# Designs for Ultra-High Efficiency Grid-Connected Power Conversion

by

Brandon J. Pierquet

S.M., Massachusetts Institute of Technology (2006)
B.S., University of Wisconsin–Madison (2004)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

Author ........ [illegible] ........
Department of Electrical Engineering and Computer Science
May 17, 2011

Certified by ........ [illegible] ........
Professor David J. Perreault
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ........ [illegible] ........
Professor Leslie A. Kolodziejski
Department of Electrical Engineering and Computer Science
Chairperson, Department Committee on Graduate Students

# Designs for Ultra-High Efficiency Grid-Connected Power Conversion

by

Brandon J. Pierquet

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2011, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Abstract

Grid connected power conversion is an absolutely critical component of many established and developing industries, such as information technology, telecommunications, renewable power generation (e.g. photovoltaic and wind), even down to consumer electronics. There is an ever present demand to reduce the volume and cost, while increasing converter efficiency and performance. Reducing the losses associated with energy conversion to and from the grid can be accomplished through the use of new circuit topologies, enhanced control methods, and optimized energy storage. The thesis outlines the development of foundational methods and architectures for improving the efficiency of these converters, and allowing the improvements to be scaled with future advances in semiconductor and passive component technologies.

The work is presented in application to module integrated converters (MICs), often called micro-inverters. These converters have been under rapid development for single-phase grid-tied photovoltaic applications. The capacitive energy storage implementation for the double-line-frequency power variation represents a differentiating factor among existing designs, and this thesis introduces a new topology that places the energy storage block in a series-connected path with the line interface. This design provides independent control over the capacitor voltage, soft-switching for all semiconductor devices, and full four-quadrant operation with the grid.

Thesis Supervisor: Professor David J. Perreault
Title: Department of Electrical Engineering and Computer Science

# Acknowledgments

Firstly, and without a doubt, most importantly, I must thank my research advisor Dave Perreault. I have had the pleasure of working with him since I started at MIT, and could not have asked for a better mentor. Over these past seven years, he has selflessly shared his humor, knowledge and extensive experience, for which I am a better engineer and person.

Robert and Bill have been two of the best colleagues and friends that I could wish for. They have been present since the beginning of this expedition, which has been filled with both dark and bright times. I owe both so much more than can be expressed here. With the inclusion of Brienne, Brooke, Nikolai, and Cara, I consider all of them family.

All other LEES students, both past and present, that have made the line between work and fun often indistinguishable. The open sharing of ideas and vast expertise is something I have rarely found outside of our humble basement, and it will be sorely missed.

There are too many other people to name that have been a part of my time at MIT. some have come and gone along the way, and others that were here when I arrived are still here as I leave. Those that have helped me know who they are, and how grateful I am.

A substantial portion of the funding for this research came from the generous support of Enphase Energy. Their interest in the outcome of this work has been extremely motivating, as has been their tolerance for the twists, turns, and dead-ends that plague this type of work.

And finally, I thank you, the reader, for your interest in this thesis. I sincerely hope that you find the information contained within to be worthy of your time.

# Contents

# List of Figures

# List of Tables

*Chapter 1*

# *Introduction*

## 1.1 Background

Initial use of power from the alternating current electricity distribution system was focused on automating machinery and providing lighting. While the basic applications are still in use a century later (albeit more refined), they've been increasingly replaced by more advanced implementations, or alternatives that don't utilize AC natively. Instead, modern grid connected machines and devices convert the grid voltage it to a more appropriate form such as DC or high-frequency modulated AC. Additionally, much of the electricity used today is in residential and commercial environments, a shift from the primarily industrial usage a century ago. This document focuses on electrical systems from the perspective of grid connected electronics, specifically photovoltaic inverter systems.

Connecting electronic devices to the AC distribution system is a well understood task, and significant work has been completed in both sourcing power from, and delivering power to the grid [ – ]. Much of this work is focused on three-phase interconnection of varying line voltages, with power levels ranging from 10–500 kW, often for applications such as motor drives [ , ], electric vehicle drivetrains [ ], wind turbines [ , ], and UPS systems [ , ].

In contrast to these existing high-power multiphase systems, the electrical systems found in commercial and residential environments often operate on a single or split-phase interface at a significantly lower power level. Renewed focus on energy efficiency and small-scale

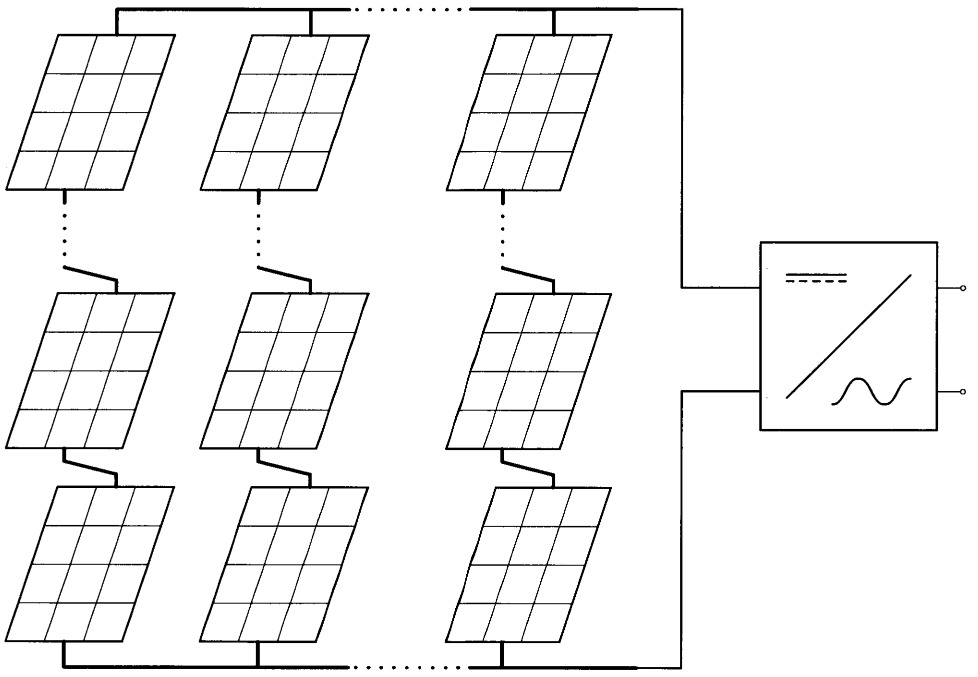


Figure 1.1: A centralized-inverter topology converts dc-power from parallel-strings of series-connected solar modules to grid-connected ac power.

distributed generation has created a demand for more effective solutions in a number of areas. One of these areas, which is the central area of application for this thesis, is scalable residential and commercial photovoltaic energy conversion.

### 1.1.1 Photovoltaic Installation Types

Grid-tied inverters for photovoltaic systems represent a rapidly developing area. Traditionally, installations use series-connected into modules info strings to increase voltage, with parallel sets of strings used to increase the output power. These arrays, as illustrated in Fig. 1.1 are interfaced to a single dc-ac inverter, as the single inverter can be optimized for a fixed-size array. Typical utility scale installations may exceed 1 MW, whereas medium and small commercial rooftop installations may range from 75–250 kW.

As centralized inverters have evolved to support smaller installation sizes, micro-inverters, also known as module-integrated converters (MICs), have been developed to interface a single, low-voltage (25–50 v, typically) panel to the AC grid [ – ]. Micro-inverters provide a number of system benefits: array redundancy and scalability, ease of installation, and increased performance in partially shaded conditions [ ]. Notable drawbacks are the duplication of components (enclosures, control circuits, etc), and the difficulty in obtaining the same efficiencies as inverters which manage multiple series-connected modules at higher power levels. A single 72-cell panel, with a nominal output voltage of 36 V, requires a much larger transformation ratio to interface with a grid voltage of 240 V than a series string of 10 modules requires.

### 1.1.2 Single Phase Challenges

The grid interconnection most commonly available in residential and small commercial systems is single phase ac (e.g. at 120, 208, or 240 $V_{rms}$). Module integrated converters typically target these electrical systems [ ], however one large challenge for these single phase converters is the sinusoidally varying power transfer to the grid. The constant power output of a solar module is poorly matched to this time varying requirement, as is shown in Fig. 1.2, where the gray shaded areas illustrate the energy storage required to compensate for the instantaneous power mismatch.

The flow of power through the converter can be modeled as a three port system such as the one in Fig. 1.3. Power flow must be managed between each port, with the fixed requirements of maintaining dc plus twice-line-frequency sinusoidal power flow with grid, and constant power draw from the solar module. The input and output power transfers can be described by

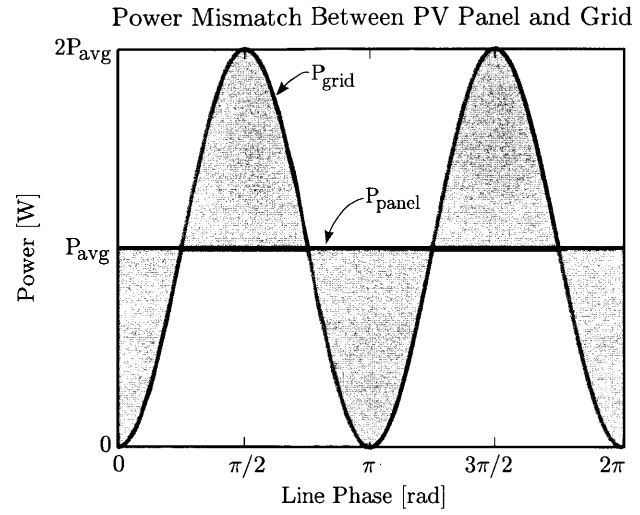


Figure 1.2: The power flow mismatch between the grid and a constant power source results in the shaded area, representing the required energy storage.

$$P_{\text{PV}} = -P_{avg}, \tag{1.1}$$

$$P_{\text{Line}} = P_{avg}(1 - cos(2\omega_l t)), \tag{1.2}$$

and the power transfer of the energy storage buffer is determined by the difference in power between these two ports, specifically

$$P_{\text{Buf}} = P_{avg}\cos(2\omega_l t). \tag{1.3}$$

The absolute minimum required energy storage for buffering the power transfer mismatch is the summation of power transferred into, or out of, the buffer over one-half of a line cycle. Integrating (1.3) yields

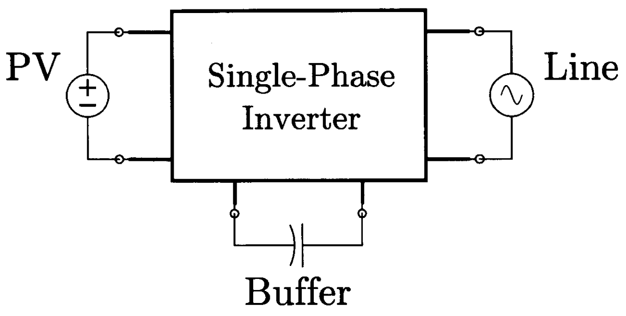


Figure 1.3: The generalization of a grid-connected power converter as a three-port system.

$$W_{\min} = \frac{P_{avg}}{\omega_l}, \tag{1.4}$$

which represents the lower bound on the energy storage required by the converter to properly manage the power flow in the converter. This, however, does not account for further constraints imposed by the converter implementation, or more specifically the circuit topology.

### 1.1.3 Existing Topologies

For single-phase module-integrated converters, past literature reviews have investigated a comprehensive set of micro-inverters designs below 500 W [ , ], and classified them into categories based on their number and type of their power conversion stages. Here, three types of converters are outlined, however, they are categorized by the location and operation of the energy storage within the converter.

