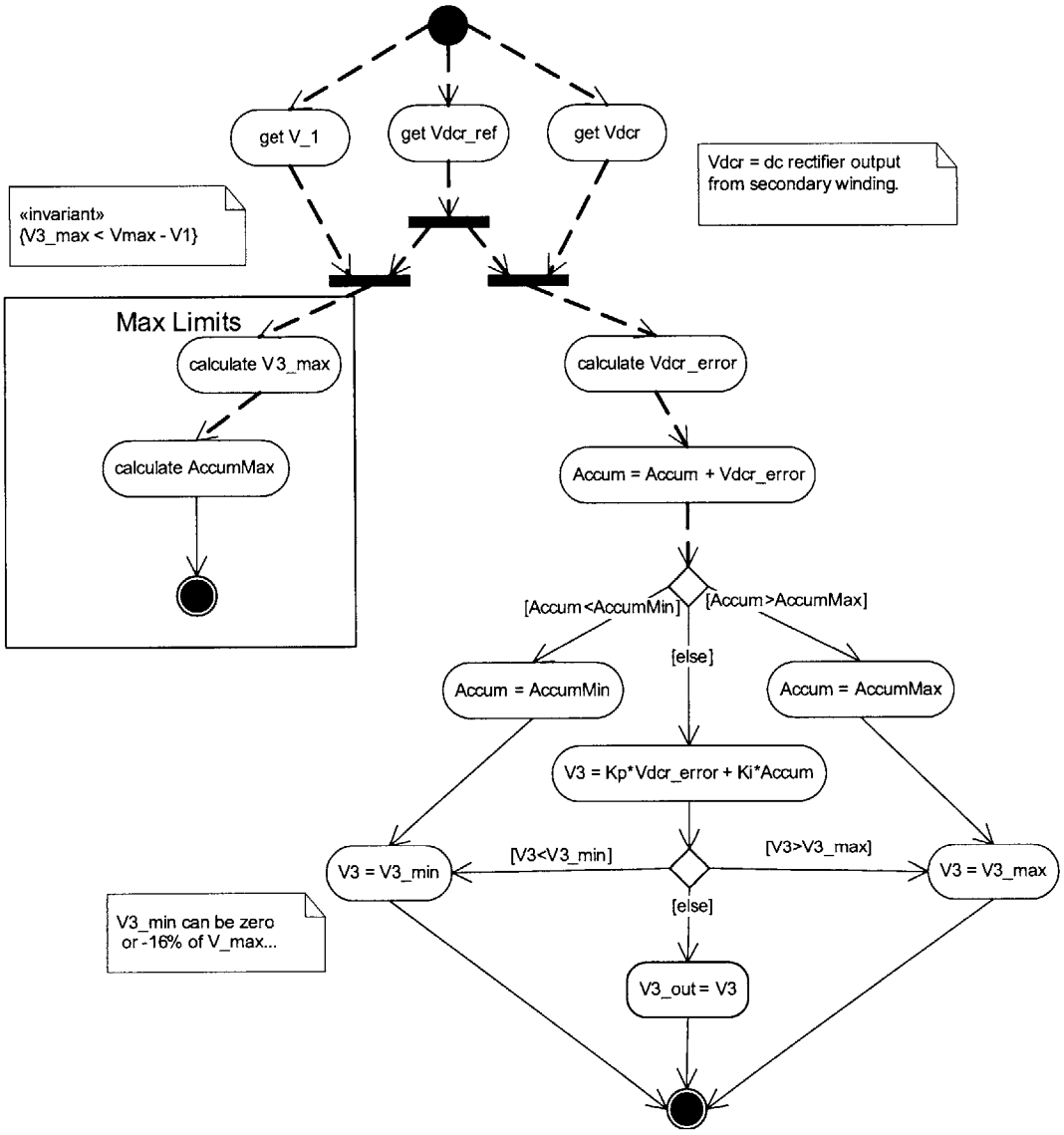


Figure 5.1: Activity diagram for closed-loop control of dc output using a proportion-integral controller with accumulator anti-windup along with overflow and underflow control.

Figure 5.2: Activity diagram for control of stator current in the synchronous frame using a proportional-integral controller with accumulator anti-windup along with overflow and underflow control.

Chapter 6

Conclusions and Future Work

A number of challenges were surmounted which resulted in a number of innovations. An enabling concept has been the use of zero sequence harmonics where the peak of the fundamental coincides with a peak of the harmonic to control the dc rectifier output of a secondary on the stator. This concept has been demonstrated in a 1.5 hp induction machine using a closed-loop, proportional-integral controller to control a π-phase third harmonic so that a dc output can be regulated. Another enabling concept has been the use of a zero sequence transformer, which allows the efficient transfer of zero sequence harmonics to the secondary.

To realize the system, an inverter and drive control had to be developed. The design of the inverter was crucial in that it had produce an accurate sine wave and superposing it with any other harmonic without overwhelming the digital signal processor. Two implementations were carefully analyzed: a table-based sine reference and a parabolic approximation to pieces of the sine. Ultimately, the parabolic approximation appears superior to a stored table that can easily fit into the DSP. In the future, it may be possible to implement a 4^{th}-order approximation using a doubly-iterated parabolic approximation, which in the literature appears to have excellent performance.

Two drive control methods have been developed. The volts per hertz drive has been successfully implemented while the indirect field-oriented controller has been shown to work in simulation, but with results of the implementation forthcoming. A number of practical and theoretical issues arise with the field-oriented controller. These include anti-windup, cross-coupling of the dq-axis variables in the plant and the implementation of current regulator. The synchronous current regulator for a multi-use machine is different than what is commonly used in field-oriented controllers in that it must supply a drive current that links flux to the rotor, but must also provide a voltage with which to regulate a dc rectifier output. In a sense, it must be both a current and a voltage source. Such a thing is possible using feedback so that

the fundamental is controlled as a current loop and the zero sequence harmonics controlled by a voltage loop. To achieve this, the inputs are conceptually the frequencies and voltage amplitudes of the fundamental and harmonics, along with a phase, which are essentially polar coordinates, as opposed to the Cartesian d- and q axis variables.

Included is a preliminary analysis of what might be possible with the idea of multi-use machine, by quoting some rough scaling laws.

To achieve this, a number of analysis and simulation tools have been developed, in MATLAB, Simulink and SPICE. Some concepts, some of which are novel, have been reframed into appropriate contexts.

Future results include implementation results from the field-oriented controller along with a careful analysis of an example design for a multi-use machine, so that better comparisons with competing technologies can be made.

Appendix A

SPICE Deck for Multistator Transformer

A.1 PSPICE–Wye-Ungrounded

```
* Schematics Netlist *

L_L4          $N_0001 $N_0002  {Lc/2}
L_L6          $N_0003 $N_0004  {Lc/2}
L_L5          vz0 $N_0002  {Lc/2}
L_L8          $N_0003 vz0  {Lc/2}
L_L7          $N_0005 $N_0006  {Lc/2}
L_L14          $N_0007 $N_0008  {Lc/2}
L_L17          $N_0009 $N_0010  {Lc/2}
L_L15          $N_0007 vz1  {Lc/2}
L_L13          $N_0011 $N_0012  {Lc/2}
L_L18          vz1 $N_0012  {Lc/2}
L_L16          vz1 $N_0010  {Lc/2}
L_L1          $N_0013 $N_0001  {Llk}
L_L2          $N_0014 $N_0004  {Llk}
L_L3          $N_0015 $N_0005  {Llk}
L_L10          $N_0011 $N_0016  {Llk}
L_L11          $N_0008 $N_0017  {Llk}
L_L12          $N_0009 $N_0018  {Llk}
V_Vb          $N_0019 $N_0020
+SIN 0 {Vk1} {f1} 0 0 240
V_Vc          $N_0021 $N_0020
+SIN 0 {Vk1} {f1} 0 0 120
V_Vn          $N_0020 0
+SIN 0 {Vk3} {f3} 0 0 0
D_D8          $N_0022 v01 Dbreak
```

Figure A.1: Wye-Ungrounded

```
D_D9          $N_0023 v01 Dbreak
D_D10          $N_0024 v01 Dbreak
D_D11          v02 $N_0022 Dbreak
D_D12          v02 $N_0023 Dbreak
D_D13          v02 $N_0024 Dbreak
Kn_K1          L_L4 L_L6
+ L_L13 L_L14   {kp}
Kn_K4          L_L5 L_L7
+ L_L17 L_L18   {kp}
Kn_K7          L_L8 L_L9
+ L_L15 L_L16   {kp}
R_R1          $N_0025 $N_0013  1
R_R2          $N_0019 $N_0014  1
R_R3          $N_0021 $N_0015  1
R_R4          $N_0016 $N_0022  1
R_R5          $N_0017 $N_0023  1
R_R6          $N_0018 $N_0024  1
V_Va          $N_0025 $N_0020
+SIN 0 {Vk1} {f1} 0 0 0
C_C1          v00 v01  1000uF
C_C2          v02 v00  1000uF
R_R7          v00 v01  200
R_R8          v02 v00  200
R_R11          v00 vz1  100MEG
R_R9          0 vz1  1
L_L9          vz0 $N_0006  {Lc/2}

.PARAM          f1=60 f3=180
.PARAM          kp=1 k0=1
.PARAM          Lc=100mH Llk=100uH
.PARAM          Vk1=300 Vk3=0.001

** Analysis setup **
.tran 0ns 2
.OPTIONS ABSTOL=10pA
.OPTIONS RELTOL=0.005
.OPTIONS VNTOL=10uV
.OP

* From [PSPICE NETLIST] section of pspiceev.ini:
.lib "nom.lib"
```

```
.INC "three-legged transformer4.net"
.INC "three-legged transformer4.als"

.probe

.END
```

A.2 PSPICE–Wye-Grounded

```
* Schematics Netlist *

L_L4          $N_0001 $N_0002  {Lc/2}
L_L6          $N_0003 $N_0004  {Lc/2}
L_L5          0 $N_0002  {Lc/2}
L_L8          $N_0003 0  {Lc/2}
L_L7          $N_0005 $N_0006  {Lc/2}
L_L9          0 $N_0006  {Lc/2}
L_L14          $N_0007 $N_0008  {Lc/2}
L_L17          $N_0009 $N_0010  {Lc/2}
L_L15          $N_0007 vz1  {Lc/2}
L_L13          $N_0011 $N_0012  {Lc/2}
L_L18          vz1 $N_0012  {Lc/2}
L_L16          vz1 $N_0010  {Lc/2}
L_L1          $N_0013 $N_0001  {Llk}
L_L2          $N_0014 $N_0004  {Llk}
L_L3          $N_0015 $N_0005  {Llk}
L_L10          $N_0011 $N_0016  {Llk}
L_L11          $N_0008 $N_0017  {Llk}
L_L12          $N_0009 $N_0018  {Llk}
C_C1          v00 v01  1000uF
C_C2          v02 v00  1000uF
R_R8          v02 v00  200
V_Vb          $N_0019 $N_0020
+SIN 0 {Vk1} {f1} 0 0 240
V_Vc          $N_0021 $N_0020
+SIN 0 {Vk1} {f1} 0 0 120
V_Vn          $N_0020 0
+SIN 0 {Vk3} {f3} 0 0 0
D_D8          $N_0022 v01 Dbreak
D_D9          $N_0023 v01 Dbreak
D_D10          $N_0024 v01 Dbreak
```

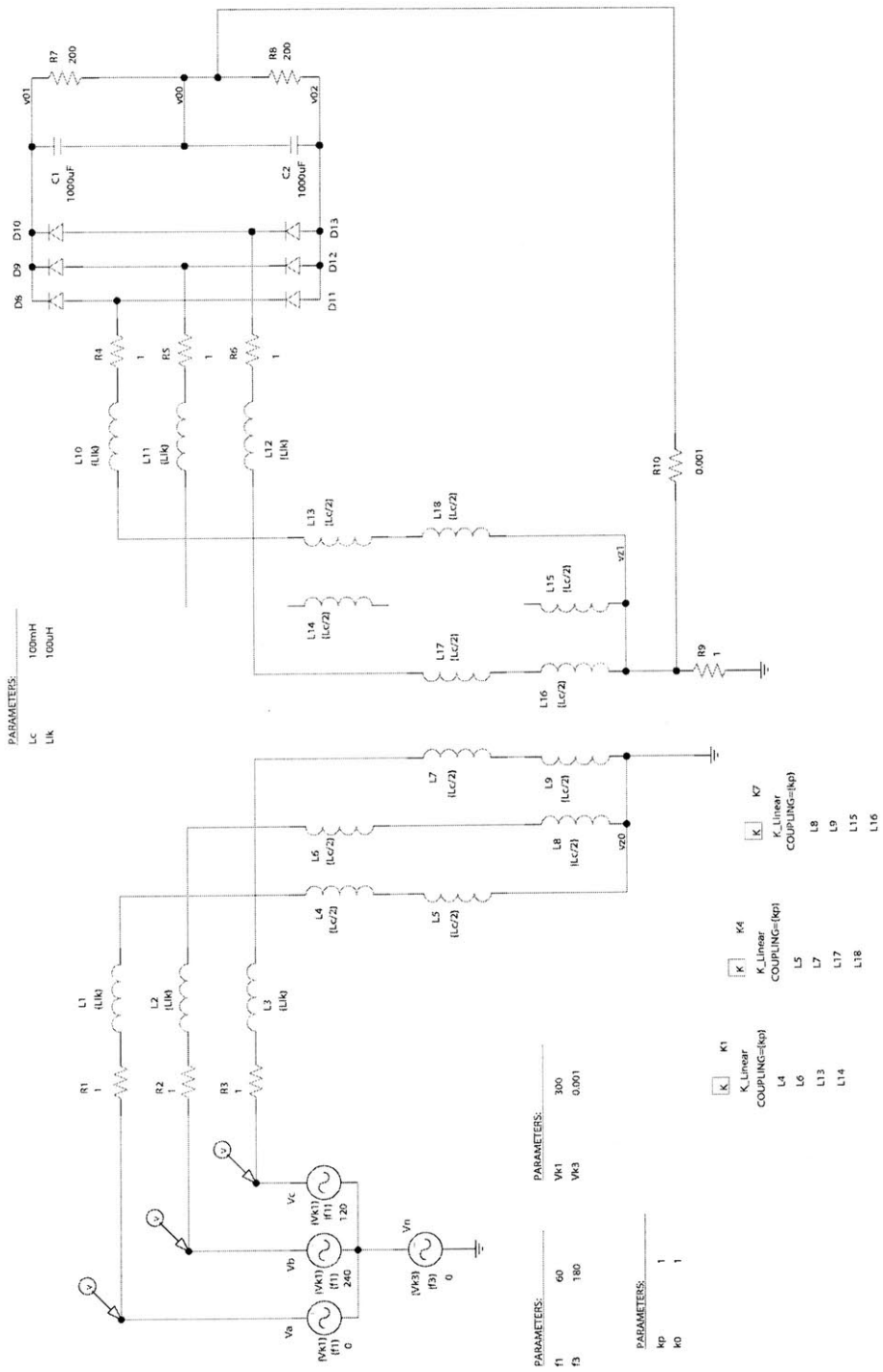


Figure A.2: Wye-Grounded

```
D_D11          v02 $N_0022 Dbreak
D_D12          v02 $N_0023 Dbreak
D_D13          v02 $N_0024 Dbreak
Kn_K1          L_L4 L_L6
+ L_L13 L_L14   {kp}
Kn_K4          L_L5 L_L7
+ L_L17 L_L18   {kp}
Kn_K7          L_L8 L_L9
+ L_L15 L_L16   {kp}
R_R1          $N_0025 $N_0013  1
R_R2          $N_0019 $N_0014  1
R_R3          $N_0021 $N_0015  1
R_R4          $N_0016 $N_0022  1
R_R5          $N_0017 $N_0023  1
R_R6          $N_0018 $N_0024  1
V_Va          $N_0025 $N_0020
+SIN 0 {Vk1} {f1} 0 0 0
R_R7          v00 v01  200
R_R9          0 vz1  1
R_R10          vz1 v00  0.001

.PARAM          Lc=100mH Llk=100uH
.PARAM          f1=60 f3=180
.PARAM          kp=1 k0=1
.PARAM          Vk1=300 Vk3=0.001

** Analysis setup **
.tran 0ns 2
.OP

* From [PSPICE NETLIST] section of pspiceev.ini:
.lib "nom.lib"

.INC "three-legged transformer2.net"
.INC "three-legged transformer2.als"

.probe

.END
```

A.3 PSPICE–Wye-Grounded with Zero Sequence Transformer

```
* Schematics Netlist *

L_L4          vpa $N_0001  {Lc/2}
L_L6          $N_0002 vpb  {Lc/2}
L_L5          vz0 $N_0001  {Lc/2}
L_L8          $N_0002 vz0  {Lc/2}
L_L14          $N_0003 vsb  {Lc/2}
L_L15          $N_0003 vz1  {Lc/2}
L_L13          vsa $N_0004  {Lc/2}
L_L1          $N_0005 vpa  {Llk}
L_L2          $N_0006 vpb  {Llk}
L_L3          $N_0007 vpc  {Llk}
L_L10          vsa $N_0008  {Llk}
L_L11          vsb $N_0009  {Llk}
L_L12          vsc $N_0010  {Llk}
C_C1          v00 v01  1000uF
C_C2          v02 v00  1000uF
R_R8          v02 v00  200
D_D11          v02 $N_0011 Dbreak
D_D12          v02 $N_0012 Dbreak
D_D13          v02 $N_0013 Dbreak
Kn_K1          L_L4 L_L6
+ L_L13 L_L14   {kp}
Kn_K4          L_L5 L_L7
+ L_L17 L_L18   {kp}
Kn_K7          L_L8 L_L9
+ L_L15 L_L16   {kp}
L_L17          vsc $N_0014  {Lc/2}
L_L7          vpc $N_0015  {Lc/2}
L_L18          vz1 $N_0004  {Lc/2}
R_R7          v00 v01  200
D_D8          $N_0011 v01 Dbreak
D_D10          $N_0013 v01 Dbreak
D_D9          $N_0012 v01 Dbreak
R_R1          $N_0016 $N_0005  1
R_R2          $N_0017 $N_0006  1
R_R3          $N_0018 $N_0007  1
R_R6          $N_0010 $N_0013  1
R_R5          $N_0009 $N_0012  1
```

Figure A.3: Wye-Grounded with Zero Sequence Transformer

```
R_R4          $N_0008 $N_0011  1
R_R9          0 v00  1
L_L16          vz1 $N_0014  {Lc/2}
L_L9          vz0 $N_0015  {Lc/2}
K_TX2          L1_TX2 L2_TX2 1
L1_TX2          $N_0019 0 100mH
L2_TX2         $N_0020 v00 100mH
R_R10          $N_0019 vz0  1
V_Va          $N_0016 $N_0021
+SIN 0 {Vk1} 60 0 0 0
V_Vb          $N_0017 $N_0021
+SIN 0 {Vk1} 60 0 0 -120
V_Vc          $N_0018 $N_0021
+SIN 0 {Vk1} 60 0 0 +120
V_Vn          $N_0021 0
+SIN 0 {Vk3} 180 0 0 180
L_L20          vz1 $N_0020  1mH

.PARAM          f1=60 f3=180
.PARAM          kp=1 k0=1
.PARAM          Lc=100mH Llk=1mH
.PARAM          Vk1=0.1 Vk3=299.9

** Analysis setup **
.tran 0ns 2
.OP


* From [PSPICE NETLIST] section of pspiceev.ini:
.lib "nom.lib"

.INC "three-legged transformer3.net"
.INC "three-legged transformer3.als"


.probe


.END
```

Appendix B

MATLAB Script for Parabolic Approximations of Sine Function

B.1 Script

```
%%M-file to generate plots and coefficients for various
%%2nd order sine approximations
clear;
npts = 10000;
tmesh = 0:pi/npts:pi;
x0 = [-4/pi^2 pi/2 1]';
str = cell(11,11);
%L1 fit
k=1;
[y,f] = fminsearch(@(x) h1fit(x,tmesh),x0)
[r,a1,df] = thd(y,tmesh,pi);
str(k,1) = {'L1 Fit'}
for i = 2:4
    str(k,i) = {y(i-1)};
end
str(k,5) = {'b-'};
str(k,6) = {f};
str(k,7) = {4*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
str(k,8) = {r};
str(k,9) = {a1};
str(k,10) = {df};

%L2 fit
k=k+1;
str(k,1) = {'L2 Fit'}
[y,f] = fminsearch(@(x) h2fit(x,tmesh),x0)
[r,a1,df] = thd(y,tmesh);
for i = 2:4
```

```
29      str(k,i) = {y(i-1)};
    end
31  str(k,5) = {'g:'};
    str(k,6) = {f};
33  str(k,7) = {4*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
    str(k,8) = {r};
35  str(k,9) = {a1};
    str(k,10) = {df};
37

39  %Linf fit
    k=k+1;
41  str(k,1) = {'Linf Fit'}
    [y,f] = fminsearch(@(x) hinffit(x,tmesh),x0)
43  [r,a1,df] = thd(y,tmesh);
    for i = 2:4
45      str(k,i) = {y(i-1)};
    end
47  str(k,5) = {'r-.'};
    str(k,6) = {f};
49  str(k,7) = {4*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
    str(k,8) = {r};
51  str(k,9) = {a1};
    str(k,10) = {df};
53

55  %Min Distortion Fit
    k=k+1;
57  str(k,1) = {'Min Distortion Fit'}
    [y,f] = fminsearch(@(x) mindistfit(x,tmesh),x0)
59  [r,a1,df] = thd(y,tmesh);
    for i = 2:4
61      str(k,i) = {y(i-1)};
    end
63  str(k,5) = {'c--'};
    str(k,6) = {f};
65  str(k,7) = {4*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
    str(k,8) = {r};
67  str(k,9) = {a1};
    str(k,10) = {df};
69
    %Min Distortion Fit2
71  k=k+1;
    str(k,1) = {'Min Dist Fit2'}
73  x00 = [-0.4177 1.5708 0.9802];  %Use L2 guess for initial fit
    [y,f] = fminsearch(@(x) thdopt(x,tmesh),x00)
75  [r,a1,df] = thd(y,tmesh);
    for i = 2:4
```

```
77      str(k,i) = {y(i-1)};
    end
79  str(k,5) = {'k--'};
    str(k,6) = {f};
81  str(k,7) = {4*mindistfit(y,tmesh)/npts}; %half wave = 4/npts;  qtr=8/npts
    str(k,8) = {r};
83  str(k,9) = {a1};
    str(k,10) = {df};
85  str(k,11)={'k+'};

87  %Zero pk and crossing error
    k=k+1;
89  str(k,1) = {'Pk and Zero Cross'}
    y = x0;
91  [r,a1,df] = thd(y,tmesh);
    for i = 2:4
93      str(k,i) = {y(i-1)};
    end
95  str(k,5) = {'b-'};
    str(k,6) = {f};
97  str(k,7) = {4*mindistfit(y,tmesh)/npts}; %half wave = 4/npts;  qtr=8/npts
    str(k,8) = {r};
99  str(k,9) = {a1};
    str(k,10) = {df};
101 str(k,11) = {'bo'};

103 %L1 fit, zero crossing error
    k=k+1;
105 str(k,1) = {'L1 Fit Zero Cross'}
    options = optimset('LargeScale','off');
107 [y,f] = fmincon(@(x) h1fit(x,tmesh),x0,[],[],[],[],[],[],...
        @confuneq,options)
109 [c,ceq] = confuneq(y)
    [r,a1,df] = thd(y,tmesh);
111 for i = 2:4
        str(k,i) = {y(i-1)};
113 end
    str(k,5) = {'g-'};
115 str(k,6) = {f};
    str(k,7) = {4*mindistfit(y,tmesh)/npts}; %half wave = 4/npts;  qtr=8/npts
117 str(k,8) = {r};
    str(k,9) = {a1};
119 str(k,10) = {df};

121 %L2 fit, zero crossing error
    k=k+1;
123 str(k,1) = {'L2 Fit Zero Cross'}
    options = optimset('LargeScale','off');
```

```
125 [y,f] = fmincon(@(x) h2fit(x,tmesh),x0,[],[],[],[],[],[],...
        @confuneq,options)
127 [c,ceq] = confuneq(y)
    [r,a1,df] = thd(y,tmesh);
129 for i = 2:4
        str(k,i) = {y(i-1)};
131 end
    str(k,5) = {'r:'};
133 str(k,6) = {f};
    str(k,7) = {4*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
135 str(k,8) = {r};
    str(k,9) = {a1};
137 str(k,10) = {df};

139 %Linf fit, zero crossing error
    k=k+1;
141 str(k,1) = {'Linf Fit Zero Cross'}
    options = optimset('LargeScale','off');
143 [y,f] = fmincon(@(x) hinffit(x,tmesh),x0,[],[],[],[],[],[],...
        @confuneq,options)
145 [c,ceq] = confuneq(y)
    [r,a1,df] = thd(y,tmesh);
147 for i = 2:4
        str(k,i) = {y(i-1)};
149 end
    str(k,5) = {'c-.'};
151 str(k,6) = {f};
    str(k,7) = {4*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
153 str(k,8) = {r};
    str(k,9) = {a1};
155 str(k,10) = {df};

157 %Max fundamental, zero crossing error
    k=k+1;
159 str(k,1) = {'Min Distort Z Cross'}
    options = optimset('LargeScale','off');
161 [y,f] = fmincon(@(x) mindistfit(x,tmesh),x0,[],[],[],[],[],[],...
        @confuneq,options)
163 [c,ceq] = confuneq(y)
    [r,a1,df] = thd(y,tmesh);
165 for i = 2:4
        str(k,i) = {y(i-1)};
167 end
    str(k,5) = {'m—'};
169 str(k,6) = {f};
    str(k,7) = {4*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
171 str(k,8) = {r};
    str(k,9) = {a1};
```

```
173 str(k,10) = {df};

175 %Min distortion fit 2, zero crossing error
    k=k+1;
177 str(k,1) = {'Min Distort2 ZCross'};
    x00 = [-0.3921 1.5708 0.9675];
179 options = optimset('LargeScale','off');
    [y,f] = fmincon(@(x) thdopt(x,tmesh),x00,[],[],[],[],[],[],...
181     @confuneq,options)
    [c,ceq] = confuneq(y)
183 [r,a1,df] = thd(y,tmesh);
    for i = 2:4
185     str(k,i) = {y(i-1)};
    end
187 str(k,5) = {'k--'};
    str(k,6) = {f};
189 str(k,7) = {4*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
    str(k,8) = {r};
191 str(k,9) = {a1};
    str(k,10) = {df};
193 str(k,11)={'k+'};

195 % Sa = y(1)*(tmesh - y(2)*ones(size(tmesh))).^2 + y(3)*ones(size(tmesh));
    % error = Sa - sin(tmesh);
197 % figure(1)
    % plot(tmesh/pi,error);
199 % figure(2)
    % plot(tmesh/pi,sin(tmesh),tmesh/pi,Sa);
201 clear i,y; y = zeros(1,3);
    figure(1);
203 for i = 1:6
        for j = 1:3;
205         y(j,1) = str{i,j+1}
        end
207     Sa = y(1)*(tmesh - y(2)*ones(size(tmesh))).^2 + ...
            y(3)*ones(size(tmesh));
209     error = Sa - sin(tmesh);
        plot(tmesh,error,str{i,5});
211     hold on;
        if isa(str{i,11},'char')
213         plot(downsample(tmesh,(npts/10)),...
                downsample(error,(npts/10)),str{i,11});
215     end
    end
217 set(gca,'XTick',[0 pi/4 pi/2 3*pi/4 pi]);
    set(gca,'XTickLabel','tgx0|tgxpi4|tgxpi2|tgx3pi4|tgxpi');
219 %set(gca,'YTick',[-0.15 -0.1 0.05 0 0.05 0.1]);
    %set(gca,'YTickLabel',...
```

```
%     'tgym0p15|tgym0p1|tgym0p05|tgy0|tgy0p05|tgym0p1');
xlabel('tgxxtheta')
ylabel('tgyyerror')
hold off;
figure(2);
for i = 6:11
    for j = 1:3;
        y(j,1) = str{i,j+1}
    end
    Sa = y(1)*(tmesh - y(2)*ones(size(tmesh))).^2 + ...
        y(3)*ones(size(tmesh));
    error = Sa - sin(tmesh);
    plot(tmesh,error,str{i,5});
    hold on;
    if isa(str{i,11},'char')
        plot(downsample(tmesh,(npts/10)),...
            downsample(error,(npts/10)),str{i,11});
    end
end
% set(gca,'XTick',[0 pi/4 pi/2 3*pi/4 pi]);
% set(gca,'XTickLabel','0|pi/4|pi/2|3pi/4|pi');
% xlabel('\theta (radians)')
% ylabel('Error (Unit Amplitude)')
set(gca,'XTick',[0 pi/4 pi/2 3*pi/4 pi]);
set(gca,'XTickLabel','tgx0|tgxpi4|tgxpi2|tgx3pi4|tgxpi');
%set(gca,'YTick',[-0.15 -0.1 0.05 0 0.05 0.1]);
%set(gca,'YTickLabel',...
%     'tgym0p15|tgym0p1|tgym0p05|tgy0|tgy0p05|tgym0p1');
xlabel('tgxxtheta')
ylabel('tgyyerror')
hold off;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Quarter Wave Approximation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tmesh = 0:pi/2/npts:pi/2;
x0 = [-4/pi^2 pi/2 1]';
str2 = cell(11,10);
%L1 fit
k=1;
[y,f] = fminsearch(@(x) h1fit(x,tmesh),x0)
[r,a1,df] = thd(y,tmesh);
str2(k,1) = {'L1_Fit'}
for i = 2:4
    str2(k,i) = {y(i-1)};
end
str2(k,5) = {'b-'};
str2(k,6) = {f};
```

```
269 str2(k,7) = {8*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
    str2(k,8) = {r};
271 str2(k,9) = {a1};
    str2(k,10) = {df};
273
    %L2 fit
275 k=k+1;
    [y,f] = fminsearch(@(x) h2fit(x,tmesh),x0)
277 [r,a1,df] = thd(y,tmesh);
    str2(k,1) = {'L2 Fit'}
279 for i = 2:4
        str2(k,i) = {y(i-1)};
281 end
    str2(k,5) = {'g:'};
283 str2(k,6) = {f};
    str2(k,7) = {8*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
285 str2(k,8) = {r};
    str2(k,9) = {a1};
287 str2(k,10) = {df};

289 %Linf fit
    k=k+1;
291 [y,f] = fminsearch(@(x) hinffit(x,tmesh),x0)
    [r,a1,df] = thd(y,tmesh);
293 str2(k,1) = {'Linf Fit'}
    for i = 2:4
295     str2(k,i) = {y(i-1)};
    end
297 str2(k,5) = {'r-.'};
    str2(k,6) = {f};
299 str2(k,7) = {8*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
    str2(k,8) = {r};
301 str2(k,9) = {a1};
    str2(k,10) = {df};
303
    %Max Fundamental Fit
305 k=k+1;
    [y,f] = fminsearch(@(x) mindistfit(x,tmesh),x0)
307 str2(k,1) = {'Min Dist Fit'}
    for i = 2:4
309     str2(k,i) = {y(i-1)};
    end
311 str2(k,5) = {'c--'};
    str2(k,6) = {f};
313 str2(k,7) = {8*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
    str2(k,8) = {r};
315 str2(k,9) = {a1};
    str2(k,10) = {df};
```

```
317
   %Min Distortion Fit 2
319 k=k+1;
   x00 = [-0.3882 1.7676 1.0325];
321 [y,f] = fminsearch(@(x) thdopt(x,tmesh),x00)
   str2(k,1) = {'Min Dist Fit2'}
323 for i = 2:4
       str2(k,i) = {y(i-1)};
325 end
   str2(k,5) = {'k--'};
327 str2(k,6) = {f};
   str2(k,7) = {8*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
329 str2(k,8) = {r};
   str2(k,9) = {a1};
331 str2(k,10) = {df};
   str2(k,11)={'k+'};
333
   %Zero pk and crossing error
335 k=k+1;
   y = x0;
337 [r,a1,df] = thd(y,tmesh);
   str2(k,1) = {'Zero pk and cross'}
339 for i = 2:4
       str2(k,i) = {y(i-1)};
341 end
   str2(k,5) = {'b-'};
343 str2(k,6) = {f};
   str2(k,7) = {8*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
345 str2(k,8) = {r};
   str2(k,9) = {a1};
347 str2(k,10) = {df};
   str2(k,11) = {'bo'};
349
   %L1 fit, zero crossing error
351 k=k+1;
   options = optimset('LargeScale','off');
353 [y,f] = fmincon(@(x) h1fit(x,tmesh),x0,[],[],[],[],[],[],...
       @confuneq2,options)
355 [c,ceq] = confuneq2(y)
   [r,a1,df] = thd(y,tmesh);
357 str2(k,1) = {'L1 Fit Z Cross'}
   for i = 2:4
359     str2(k,i) = {y(i-1)};
   end
361 str2(k,5) = {'g-'};
   str2(k,6) = {f};
363 str2(k,7) = {8*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
   str2(k,8) = {r};
```

```
365 str2(k,9) = {a1};
    str2(k,10) = {df};
367
    %L2 fit, zero crossing error
369 k=k+1;
    options = optimset('LargeScale','off');
371 [y,f] = fmincon(@(x) h2fit(x,tmesh),x0,[],[],[],[],[],[],...
        @confuneq2,options)
373 [c,ceq] = confuneq2(y)
    [r,a1,df] = thd(y,tmesh);
375 str2(k,1) = {'L2 Fit Z Cross'}
    for i = 2:4
377     str2(k,i) = {y(i-1)};
    end
379 str2(k,5) = {'r:'};
    str2(k,6) = {f};
381 str2(k,7) = {8*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
    str2(k,8) = {r};
383 str2(k,9) = {a1};
    str2(k,10) = {df};
385
    %Linf fit, zero crossing error
387 k=k+1;
    options = optimset('LargeScale','off');
389 [y,f] = fmincon(@(x) hinffit(x,tmesh),x0,[],[],[],[],[],[],...
        @confuneq2,options)
391 [c,ceq] = confuneq2(y)
    [r,a1,df] = thd(y,tmesh);
393 str2(k,1) = {'Linf Fit Z Cross'}
    for i = 2:4
395     str2(k,i) = {y(i-1)};
    end
397 str2(k,5) = {'c-.'};
    str2(k,6) = {f};
399 str2(k,7) = {8*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
    str2(k,8) = {r};
401 str2(k,9) = {a1};
    str2(k,10) = {df};
403
    %Max fundamental, zero crossing error
405 k=k+1;
    options = optimset('LargeScale','off');
407 [y,f] = fmincon(@(x) mindistfit(x,tmesh),x0,[],[],[],[],[],[],...
        @confuneq2,options)
409 [c,ceq] = confuneq2(y)
    [r,a1,df] = thd(y,tmesh);
411 str2(k,1) = {'Min Dist Z Cross'}
    for i = 2:4
```

```
413       str2(k,i) = {y(i-1)};
    end
415 str2(k,5) = {'m--'};
    str2(k,6) = {f};
417 str2(k,7) = {8*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
    str2(k,8) = {r};
419 str2(k,9) = {a1};
    str2(k,10) = {df};
421
    %Min distortion fit 2, zero crossing error
423 k=k+1;
    x00 = [-0.3054 1.8563 1.0524];
425 options = optimset('LargeScale','off');
    [y,f] = fmincon(@(x) thdopt(x,tmesh),x0,[],[],[],[],[],[],...
427       @confuneq2,options)
    [c,ceq] = confuneq2(y)
429 [r,a1,df] = thd(y,tmesh);
    str2(k,1) = {'Min Dist2 ZCross'}
431 for i = 2:4
        str2(k,i) = {y(i-1)};
433 end
    str2(k,5) = {'k--'};
435 str2(k,6) = {f};
    str2(k,7) = {8*mindistfit(y,tmesh)/npts}; %half wave = 4/npts; qtr=8/npts
437 str2(k,8) = {r};
    str2(k,9) = {a1};
439 str2(k,10) = {df};
    str2(k,11)={'k+'};
441
    figure(3);
443 for i = 1:6
        for j = 1:3;
445         y(j,1) = str2{i,j+1}
        end
447     Sa = y(1)*(tmesh - y(2)*ones(size(tmesh))).^2 +...
            y(3)*ones(size(tmesh));
449     error = Sa - sin(tmesh);
        plot(tmesh,error,str2{i,5});
451     hold on;
        if isa(str2{i,11},'char')
453         plot(downsample(tmesh,(npts/10)),...
                downsample(error,(npts/10)),str2{i,11});
455     end
    end
457 % set(gca,'XTick',[0 pi/4 pi/2 3*pi/4 pi]);
    % set(gca,'XTickLabel','0|pi/4|pi/2|3pi/4|pi');
459 % xlabel('\theta (radians)')
    % ylabel('Error (Unit Amplitude)')
```

```
461 set(gca,'XTick',[0 pi/4 pi/2 3*pi/4 pi]);
    set(gca,'XTickLabel','tgx0|tgxpi4|tgxpi2|tgx3pi4|tgxpi');
463 %set(gca,'YTick',[-0.15 -0.1 0.05 0 0.05 0.1]);
    %set(gca,'YTickLabel',...
465 %     'tgym0p15|tgym0p1|tgym0p05|tgy0|tgy0p05|tgym0p1');
    xlabel('tgxxtheta')
467 ylabel('tgyyerror')
    hold off;
469 figure(4);
    for i = 6:11
471     for j = 1:3;
            y(j,1) = str2{i,j+1}
473     end
        Sa = y(1)*(tmesh - y(2)*ones(size(tmesh))).^2 +...
475         y(3)*ones(size(tmesh));
        error = Sa - sin(tmesh);
477     plot(tmesh,error,str2{i,5});
        hold on;
479     if isa(str2{i,11},'char')
            plot(downsample(tmesh,(npts/10)),...
481             downsample(error,(npts/10)),str2{i,11});
        end
483 end
    % set(gca,'XTick',[0 pi/4 pi/2 3*pi/4 pi]);
485 % set(gca,'XTickLabel','0|pi/4|pi/2|3pi/4|pi');
    % xlabel('\theta (radians)')
487 % ylabel('Error (Unit Amplitude)')
    set(gca,'XTick',[0 pi/4 pi/2 3*pi/4 pi]);
489 set(gca,'XTickLabel','tgx0|tgxpi4|tgxpi2|tgx3pi4|tgxpi');
    %set(gca,'YTick',[-0.15 -0.1 0.05 0 0.05 0.1]);
491 %set(gca,'YTickLabel',...
    %     'tgym0p15|tgym0p1|tgym0p05|tgy0|tgy0p05|tgym0p1');
493 xlabel('tgxxtheta')
    ylabel('tgyyerror')
495 hold off;
```