Most single-stage topologies, such as the flyback and ac-link converters, place capacitance in parallel with the input [ , ]. This is an effective low-complexity implementation, but in order to avoid interfering with the peak-power tracking efficiency, substantial energy

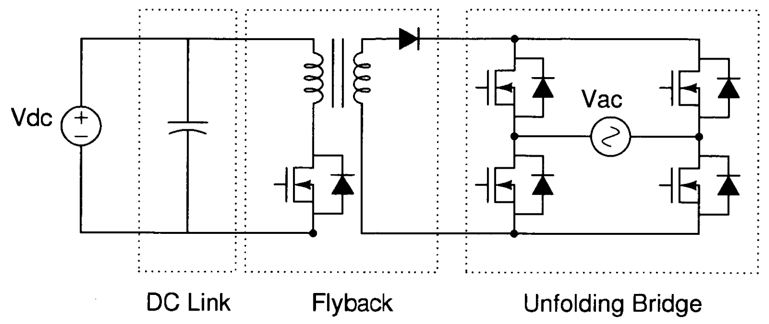


Figure 1.4: Grid-connected inverter with primary energy storage located across the input source. Implemented using a flyback converter and unfolding bridge.

storage is required to keep the voltage ripple extremely low across the panel. A common second method involves two cascaded conversion stages, providing energy storage at an intermediate dc bus. This arrangement can be implemented with less energy storage than the previous method, as a much larger voltage fluctuation on the intermediate bus can be tolerated. Additionally, recent work has investigated alternate "third-port" topologies (e.g. [ , ]), which can control the voltage on the energy storage capacitor independent of the input and output voltages; the series-buffer-block converter presented in Chapter 2 belongs to this category.

The first converter topology considered places the energy storage buffer across the low voltage dc port, in parallel with the PV panel. This type of connection has a significant energy storage requirement, as fluctuation of the voltage across the panel impacts the peak-power tracking effectiveness of the converter, reducing system efficiency. The conversion from the low-voltage dc can be accomplished with a number of combinations of high-frequency inverters and rectification schemes, such as the flyback converter followed by an unfolding stage shown in Fig. 1.4, or a single-stage ac-link structure with cycloconverter output [ ] (not shown). Due to the location of the energy storage, the entire converter must process the full range of power into the line (e.g. $0 - 2P_{\text{avg}}$).

The second converter topology considered decouples the power flow from the low-voltage

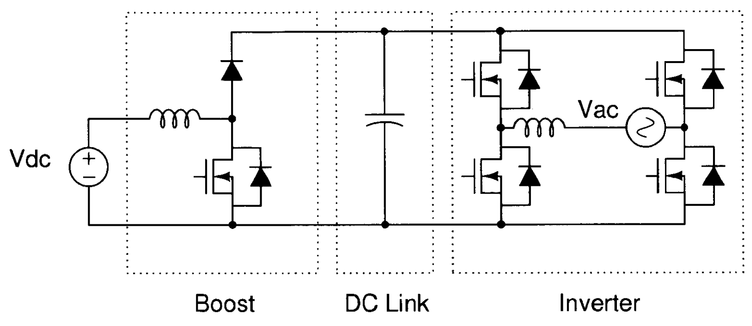
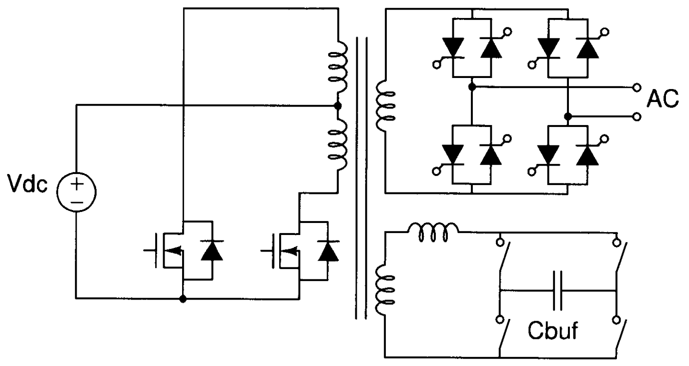


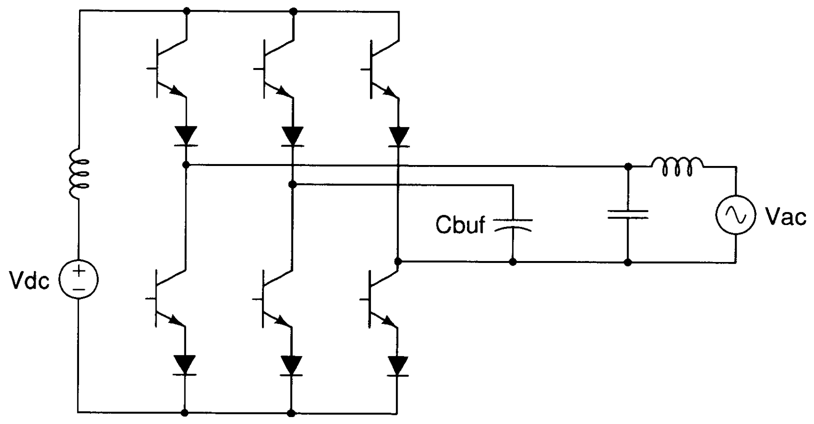
Figure 1.5: Grid-connected inverters with an intermediate DC bus for primary energy storage. Implemented using a boost stage to feed the DC bus, and a full-bridge inverter across the line.

dc port and the line interface through the use of a shared intermediate dc bus. A dc-dc boost stage operates to isolate (if necessary), and scale the voltage up to the high-voltage bus, which is held within a voltage range above the peak of the ac line voltage. Interfacing to this intermediate dc bus to the line then is performed by a dc-ac converter, either in the form of a full-bridge inverter, as shown in Fig. 1.5, or using a buck converter with an unfolding bridge. In this topology, the second stage is required to process up to double the average power, however the first stage only needs to convert the constant power output from the panel.

A number of alternative topologies decouple the energy storage from both the input and output stages of the converter, using a actively controlled "third-port". Two of these topologies [ , ], shown in Fig. 1.6, implement the storage port as a separate branch that is in parallel to the line and PV ports. This additional port is controlled to allow the capacitor voltage to vary independently from the input and output voltages, which can permit a substantial reduction in the energy storage compared to the topologies in Figs. 1.4 and 1.5.



(a)



(b)

Figure 1.6: Grid-connected inverters with a third port for primary energy storage. (a) Implemented using an isolated ac-link structure [ ], and (b) implemented as a non-isolated current-fed converter [ ].

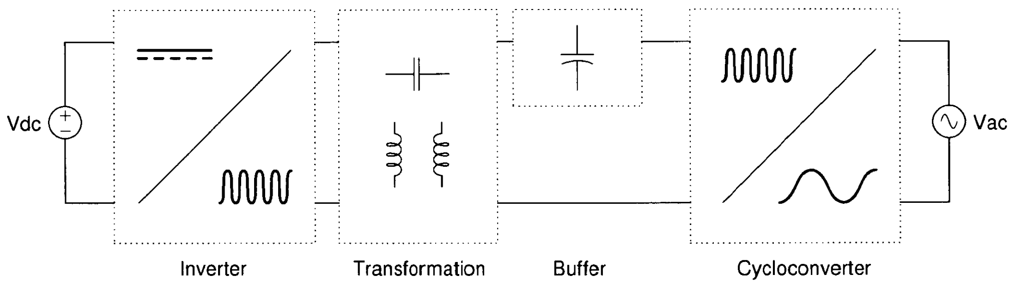


Figure 1.7: Block diagram of the series-buffer-block converter, illustrating the single-port nature of the buffer.

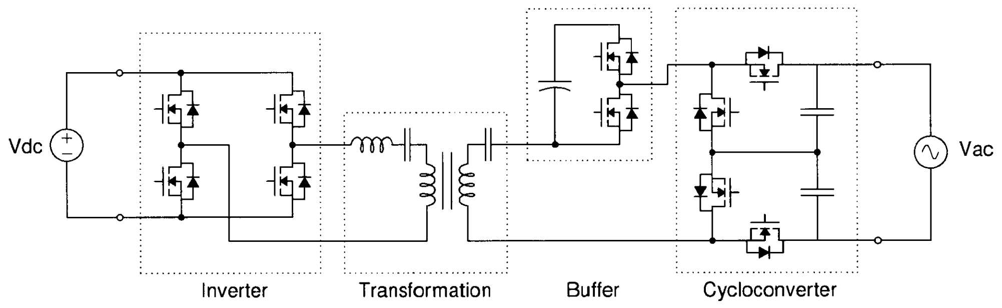


Figure 1.8: Schematic of proposed PV micro inverter, corresponding to the block diagram in Fig. 1.7.

## 1.2 Series-Buffer-Block Converter Topology

An additional third-port topology, which is the primary focus of this thesis, has been developed that places the energy storage in series with the input and output ports instead of parallel as in Fig. 1.6. The high-level block diagram and the schematic of the proposed converter are presented in Figs. 1.7 and 1.8. The dc-connected inverter transforms the dc source into high frequency ac, with the transformation stage providing both voltage gain and appropriate impedance shaping. The cycloconverter is controlled to modulate the transfer of the high frequency resonant current in response to the changing voltage of the ac port; the buffer-block acts in a similar manner, but is controlled to absorb or deliver power to the storage port to compensates for the power mismatch between the dc and ac ports.

The proposed topology in Fig. 1.8 contains four low-voltage devices for the full-bridge inverter, four high-voltage devices for the bi-directional half-bridge cycloconverter, and two additional high-voltage devices for the half-bridge series buffer. The half-bridge buffer is positioned on the secondary side of the transformation stage, which substantially reduces the volt-second magnitude imposed on the transformer, and permits use of the higher energy density of high-voltage capacitors in the buffer. Alternatively, if improved energy storage and semiconductor devices are available at lower voltages, the buffer-block can be placed on the primary side of the transformer.

In comparison to existing designs outlined in Section 1.1.3, including the third-port topologies, this topology effectively places all major power processing blocks, (e.g. the high-frequency inverter, buffer-block, and cycloconverter) in a series path with respect to the high-frequency resonant current. This allows the power-flow to be modulated in each block by controlling the switching function relative to the current.

The placement of each block in series with the drive current seems, at first glance, to impose a heavy conduction loss penalty. However, the proposed approach provides means to mitigate this loss, in addition to presenting opportunities not found in previous designs. Using unipolar devices such as MOSFETs, and implementing zero-voltage switching (ZVS) for the primary switches, allows the semiconductor area to be scaled up to reduce conduction loss [ ]. Devices such as IGBTs, SCRs, and diodes operate with a fixed on-state voltage drop, an intrinsic property of the p-n junction, which does not scale with device area.

MOSFET device figure-of-merit values have improved steadily since their introduction, and the recent use of charge-compensation principles has allowed high-voltage silicon MOSFETs to surpass the "silicon limit" [ – ] and become viable for voltage ranges once relegated to IGBT devices alone. Additionally, the emergence of wide-bandgap based devices, implemented in SiC and GaN, have the potential to dramatically reduce the on-state resistance of devices even further while reducing undesirable parasitics [ , ]. This historical semiconductor device progress, combined with these and other anticipated future improvements, are a motivating factor in the elimination of p-n junction devices with this topology development.

## 1.3 Thesis Objectives and Organization

The primary objective of this thesis is to develop a method for implementing a high-efficiency grid-tied power converter, using the unexplored circuit topology of Fig. 1.8, including the control methods for single-phase grid interconnection. Additional functional goals for the operation of the converter include

- Independent control of the power buffering, allowing variable-voltage energy storage,
- Stable and controllable over wide input and output voltages, and output power levels,
- Bi-directional power transfer, including reactive power transfer capability,
- Scalability to future semiconductor device technology, voltage levels, and power requirements.

Following this introductory chapter, Chapter 2 develops the converter model, operating fundamentals, and control methodology. A proof-of-concept prototype is presented in Chapter 3, demonstrating the converter operation and performance results for single-phase operation. Chapter 4 presents a variation of the converter topology, as applied to a three-phase grid interface. The thesis is concluded in Chapter 5 with remarks on possible areas for future development.

The appendices which follow the thesis include associated derivations referenced in the text, the developed computer codes used for converter simulation, the printed circuit board artwork, and digital controller implementation details.