B.2 Functions

B.2.1 h1fit()

```
    function f = h1fit(x,tmesh)
2   %L1 fitting sine fitting function
    %y1 = x(3)*tmesh.^2 + x(2)*tmesh + x(1)*ones(size(tmesh));
4   y1 = x(1)*(tmesh - x(2)*ones(size(tmesh))).^2 + x(3)*ones(size(tmesh));
    y2 = abs(y1 - sin(tmesh));
6   f = sum(y2);
```

B.2.2 h2fit()

```
function f = h2fit(x,tmesh)
%L2 fitting sine fitting function
%y1 = x(3)*tmesh.^2 + x(2)*tmesh + x(1)*ones(size(tmesh));
y1 = x(1)*(tmesh - x(2)*ones(size(tmesh))).^2 + x(3)*ones(size(tmesh));
y2 = (y1 - sin(tmesh)).^2;
f = sum(y2);
```

B.2.3 hinffit()

```
function f = hinffit(x,tmesh)
%Linf fitting sine fitting function
%y1 = x(3)*tmesh.^2 + x(2)*tmesh + x(1)*ones(size(tmesh));
y1 = x(1)*(tmesh - x(2)*ones(size(tmesh))).^2 + x(3)*ones(size(tmesh));
y2 = abs(y1 - sin(tmesh));
f = max(y2);
```

B.2.4 mindistfit()

```
function f = mindistfit(x,tmesh)
%Min Distortion fitting sine fitting function
%y1 = x(3)*tmesh.^2 + x(2)*tmesh + x(1)*ones(size(tmesh));
y1 = x(1)*(tmesh - x(2)*ones(size(tmesh))).^2 + x(3)*ones(size(tmesh));
y2 = sum(y1.*sin(tmesh));
y3 = sum(sin(tmesh).^2);
f = abs(y2-y3);
```

B.2.5 thdopt()

```
function f = thdopt(x,tmesh)
%Minimum THD fitting for sine wave approximation
y1 = x(1)*(tmesh - x(2)*ones(size(tmesh))).^2 + x(3)*ones(size(tmesh));
y2 = sum(y1.^2)/length(tmesh);
a1 = (2/length(tmesh))*sum(y1.*sin(tmesh));

df = (a1/sqrt(2))/sqrt(y2); %Distortion Factor

%f = sqrt((1/df)^2 - 1);
%f = (1/df)^2 - 1;
%f = -(df)^2;
f = (1/df)^2;
```

B.2.6 thd()

```
function [f,a1,df] = thd(x,tmesh,width)
%THD calculation for sine wave approximation
y1 = x(1)*(tmesh - x(2)*ones(size(tmesh))).^2 + x(3)*ones(size(tmesh));
```

```
y2 = sum(y1.^2)/length(tmesh);

%a1 = (2/(width*length(tmesh)))*sum(y1.*sin(tmesh));
%a1 = (2/(width*length(tmesh)))*sum(sin(tmesh).^2);
a1 = (2/length(tmesh))*sum(y1.*sin(tmesh));
%y3 = sqrt(2)*sqrt(y2)/length(tmesh)/a1; %1/Distortion Factor

df = (a1/sqrt(2))/sqrt(y2); %Distortion Factor

f = sqrt((1/df)^2 - 1);
```

B.2.7 infnorm()

```
function f = infnorm(tmesh,yvals)
%Linf fitting sine fitting function
%y1 = x(3)*tmesh.^2 + x(2)*tmesh + x(1)*ones(size(tmesh));
y1 = yvals;
y2 = abs(y1 - sin(tmesh));
f = max(y2);
```

Appendix C

Field Oriented Control Simulink Model

C.1 Block Models

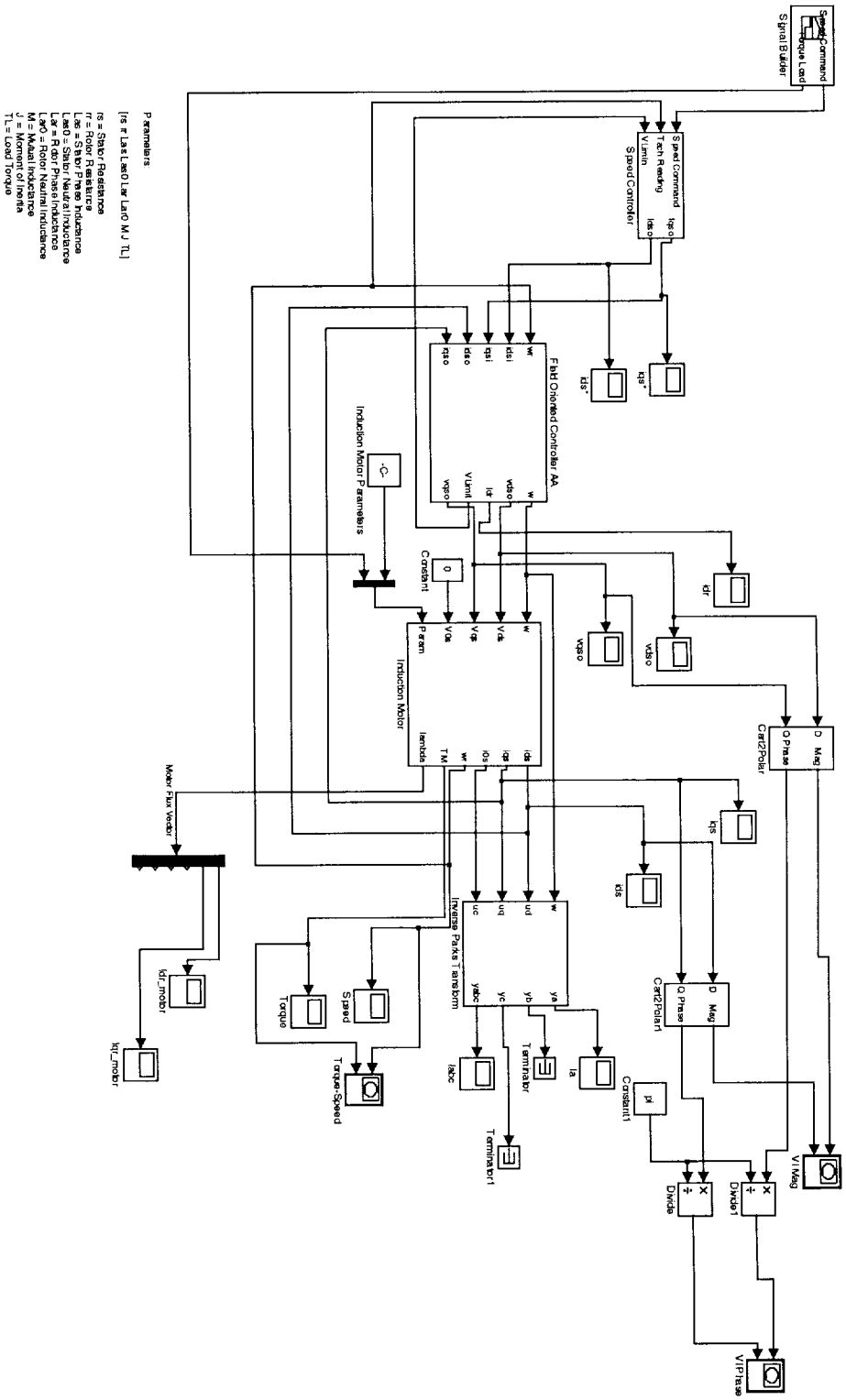


Figure C.1: Top Level Model

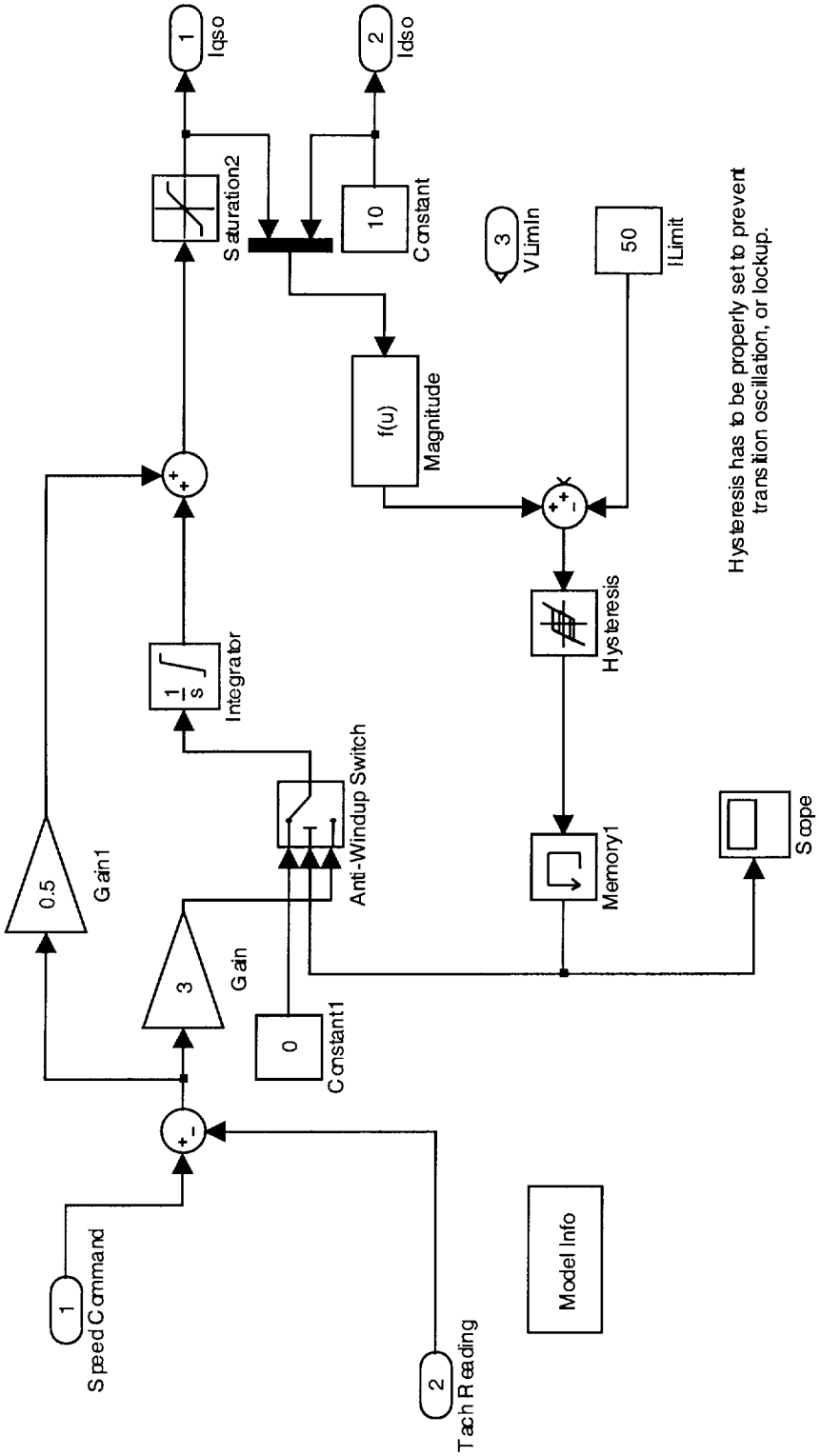


Figure C.2: Speed Controller

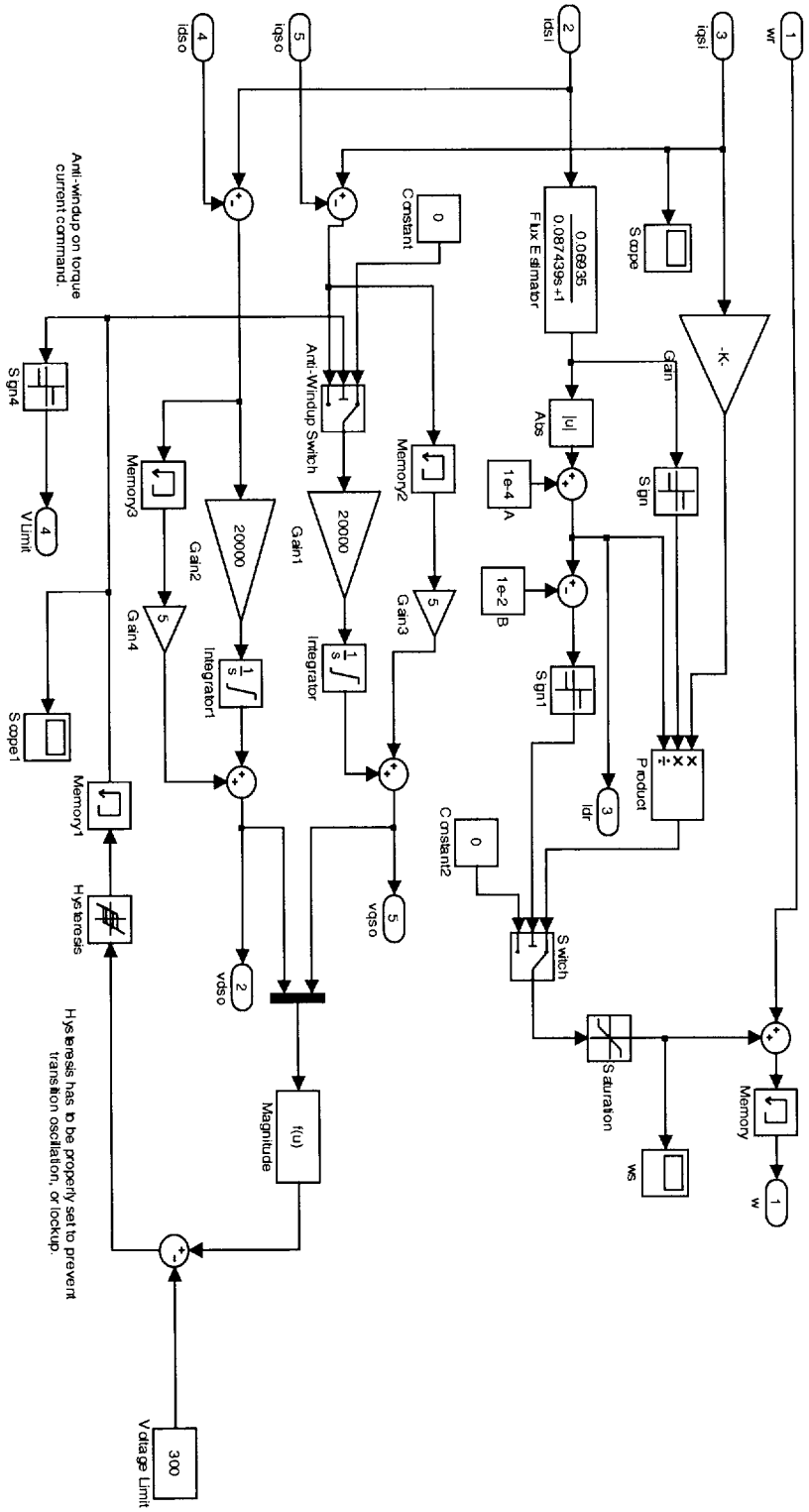


Figure C.3: Field Oriented Controller

C.2 Induction Motor S-Function

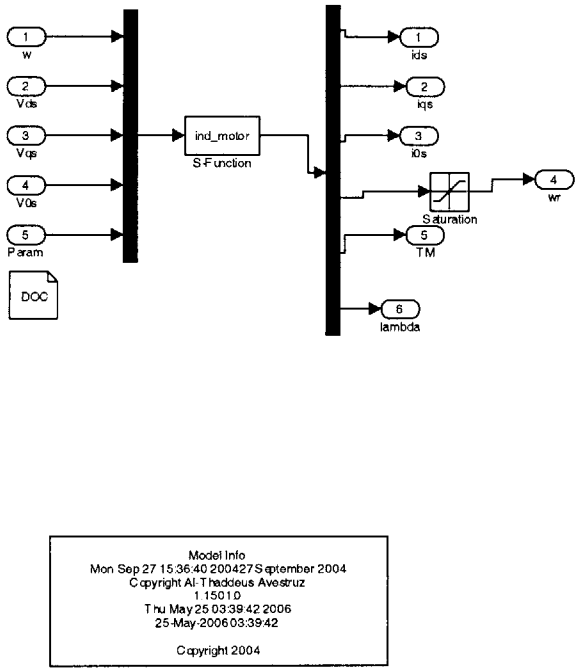


Figure C.4: Induction Motor Model

```
function [sys,x0,str,ts] = ind_motor(t,x,u,flag,P,X0)
%SFUNTMPL General M-file S-function template
%   With M-file S-functions, you can define you own ordinary differential
%   equations (ODEs), discrete system equations, and/or just about
%   any type of algorithm to be used within a Simulink block diagram.
%
%   The general form of an M-File S-function syntax is:
%       [SYS,X0,STR,TS] = SFUNC(T,X,U,FLAG,P1,...,Pn)
%
%   What is returned by SFUNC at a given point in time, T, depends on the
%   value of the FLAG, the current state vector, X, and the current
%   input vector, U.
%
%   FLAG    RESULT              DESCRIPTION
%   ----    ------              -----------------------------------------
%   0       [SIZES,X0,STR,TS]   Initialization, return system sizes in SYS,
%                               initial state in X0, state ordering strings
%                               in STR, and sample times in TS.
%   1       DX                  Return continuous state derivatives in SYS.
```

```
%   2       DS                  Update discrete states SYS = X(n+1)
%   3       Y                   Return outputs in SYS.
%   4       TNEXT               Return next time hit for variable step sample
%                               time in SYS.
%   5                           Reserved for future (root finding).
%   9       []                  Termination, perform any cleanup SYS=[].
%
%
%   The state vectors, X and X0 consists of continuous states followed
%   by discrete states.
%
%   Optional parameters, P1,...,Pn can be provided to the S-function and
%   used during any FLAG operation.
%
%   When SFUNC is called with FLAG = 0, the following information
%   should be returned:
%
%      SYS(1) = Number of continuous states.
%      SYS(2) = Number of discrete states.
%      SYS(3) = Number of outputs.
%      SYS(4) = Number of inputs.
%               Any of the first four elements in SYS can be specified
%               as -1 indicating that they are dynamically sized. The
%               actual length for all other flags will be equal to the
%               length of the input, U.
%      SYS(5) = Reserved for root finding. Must be zero.
%      SYS(6) = Direct feedthrough flag (1=yes, 0=no). The s-function
%               has direct feedthrough if U is used during the FLAG=3
%               call. Setting this to 0 is akin to making a promise that
%               U will not be used during FLAG=3. If you break the promise
%               then unpredictable results will occur.
%      SYS(7) = Number of sample times. This is the number of rows in TS.
%
%
%      X0     = Initial state conditions or [] if no states.
%
%      STR    = State ordering strings which is generally specified as [].
%
%      TS     = An m-by-2 matrix containing the sample time
%               (period, offset) information. Where m = number of sample
%               times. The ordering of the sample times must be:
%
%                  TS = [0      0,      : Continuous sample time.
%                        0      1,      : Continuous, but fixed in minor step
%                                         sample time.
%                        PERIOD OFFSET, : Discrete sample time where
%                                         PERIOD > 0 & OFFSET < PERIOD.
%                        -2     0];     : Variable step discrete sample time
```

```
%                                    where FLAG=4 is used to get time of
%                                    next hit.
%
%               There can be more than one sample time providing
%               they are ordered such that they are monotonically
%               increasing. Only the needed sample times should be
%               specified in TS. When specifying than one
%               sample time, you must check for sample hits explicitly by
%               seeing if
%                  abs(round((T-OFFSET)/PERIOD) - (T-OFFSET)/PERIOD)
%               is within a specified tolerance, generally 1e-8. This
%               tolerance is dependent upon your model's sampling times
%               and simulation time.
%
%               You can also specify that the sample time of the S-function
%               is inherited from the driving block. For functions which
%               change during minor steps, this is done by
%               specifying SYS(7) = 1 and TS = [-1 0]. For functions which
%               are held during minor steps, this is done by specifying
%               SYS(7) = 1 and TS = [-1 1].

%   Copyright 1990-2002 The MathWorks, Inc.
%   $Revision: 1.18 $

%
% The following outlines the general structure of an S-function.
%
switch flag,

  %%%%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts]=mdlInitializeSizes(X0);

  %%%%%%%%%%%%%%%%%%
  % Derivatives %
  %%%%%%%%%%%%%%%%%%
  case 1,
    sys=mdlDerivatives(t,x,u,P);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
  case 2,
    sys=mdlUpdate(t,x,u);

  %%%%%%%%%%
```

```
116   % Outputs %
      %%%%%%%%%%%
118   case 3,
        sys=mdlOutputs(t,x,u,P);
120
      %%%%%%%%%%%%%%%%%%%%%%%%%
122   % GetTimeOfNextVarHit %
      %%%%%%%%%%%%%%%%%%%%%%%%%
124   case 4,
        sys=mdlGetTimeOfNextVarHit(t,x,u);
126
      %%%%%%%%%%%%%
128   % Terminate %
      %%%%%%%%%%%%%
130   case 9,
        sys=mdlTerminate(t,x,u);
132
      %%%%%%%%%%%%%%%%%%%%
134   % Unexpected flags %
      %%%%%%%%%%%%%%%%%%%%
136   otherwise
        error(['Unhandled flag = ',num2str(flag)]);
138
    end
140
    % end sfuntmpl
142
    %
144 %==========================================================================
    % mdlInitializeSizes
146 % Return the sizes, initial conditions, and sample times for
    %the S-function.
148 %==========================================================================
    %
150 function [sys,x0,str,ts]=mdlInitializeSizes(X0)

152 %
    % call simsizes for a sizes structure, fill it in and convert it to a
154 % sizes array.
    %
156 % Note that in this example, the values are hard coded.  This is not a
    % recommended practice as the characteristics of the block are typically
158 % defined by the S-function parameters.
    %
160 sizes = simsizes;

162 sizes.NumContStates  = 7;
    sizes.NumDiscStates  = 0;
```

```
sizes.NumOutputs      = 11;
sizes.NumInputs       = 13;
sizes.DirFeedthrough = 1;
sizes.NumSampleTimes = 1;   % at least one sample time is needed

sys = simsizes(sizes);

%
% initialize the initial conditions
%

x0 = X0;

%
% str is always an empty matrix
%
str = [];

%
% initialize the array of sample times
%
ts  = [0  0];

% end mdlInitializeSizes

%
%=============================================================
% mdlDerivatives
% Return the derivatives for the continuous states.
%=============================================================
%
function sys=mdlDerivatives(t,x,u,P)

lds = x(1);
lqs = x(2);
l0s = x(3);
ldr = x(4);
lqr = x(5);
l0r = x(6);
wr = x(7);

w = u(1);
vds = u(2);
vqs = u(3);
v0s = u(4);
rs = u(5);
rr = u(6);
Las = u(7);
```

```
Las0 = u(8);
Lar = u(9);
Lar0 = u(10);
M = u(11);
J = u(12);
TL = u(13); %Load torque

ids = (Lar*lds - M*ldr)/(Las*Lar - M^2);
iqs = (Lar*lqs - M*lqr)/(Las*Lar - M^2);
i0s = l0s/Las0;
idr = (-M*lds + Las*ldr)/(Las*Lar - M^2);
iqr = (-M*lqs + Las*lqr)/(Las*Lar - M^2);
i0r = l0r/Lar0;

v0r = 0;

TM = (3/2)*P*(lqr*idr - ldr*iqr); %Motor torque

dlds = -rs*ids + w*lqs + vds;
dlqs = -rs*iqs - w*lds + vqs;
dl0s = v0s - rs*i0s;
dldr = -rr*idr + (w-wr)*lqr;
dlqr = -rr*iqr - (w-wr)*ldr;
dl0r = v0r - rr*i0r;
dwr = (TM-TL)/J;

sys = [dlds dlqs dl0s dldr dlqr dl0r dwr];

% end mdlDerivatives

%
%=============================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================
%
function sys=mdlUpdate(t,x,u)

sys = [];

% end mdlUpdate

%
%=============================================================
% mdlOutputs
% Return the block outputs.
%=============================================================
```

```
260 %
    function sys=mdlOutputs(t,x,u,P)
262
    lds = x(1);
264 lqs = x(2);
    l0s = x(3);
266 ldr = x(4);
    lqr = x(5);
268 l0r = x(6);
    wr = x(7);
270
    w = u(1);
272 vds = u(2);
    vqs = u(3);
274 v0s = u(4);
    rs = u(5);
276 rr = u(6);
    Las = u(7);
278 Las0 = u(8);
    Lar = u(9);
280 Lar0 = u(10);
    M = u(11);
282 J = u(12);
    TL = u(13); %Load torque
284
    ids = (Lar*lds - M*ldr)/(Las*Lar - M^2);
286 iqs = (Lar*lqs - M*lqr)/(Las*Lar - M^2);
    i0s = l0s/Las0;
288 idr = (-M*lds + Las*ldr)/(Las*Lar - M^2);
    iqr = (-M*lqs + Las*lqr)/(Las*Lar - M^2);
290 i0r = l0r/Lar0;

292 TM = (3/2)*P*(lqr*idr - ldr*iqr); %Motor torque

294 sys = [ids iqs i0s wr TM lds lqs l0s ldr lqr l0r];

296 % end mdlOutputs

298 %
    %=============================================================
300 % mdlGetTimeOfNextVarHit
    % Return the time of the next hit for this block.  Note that the result is
302 % absolute time.  Note that this function is only used when you specify a
    % variable discrete-time sample time [-2 0] in the sample time array in
304 % mdlInitializeSizes.
    %=============================================================
306 %
    function sys=mdlGetTimeOfNextVarHit(t,x,u)
```

```
308
   %sampleTime = 1;      % Example, set the next hit to be one second later.
310 %sys = t + sampleTime;
   sys = [];
312
   % end mdlGetTimeOfNextVarHit
314
   %
316 %=============================================================================
   % mdlTerminate
318 % Perform any end of simulation tasks.
   %=============================================================================
320 %
   function sys=mdlTerminate(t,x,u)
322
   sys = [];
324
   % end mdlTerminate
```

Appendix D

Motor Control Embedded Firmware

D.1 Main Motor Control Module

D.1.1 mot_cntl.c

```
//TMS320LF2406A Main Module Module
//
//Peripheral TMS320LF2406A
//Author: Al-Thaddeus Avestruz
//Created: 1 Dec 2004
//Copyright 2004 Al-Thaddeus Avestruz
//
//mot_cntl.c
/*  */

#include <stdlib.h>
#include <string.h>
#include "regs240x.h"
#include "pwm/include/F2407pwm.h"
#include "pwm/include/svgen.h"
#include "sysvecs.h"
#include "sine_pwm_init.h"
#include "umacros.h"
#include "serialcomm.h"
#include "periphs.h"
#include "vfcontrol.h"


#define WAIT_STATES 0x40;

#define SET_LO(x,b) ((x)&=~(1<<(b)))
#define SET_HI(x,b) ((x)|=(1<<(b)))