*Chapter 2*

# *Converter Operation and Control*

## 2.1 Introduction

In the general form, operation of the converter requires control over the switching functions of each block relative to others. The combined voltage pattern of all active blocks, imposed on the transformation stage, is responsible for generating the resonant current that links the converter. In turn, the switching pattern of each block relative to this resonant current is what determines the average power delivery for that block. This results in a tightly-coupled non-linear relationship between the output voltage waveform of each block and their respective power deliveries.

To approach the control of the converter, given this initial complexity, it is broken down into a common switching sub-circuit, which is then analyzed and used to construct a generalized model of the converter operation. This model is then used to illustrate the development of the system control methods and a functional prototype design.

## 2.2 System Modeling

At the outset of the analysis, two reasonable assumptions are made about the converter's operation:

1. The voltage at each terminal of the converter (PV, buffer, and line) changes slowly enough, relative to the switching frequency, that they can be approximated as constant over a switching cycle. This effectively decouples the high-frequency switching model of the converter from the low-frequency power transfer model used over a line cycle.

2. The quality factor of the series resonant circuit is sufficiently high to approximate it as a sinusoidal current source operating at the switching frequency. This offers the opportunity to use of phasor analysis, and calculate equivalent impedances.

### 2.2.1 Power Transfer Modulation

The modulation of power through the blocks of the converter is accomplished by controlling the switching function of each block relative to the series resonant current. To quantify this operation, the canonical switching module of the converter is used for illustration. In its most general form, the canonical switch model shown in Fig. 2.1a is composed of a single pole, and two throws. A voltage source $V_t$ is placed across the throws, and a current source $I_r$ is connected between the pole and a single throw. The operation of the three-terminal switch prevents an open-circuit of the current source, and the short-circuit of the voltage source. The operation of the three-terminal switch can also be implemented with two complimentary operated two-terminal switches, as shown in Fig. 2.1b. For purposes of this analysis, we ignore the details of the switching transitions, including means providing zero-voltage switching in the actual converter.

The modulation of power between the current and voltage sources is determined by the values of these sources, and the function $Q$ controlling the switch operation. The current in this switching module represents the resonant current through the converter, and the voltage source as one of the terminal voltages (e.g. cycloconverter or buffer-block), which can be written as

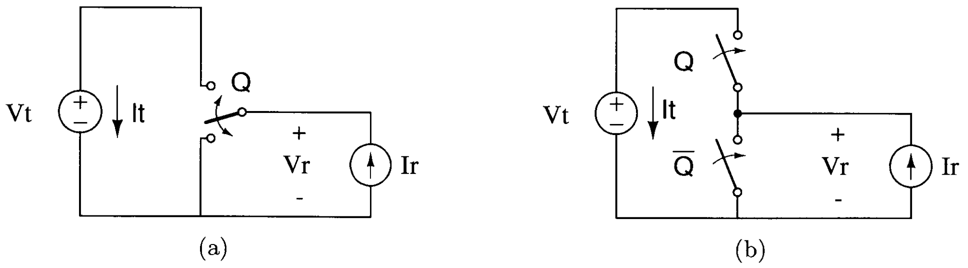


Figure 2.1: The standard switching module implemented with (a) a canonical single-pole-dual-throw switch, and (b) two complimentary single-pole-single-throw switches.

$$i_r(t) = I_r \sin(\omega_{\text{sw}} t), \tag{2.1}$$

$$v_t(t) = V_t. \tag{2.2}$$

The control of the switch, and its influence on operating waveforms, is most easily illustrated by waveforms in Fig. 2.2. The operation can be considered from the perspective of either the current or voltage source. During the time in which $Q$ is on, the current source has a voltage $V_t$ applied across it, with the voltage being otherwise zero; when $Q$ is on, the voltage source is fed by a current $i_r(t)$, with the current being otherwise zero.

When average transfer of power *from* the current source can over a switching cycle be written as

$$P_r = \frac{1}{T_{\text{sw}}} \int_0^{T_{\text{sw}}} v_r(t) i_r(t) dt, \tag{2.3}$$

where the current $i_r(t)$ is defined to be sinusoidal in (2.1), and the voltage $v_r(t)$ is the product of $v_t(t)$ with the switching function $Q(t)$. The switching function for the module

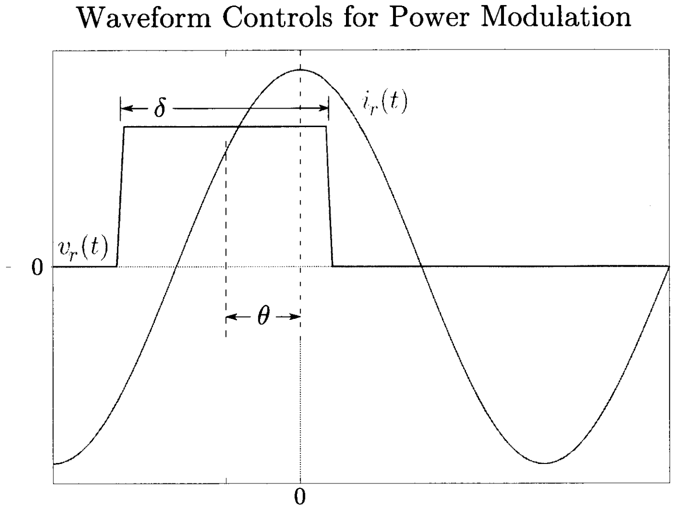


Figure 2.2: The relationship between the series-path current and switching function determines the transfer of energy through the converter.

(as illustrated in Fig. 2.2) can be written with the conditional assignment

$$Q(t) = \begin{cases} 1, & \text{if } \dfrac{\omega_{\text{sw}} t}{2\pi} \in \left[\theta - \dfrac{\delta}{2}, \theta + \dfrac{\delta}{2}\right] \\ 0, & \text{else} \end{cases}, \tag{2.4}$$

and effectively provides a windowing effect. The above definitions can be used to find the cycle-averaged power transfer of (2.3) to be

$$P_r = \frac{V_t I_r}{2\pi} \int_{\theta - \frac{\delta}{2}}^{\theta + \frac{\delta}{2}} \sin(\omega_{\text{sw}} t) d\omega_{\text{sw}} t \tag{2.5}$$

$$P_r = \frac{V_t I_r}{\pi} \sin(\delta/2) \cos(\theta), \tag{2.6}$$

given the parameters $\delta$ and $\theta$, expressed in (switching cycle) radians, that are the direct results of chosen switching function $Q(t)$.

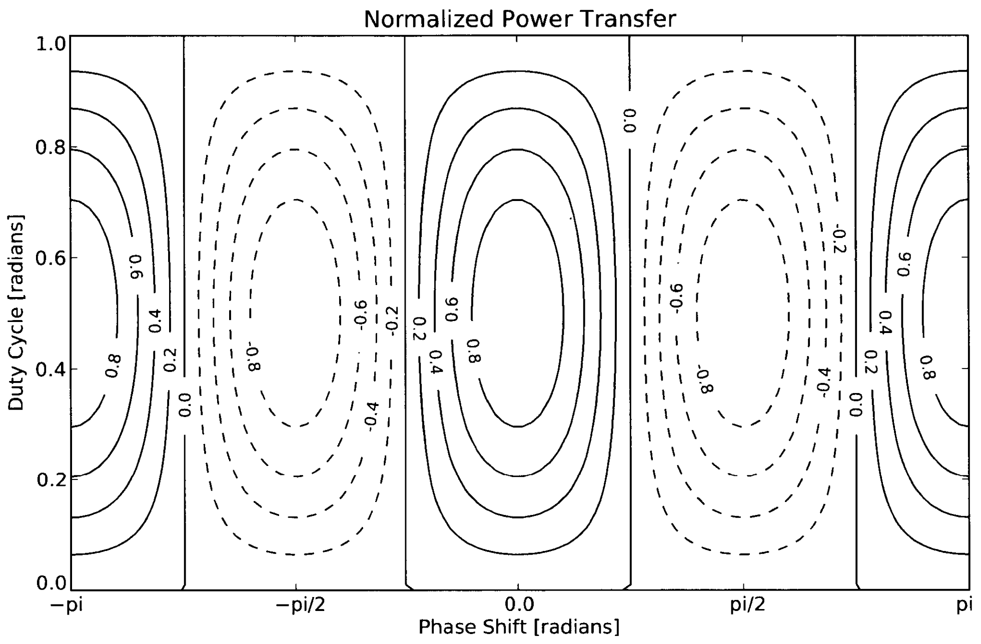


Figure 2.3: Power transfer relationship for voltage phase-shift ($\theta$), and pulse width ($\delta$).

To visualize the power transfer of (2.6), Fig. 2.3 illustrates the normalized power transfer for the phase space of $\{\theta, \delta\} \in [-\pi, \pi]$, given $V_t > 0$. The result is symmetric along both $\theta = 0$ and $\delta = \pi$, providing multiple solutions for a given power transfer, if the set is not constrained further.

To address the continuum of parameter combinations, two specific switch modulation cases are considered: phase-shift modulation, and pulse-width modulation. The basis for the phase-shift modulation is to maintain a fixed pulse width, $\delta$, and shift the phase of the switching function, $\theta$, relative to the resonant current. Alternatively, in the pulse-width modulation (PWM) method, the pulse width, $\delta$, is controlled such that the high side switch remains on for the duration required to obtain the required energy transfer at a chosen $\theta$. In both cases, the turn-on and turn-off transitions for all devices can be selected such that they occur under zero-voltage conditions. The selection process is explored further in

Section 2.5.

The primary side full-bridge inverter is controlled by phase-shifting the two halves of the canonical switching modules in opposing directions relative to a reference, with each operating at a fixed one-half duty cycle. This phase shift is what controls the pulse width, seen at the output as a differential waveform. The average power transfer over a switching cycle can be found, using the same method in (2.3), to be

$$P_r = 2\frac{V_t I_r}{\pi} \sin(\delta/4)\cos(\theta), \tag{2.7}$$

where $\delta$ denotes the pulse width, and the phase $\theta$ denotes the difference in phase between the output voltage waveform and the series resonant current, expressed in (switching cycle) radians.

### 2.2.2 Equivalent Impedance

The input impedance of the canonical switching module, as shown in Fig. 2.4, is important in the design of the converter, particularly the full-bridge inverter and transformation stage. The combination of the buffer-block and cycloconverter act as an effective load during the converter operation, and understanding how this load changes over time, or through changes in the control parameters, can significantly influence the implementation.

With the resonant current waveform $i_r(t)$ defined as a sinusoid in (2.1), the fundamental component of $v_r(t)$ can be used to calculate the effective input impedance of the switching block at the operating frequency $\omega_{\text{sw}}$. Using phasors, $i_r(t)$ and $v_r(t)$ can be written (with the $e^{j\omega_{\text{sw}}t}$ factor omitted) as

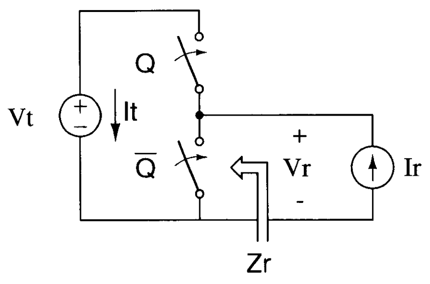


Figure 2.4: The input impedance notation of the canonical switching module.

$$\bar{I_r} = I_r \tag{2.8}$$

$$\bar{V_r} = V_t \frac{2}{\pi} \sin(\delta/2) e^{j\theta}. \tag{2.9}$$

The impedance driven by the current source, as shown in Fig. 2.4, is written simply as

$$Z_r = \frac{\bar{V_r}}{\bar{I_r}} \tag{2.10}$$

$$Z_r = \frac{V_t}{I_r} \frac{2}{\pi} \sin(\delta/2) e^{j\theta}. \tag{2.11}$$

This result indicates that the canonical switching module can present a variable magnitude complex load based on the selection of control variables $\delta$ and $\theta$.