```

```
30

32 #define CLKOUT 40000000

34 //LIMITS

36 //#define RAMP_END 256000L
   #define RAMP_END 12800L  //need the L postfix
38 #define V1_MAX 32767
   #define F1_MAX (60*256)
40 #define RAMP_dV 100L
   #define RAMP_dF 100L
42 #define RAMP_VINTVL ((long)(RAMP_dV*RAMP_END)/(long)V1_MAX)
   #define RAMP_FINTVL ((long)(RAMP_dF*RAMP_END)/(long)F1_MAX)
44

46 void interrupt periodic_interrupt_isr(void);
   void interrupt phantom(void);
48 void trap(void);
   void setup_PLL(void);
50
   //void RampVF(void);
52


54
   /* pwm stuff */
56
   volatile unsigned long isr_count = 0;
58 volatile unsigned long evCounterA = 0;
   volatile unsigned long evCounterB = 0;
60 volatile int tmpregister1 = 0;
   volatile int tmpregister2 = 0;
62 extern volatile unsigned long Ramp_Count;
   /*
64 typedef struct
   {
66     char * array;
       unsigned int index;
68     unsigned int length;
       int lock;
70     int full;
       int empty;
72     int reading;
   } typecBuffer;
74 */

76 typecBuffer SerTxBuffer = {0, 0, 64, 0, 0, 0, 0};
```

```
78 typeRamp VFRamping = {1,0,1,1,0,0,0};
   typeVout VControl = {0, {0,0,0,0,0,0,0,0},0,0,0,0,0,0,0};
80
   main()
82 {

84     ldiv_t ldiv_r;
       unsigned long ltmp = 0;
86     signed int itmp = 0;
       signed int itoggle = 1;
88
       disable_ints();
90
       WDCR = 0x68;
92     MCRA = MCRA&(~0x4000); //Makes port IOBP6

94     setup_PLL();

96     setup_PWM(&sin_pwm);

98     setupEVB();

100    setupTimer3();

102    SetupSerial();  sendChar('E');

104 //  setupADC();

106    setupV3ADC();

108    SCSR1 |= 0x0001;  //Clear ILLADR bit

110
       IFR = 0xffff; /* Clear all interrupts. */
112    IMR = 0x0002 + 0x0001; // Enable INT2. and INT1 interrupt mask register

114 //  strcpy(strtmp,"v.1.0");
   //  sendString(strtmp);
116
   //  EVAIFRA = 0xffff; /* Clear all EV1 group A EV interrupt flags. */
118 //  EVAIMRA = 0x0080; /* Enable Timer 1 period interrupt interrupts */

120    ENABT3();

122    enable_ints();

124    sendChar('F');
   //  RampVF();  //Volts/Hz Ramp
```

```
126     sin_pwm.V1 = 0;
        sin_pwm.V3 = 0;
128     sin_pwm.F1.f = 1;
        sin_pwm.F1.n = 8;
130     update_PWM(&sin_pwm);

132     VFRamping.direction = 1;
        VFRamping.period = 25600L;
134     VFRamping.dF = 60*256;
        VFRamping.dV = 32767;
136     VFRamping.fstep = 100L;
        VFRamping.vstep = 100L;
138     RampVF(&sin_pwm, &VFRamping);
        sendChar('G');
140 /*
        //12/23/04
142     VFRamping.direction = -1;
        VFRamping.period = 6000L;
144 //  VFRamping.dF = 20*256;
        VFRamping.dF = 1;
146     VFRamping.dV = 10000;
    //  VFRamping.fstep = 350L;
148     VFRamping.fstep = 0;
        VFRamping.vstep = 250L;
150     RampVF(&sin_pwm, &VFRamping);
    */
152
        VFRamping.direction = -1;
154     VFRamping.period = 12800L;
        VFRamping.dF = 1;
156 //  VFRamping.dV = 6554;
    //  VFRamping.dV = 15000;
158 //  VFRamping.dV = 12000;
        VFRamping.dV = 10923;
160     VFRamping.fstep = 0;
        VFRamping.vstep = 100L;
162     RampVF(&sin_pwm, &VFRamping);

164 /*
        VFRamping.direction = -1;
166     VFRamping.period = 2400L;
        VFRamping.dF = 12*256;
168 //  VFRamping.dV = 6554;
        VFRamping.fstep = 1000L;
170     VFRamping.vstep = 0;
        RampVF(&sin_pwm, &VFRamping);
172 */
    /*
```

```
174     VFRamping.direction = -1;
        VFRamping.period = 2400L;
176     VFRamping.dF = 1;
        VFRamping.dV = 6554;
178     VFRamping.fstep = 0;
        VFRamping.vstep = 1000L;
180     RampVF(&sin_pwm, &VFRamping);
    */
182 /*
        VFRamping.direction = -1;  //12-23-04
184     VFRamping.period = 8000L;
        VFRamping.dF = 1;
186     VFRamping.dV = 5000;
        VFRamping.fstep = 0;
188     VFRamping.vstep = 100L;
        RampVF(&sin_pwm, &VFRamping);
190 */
    //  sin_pwm.V3 = -6554;
192
    //  VControl.igain = 5;
194 //  VControl.pgain = 5;

196     VControl.igain = 60000;
        VControl.pgain = 60000;
198 //  VControl.igain = 50;
    //  VControl.pgain = 50;
200 //  VControl.igain = 25;
    //  VControl.pgain = 25;
202 //  VControl.vcommand = 389;
    //  VControl.vcommand = 76;
204 //  VControl.vcommand = 420;
    //  VControl.vcommand = 480;
206     VControl.vcommand = 275;  //old value 225

208 //  sin_pwm.V3 = -10000;  update_PWM(&sin_pwm);
    //  sin_pwm.V3 = -10000;
210
    //  sin_pwm.V3 = 1000;  update_PWM(&sin_pwm);
212 //  sin_pwm.V3 = -10923; update_PWM(&sin_pwm);

214     createcBuffer(&SerTxBuffer);
        SerTxBuffer.empty = 1;
216     while(1)
        {
218
    /*
220         if (isr_count >= 2000L)
            {
```

```
222             //VControl.vcommand = 389 + itoggle*30;
                VControl.vcommand = VControl.vcommand + itoggle*30;
224             itoggle = -1*itoggle;
                isr_count = 0;
226         }
    */
228 /*      if (0 == VFRamping.ramping)
        {
230         VFRamping.direction = -1;
            VFRamping.period = 12000L;
232         VFRamping.dF = 20*256;
            VFRamping.dV = 10000;
234         VFRamping.fstep = 100L;
            VFRamping.vstep = 100L;
236     }
        RampVFControl(&sin_pwm, &VFRamping, &evCounterB, 4000L, 1);
238
        if (0 == VFRamping.ramping)
240     {
            VFRamping.direction = 1;
242         VFRamping.period = 12000L;
            VFRamping.dF = 20*256;
244         VFRamping.dV = 10000;
            VFRamping.fstep = 100L;
246         VFRamping.vstep = 100L;
        }
248     RampVFControl(&sin_pwm, &VFRamping, &evCounterB, 4000L, 2);
    */
250
        if (0 == VFRamping.ramping) //12-20-04
252     {
            VFRamping.direction = -1;
254         VFRamping.period = 6000L;
            VFRamping.dF = 20*256;
256         VFRamping.dV = 10000;
            VFRamping.fstep = 250L;
258         VFRamping.vstep = 250L;
        }
260     RampVFControl(&sin_pwm, &VFRamping, &evCounterB, 2000L, 1);

262     if (0 == VFRamping.ramping)
        {
264         VFRamping.direction = 1;
            VFRamping.period = 6000L;
266         VFRamping.dF = 20*256;
            VFRamping.dV = 10000;
268         VFRamping.fstep = 250L;
            VFRamping.vstep = 250L;
```

```
270         }
            RampVFControl(&sin_pwm, &VFRamping, &evCounterB, 2000L, 2);
272
    //      sin_pwm.V3 = -(32767-sin_pwm.V1);
274 //Hack to demonstrate third harmonic voltage regulation
            sin_pwm.V3 = 0;
276 //      controlV3(&VControl, &sin_pwm);

278         update_PWM(&sin_pwm);
            //Single update for everybody to ensure synchronous update
280

282         writeSerBuffer(&SerTxBuffer, VControl.insum, &evCounterA, 500,1);
            writeSerBuffer(&SerTxBuffer, VControl.verror, &evCounterA, 500,2);
284         writeSerBuffer(&SerTxBuffer, VControl.vcommand, /
                &evCounterA, 500,3);
286         writeSerBuffer(&SerTxBuffer, VControl.accum, &evCounterA, 500,4);
            writeSerBuffer(&SerTxBuffer, sin_pwm.V3, &evCounterA, 500,5);
288         writeSerBuffer(&SerTxBuffer, VControl.vbus, &evCounterA, 500,6);
            terminate_wrtSerBuffer(&SerTxBuffer, &evCounterA, 500);
290
    //      sin_pwm.V1 = 0x7fff;
292 //      sin_pwm.V1 = 26213;
    //      sin_pwm.V1 = 0;
294 //      sin_pwm.V3 = 0x7fff;
    //      sin_pwm.V3 = 0;
296 //      sin_pwm.F1.f = 60*256;
    //      sin_pwm.F1.n = 8;
298
    /*
300 //      if (isr_count >= 5000L)
            if (isr_count >= 5000L)
302         {
    //          sin_pwm.V3 = -sin_pwm.V3;
304             sin_pwm.V3  = sin_pwm.V3 + itoggle*10000;
                update_PWM(&sin_pwm);
306             itoggle = -1*itoggle;
                isr_count = 0;
308         }

310 */

312 //      sendChar('V');
    //      printADC(8);
314 //      printADC(5);
    //      sendChar('B');
316 //      printADC(3);
```

```
318 //        sendChar('H');  //this works

320

322     }

324     /* end main() */
    }
326
    interrupt void high_interrupt_isr()
328 {
        if (1 == BITGET(EVAIFRA,0))  //Power Drive Interrupt
330     {
            sin_pwm.fault.flag = 1;
332         sin_pwm.fault.count++;
            sin_pwm.V3 = 0;
334         sin_pwm.V1 = 0;
            update_PWM(&sin_pwm);
336         sendChar('*');
            sendChar('f');
338         asm("__NOP");
        }
340
        if (1 == BITGET(SCICTL2,7)) //TXRDY
342     //To do: no interrupts;  something is masking these interrupts
        {
344         sendBuffer(&SerTxBuffer);
        }
346 }

348 interrupt void periodic_interrupt_isr()
    {
350     disable_ints();
        PBDATDIR |= 0x0040; //Set IOBP6 High
352 //  print_reg('A', IFR);
    //  print_reg('B', IMR);
354 //  print_reg('C',EVAIFRA);
    //  print_reg('D',EVAIMRA);
356 //  sendChar('\r'); sendChar('\n');
        if (1 == BITGET(EVAIFRA,7))  //T1PINT Flag
358     {
    //      PBDATDIR |= 0x0040; //Set IOBP6 High
360
            handle_PWM_interrupt(&sin_pwm);
362
            EVAIMRA = 0x0080; /* Enable Timer 1 period interrupt interrupts */
364         EVAIFRA = 0xffff; /* Clear all EV1 group A EV interrupt flags. */
```

```
366 //        sendChar('P');
    //        print_reg('A', IFR);
368 //        print_reg('B', IMR);
    //        print_reg('C',EVAIFRA);
370 //        print_reg('D',EVAIMRA);
    //        sendChar('\r'); sendChar('\n');
372 //        PBDATDIR &= ~0x0040;  //Sets IOBP6 Low
        }
374
        if (1 == BITGET(EVBIFRA,7))  //T3PINT Flag
376     {
            isr_count++;
378         Ramp_Count++;
            evCounterA++;
380         evCounterB++;
            update_Vout(&VControl);
382         EVBIFRA = 0xffff;

384     }

386     /* Done with the ISR */

388 //  IFR = 0xffff; /* Clear all interrupts.
    */   //To do: clear only the flag that was handled;
390     this should automatically clear w/o intervention
    //  IMR = 0x0002; /* Enable INT2. */
392
    //  EVAIMRA = 0x0080; /* Enable Timer 1 period interrupt interrupts */
394     EVAIFRA = 0xffff; /* Clear all EV1 group A EV interrupt flags. */

396     EVBIFRA = 0xffff;

398     PBDATDIR &= ~0x0040;  //Sets IOBP6 Low

400     enable_ints();
    }
402


404


406
    void trap()
408 {
        //Square wave on the contactor pin (#13) on the serial connector.
410     MCRA = MCRA&(~0x4000);

412     //make that pin an output pin
        PBDATDIR = PBDATDIR|0x4000;
```

```
414
        while(1) {
416         if(GPTCONA & (1<<13))
                PBDATDIR |= 0x0040;
418         else
                PBDATDIR &= ~0x0040;
420     }
    }
422

424 void setup_PLL()
    {
426     /* setup the PLL module */
        /*
428     asm("SPLK    #0041h,PLL_CNTL1   ;Disable PLL first.=CPUCLK/2,");
        asm("SPLK    #00B1h,PLL_CNTL2   ;CLKIN(XTAL)=10MHz, PLL*2.0=20MHz;");
430     asm("SPLK    #0081h,PLL_CNTL1   ;CLKMD=PLL Enable,f_SYSCLK=f_CPUCLK/2");
        asm("SPLK    #0080h,PLL_CNTL1   ;CLKMD=PLL Enable,f_SYSCLK=f_CPUCLK/4");
432     asm("SPLK    #40C0h,SYSCR       ;CLKOUT=CPUCLK");
        */
434     SCSR1 = 0x0000;//0x0600;
    }
436


438
    void interrupt phantom()
440 {
    //  IFR = 0xffff; /* Clear all interrupts. */
442 }

444 /*
    void RampVF(void)
446 {
        int ramping = 0;
448     unsigned long ltmp = 0;
    //  ldiv_t ldiv_r;
450 //  unsigned long lcount = 0;
        unsigned long prevFCount = 0;
452     unsigned long prevVCount = 0;

454     sin_pwm.V1 = 0;
        sin_pwm.V3 = 0;
456     sin_pwm.F1.f = 1*256;
        sin_pwm.F1.n = 8;
458     update_PWM(&sin_pwm);

460     disable_ints();
        isr_count = 0;
```

```
462     enable_ints();
        ramping = 1;
464
        ltmp = (long)RAMP_VINTVL;
466
        while(1 == ramping)
468     {
            tmpregister1 = T3PR;
470         tmpregister2 = T3CNT;
            if (sin_pwm.F1.f < F1_MAX)
472         {
                if ((isr_count - prevFCount)> (long)RAMP_FINTVL)
474             {
                    sin_pwm.F1.f = sin_pwm.F1.f + RAMP_dF;
476                 update_PWM(&sin_pwm);
                    prevFCount = isr_count;
478             }
            }
480
            if (sin_pwm.V1<(unsigned int)V1_MAX)
482         {
                if ((isr_count - prevVCount)> (long)RAMP_VINTVL)
484             {
                    sin_pwm.V1 = sin_pwm.V1 + RAMP_dV;
486 //                  PBDATDIR |= 0x0040; //Set IOBP6 High
                    update_PWM(&sin_pwm);
488 //                  PBDATDIR &= ~0x0040;  //Sets IOBP6 Low
                    prevVCount = isr_count;
490             }
            }
492
            if(isr_count >= RAMP_END)
494             ramping = 0;

496     }
    }
498 */
```

D.1.2 umacros.h

```
    //Utility Macros
2   //
    //Utility Macros for TMS320LF2406A
4   //Author: Al-Thaddeus Avestruz
    //Created: 7 November 2004
6   //Copyright 2004 Al-Thaddeus Avestruz
    //
8   //umacros.h
```

```
  //REV 1.0
10
  #ifndef __UMACROS__
12 #define __UMACROS__

14 #define BITSET_L(x,b) ((x)&=~(1<<(b)))
  #define BITSET_H(x,b) ((x)|=(1<<(b)))
16 #define BITGET(x,b) (((x)>>(b))& 0x0001)

18 #define SET_LO(x,b) ((x)&=~(1<<(b)))
  #define SET_HI(x,b) ((x)|=(1<<(b)))
20
  #define PI 3.1415927
22 #define TWOPI 6.2831853
  #define TWOPIBYTHREE 2.0943951
24 #define SEVENPI 21.9911485751286
  #define ONEBYTWOPI 0.159154943091895
26
  #define disable_ints() asm("    setc    intm    ")
28
  #define enable_ints()  asm("    clrc    intm    ");
30 asm("_NOP");asm("_NOP");asm("_NOP") //Bug SDSsq29090

32 #define sgn(x) (-((x)<0) + ((x)>0))

34 typedef struct
  {
36     unsigned int f;
       int n;
38 } typeuQint;

40 typedef volatile unsigned int typeFlag;
  typedef volatile struct
42 {
       unsigned int locked;
44     unsigned int id;
       unsigned int life;
46 } typeSemaphore;

48 typedef volatile struct
  {
50     typeSemaphore R;     //Read
       typeSemaphore W;     //Write
52     typeFlag U;          //Update
  } typeRWSemaphore;
54
  typedef volatile struct
56 {
```

```
    union
58      {
            int * iarray;
60          unsigned int * uiarray;
            char * txtarray;
62          float * farray;
            long * larray;
64      } addr;
    unsigned int length;
66  unsigned int rindex;
    unsigned int windex;
68  typeFlag overflow;
    typeRWSemaphore sem;
70 } typeCircBuffer;

72 typedef struct
   {
74  char * array;
    volatile unsigned int index;
76  unsigned int length;
    volatile int lock;
78  volatile int full;
    volatile int empty;
80  volatile int reading;
   } typecBuffer;
82

84 union TwoBytes
   {
86  unsigned int lt;
    unsigned char bt[2];
88  //To do:  Check array alignments  chars are 16 bits
   };
90
   struct bitfield
92 {
    unsigned int b7:1;
94  unsigned int b6:1;
    unsigned int b5:1;
96  unsigned int b4:1;
    unsigned int b3:1;
98  unsigned int b2:1;
    unsigned int b1:1;
100 unsigned int b0:1;
   };
102 typedef union Byte
   {
104 struct bitfield bit;
```

```
    unsigned char byte;
} byte;

#endif // __UMACROS_
```

D.2 Sine Inverter PWM Module

D.2.1 sin_pwm.c

```
//Sine PWM Module
//
//SCI Peripheral TMS320LF2406A
//Author: Al-Thaddeus Avestruz
//Created: 11 November 2004
//Copyright 2004 Al-Thaddeus Avestruz
//
//sine_pwm.c
//REV 1.0
//12/1/04  Nearly constant switching frequency

#include "regs240x.h"
#include <stdlib.h>
#include "pwm/include/F2407pwm.h"
#include "pwm/include/svgen.h"
#include "umacros.h"
#include "serialcomm.h"
#include "sine_pwm.h"

/*
typedef struct
{
    signed int V1;
    unsigned int F1;
    signed int V3;
    unsigned int F3;
    signed int Ph3;     //not yet implemented
    unsigned int fs;
    unsigned int N1;
    unsigned int N3;
    unsigned int tpd;
    int mflag;
} typeSIN_PWM;
*/

//Data Tables
#define TBL_LEN 108     //Should always be a multiple of 3
#define TBL_LEN_3 TBL_LEN/3
#define TBL_LEN2_3 2*TBL_LEN/3
```

```
#define TBL_LEN_2 TBL_LEN/2

#pragma DATA_SECTION(SinTab, ".tables")
//#pragma CODE_SECTION(update_PWM, "fast")

signed int SinTab[] =
{0,1905,3804,5690,7557,9398,11207,12978,14706,16383,
18006,19567,21062,22486,23834,25101,26283,27376,28377,
29282,30087,30791,31390,31884,32269,32545,32712,32767,
32712,32545,32269,31884,31390,30791,30087,29282,28377,
27376,26283,25101,23834,22486,21062,19567,18006,16384,
14706,12978,11207,9398,7557,5690,3804,1905,
0,-1905,-3804,-5690,-7557,-9398,-11207,-12978,-14706,-16383,
-18006,-19567,-21062,-22486,-23834,-25101,-26283,-27376,-28377,
-29282,-30087,-30791,-31390,-31884,-32269,-32545,-32712,-32767,
-32712,-32545,-32269,-31884,-31390,-30791,-30087,-29282,-28377,
-27376,-26283,-25101,-23834,-22486,-21062,-19567,-18006,-16384,
-14706,-12978,-11207,-9398,-7557,-5690,-3804,-1905};

////Functions
//To do: run this interupt handler out of RAM
void handle_PWM_interrupt(typeSIN_PWM * pwm)
{
    div_t idiv_r;
    ldiv_t ldiv_r;

    signed int fna = 0;
    signed int fnb = 0;
    signed int fnc = 0;
    signed int fn3 = 0;
    signed int itmp = 0;

    //Private variables
    static unsigned int ncnt = 0;
    static unsigned int N1 = 0;
    static unsigned int n3 = 0;
    static unsigned int na = 0;
    static unsigned int nb = 0;
    static unsigned int nc = 0;
    static unsigned int tpd_2 = TMR_PERIOD/2;
    static signed int V1 = 0;
    static signed int V3 = 0;
    static signed int V3tmp = 0;
    static signed int tpd = TMR_PERIOD;
        //must be signed for the compiler to do the
        //multiply properly

    //To do: fix tpd so it can be unsigned and still multiply properly
```

```
89      if (pwm->mflag)          //Check mutex flag for update
        {
91          V1 = pwm->V1;
            V3tmp = pwm->V3;
93          tpd = pwm->tpd;
            tpd_2 = tpd>>1;
95          N1 = pwm->N1;
            T1PR = pwm->tpd; /* Timer Period Register*/
97      }

99      if ((0==n3)||(TBL_LEN_2==n3)) V3=V3tmp;
        // update only on zero crossings of phi3
101
        ncnt++;
103     if (ncnt >= N1)
        {
105         //Fundamental
  //        if (na<TBL_LEN) na++; else na=0;
107         //To do: error here maybe should be na++<TBL_LEN
            if (++na<TBL_LEN); else na=0;
109         if (na>TBL_LEN2_3) nb = na - TBL_LEN2_3;
                else nb = na + TBL_LEN_3;    //if (ka + K/3)>K
111         if (na>TBL_LEN_3) nc = na - TBL_LEN_3;
                else nc = na + TBL_LEN2_3;    //if (ka + 2*K/3)>K
113
            //To do:  MSK V1 also.  e.g.
115         //if ((0==na)||(TBL_LEN_2==na)) V1=V1new;

117         itmp = ((long)V1 * (long)SinTab[na])>>16;
                //Voltage Scale (14 bits max)
119         itmp = 2*itmp;
            fna = ((long)itmp * (long) tpd)>>16;//really a mult by tpd/2
121         if (fna>0) fna-=1;
            if (fna<0) fna+=1;
123         itmp = ((long)V1 * (long)SinTab[nb])>>16;
            itmp = 2*itmp;
125         fnb = ((long)itmp * (long) tpd)>>16;
            if (fnb>0) fnb-=1;
127         if (fnb<0) fnb+=1;
            itmp = ((long)V1 * (long)SinTab[nc])>>16;
129         itmp = 2*itmp;
            fnc = ((long)itmp * (long) tpd)>>16;
131         if (fnc>0) fnc-=1;
            if (fnc<0) fnc+=1;
133
            //3rd Harmonic
135         if ((n3+3)<TBL_LEN) n3=n3+3; else n3=0;
```

```
            itmp = ((long)V3 * (long)SinTab[n3])>>16;
137         itmp = 2*itmp;
            fn3 = ((long)itmp * (long) tpd)>>16;
139         if (fn3>0) fn3-=1;
            if (fn3<0) fn3+=1;
141
            /* Phase A duty cycle */
143         CMPR1 = ((unsigned int)((fna + fn3) + tpd_2));
                //Compare threshold
145         itmp = CMPR1;
            /* Phase B duty cycle */
147         CMPR2 = ((unsigned int)((fnb + fn3) + tpd_2));
            /* Phase C duty cycle */
149         CMPR3 = ((unsigned int)((fnc + fn3) + tpd_2));

151         ncnt = 0;
        };
153
        pwm->mflag = 0;  //release mutex
155
    //      if (27==na)
157 //          {
    //              print_reg('M',CMPR1);
159 //              print_reg('T',tpd);
    //          }
161 //      if (81==na)
    //          {
163 //              print_reg('N',CMPR1);
    //              print_reg('U',tpd);
165 //          }
    }
167
    void update_PWM(typeSIN_PWM * pwm)
169 {
        div_t idiv_r;
171     ldiv_t ldiv_r;
        unsigned long ltmp = 0;
173     static unsigned long fs0thresh = FS0;

175     while (pwm->mflag);  //Wait for flag to clear

177     pwm->F3.f = 3*pwm->F1.f;
        pwm->F3.n = pwm->F1.n;
179
        ltmp =  ((unsigned long)TBL_LEN * (unsigned long)pwm->F1.f);
181     ltmp = ltmp>>pwm->F1.n;
        ldiv_r = ldiv((unsigned long) fs0thresh, ltmp);
183
```

```
        pwm->N1 = (unsigned int)ldiv_r.quot;
185     if ((2*ldiv_r.rem) > ltmp) pwm->N1+=1;  //rounding
        //To do:  add rounding hysteresis so won't oscillate between fs's
187
        ltmp = pwm->fs;
189     pwm->fs = ((unsigned long)pwm->N1 /
            *(unsigned long)TBL_LEN * /
191             (unsigned long)(pwm->F1.f))>>pwm->F1.n;
                    //align switching period with the table
193
        if (ltmp > pwm->fs) fs0thresh = FS0 - FS0HYST;
195         else if (ltmp < pwm->fs) fs0thresh = FS0 + FS0HYST;

197     ldiv_r = ldiv((unsigned long) CLKOUT, /
            2*PRESCALE*(unsigned long) pwm->fs);
199
        pwm->tpd = (unsigned int) ldiv_r.quot;  //calc new timer period
201     //To do:  is this less than 32767
        if (pwm->tpd > 32767) pwm->tpd=32767;
203     //added 2/20/05 because handle_PWM_interrupt() won't do >32767

205     if (pwm->tpd< (unsigned int)MIN_TMR1PD) /
            pwm->tpd = (unsigned int)MIN_TMR1PD;
207     //limit the switching freq

209     //Update parameters -- setting the flag causes isr to
        //update all parameters
211     disable_ints();
        pwm->mflag = 1;
213     enable_ints();
    }
215
    void setup_PWM(typeSIN_PWM * pwm)
217 {
        //To do: disable interrupts
219     /* setup the modules */

221     MCRA = MCRA&(~0x4000);
        PBDATDIR = PBDATDIR|0x4000;
223
        /* Set T3PWM low */
225     SET_LO(MCRC,10);     // Select IOPF2
        SET_HI(PFDATDIR,10);// Output
227     SET_LO(PFDATDIR,2); // Pin low

229     /* reset any faults */
        reset_PWM_fault();
231
```

```
        PIACKR1 = 0x0001;   //Ack PDP Interrupt — Si Errata
233     EVAIFRA = 0xFFFF;   //Clear EVA interrupt flags

235     /* Set up PWM, the correct way: */
        SCSR1 |= 0x0004;   //EVA Clock Enable
237
        pwm->V1 = 0;
239     pwm->V3 = 0;
        pwm->F1.f = 1*256;
241     pwm->F1.n = 8;
        update_PWM(pwm);
243     T1PR = pwm->tpd;
        /* Set Timer Period explicitly so that we can go into the interrupt*/
245


247
        DBTCONA = DBTCON_INIT_STATE;
249     /* Setup the bridging IGBT gate driver polarities */
        ACTRA = COMPARE1_AL +
251             COMPARE2_AH +
                COMPARE3_AL +
253             COMPARE4_AH +
                COMPARE5_AL +
255             COMPARE6_AH;

257     CMPR1 = 0; /* Phase A duty cycle */
        CMPR2 = 0; /* Phase B duty cycle */
259     CMPR3 = 0; /* Phase C duty cycle */

261     COMCONA=0xa200;   //10100010 CENABLE CLD0 FCOMPOE
        T1CON = PWM_INIT_STATE;
263     MCRA |= 0x0fc0;

265     EVAIFRA = 0xffff; /* Clear all EV1 group A EV interrupt flags. */
        EVAIMRA = 0x0080 + 0x0001;
267         /* Enable Timer 1 period interrupts and PDP*/

269     /* Setup done */

271     /* reset any faults */
        reset_PWM_fault();
273
        //To do:  enable interrupts
275
    }
277
    void reset_PWM_fault(void)
279 {
```

```
    int i;
281 /* Clear faults on drivers */
    SET_LO(MCRC,3);      // Select IOPE3
283 SET_HI(PEDATDIR,11);// Output
    SET_HI(PEDATDIR,3); // Pin high
285 SET_LO(PEDATDIR,3); // Pin low

287 /* Assert latch reset */
    SET_LO(MCRB,8);      // Select IOPD0
289 SET_HI(PDDATDIR,8); // Output
    SET_LO(PDDATDIR,0); // Pin low
291 for(i=0;i<100;i++)
        asm("nop");      // Delay to allow the latch to reset
293 SET_HI(PDDATDIR,0); // Pin high
}
```

D.2.2 Header Files

sin_pwm_init.h

```
1  //Sine PWM Initializations Header File
   //
3  //SCI Peripheral TMS320LF2406A
   //Author: Al-Thaddeus Avestruz
5  //Created: 11 November 2004
   //Copyright 2004 Al-Thaddeus Avestruz
7  //
   //sine_pwm.h
9  //REV 1.0

11 #include "sine_pwm.h"

13 //Data Structures
   /*
15 typedef volatile struct
   {
17     signed int V1;
       typeuQint F1;
19     signed int V3;
       typeuQint F3;
21     signed int Ph3;      //not yet implemented
       unsigned long fs;
23     unsigned int N1;
       unsigned int N3;
25     unsigned int tpd;
       unsigned int prescale;
27     typeFault fault;
       unsigned int vbus;
```

```
29      volatile int mflag;
  } typeSIN_PWM;
31 */

33 typeSIN_PWM sin_pwm =
  {
35 0,        //V1
  {1*256,8},        //F1
37 0,        //V3
  {3*256,8},        //F3
39 0,        //Ph3
  FS0,      //fs
41 1,   //N1
  1,   //N3
43 TMR_PERIOD, //tpd
  1,            //prescale
45 {0,0},   //fault.flag, fault.count
  0,
47 0         //mflag 0=exclude
  };
```

D.2.3 sin_pwm.h

```
1 //Sine PWM Module Header File
  //
3 //SCI Peripheral TMS320LF2406A
  //Author: Al-Thaddeus Avestruz
5 //Created: 11 November 2004
  //Copyright 2004 Al-Thaddeus Avestruz
7 //
  //sine_pwm.h
9 //REV 1.0

11 #include <stdio.h>
  #include "umacros.h"
13
  //#define FS0 10800
15      //Nominal Switching Frequency needs to be a multiple of TBL_LEN
  //#define FS0HYST 108
17
  #define FS0 30800
19      //Nominal Switching Frequency needs to be a multiple of TBL_LEN
  #define FS0HYST 308
21
  #define PRESCALE 1
23 #define TMR_PERIOD (5000/PRESCALE)
      //Nominal Timer Period— Must be less than 32767
25      //for handle_PWM_interrupt() to work properly
```

```
#define MIN_TMR1PD (200/PRESCALE)

//To do:  Make an ifndef here
#define CLKOUT 40000000

//Data Structures
/*
typedef struct
{
    signed int V1;
    unsigned int F1;
    signed int V3;
    unsigned int F3;
    signed int Ph3;      //not yet implemented
    unsigned int fs;
    unsigned int N1;
    unsigned int N3;
    unsigned int tpd;
    volatile int mflag;
} typeSIN_PWM;
*/
typedef volatile struct
{
    unsigned int flag;
    unsigned int count;
} typeFault;


typedef volatile struct
{
    signed int V1;
    typeuQint F1;
    signed int V3;
    typeuQint F3;
    signed int Ph3;      //not yet implemented
    unsigned long fs;
    unsigned int N1;
    unsigned int N3;
    unsigned int tpd;
    unsigned int prescale;
    typeFault fault;
    unsigned int vbus;
    volatile int mflag;
} typeSIN_PWM;

//Function Prototypes

void handle_PWM_interrupt(typeSIN_PWM *);
```

```
void update_PWM(typeSIN_PWM *);
void setup_PWM(typeSIN_PWM * pwm);
void reset_PWM_fault(void);
```