Use of the current and voltage phasors in (2.8) and (2.9) can also be used to calculate the power transfer from the current source, yielding

$$P_r = \frac{1}{2} Re \left\{ I_r V_t \frac{2}{\pi} \sin(\delta/2) e^{j\theta} \right\} \tag{2.12}$$

$$P_r = \frac{V_t I_r}{\pi} \sin(\delta/2) Re \left\{ e^{j\theta} \right\} \tag{2.13}$$

$$P_r = \frac{V_t I_r}{\pi} \sin(\delta/2) \cos(\theta), \tag{2.14}$$

which matches the result found in (2.6).

### 2.2.3 Time-Dependent Analysis

The calculation of the time-averaged power transfer and effective load impedances in are useful for understanding the steady-state operation and driving requirements for the canonical module. However, the operation over time scales much longer than the switching period are also of interest, particularly when a sinusoidal (e.g. grid) voltage is present, or when power transfer requirements vary.

Provided that the rate at which the control variables and circuit parameters change allows them to be considered constant over the switching cycle, then the solution for the time-averaged power in (2.14) can directly augmented to make the expression time dependent

$$P(t) = \frac{V_t(t) I_r(t)}{\pi} \sin(\delta(t)/2) \cos(\theta(t)). \tag{2.15}$$

This same time dependence can be associated with other derivations, such as modeling the time dependent input impedance as

$$Z_r(t) = \frac{V_t(t)}{I_r(t)} \frac{2}{\pi} \sin(\delta(t)/2) e^{j\theta(t)}. \tag{2.16}$$

This simple extension to create time-dependent relationships is due to the use of time-averaged quantities over a switching cycle, eliminating the need for the switching details similar to time-dependent (dynamic) phasor methods [ ].

One specific case of interest for the time-varying parameters is when a sinusoidal voltage source is present and a proportional current is desired, as is the case for single-phase power generation. This situation can be described by

$$V_t(t) = V_t \sin(\omega_l t) \tag{2.17}$$

$$I_t(t) = I_t \sin(\omega_l t) \tag{2.18}$$

$$P_t(t) = V_t \sin(\omega_l t) I_t \sin(\omega_l t), \tag{2.19}$$

where $\omega_l$ is the angular frequency of the waveforms (e.g., the line voltage angular frequency, such as $2\pi 60\,\mathrm{Hz}$). If the above desired terminal power transfer $P_t(t)$ is equated to time-averaged result in (2.15),

$$V_t \sin(\omega_l t) I_t \sin(\omega_l t) = \frac{V_t \sin(\omega_l t) I_r(t)}{\pi} \sin(\delta(t)/2) \cos(\theta(t)) \tag{2.20}$$

$$I_t \sin(\omega_l t) = \frac{I_r(t)}{\pi} \sin(\delta(t)/2) \cos(\theta(t)) \tag{2.21}$$

an equivalency stating that the terminal current in (2.18) can be defined as a function of the resonant current, $I_r(t)$, and the switching function parameters, $\theta(t)$ and $\delta(t)$. From this relationship, it is clear that there exists a minimum resonant current for which the expression remains true, and is at its lowest when the switching function pulse width is half of the period ($\delta = \pi$) and in phase with the resonant current ($\theta = 0$)

$$I_r(t) = \frac{\pi I_t \sin(\omega_l t)}{\sin(\delta(t)/2)\cos(\theta(t))} \tag{2.22}$$

$$I_r(t)_{\min} = \pi I_t \sin(\omega_l t). \tag{2.23}$$

### 2.2.4 Resonant-Current Envelope

The minimum resonant current magnitude defined in (2.23) from the previous section is valid for the single terminal constraint considered. If the power transfer constraints for both the cycloconverter and buffer block are defined as

$$P_C(t) = 2P_{avg}\sin^2(\omega_l t) \tag{2.24}$$

$$P_B(t) = P_{avg}\cos(2\omega_l t), \tag{2.25}$$

respectively, each will have its own time-dependent current required for operation. When both requirements are combined, a minimum current-magnitude envelope can be found that will satisfy both blocks' simultaneously. If the same process is followed which led to the result in (2.23), the resonant current required for each can be defined as

$$I_r(t)_C = \frac{\pi I_C \sin(\omega_l t)}{\sin(\delta_C(t)/2)\cos(\theta_C(t))} \tag{2.26}$$

$$I_r(t)_B = \frac{\pi I_B \cos(2\omega_l t)}{\sin(\delta_B(t)/2)\cos(\theta_B(t))}, \tag{2.27}$$

and the minimum resonant current defined to be

$$I_r(t)_{C,\min} = \pi I_C \sin(\omega_l t) \tag{2.28}$$

$$I_r(t)_{B,\min} = \pi I_B \cos(2\omega_l t), \tag{2.29}$$

where $I_B$ and $I_C$ are magnitudes of the buffer and cycloconverter terminal currents. An example of these constraints applied when $I_B{=}I_C$, can be seen in Fig. 2.5, which plots both blocks' minimum current magnitude, and the resulting envelope over a half-line cycle. Other envelopes magnitudes or shapes can be used, provided they are greater than the minimum. One such envelope to be considered is one with simply a constant magnitude maintained over the line cycle; the value being the maximum value of the minimum-current envelope.

The choice of current envelope has implications on the resulting control parameters for each block. If a phase-shift control is implemented (constant duty-cycle $\delta{=}\pi$), the expressions for the required resonant current in (2.26) and (2.27) can be expressed in terms of their phase shift solutions

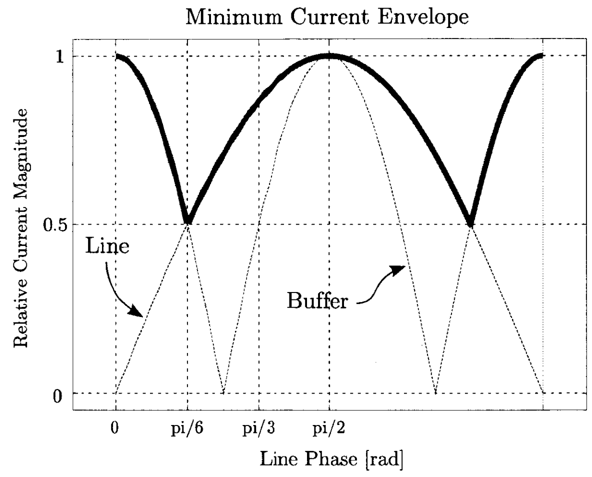


Figure 2.5: The minimum resonant current magnitude requires for the buffer-block and line-connected cycloconverter are illustrated, with the bold line representing the combined minimum-current envelope.

$$\theta_C(t) = \pm\cos^{-1}\left(\frac{\pi I_C \sin(\omega_l t)}{I_r(t)}\right) \tag{2.30}$$

$$\theta_B(t) = \pm\cos^{-1}\left(\frac{\pi I_B \cos(2\omega_l t)}{I_r(t)}\right) \tag{2.31}$$

where each are connected by the same resonant current $I_r(t)$. With a constant current envelope, and $I_B{=}I_C$ (the two clocks $B$ and $C$ in electrical series, and thus having the same current), the resonant current is defined to be $I_r(t){=}\pi I_C$. The solutions for the angles can then be evaluated to be

$$\theta_C(t) = \pm\cos^{-1}\left(\sin(\omega_l t)\right) \tag{2.32}$$

$$\theta_B(t) = \pm\cos^{-1}\left(\cos(2\omega_l t)\right) \tag{2.33}$$

These phase expressions result in two valid solutions for each of the angles. This is a byproduct of the even nature of the cosine function, and allows the choice to be made by an external constraint or preference (in this case, a desire for ZVS switching conditions). Additionally, the solutions are left in an unreduced state, as to avoid the need for multipart expressions.

The solution for the minimum current phase angles must be presented over multiple domains, corresponding to those of the minimum current envelope in Fig. 2.5. Over those domains, the block which defines the minimum current will maintain a constant phase relative to the current, while the other will vary in a non-linear manner. This can be expressed multipart form by

$$\theta_C(t) = \begin{cases} \pm\cos^{-1}\left(\dfrac{\pi I_C \sin(\omega_l t)}{I_B \cos(2\omega_l t)}\right) & \text{if} \quad \omega_l t \in \left\{\left[0, \dfrac{\pi}{6}\right], \left[\dfrac{5\pi}{6}, \pi\right]\right\} \\ \pm\cos^{-1}(1), & \text{if} \quad \omega_l t \in \left[\dfrac{\pi}{6}, \dfrac{5\pi}{6}\right] \end{cases}, \tag{2.34}$$

$$\theta_B(t) = \begin{cases} \pm\cos^{-1}(1), & \text{if} \quad \omega_l t \in \left\{\left[0, \dfrac{\pi}{6}\right], \left[\dfrac{5\pi}{6}, \pi\right]\right\} \\ \pm\cos^{-1}\left(\dfrac{\pi I_B \cos(2\omega_l t)}{I_C \sin(\omega_l t)}\right) & \text{if} \quad \omega_l t \in \left[\dfrac{\pi}{6}, \dfrac{5\pi}{6}\right] \end{cases}. \tag{2.35}$$

where the results of these solutions are plotted over a half-line cycle in Fig. 2.6 for both angles, and both the constant and minimum current envelopes.

Additionally, an evaluation of the differences between the minimum and constant envelope examples is apparent when the load impedance of the buffer-block and cycloconverter combination are considered. The impedance for the constant- and minimum-current profiles are evaluated and shown in Fig. 2.7, using the control variables found in (2.26), (2.33), (2.34), and (2.35).

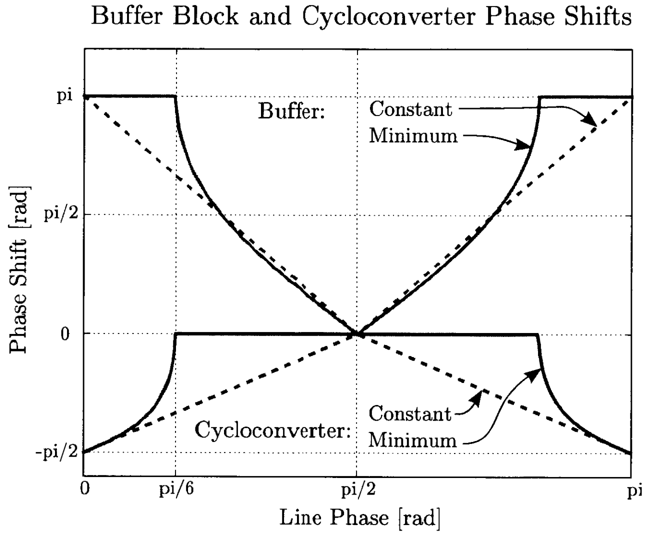
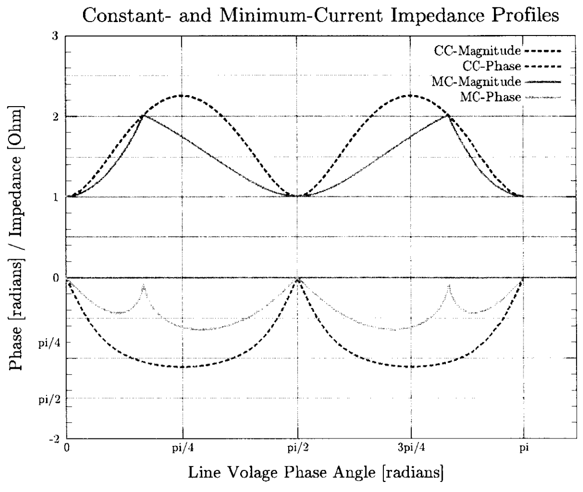


Figure 2.6: the phase relationships for the cycloconverter and buffer blocks when operated in phase-shift modulation.