D.3 Volts per Hertz Module

D.3.1 vfcontrol.c

```
//TMS320LF2406A Controller Driver Module
//
//Peripheral TMS320LF2406A
//Author: Al-Thaddeus Avestruz
//Created: 13 Dec 2004
//Copyright 2004 Al-Thaddeus Avestruz
//
//vfcontrol.c

#include <stdlib.h>
#include "regs240x.h"
#include "periphs.h"
#include "umacros.h"
#include "sine_pwm.h"
#include "vfcontrol.h"

#define NEGTHRESH 100

/*
typedef struct
{
    int direction; //(plus or minus 1)
    unsigned long period;
    unsigned int dV;
    unsigned int dF;
    unsigned int fstep;
    unsigned int vstep;
} typeRamp;

extern volatile unsigned long RampCount = 0;
*/

volatile unsigned long Ramp_Count = 0;

void RampVF(typeSIN_PWM * pwm, typeRamp * ramp)
{
    int ramping = 0;
    unsigned long prevFCount = 0;
    unsigned long prevVCount = 0;
    unsigned long fintvl = 0;
```

```
41      unsigned long vintvl = 0;
        unsigned int currentdF = 0;
43      unsigned int currentdV = 0;
        signed int dir = 1;
45      signed int currentF = 0;
        signed int currentV = 0;
47      ldiv_t ldiv_r;

49      ldiv_r = ldiv((long)ramp->vstep * ramp->period, (long)ramp->dV);
        vintvl = ldiv_r.quot;
51
        ldiv_r = ldiv((long)ramp->fstep * ramp->period, (long)ramp->dF);
53      fintvl = ldiv_r.quot;

55      disable_ints();
        Ramp_Count = 0;
57      enable_ints();


59
        while(Ramp_Count <= ramp->period)
61      {
  //        tmpregister1 = T3PR;
63 //        tmpregister2 = T3CNT;
          if (currentdF < ramp->dF)
65        {
              if ((Ramp_Count - prevFCount) > fintvl)
67            {
                  currentdF += ramp->fstep;
69                currentF = pwm->F1.f;
                  currentF = currentF + (ramp->direction * ramp->fstep);
71                if (currentF >0) pwm->F1.f = (unsigned int) currentF;
                      else pwm->F1.f = 1;
73                update_PWM(pwm);
                  prevFCount = Ramp_Count;
75            }
          }
77
          if (currentdV < ramp->dV)
79        {
              if ((Ramp_Count - prevVCount) > vintvl)
81            {
                  currentdV += ramp->vstep;
83                currentV = (unsigned int) pwm->V1;
                  currentV = currentV + (ramp->direction * ramp->vstep);
85                if (currentV >=0) pwm->V1 = currentV;
                      else pwm->V1 = -currentV;
87 //               PBDATDIR |= 0x0040; //Set IOBP6 High
                  update_PWM(pwm);
```

```
89 //                PBDATDIR &= ~0x0040;   //Sets IOBP6 Low
                     prevVCount = Ramp_Count;
91                 }
               }
93
       }  //while
95
   }
97
   void RampVFControl(typeSIN_PWM * pwm, typeRamp * ramp,
99     volatile unsigned long *ptimer, /
       unsigned long trigger,unsigned int myinstance)
101 {
       static unsigned long prevFCount = 0;
103    static unsigned long prevVCount = 0;
       static unsigned long fintvl = 0;
105    static unsigned long vintvl = 0;
       static unsigned int currentdF = 0;
107    static unsigned int currentdV = 0;
       static unsigned int triggered = 0;
109
       signed int currentF = 0;  //temp variables
111    signed int currentV = 0;
       ldiv_t ldiv_r;
113
       if (0 == ramp->ramping)
115    {
           ramp->ramping = myinstance;
117
           ldiv_r = ldiv((long)ramp->vstep * ramp->period, (long)ramp->dV);
119        vintvl = ldiv_r.quot;

121        ldiv_r = ldiv((long)ramp->fstep * ramp->period, (long)ramp->dF);
           fintvl = ldiv_r.quot;
123
           prevFCount = 0;
125        prevVCount = 0;
           currentdF = 0;
127        currentdV = 0;

129        triggered = 0;

131        disable_ints();
           *ptimer = 0;
133        enable_ints();
       }
135
       if ((*ptimer>trigger) && (myinstance == ramp->ramping) && !triggered)
```

```
137     {
            *ptimer=0;
139         triggered = 1;
        }
141     if (triggered && (myinstance == ramp->ramping))
        {
143         if(*ptimer <= ramp->period)
            {
145     //      tmpregister1 = T3PR;
        //      tmpregister2 = T3CNT;
147
                if (currentdF < ramp->dF)
149             {
                    if ((*ptimer - prevFCount) > fintvl)
151                 {
                        currentdF += ramp->fstep;
153                     currentF = pwm->F1.f;
                        currentF = currentF + (ramp->direction * ramp->fstep);
155                     if (currentF >0) pwm->F1.f = (unsigned int) currentF;
                            else pwm->F1.f = 1;
157 //                  update_PWM(pwm);
                        prevFCount = *ptimer;
159                 }
                }
161
                if (currentdV < ramp->dV)
163             {
                    if ((*ptimer - prevVCount) > vintvl)
165                 {
                        currentdV += ramp->vstep;
167                     currentV = (unsigned int) pwm->V1;
                        currentV = currentV + (ramp->direction * ramp->vstep);
169                     if (currentV >=0) pwm->V1 = currentV;
                            else pwm->V1 = -currentV;
171     //              PBDATDIR |= 0x0040; //Set IOBP6 High
    //                  update_PWM(pwm);
173     //              PBDATDIR &= ~0x0040;  //Sets IOBP6 Low
                        prevVCount = *ptimer;
175                 }
                } //if
177         } else
                {
179                 ramp->ramping = 0;
                    triggered = 0;
181                 disable_ints();
                    *ptimer = 0;
183                 enable_ints();
                } //else
```

```
185     }
    }
187
    /*
189 typedef struct
    {
191     unsigned int vbus
        unsigned int vcommand;
193     unsigned int vmeas[8];
        unsigned int insum;
195     signed int verror;
        signed int accum;
197     unsigned int pgain;
        unsigned int igain;
199     unsigned int mflag;
    } typeVout;
201 */
    void controlV3(typeVout * vout, typeSIN_PWM * pwm)  //PI Controller
203 {

205     int i = 0;
        signed int v3max = 0;
207     signed long accum_max = 0;
        signed int poutput = 0;
209     static signed long accumout = 0;
        signed int tmpoutput = 0;
211     div_t idiv_r;

213     if (1 == vout->mflag)
        {
215 //      PBDATDIR |= 0x0040; //Set IOBP6 High
            vout->insum = 0;
217         for (i=0; i<8; i++)
            {
219             vout->insum = vout->insum + (vout->vmeas[i]>>5);
                //Sum eight 10 bit left justified unsigned inputs.
221             //Full count 14 bits right justified.
            }
223

225         v3max = 32767 - pwm->V1;
            accum_max = ((signed long) v3max)<<16;  //32 bit accumulator
227
            vout->vbus = vout->vbus>>4;
229             //Bus voltage measurement 10 bits.  Full count 14 bits

231         vout->verror = (signed int) /
                (vout->vcommand<<4) - (signed int) vout->insum;
```

```
233
            accumout += 4L * /
235         ((signed long) vout->igain * (signed long) vout->verror);

237
            //Anti-Windup:  Saturate accumulator for underflow and overflow
239
            if (accumout < 0) accumout = 0;
241         if (accumout > accum_max) accumout = accum_max;

243         vout->accum = (signed int) (accumout>>16);

245
            //Calculate proportional output

247
            poutput = (10L * /
249         (signed long)vout->pgain * (signed long)vout->verror)>>16;

251 //      poutput = 5*poutput;

253         //Check for possible overflow and then add gain outputs

255         if ((sgn(poutput) == sgn(vout->accum)) && (0 != sgn(poutput)))
            {
257             if (abs(poutput) > (32767 - abs(vout->accum)))
                {
259                 if (1 == sgn(poutput)) tmpoutput = 32767;  //overflow
                    else tmpoutput = -32767; //underflow
261             }
                else tmpoutput = poutput + vout->accum;
263         }
            else tmpoutput = poutput + vout->accum;

265
            //Saturate V3 for underflow and overflow
267         //negative V3 values for phi = pi control
            if (tmpoutput < 0) pwm->V3 = 0;
269         if (tmpoutput > v3max) pwm->V3 = -v3max;
            else pwm->V3 = -abs(tmpoutput);
271


273
    //      PBDATDIR |= 0x0040; //Set IOBP6 High
275 //      update_PWM(pwm);
    //      PBDATDIR &= ~0x0040;  //Sets IOBP6 Low
277
            disable_ints();
279         vout->mflag = 0;
            enable_ints();
```

```
281 //      PBDATDIR &= ~0x0040;  //Sets IOBP6 Low
    }
283 }

285 void update_Vout(typeVout * vout)
    {
287     static int i = 0;

289     if ((0 == vout->mflag))// && (0 == (ADCTRL2 & SEQ1_BSY)))
        {
291         if (i<8)
            {
293             MAXCONV = 0x0001;    //Two conversions
                CHSELSEQ1 = 0x0008 + (0x0003<<4);
295
                ADCTRL2 |= RESET_SEQ1;
297             ADCTRL2 |= SOC_SEQ1;     //start conversion
                    asm("_NOP");
299                 asm("_NOP");
                    asm("_NOP");
301                 asm("_NOP");
                while (ADCTRL2 & SEQ1_BSY);
303             vout->vmeas[i] = RESULT0;
                vout->vbus = RESULT1;
305             i++;
            } else
307         {
                i = 0;
309             vout->mflag = 1;
            }
311     }
    }
313
    void setupV3ADC(void)
315 {
        //Dedicated ADC Pins
317     //Setup ADC Timer
        SCSR1 |= 0x0080;
319     //Setup ADC Control Register
        ADCTRL1 = RESET_ADC + ADC_SOFT + \
321         ACQ_PRESCALE_X2 + CPS_CLK_2 + START_STOP;
        ADCTRL2 = INT_DIS_SEQ1 + INT_DIS_SEQ2;
323
        //Maximum Conversions per autoconvert
325     MAXCONV = 0x0001;  //2 conversions

327     //ADC input channel sequencing
        CHSELSEQ1 = 8;  //ADC in from DC Reg Output
```

```
329     CHSELSEQ1 += 0x0003<<4; //Inverter DC Bus Voltage

331     //Reset ADC
        resetADC();
333
}
```

D.3.2 vfcontrol.h

```
1  //Motor Controller Module Header File
   //
3  //SCI Peripheral TMS320LF2406A
   //Author: Al-Thaddeus Avestruz
5  //Created: 13 December 2004
   //Copyright 2004 Al-Thaddeus Avestruz
7  //
   //vfcontrol.h
9  //REV 1.0

11 #include "umacros.h"

13 #ifndef __VFCONTROL__
   #define __VFCONTROL__
15
   typedef struct
17 {
       int direction; //(plus or minus 1)
19     unsigned long period;
       unsigned int dV;
21     unsigned int dF;
       unsigned int fstep;
23     unsigned int vstep;
       int ramping;
25 } typeRamp;

27
   typedef struct
29 {
       unsigned int vcommand;
31     unsigned int vmeas[8];
       unsigned int vbus;
33     unsigned int insum;
       signed int verror;
35     signed int accum;
       unsigned int pgain;
37     unsigned int igain;
       unsigned int mflag;
39 } typeVout;
```

```
41
   //Prototypes
43
   void RampVF(typeSIN_PWM * pwm, typeRamp * ramp);
45 void RampVFControl(typeSIN_PWM * pwm, typeRamp * ramp,
       volatile unsigned long *ptimer, unsigned long trigger,\
47     unsigned int myinstance);
   void controlV3(typeVout * vout, typeSIN_PWM * pwm);
49 void update_Vout(typeVout * vout);
   void setupV3ADC(void);
51
   //Globals
53


55


57 //typeRamp VFRamping = {1,0,1,1,0,0};
   //volatile unsigned long Ramp_Count = 0;
59
   #endif
```

D.4 Serial Communications Module

D.4.1 serialcomm.c

```
1  //Serial Communications Module
   //
3  //SCI Peripheral TMS320LF2406A
   //Author: Al-Thaddeus Avestruz
5  //Created: 2 November 2004
   //Copyright 2004 Al-Thaddeus Avestruz
7  //
   //serialcomm.c
9  //REV 1.0

11 #define ODD 0
   #define EVEN 1
13
   #include "regs240x.h"
15 #include "umacros.h"
   #include "serialcomm.h"
17 #include <stdlib.h>

19
   //SetupSerial tested 11/17/04
21 unsigned char SetupSerial(void)
   {
```

```
23 //  union TwoBytes brr;  //Bit Rate Register
   //  byte bSCICCR;
25
   //  bSCICCR.byte = 0;
27
   //Configure SCITx and SCIRx pins
29     MCRA |=0x0003;

31 //Enable SCI CLK
       SCSR1 |= 0x0040;  //Bit 6 SCI CLKEN
33
   //Enable RS485 driver --  needed for PIIPM
35     MCRA = MCRA&(~0x0004); //Makes port IOPA2
       PADATDIR |= 0x0400; //Set IOPA2 as output
37     PADATDIR |= 0x0004; //Set IOPA2 High -- Enable RS485 Driver
   //  PADATDIR &= ~0x0004;  //Sets IOPA2 Low
39
       resetSCI();
41 //Register Setup SCICCR - SCI Communication Control Register

43 //  bSCICCR.byte = DATA-1;  //Sets lower 3 bits
   //  bSCICCR.bit.b7 = STOP-1;
45 //  bSCICCR.bit.b6 = PARITY;
   //  bSCICCR.bit.b5 = PARITY_EN;
47 //  bSCICCR.bit.b4 = 0; //Disable Loopback
   //  bSCICCR.bit.b3 = 0; // 0 - Select Idle Line Mode; 1 - Address Bit Mode
49
   //  SCICCR=bSCICCR.byte;
51     SCICCR = 0x07;
       BITSET_L(SCICCR,7);
53     BITSET_L(SCICCR,6);
       BITSET_L(SCICCR,5);
55     BITSET_L(SCICCR,4);
       BITSET_L(SCICCR,3);
57
   //SCI Control Register
59     SCICTL1 = ((RXERRINT<<6)+(SWRESET<<5)+(TXWAKE<<3)+/
           (SLEEP<<2)+(TXENA<<1)+RXENA);
61
       SCICTL2 = ((RXBKINT<<1) + TXINT);
63


65


67


69 //Baud Rate Register Setup
```

```
71
   //      brr.lt = ((int) CLKOUT)/((int) BAUD*8) - 1;
73              // Save the 16-bit value in local

75  //      SCIHBAUD = brr.bt[1];
        // Write low byte to Baud Select Register High byte
77  //      SCILBAUD = brr.bt[0];
        // Write high byte to Baud Select Register Low byte
79
        SCIHBAUD = 0x00;  //38400 baud with a 40 MHz CLKOUT
81      SCILBAUD = 0x81;

83 //Enable Receive Buffer Interrupt
   //  BITSET_H(SCIPRI,5);  //Low priority
85 //  BITSET_H(SCICTL2,1); //Enable
   //  BITSET_L(SCICTL2,1); //Disable
87
        SCIPRI = ((TXPRIORITY<<6)+(RXPRIORITY<<5)+(SCISOFTFREE<<3));
89
        setSCI();
91


93


95 //Enable SCI
   //  BITSET_H(SCICTL1,1); //Transmitter enable
97 //  BITSET_L(SCICTL1,0); //Receiver disable

99      SCICTL1 |=0x0020;
   }
101
   //sendChar() tested  11/17/04
103 unsigned char sendChar(unsigned char cinput)
   {
105     while (0==BITGET(SCICTL2,7));    //Wait until buffer is empty
        SCITXBUF=cinput;
107     return(1);
   }
109
   unsigned char sendc(unsigned char cinput)
111 {
        if (1==BITGET(SCICTL2,7))    //Check if buffer empty
113     {
            SCITXBUF=cinput;
115         return(1);
        }
117     else return(0);
   }
```

```
119
   void print_reg(char label, int reg)
121 {
        int i = 0;
123     sendChar(label);
        sendChar('␣');
125     for (i=16; i>0; i--)
        {
127         if ((12==i)||(8==i)||(4==i)) sendChar('␣');
            if (1 == ((reg>>(i-1))& 0x0001)) sendChar('1');
129             else sendChar('0');
        }
131     sendChar('\r'); sendChar('\n');
   }
133
   inline unsigned char in_sendChar(unsigned char cinput)
135 {
        if (1==BITGET(SCICTL2,7))
137     {
            SCITXBUF=cinput;
139         return(1);
        }
141     return(0);
   }
143

145 unsigned char sendStrLit(const char *inputstr)
        //To do;  doesn't work.
147     //need to do extra stuff because can't get a pointer to program memory
   {
149     while ('\0' != *inputstr)
        {
151         if (1==sendChar(*inputstr)) inputstr++;
        }
153 }

155 unsigned char sendString(char *inputstr)
   {
157     while ('\0' != *inputstr)
        {
159         sendChar(*inputstr);
            inputstr++;
161     }
   }
163
   unsigned char sendStringTask(char *inputstr, char *strqueue)
165 {
```

```
167 }

169 inline void resetSCI(void)
    {
171     asm("␣NOP");
        BITSET␣L(SCICTL1,5);  //Active low reset
173     asm("␣NOP");

175 }

177 inline void setSCI(void)
    {
179     asm("␣NOP");
        BITSET␣H(SCICTL1,5);     //Enable
181     asm("␣NOP");
    }
183
    void iprintd(char *string, unsigned int n)
185 {
        unsigned int i = 0;
187     const long a[5] = {10000,1000,100,10,1};
        ldiv_t ldiv_r;
189
        for (i=0; i<5; i++)
191     {
            ldiv_r = ldiv((long)n,a[i]);
193         *string = 0x0030 + (unsigned int) ldiv_r.quot;
            n = (unsigned int)ldiv_r.rem;
195         string++;
        }
197     *string = '\0';  //Terminate string
    }
199


201
    void createcBuffer(typecBuffer * buffer)
203 {
        buffer->array = (char *) calloc(buffer->length, sizeof(char));
205 }

207 void writeSerBuffer(typecBuffer * buffer, int invar,
        volatile unsigned long *ptimer, unsigned long trigger, \
209         unsigned int myinstance)
    {
211     static int i = 0;
        //To do: put these static in the struct
213     //so each instance doesn't have to store its own
        static int j = 0;
```

```
215     static int n = 0;

217     ldiv_t ldiv_r;
        const long a[5] = {10000,1000,100,10,1};
219

221     if (0 == buffer->lock)
        {
223         n = invar;  //To do: this is redundant
            buffer->lock = myinstance;
225     }

227     if (buffer->empty && (myinstance == buffer->lock) \
            && !buffer->full && (*ptimer>trigger))
229     {
            if (i<(buffer->length - 2))
231         {
                if (0==j)
233                 {
                        if (n<0) *(buffer->array + buffer->index) = '-';
235                     else *(buffer->array + buffer->index) = ' ';
                        j++;
237                 }
                else
239                 {
                        if (j<6)
241                     {
                            ldiv_r = ldiv((long)n,a[j-1]);
243                         *(buffer->array + buffer->index) = 0x0030 + \
                                (unsigned int) labs(ldiv_r.quot);
245                         n = (unsigned int)ldiv_r.rem;
                            j++;
247                     } else
                        {
249                         *(buffer->array + buffer->index) = ' ';
                            j = 0;
251                         buffer->lock = 0;
                        }
253                 }
                buffer->index++;
255
            } else
257             {
                    *(buffer->array + buffer->index) = 'V';
259                     //Signal overflow
                    *(buffer->array + buffer->index) = '\0';
261                     //Terminate string
```

```
263                 disable_ints();
                    *ptimer = 0;
265                 buffer->lock = 0;
                    buffer->full = 1;
267                 enable_ints();
                }
269     }

271 }

273 void terminate_wrtSerBuffer(typecBuffer * buffer,
        volatile unsigned long *ptimer, unsigned long trigger)
275 {   //this guy closes up the buffer and sends it off to the interrupt
        //driven serial out routine
277     if (buffer->empty && !buffer->lock && !buffer->full && \
            (*ptimer>trigger))
279     {
            *(buffer->array + buffer->index) = ';';
281         buffer->index++;
            *(buffer->array + buffer->index) = '\0';
283         disable_ints();
            *ptimer = 0;
285         buffer->full = 1;
            enable_ints();
287     }

289     if (buffer->full && !buffer->reading) SCITXBUF = '\0';
        //wakes up the interrupt
291 }

293 void sendBuffer(typecBuffer * buffer)
    {
295     static int i = 0;

297     if (buffer->full)
        {
299         if (i<buffer->index)
            {
301             SCITXBUF = *(buffer->array + i);
                buffer->reading = 1;
303             i++;
            } else
305         {
                i=0;
307             buffer->index = 0;
                buffer->full = 0;
309             buffer->empty = 1;
                buffer->reading = 0;
```

```
311             }
        }
313 }
```