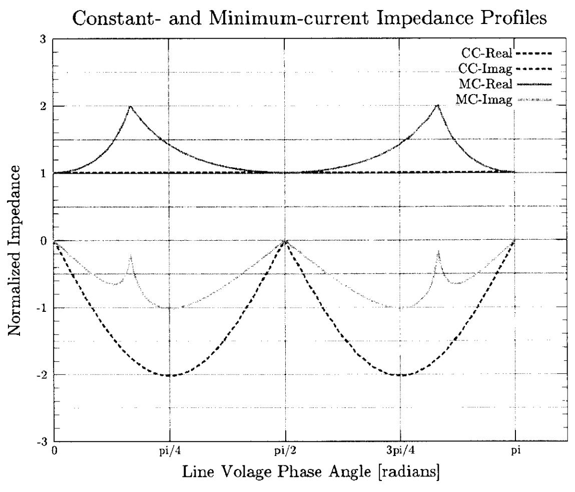
The lower reactive impedance for the minimum current envelope can be understood from the length of time the blocks' phase deviates from the current reference angle (0 or $\pi$). Minimum current always maintains one block in phase, while the constant current method is only in phase at three points over the cycle, increasing reactance. The variation of impedance over a line cycle has a direct effect on the design challenges for the full-bridge inverter and transformation network — large impedance variations reduce the opportunity for optimization, and can ultimately limit the useful operating range of the converter.

### 2.2.5 Transformation Stage Design

The purpose of the transformation stage is to take the effective load presented by they cycloconverter and buffer block and *transform* it to an impedance appropriate for the primary side driving circuit (e.g. a full bridge inverter). For a bridge converter to achieve zero-voltage switching transitions, the load it drives must appear inductive, or equivalently, present a positive reactive impedance. Additionally, the magnitude of the impedance must



(a)

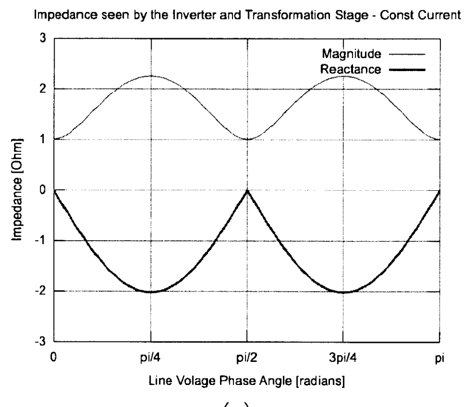


(b)

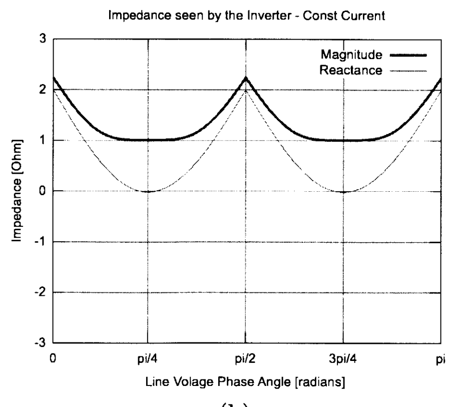
Figure 2.7: The normalized complex impedance of the buffer-block and cycloconverter are shown to vary over a line cycle based on the constant- or minimum-current drive method. Both the (a) magnitude/phase and (b) real/reactive relationships are presented.

be such that the driving circuit can deliver the required power, or synthesize the appropriate waveforms.

The reactance presented by the cycloconverter and buffer-block combination, as illustrated in Fig. 2.8a, is capacitive. To offset this negative reactance, a series inductance is used to compensate, but this also acts to influence the overall impedance magnitude, as seen in Fig. 2.8b. In this case, the peak magnitude remains the same, however the pattern over the line cycle has changed substantially. If the magnitude of the compensated load impedance is not appropriate for the driving circuit, which is likely the case for an input voltage much lower than the output voltage, a transformer with an appropriate winding ratio may be required to match the two circuits.



(a)



(b)

Figure 2.8: The normalized load impedance presented to the full-bridge inverter by (a) the buffer-block, and cycloconverter stages, and (b) including the series-resonant tank. Without the inductance of the resonant tank, the reactance presented to the inverter prevents zero-voltage switching.

## 2.3 Control Parameter Solutions

The development of the control parameters in the previous section found phasor analysis to be an effective modeling tool. This is further developed and applied to model the full converter by approximating each switching block as a complex voltage source, and the transformation stage lumped into a single complex impedance $z_T = R + jX_T$. Fig. 2.9b illustrates the new equivalent circuit of the converter, where each block has been replaced by its phasor equivalent.

If the resonant current is defined in terms of the circuit voltages and tank impedance, then

$$\bar{I} = \frac{1}{Z_T e^{j\theta_Z}} \left( V_A e^{j\theta_A} + V_B e^{j\theta_B} + V_C e^{j\theta_C} \right), \tag{2.36}$$

and the power transfer through the source $k$ is

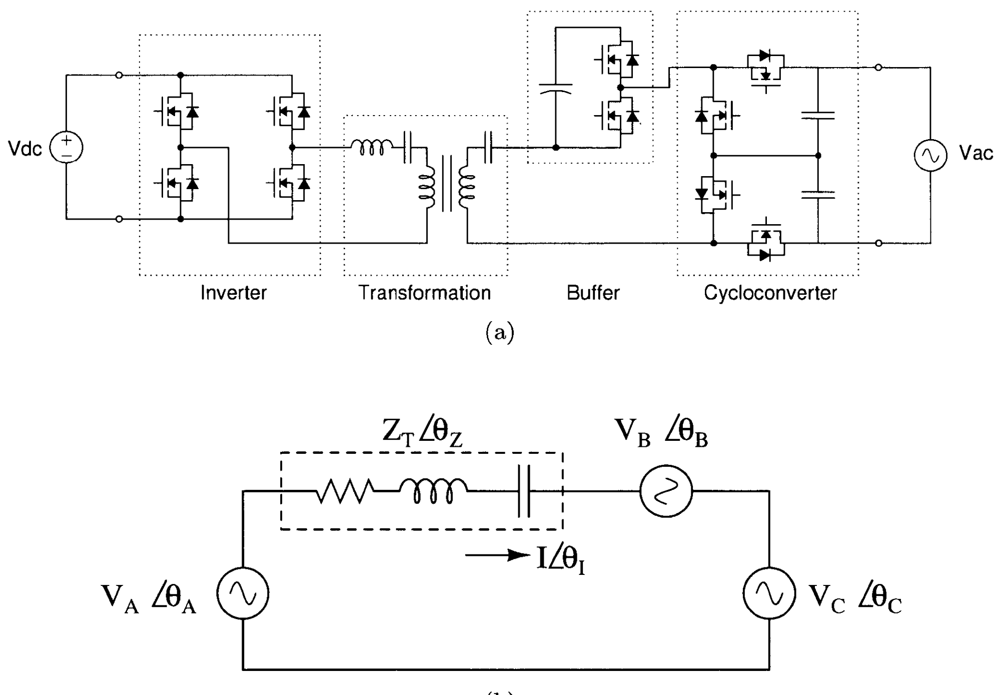


Figure 2.9: The series buffer-block converter schematic in (a) is approximated using phasors (at the switching frequency) in (b). Each switching block is approximated by a sinusoidal source, and a complex impedance in place of the transformation stage.

$$
\begin{aligned}
P_k &= \frac{1}{2} Re\left\{\bar{V}_k \bar{I}^*\right\} \\
P_k &= \frac{1}{2} Re\left\{ V_k e^{j\theta_k} \frac{\left(V_A e^{-j\theta_A} + V_B e^{-j\theta_B} + V_C e^{-j\theta_C}\right)}{\left(Z e^{-j\theta_Z}\right)} \right\}.
\end{aligned}
\tag{2.37}
$$

In this formulation, it is clear that the voltage of each block influences both the magnitude and phase of the current, and this results in a coupled non-linear system of equations for power modulation as well.

The difficulty in applying classical feedback techniques to this type of relationship is greatly increased due to the number of control variables to the system and the lack of independence to the desired outputs. Therefore an open-loop strategy is initially pursued to precompute the control parameters needed obtain desired converter responses. This table of input-output relationships can then be used to compensate the system, which could then permit classical control methods to be implemented.

Calculating the power transfer for a single source, as defined by (2.37), requires seven parameters: the switching frequency and three magnitude/phase pairs for the voltage sources. However, the magnitude of the sources, as defined in (2.9), is itself dependent on two variables: the terminal voltage $V_k$, and the switching pulse width $\delta_k$. This yields a total of ten variables.

The desired outputs of the converter, the power transfer through each source, are defined by

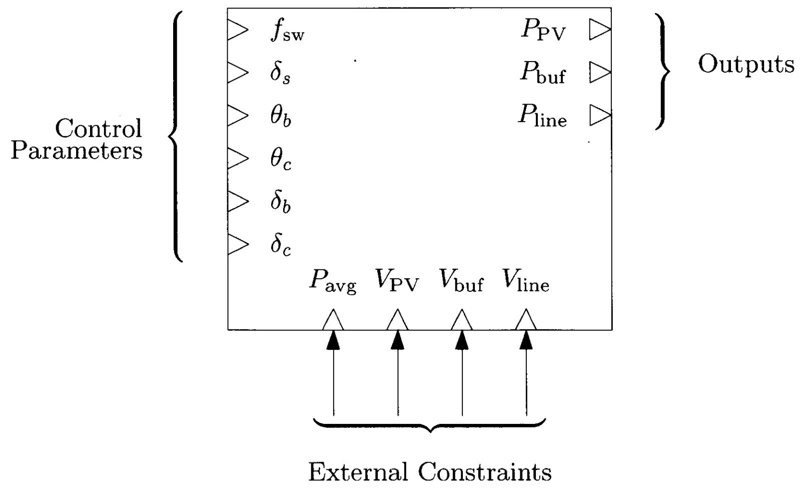


Figure 2.10: A multiple-input multiple-output model for the converter control.

$$P_A = -P_{avg} \tag{2.38}$$

$$P_B = P_{avg}\cos(2\omega_l t) \tag{2.39}$$

$$P_C = P_{avg}(1 - cos(2\omega_l t)), \tag{2.40}$$

which only contain two independent constraints.

The remaining set of eight unconstrained variables can be reduced by defining the terminal voltages as pseudo-static external constraints, and by selecting $\theta_A$ as the reference phase for all angles. The remaining number of unknowns is reduced to six: the switching frequency $f_{sw}$, the two remaining phase shifts $(\theta_B, \theta_C)$, the three duty cycles $(\delta_A, \delta_B, \delta_C)$. The system control block for the converter model is shown in Fig. 2.10, where the control variables, externally applied constraints, and the desired outputs are grouped and enumerated.

Of these six remaining control parameters, only the switching frequency $f_{sw}$ and full-bridge pulse-width $\delta_A$ are unrestricted by the choice of the power modulation methods in

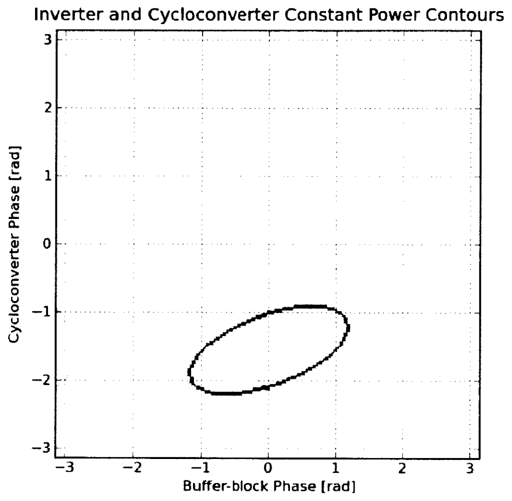


Figure 2.11: Contour plots for the valid solution sets of two power transfer constraints over the $(\theta_B, \theta_C)$ phase space. The intersection of the two contours yields a set of solutions that meet both of these requirements.

Section 2.2. The bounds placed on these variables also have physical implication on the implementation of the high-frequency inverter and transformation stages, and therefore they are chosen to remain independent. As for the four remaining control variables in the cycloconverter and buffer blocks, their use is dependent on the power modulation method chosen. Phase-shift modulation holds the values $\delta_B$ and $\delta_C$ constant, but the pulse-width modulation requires both $\delta$ and $\theta$ of each block if ZVS is to be maintained. For this reason, the phase-shift modulation is selected, leaving $\theta_B$ and $\theta_C$ as dependent variables (e.g. the unknowns), for which solutions are sought.

In determining solutions for the unknown control angles, each power transfer constraint from (2.37) is considered separately. This requires supplying values for the four external constraints, $(V_A, V_B, V_C, P_{\text{avg}})$, two independent control values $(f_{\text{sw}}, \delta_A)$, and desired output powers for the source of interest. To calculate a valid set of phase solutions for each source, a simple brute-force map of the solution-space $(\theta_B, \theta_C)$ is performed, determining the resulting power transfer at each point; the locus of solutions provides a valid set for the single power

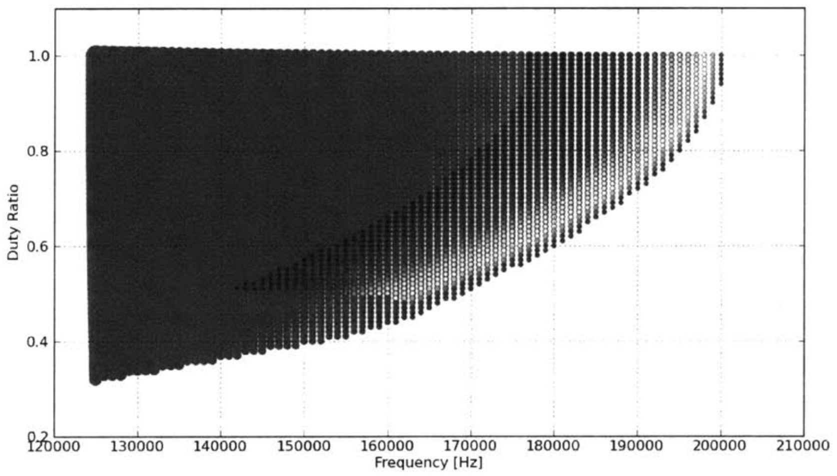


Figure 2.12: A map of valid solutions for varying combinations of switching frequency and full-bridge inverter duty-ratio. The smaller blue points indicate low resonant current magnitude, with larger red dots indicating larger currents. (Calculated for $P_{avg}$=100 W, $V_{buf}$=170 V, $V_{PV}$=32 V, $V_{line}$=0 V, given the converter matching the specifications in Table 3.2.)

transfer constraint. A valid solution is then found for each independent power transfer, and the intersection of these sets provides a new set representing solutions that meet the full set of constraints.

To visualize the valid-set intersections, Fig. 2.11 presents two contours in the phase-space that corresponds to $P_{\text{PV}}$ and $P_{\text{Line}}$ valid sets. In this example, the intersection results in two solutions, although other numbers of solutions may exist for different operating points. If the two sets do not intersect, then there is no solution for the inputs to the system, given the power transfer requirements.

With a procedure in place for finding the unknown phase angles, it is repeated for additional combinations of the independent control variables, $f_{\text{sw}}$ and $\delta_A$, until a map of solutions emerges. An example is shown in Fig. 2.12, which has been limited to include only solutions that provide zero-voltage switching transitions for all switching devices. A

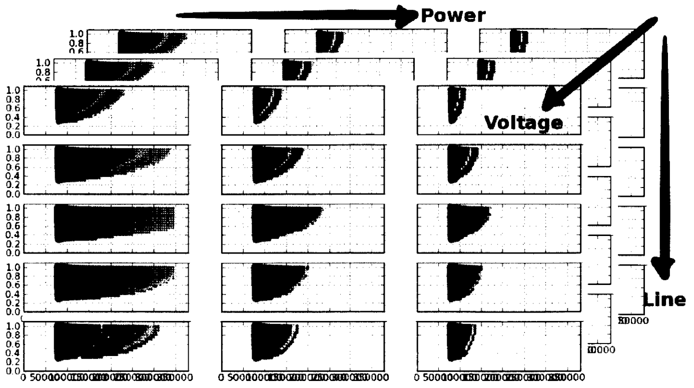


Figure 2.13: Multiple solution maps created to cover the ranges of applied terminal voltages, and the constraints of desired power transfer constraints among the three ports, results in a multidimensional space of valid converter operating parameters.

number of the $(f_{\text{sw}}, \delta_A)$ points contain two valid solutions, each with a different $(\theta_B, \theta_C)$ pair, and consequently different resonant current magnitudes. The large blue dots indicate the lowest resonant current magnitude relative to the small red points which are the largest.

The solution map presented is valid for the single operating condition defined by the applied terminal voltages and power transfer constraints, and therefore the process must be repeated for each operating condition of interest. The additional mappings required for changes in the applied external constraints significantly increases the size of the solution space that needs to be searched, particularly if a fine granularity is desired. Fig. 2.13 illustrates the additional dimensions created by varying $V_{\text{line}}$, $V_{\text{PV}}$, and $P_{\text{avg}}$.

## 2.4 Waveform Prediction

Up to this point, evaluation of the converter has relied on the sinusoidal approximation of waveforms, particularly for the resonant current. While this allows for the system to be described symbolically, it clearly lacks the higher order harmonics that are generated by the square-wave voltage excitations from the individual blocks' switching. These details are critical to evaluating the fitness of a solution, such as for ZVS detection, resonant current harmonic analysis, and even accurate power transfer calculations.

Existing software packages, such as SPICE, piecewise-linear solvers (e.g. NL5, PLECS), and Mathworks' Simulink, are effective at modeling detailed behavior of circuits and systems, however, they must start from an initial condition and then converge to the steady-state solution. The alternative method implemented here is to directly solve for the steady-state operating waveforms, given ideal switches and linear passive components. A finite difference description of the system state equations is constructed, and periodic boundary conditions are applied

To start, a set of differential equations is constructed to describe the state variables of the system. The series connected topology of the circuit requires only a single equation describing the sum of voltages around its loop. This can be expressed in time domain form by

$$0 = v_T(t) + \underbrace{L\frac{di(t)}{dt}}_{v_L(t)} + \underbrace{Ri(t)}_{v_R(t)} + \underbrace{\frac{1}{C}\int_0^t i(t)\,\mathrm{d}t}_{v_C(t)}, \tag{2.41}$$

where $v_T(t)$ is the superposition of the switching waveforms from the full-bridge, buffer, and cycloconverter. This result is rewritten into a fully differential form,

$$0 = \frac{dv_T(t)}{dt} + L\frac{d^2i(t)}{dt^2} + R\frac{di(t)}{dt} + \frac{1}{C}i(t), \tag{2.42}$$

and directly converted to the difference equation

$$\begin{aligned} 0 = & \frac{v_T[n] - v_T[n-1]}{\Delta t} + L\frac{i[n] - 2i[n-1] + i[n-2]}{\Delta t^2} \\ & + R\frac{i[n] - i[n-1]}{\Delta t} + \frac{1}{C}i[n], \end{aligned} \tag{2.43}$$

where $\Delta t$ is defined as the step size for which the difference equation will be solved; equivalently, $\Delta t = T_{\text{sw}}/N$ where $T_{\text{sw}}$ is the switching period and $N$ is the number of samples over the switching period. The periodic boundary condition is implemented such that $i[-k] = i[N-k]$ for $k < N$. Expanding (2.43) for each step $n$ results in a set of equations that can be represented in matrix form as $\mathbf{A}x = b$, and solved as a standard set of linear equations [ ]. The construction of the full $\mathbf{A}$ matrix is illustrated as the sum of its constituent parts

$$\mathbf{A} = \underbrace{\begin{bmatrix} -2 & 1 & & & 1 \\ 1 & -2 & \ddots & & \\ & \ddots & \ddots & 1 & \\ & & 1 & -2 & 1 \\ 1 & & & 1 & -2 \end{bmatrix} \frac{L}{\Delta t}}_{\frac{dv_L(t)}{dt}} + \underbrace{\begin{bmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \\ 1 & & & & -1 \end{bmatrix} \frac{R}{\Delta t}}_{\frac{dv_R(t)}{dt}} + \underbrace{\begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \frac{1}{C\Delta t}}_{\frac{dv_C(t)}{dt}},$$

and the vectors $b$ and $x$ constructed as

$$b = \underbrace{\begin{bmatrix} v[1]-v[0] \\ v[2]-v[1] \\ \vdots \\ v[N-1]-v[N-2] \\ v[0]-v[N-1] \end{bmatrix} \frac{1}{\Delta t}}_{\frac{dv_T(t)}{dt}} \qquad x = \underbrace{\begin{bmatrix} i[0] \\ i[1] \\ \vdots \\ i[N-2] \\ i[N-1] \end{bmatrix}}_{i(t)}.$$

In addition, the system must constrain the resonant current to have a zero dc component, which can be implemented by

$$\sum_{t=0}^{N-1} i[n] = 0. \tag{2.44}$$

This constraint can be enforced by subtracting the mean of the resulting current once a solutions if found, or by modifying the system that has been formulated. To modify the system of equations, the matrix $\mathbf{A}$ and vectors $b$ and $x$ are rewritten in terms of their original construction

$$\mathbf{A} = \begin{bmatrix} & & 1 \\ & [\mathbf{A}] & \vdots \\ & & 1 \\ 1 & \dots \; 1 & 0 \end{bmatrix}, \qquad b = \begin{bmatrix} \\ [v] \\ \\ 0 \end{bmatrix}, \qquad x = \begin{bmatrix} \\ [i] \\ \\ i_{\text{dc}} \end{bmatrix},$$

then the solution for the resonant current can then be found by solving for the vector $x$.

This approach is very efficient for fine time discretizations, as the formulation creates

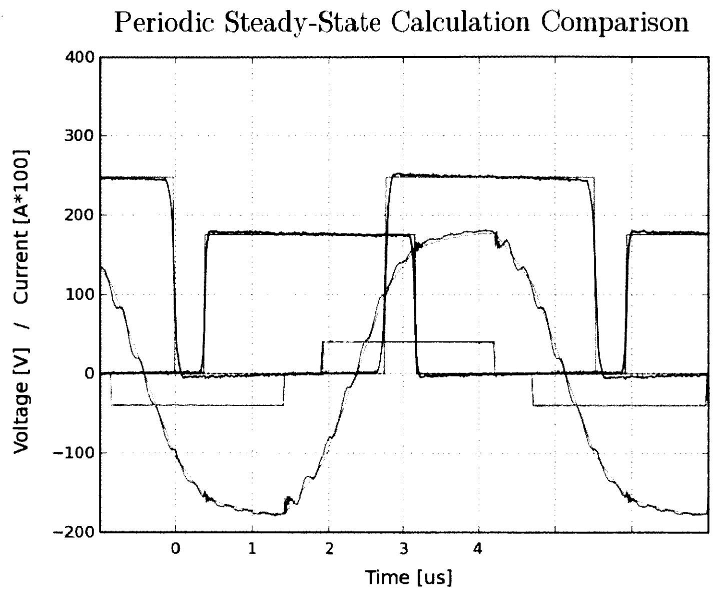


Figure 2.14: A comparison of the calculated finite-difference waveform solution and measured in-circuit waveforms for a converter matching the specifications in Table 3.2 (operating with $P_{avg}$=200 W, $V_{buf}$=170 V, $V_{PV}$=32 V, $V_{line}$=240 V, $\delta_B$=$\delta_C$=$\pi$, $\delta_A$=$4\pi/5$, $f_{sw}$=180 kHz, $\theta_B$=$\pi/2$, $\theta_C$= $-\pi/2$).

banded sparse matrices, which have highly optimized solvers. The performance has been observed to provide a *significant* performance improvement for obtaining the steady-state operation for this converter, while providing sufficiently accurate solutions. An example showing the efficacy of a solution found using this method is shown in Fig. 2.14, where the components of the imposed voltage, $v_T(t)$, and calculated current, $i(t)$, are overlaid on in-circuit measured waveforms. Appendix B contains the code used to implement this finite difference solution method.