D.4.2 serialcomm.h

```
1  //Serial Communications Module
   //
3  //SCI Peripheral TMS320LF2406A
   //Author: Al-Thaddeus Avestruz
5  //Created: 2 November 2004
   //Copyright 2004 Al-Thaddeus Avestruz
7  //
   //serialcomm.h
9  //REV 1.0

11 #ifndef _SERIALCOMM_
   #define _SERIALCOMM_
13
   #define BAUD 38400
15 #define CLKOUT 40000000

17 //Setup Parameters
   #define DATA 8
19 #define PARITY_EN 0
   #define PARITY ODD
21 #define STOP 1 //One or two stop bits

23 //SCICTL1
   #define RXERRINT     0 // 0: disable rx error interrupts
25 #define SWRESET      0 // 0: Reset
   #define TXWAKE       0 // 0: no wakefunc now
27 #define SLEEP        0 // 0: sleep disabled
   #define TXENA        1 // 1: tx enable
29 #define RXENA        0 // 1: rx_enable
   //SCICTL2
31 #define RXBKINT      0 // 0: disable RX and break interr
   #define TXINT        1 // 0: disable TXRDY-interr
33 //SCIPRI
   #define TXPRIORITY   0 // 0: TXD-int on high priority (INT1)
35 #define RXPRIORITY   0 // 0: RXD-int on high priority (INT1)
   #define SCISOFTFREE 2 // on emulator suspend complete SCI
37
   //Function Prototypes
39
   unsigned char SetupSerial();
41
   unsigned char sendChar(unsigned char);
```

```
43 unsigned char recvChar();
   unsigned char sendc(unsigned char);
45
   unsigned char sendString(char *);
47
   inline unsigned char in_sendChar(unsigned char);
49 unsigned char sendStringTask(char *, char *);

51 inline void resetSCI(void);
   inline void setSCI(void);
53
   unsigned char sendStrLit(const char *);
55
   void print_reg(char, int);
57 //void printf(char *);

59 void iprintd(char *string, unsigned int number);

61 void createcBuffer(typecBuffer * buffer);
   void writeSerBuffer(typecBuffer * buffer, int invar,
63     volatile unsigned long *ptimer, unsigned long trigger, \
           unsigned int myinstance);
65 void terminate_wrtSerBuffer(typecBuffer * buffer,
       volatile unsigned long * ptimer, unsigned long trigger);
67 void sendBuffer(typecBuffer * buffer);

69
   #endif
```

D.5 Peripheral Driver Module

D.5.1 periphs.c

```
1 //TMS320LF2406A Peripheral Driver Module
  //
3 //Peripheral TMS320LF2406A
  //Author: Al-Thaddeus Avestruz
5 //Created: 11 November 2004
  //Copyright 2004 Al-Thaddeus Avestruz
7 //
  //periphs.c
9
  #include "regs240x.h"
11 #include "umacros.h"
   #include "periphs.h"
13 #include "serialcomm.h"

15 void setupEVB(void)
```

```
{
    EVBIMRA = 0x0000;    //Mask all EVB interrupts
//  BITSET_H(SCSR1,3);   //EVB Clock enable
    SCSR1 |= 0x0008;
        //Register may be Read-Modify-Write;  BITSET may not work.
    EVBIFRA = 0xffff;    //Reset all EVB flags
}

void setupTimer3(void)
{
//  BITSET_H(EVBIMRA,7); //enable Tmr3 period interrupts
    EVBIMRA = 0x0040;
    T3CON = T3SETUP;
    GPTCONB = 0x0000;

    T3PR = T3PERIOD; //Set period register
//  T3CNT = 0;   //Clear T3 Counter
    EVBIFRA = 0xffff;    //Reset Tmr3 interrupt flags
}

void setupADC(void)
{
    //Dedicated ADC Pins
    //Setup ADC Timer
    SCSR1 |= 0x0080;
    //Setup ADC Control Register
    ADCTRL1 = RESET_ADC + ADC_SOFT + ACQ_PRESCALE_X2 + \
        CPS_CLK_2 + START_STOP;
    ADCTRL2 = INT_DIS_SEQ1 + INT_DIS_SEQ2;

    //Maximum Conversions per autoconvert
    MAXCONV = 0x0000;   //1 conversion

    //ADC input channel sequencing
    CHSELSEQ1 = 0x0008; //ADC in from DC Reg Output
    CHSELSEQ1 += 0x0003>>4; //Inverter DC Bus Voltage

    //Reset ADC
    resetADC();

}

unsigned int readADC(unsigned int channel)
{
    MAXCONV = 0x0000;    //Single conversion
    CHSELSEQ1 = channel;
    ADCTRL2 |= RESET_SEQ1;
    ADCTRL2 |= SOC_SEQ1;     //start conversion
```

```
    asm("␣NOP");
    asm("␣NOP");
    asm("␣NOP");
    asm("␣NOP");
    while (ADCTRL2 & SEQ1_BSY);
    return(RESULT0);
}

void printADC(unsigned int channel)
{
    unsigned int result = 0;
    char strtmp[10] = "";

    result = readADC(channel);
    iprintd(strtmp,result);
    sendString(strtmp);
    sendChar('␣');

}

void resetADC(void)
{
    ADCTRL1 |= 0x4000;  //Reset ADC
    asm("␣NOP");
    ADCTRL1 &= ~0x4000;  //Unreset ADC
}
```

D.5.2 periphs.h

```
//TMS320LF2406A Peripheral Driver Header Module
//
//Peripheral TMS320LF2406A
//Author: Al-Thaddeus Avestruz
//Created: 11 November 2004
//Copyright 2004 Al-Thaddeus Avestruz
//
//periphs.h
//REV 1.0

#include "./pwm/include/F2407BMSK.h"
#include "umacros.h"

//Timer Parameters
//#define T3PERIOD 500
#define T3PERIOD 20000

//T3CON
```

```
#define T3SETUP (SOFT_STOP_FLAG + TIMER_CONT_UP + \
        TIMER_CLK_PRESCALE_X_1 + \
        TIMER_ENABLE_BY_OWN + TIMER_DISABLE + \
        TIMER_CLOCK_SRC_INTERNAL   )
        //Timer compare disabled; Own period register

//      For a 40 Mhz clock w/ prescale 1, \
\\      tick = 500us for T3Period 20000


//GPTCONB

//ADC Bit Masks
 //ADCTRL1
#define RESET_ADC          0x4000
 #define ADC_SOFT          0x2000
#define ADC_NOSOFT         0x0000
 #define ADC_FREE          0x1000
#define ADC_NOFREE         0x0000
 #define ACQ_PRESCALE_X1       0x0000
#define ACQ_PRESCALE_X2       0x0100
 #define ACQ_PRESCALE_X3       0x0200
#define ACQ_PRESCALE_X4       0x0300
 #define CPS_CLK_1    0x0000
#define CPS_CLK_2     0x0080
 #define CONT_RUN     0x0040
#define START_STOP    0x0000
 #define INT_H_PRIORITY        0x0000
#define INT_L_PRIORITY         0x0020
 #define DUAL_SEQUENCE         0x0000
#define CASCADE                0x0010
 #define CAL_ENABLE            0x0008
#define BRIDGE_EN              0x0004
 #define REFLO                 0x0000
#define REFHI                  0x0001
 //ADCTRL2
#define RESET_SEQ1         0x4000
 #define START_CAL         0x4000
#define SOC_SEQ1           0x2000
 #define SEQ1_BSY          0x1000
#define INT_DIS_SEQ1       0x0000
 #define INT_ENA1_SEQ1     0x0400
#define INT_ENA2_SEQ1      0x0800
 #define INT_FLAG_SEQ1     0x0200
#define EVA_SOC_SEQ1       0x0100
 #define RESET_SEQ2        0x0040
#define SOC_SEQ2           0x0020
 #define SEQ2_BSY          0x0010
```

```
67 #define INT_DIS_SEQ2     0x0000
   #define INT_ENA1_SEQ2    0x0004
69 #define INT_ENA2_SEQ2    0x0008
   #define INT_FLAG_SEQ2    0x0002
71 #define EVB_SOC_SEQ2     0x0001

73


75
   //Prototypes
77 void setupEVB(void);
   void setupTimer3(void);
79
   void setupADC(void);
81 unsigned int readADC(unsigned int _channel);
   void resetADC(void);
83 void printADC(unsigned int _channel);

85 void setupCapture(unsigned int _timer);
   void handleCapture(void);
87

89 //Macros
   #define ENABT3()  (T3CON |= TIMER_ENABLE)
91 #define DISBT3()  (T3CON &= ~TIMER_ENABLE)
```

Bibliography

[1] T.M. Jahns and E.L. Owen. Ac adjustable-speed drives at the millennium: how did we get here? *Power Electronics, IEEE Transactions on*, 16(1):17–25, 2001. TY - JOUR.

[2] Jack Wade Holloway. *Harmonic control of multiple-stator machines for voltage regulation.* M.eng, Massachusetts Institute of Technology, 2004.

[3] Steven B. Leeb. Notes on field oriented control of induction motors, 2001.

[4] H. Wayne Beaty and Jr. James L. Kirtley. *Electric Motor Handbook.* McGraw-Hill, New York, 1998.

[5] A.-T. Avestruz, J.W. Holloway, R. Cox, and S.B. Leeb. Voltage regulation in induction machines with multiple stator windings by zero sequence harmonic control. In *Applied Power Electronics Conference and Exposition, 2005. APEC 2005. Twentieth Annual IEEE*, volume 2, pages 746–752 Vol. 2, 2005. TY - CONF.

[6] Philip Langdon Alger. *Induction Machines, Their Behavior and Uses.* Gordon and Breach, New York, 1970.

[7] Colonel William T. McLyman. *Transformer and inductor design handbook.* Marcel Dekker, New York, 3rd edition, 2004. Colonel WM. T. McLyman. ill. ; 29 cm. Electrical and computer engineering ; 121.

[8] Furukawa Electric. Triple insulated wire - standard type tex-e, 2006.

[9] A. Munoz-Garcia, T.A. Lipo, and D.W. Novotny. A new induction motor v/f control method capable of high-performance regulation at low speeds. *Industry Applications, IEEE Transactions on*, 34(4):813–821, 1998. TY - JOUR.

[10] G.-W. Chang, G. Espinosa-Perez, E. Mendes, and R. Ortega. Tuning rules for the pi gains of field-oriented controllers of induction motors. *Industrial Electronics, IEEE Transactions on*, 47(3):592–602, 2000. TY - JOUR.

[11] James K. Roberge. *Operational amplifiers : theory and practice.* Wiley, New York, 1975. James K. Roberge. ill. ; 24 cm.

[12] International Rectifier. Piipm15p12d007 programmable isolated integrated power module, September 2003 2003.

[13] Ned Mohan, Tore M. Undeland, and William P. Robbins. *Power Electronics.* John Wiley and Sons, New York, 2nd edition, 1995.

[14] T. Abeyasekera, C.M. Johnson, D.J. Atkinson, and M. Armstrong. Elimination of subharmonics in direct look-up table (dlt) sine wave reference generators for low-cost microprocessor-controlled inverters. *Power Electronics, IEEE Transactions on*, 18(6):1315–1321, 2003. TY - JOUR.

[15] A.M. Sodagar and G.R. Lahiji. A pipelined rom-less architecture for sine-output direct digital frequency synthesizers using the second-order parabolic approximation. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, 48(9):850–857, 2001. TY - JOUR.

[16] J. Vankka. Methods of mapping from phase to sine amplitude in direct digital synthesis. *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on*, 44(2):526–534, 1997. TY - JOUR.

[17] A.M. Sodagar and G.R. Lahiji. Second-order parabolic approximation: a new mathematical approximation dedicated to rom-less ddfss. In *Microelectronics, 2000. ICM 2000. Proceedings of the 12th International Conference on*, pages 47–50, 2000. TY - CONF.

[18] A. Consoli, G. Oriti, A. Testa, and A.L. Julian. Induction motor modeling for common mode and differential mode emission evaluation. In *Industry Applications Conference, 1996. Thirty-First IAS Annual Meeting, IAS '96., Conference Record of the 1996 IEEE*, volume 1, pages 595–599 vol.1, 1996. TY - CONF.

[19] M. Cacciato, A. Consoli, G. Scarcella, and A. Testa. Continuous pwm to square wave inverter control with low common mode emissions. In *Power Electronics Specialists Conference, 1998. PESC 98 Record. 29th Annual IEEE*, volume 1, pages 871–877 vol.1, 1998. TY - CONF.

[20] T. Kawabata, T. Miyashita, and Y. Yamamoto. Digital control of three-phase pwm inverter with lc filter. *Power Electronics, IEEE Transactions on*, 6(1):62–72, 1991. TY - JOUR.

[21] J.D. Park, C. Khalizadeh, and H. Hofmann. Design and control of high-speed solid-rotor synchronous reluctance drive with three-phase lc filter. In *Industry Applications Conference, 2005. Fourtieth IAS Annual Meeting. Conference Record of the 2005*, volume 1, pages 715–722 Vol. 1, 2005. TY - CONF.

[22] P. Barbosa, F. Canales, and F. Lee. Passive input current ripple cancellation in three-phase discontinuous conduction mode rectifiers. In *Power Electronics Specialists Conference, 2001. PESC. 2001 IEEE 32nd Annual*, volume 2, pages 1019–1024 vol.2, 2001. TY - CONF.

[23] R. Balog and P.T. Krein. Automatic tuning of coupled inductor filters. In *Power Electronics Specialists Conference, 2002. pesc 02. 2002 IEEE 33rd Annual*, volume 2, pages 591–596 vol.2, 2002. TY - CONF.

[24] S. Senini and P.J. Wolfs. The coupled inductor filter: analysis and design for ac systems. *Industrial Electronics, IEEE Transactions on*, 45(4):574–578, 1998. TY - JOUR.

[25] B. Jeftenic, S. Milosavljevic, N. Mitrovic, and M. Rodic. Speed control in inverter induction motor drives based on only one current sensor. In *Power Electronics Specialists Conference, 1996. PESC '96 Record., 27th Annual IEEE*, volume 1, pages 364–369 vol.1, 1996. TY - CONF.

[26] H. Kim and T.M. Jahns. Phase current reconstruction for ac motor drives using a dc-link single current sensor and measurement voltage vectors. In *Power Electronics Specialists, 2005 IEEE 36th Conference on*, pages 1346–1352, 2005. TY - CONF.