## 2.5 Refining Parameter Selection

In the preceding sections, tools to calculate the converter control parameters, and evaluate them accurately in the time-domain, have been presented. It can be inferred from Fig. 2.12, based on resonant current magnitude alone, that most solutions generated are not necessarily desirable. Ideally, this allows an objective function to be written to rank the desirability of each solution (e.g. based on the lowest power loss) for a given set of constraints, with only a small subset that are likely to be of practical interest. Even if the number of solutions for a given map are substantially reduced, they may all be nearly equivalent, and the difference between them below the accuracy of the models used.

The goal is to limit the feasible operating point solutions to reduce the complexity of real-time operation. By selecting a single element from each solution map in Fig. 2.13, the dimensionality of the solution space is greatly reduced, and the relationship of operating conditions and corresponding operating parameters can be investigated in a tractable manner. Even if an objective function can be composed, based on the steady-state converter operation, there are additional factors to consider when evaluating the fitness of the solutions.

### 2.5.1 Path Definition

In the dynamic behavior of the converter, as its external constraints evolve, the *smoothness* of changes between adjacent solutions in the neighboring maps can impact stability and performance. For example, if the external constraints $P_{\text{avg}}$, $V_{\text{PV}}$, and $V_{\text{buf}}$ are fixed, and the only dimension with adjacency is along $V_{\text{line}}$, the solutions along the steps of discretized line voltage should avoid sharp transitions in each of the control variables (e.g. $f_{\text{sw}}$, $\delta_A$, $\theta_B$, $\theta_C$, $\delta_B$, $\delta_C$). A *cost* value can be assigned to each step, where the lowest cost path should be the most preferred.

Fig. 2.15 attempts to illustrate that at each step, a number of possible operating parameter solutions exist, and that a step can be made to any of the solutions in the next step in the sequence. As the number of steps increases, or the number of solutions at each step, the

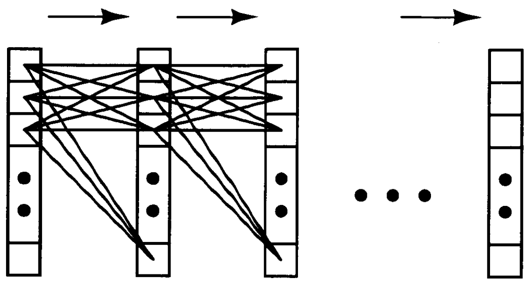


Figure 2.15: The permutations of path selections along the discretized path of an external constraint.

total number of possible paths grows exponentially. To compare the desirability of possible paths, the cost associated with a each step is defined as a measure of the change in each of the operating parameter solutions. These changes are then weighted and combined to form a single cost value for the transition.

To normalize the operating parameter changes, the change in each variable ($f_{\mathrm{sw}}$, $\delta_A$, $\theta_B$, $\theta_C$, $\delta_B$, $\delta_C$) is reduced down to a difference in phase. The greater the change in phase, the greater the step disturbance. These changes can be expressed as

$$k_f = f_{\mathrm{sw}_1}/f_{\mathrm{sw}_2}$$

$$\Delta_{f_{\mathrm{sw}}} = 2\pi(1-k_f) \tag{2.45}$$

$$\Delta_{\delta_A} = 2\pi\left[(1-\delta_{A_1})-(1-\delta_{A_2})k_f\right] \tag{2.46}$$

$$\Delta_{\delta_B} = 2\pi\left[(1-\delta_{B_1})-(1-\delta_{B_2})k_f\right] \tag{2.47}$$

$$\Delta_{\delta_C} = 2\pi\left[(1-\delta_{C_1})-(1-\delta_{C_2})k_f\right] \tag{2.48}$$

$$\Delta_{\theta_B} = \theta_{B_1}-\theta_{B_2}k_f \tag{2.49}$$

$$\Delta_{\theta_C} = \theta_{C_1}-\theta_{C_2}k_f, \tag{2.50}$$

and their differences then combined into a single cost function. A number of cost functions can be imagined, particularly if the variation in one parameter is more or less influential than others. A simple implementation is the 2-norm,

$$\text{Cost} = \sqrt{|\Delta_{f_{\text{sw}}}|^2 + |\Delta_{\delta_A}|^2 + |\Delta_{\delta_B}|^2 + |\Delta_{\delta_C}|^2 + |\Delta_{\theta_B}|^2 + |\Delta_{\theta_C}|^2}, \tag{2.51}$$

which is a reasonably appropriate choice, as this is effectively a vector norm applied across the dimensions of $\Delta$.

### 2.5.2 Path Finding

If the search for the lowest cost path is approached by direct evaluation of each unique path, a total of $N^k$ paths must be evaluated, if there are $k$ steps along the path and $N$ choices at each step. This brute-force method is functional, however extremely intensive for even few steps and choices. A number of formalized algorithms exist that provide elegant methods to evaluate the path-finding problem, most under the guise of network theory [ ].

Two provable algorithms for finding the lowest-cost path in a graph are Dijkstra and A* (A-star). At the heart of the algorithms is the ability to prune sub-graphs from the search space as lower cost paths are discovered. The graph traversal problem posed here is quite simplistic and well defined in scope when compared to traditional applications in vehicular route planning. To evaluate the use of path finding for determining the trajectory through the operating points, the external constraints were held constant except for $V_{line}$, which was broken into 100 equally spaced voltage steps. At each of these steps, a solution map was created, as described in Section 2.3.

The initial graph construction involved selecting the 50 solutions with the lowest resonant

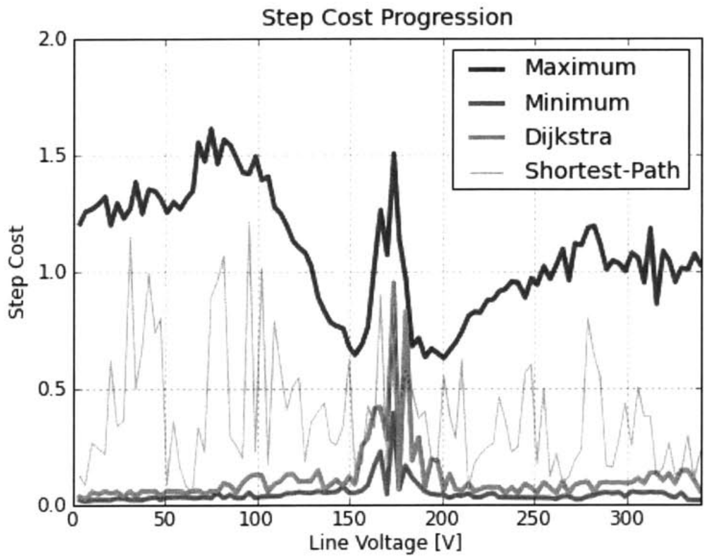


Figure 2.16: The step-costs associated with the path in Fig. 2.17, clearly illustrating the sharp transitions of the associated path, found by Dijkstra, to the increased step costs between 150–200 V.

current magnitude, a set with less than 2% difference at each step. The resulting path-cost through the parameter space, generated by the Dijkstra algorithm, is shown in Fig. 2.16. Unfortunately, the resulting path contains sharp transitions between 150–200 V, which can be seen in the path as overlaid on the parameter space in Fig. 2.16.

To improve upon the generated path, a slightly modified operating point selection method is employed. Instead of limiting the number of solutions per step to a fixed number, additional solutions are added to each step in proportion to the path cost leading into or out of that step. This process is performed iteratively, until either the path cost reaches its minimum or a predetermined level. Figs. 2.19 and 2.18 illustrate the resulting path and its cost progression for the improved solution. This new solution greatly improves the smoothness of the resulting path, however, portions of the path have increased resonant current magnitudes (thus power losses) than the original; incorporating the operating-point solution objective-function into the step-cost function can be used to weigh these trade-offs.

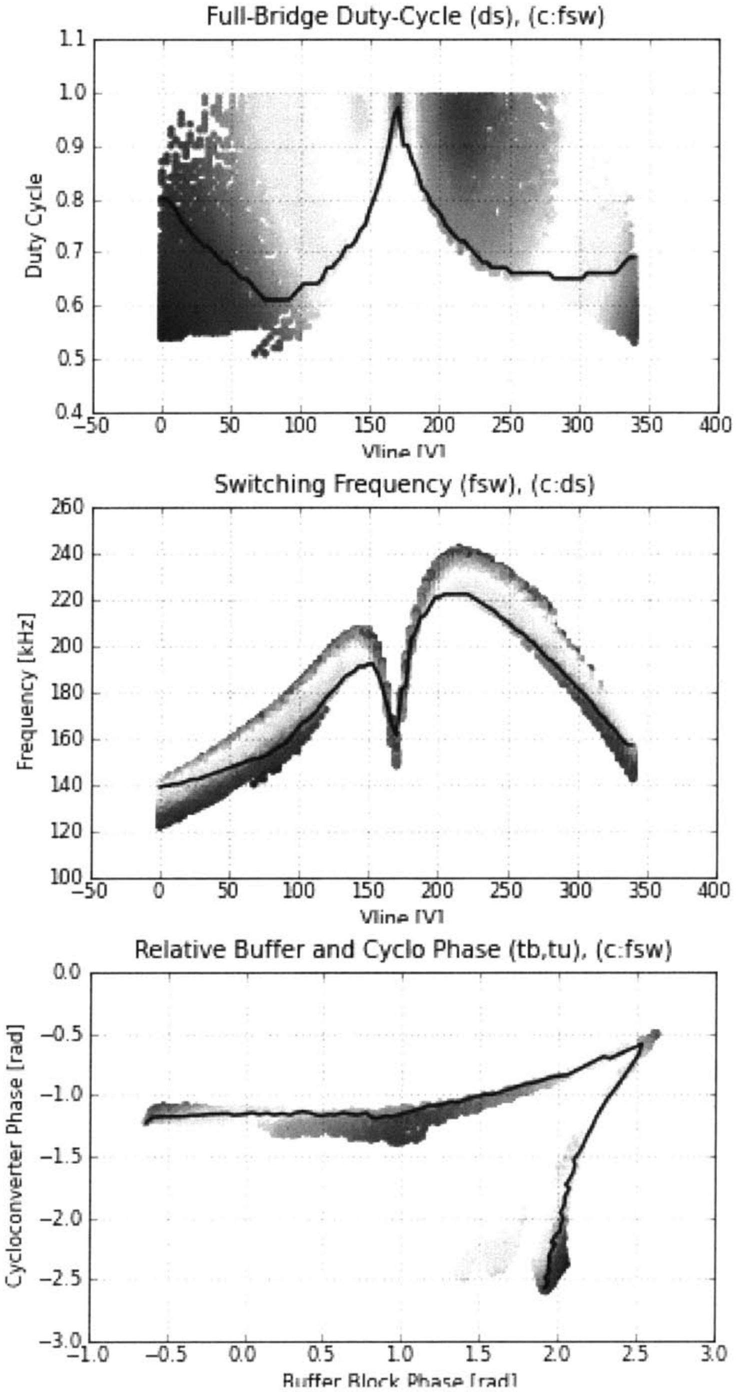


Figure 2.17: The path generated for operation over a line-cycle, with the 50 lowest resonant current magnitude solutions at each point. The path solution exhibits sharp transitions in all variables near $V_{\text{line}} = 170\,\text{V}$.

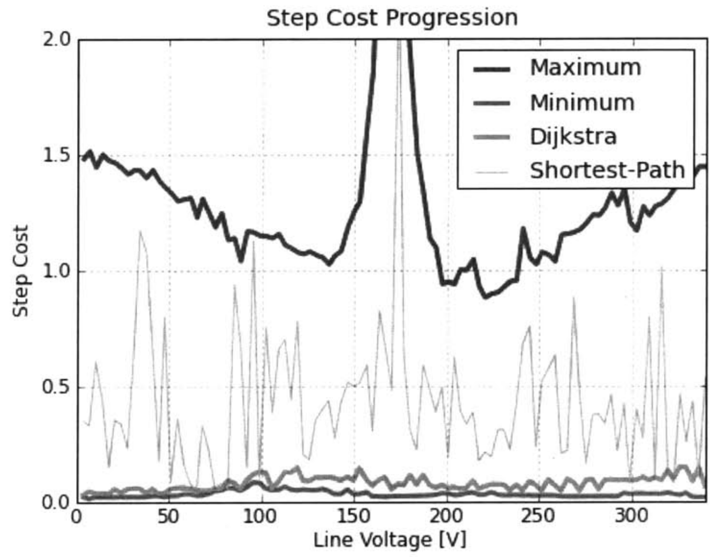


Figure 2.18: The step costs associated with the path in Fig. 2.19, illustrating the uniform and low cost steps found by the Dijkstra algorithm.

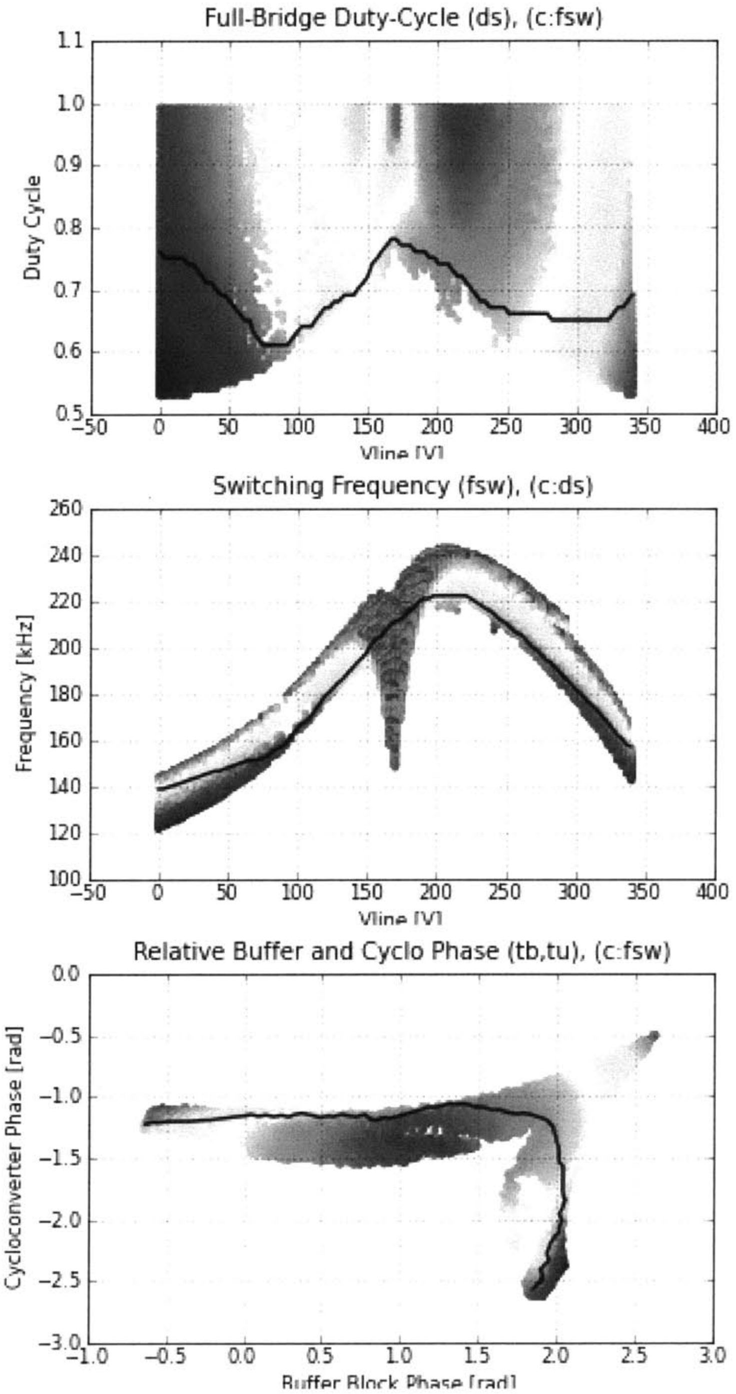


Figure 2.19: The path generated for operation over a line-cycle, with a relaxed resonant current magnitude requirement compared with Fig. 2.17. The path solution eliminates the previous sharp transitions.

*Chapter 3*

# *Prototype Design and Verification*

## 3.1 Introduction

This chapter illustrates the performance and functionality of the series connected buffer-block topology described in this thesis. The control and design methodology presented in Chapter 2 is applied to an operational prototype, and the efficacy of the series buffer block converter is shown.

First, the target design constraints are presented for the prototype, as well as the high-level circuit design and implementation. Secondly, the testing methods employed, for both static and dynamic operation, are described along side the results from their measurements.

## 3.2 Reference Implementation

The target implementation for the prototype platform is for an input from a single 72-cell 200 W photovoltaic module, and an output to a single-phase 240 $V_{ac}$ residential service. The operating requirements for the inverter are outlined in Table 3.1.

The prototype implementation considered in this chapter can be see in Fig. 3.1, the FPGA and microcontroller control boards (not shown) are presented in Appendix C along with their associated code. The PCB is fabricated on FR4 with four 1 oz. copper layers, and

Table 3.1: Prototype converter target requirements.

| Parameter | Value | |
|---|---|---|
| Input Voltage | 25–40 | $V_{DC}$ |
| Output Voltage | 240 ± 10% | $V_{AC}$ |
| Input Power | 0–200 | W |
| Line Frequency | 50–60 | Hz |

has outer dimensions of 7 inches by 4 inches. The PCB artwork and bill-of-materials can be found in Appendix A.

### 3.2.1 Converter Design

The converter's central design follows directly on the circuit topology shown in Fig. 3.2 (originally presented in Chapter 3). Each of the three active blocks contains additional supporting hardware such as gate-drives, communication hardware, power supplies, and protection circuits which are not shown in the figure. The three blocks' gate drive power and digital signals are each independently isolated, then connected externally to a common voltage source and digital-control development board. This provides added flexibility and safety to the circuit while under test. A full list of the converter components, and detailed schematic are located in Appendix A; a abridged listing of the primary circuit components and operating parameters is found in Table 3.2.

The design of the transformer is one key element that benefits from the presence and operation of the buffer-block; the turns ratio and secondary side volt-seconds are both reduced. In a directly comparable topology [ ], implemented without the use of the series buffer, a turns ratio of 1:7.5 (of an ideal minimum 1:6.8) was used, while the implementation here uses a 1:5.0 ratio, with a lower limit below 1:4.8, depending on the switching modulation and current drive method selected.

The transformation stage design closely ties the impedance of the series-resonant tank

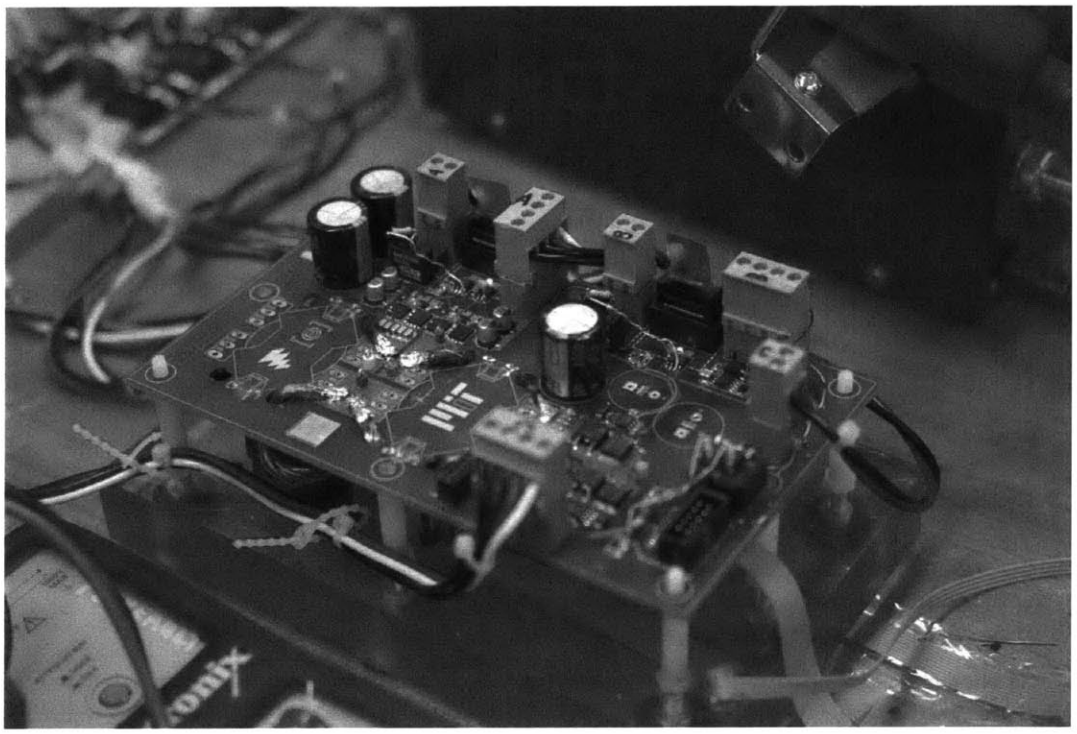


Figure 3.1: Photograph of the proof-of-concept implementation for the series buffer-block topology, not including measurement probes and digital control board.

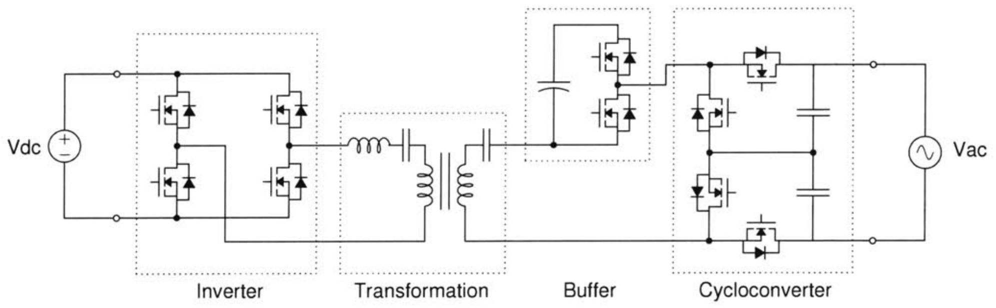


Figure 3.2: Power-stage circuit topology for the prototype evaluated in this chapter.

Table 3.2: Operating range and Component listing for proof-of-concept converter implementation. The resonant component values are listed as the values measured in-circuit.

| Parameter | Value | |
|---|---|---|
| Switching Frequency | 100–500 | kHz |
| Buffer Voltage | 170 | $V_{DC}$ |
| Buffer Capacitance | 141 | $\mu$F |
| Resonant Inductor | 6.44[1] | $\mu$H |
| Resonant Capacitance | 0.60[2] | $\mu$F |
| Transformer | 1:5[3] | |
| Full-bridge MOSFETs[4] | 60 | V |
| | 8.0 | m$\Omega$ |
| Cycloconverter and Buffer-block MOSFETs[5] | 650<br>299 | V<br>m$\Omega$ |

[1] Resonant Inductor — 9 turns, 325 strand 38 AWG litz; RM14-3F3 core, 3.8 mm center-post gap.

[2] Resonant Capacitors — 6 Murata 0.1 $\mu$F 50 V C0G, part number GCM31C5C1H104JA16L.

[3] Transformer — Primary: 5 turns, 300 strand 40 AWG litz; Secondary: 25 turns, 100 strand 40 AWG litz; RM14-3F3 core, ungapped. Leakage Inductance (Primary): 0.288 $\mu$H, Magnetizing Inductance (Primary): 154.5 $\mu$H, Parallel Capacitance (Secondary): 23.9 pF.

[4] NXP part number: PSMN8R5-60YS

[5] STMicroelectronics part number: STD16N65M5

with the transformer turns ratio. The resonant inductor value was selected such that the full-bridge was presented with a low enough impedance to meet the highest power transfer requirement at the lowest input voltage, while also providing enough inductive energy to provide ZVS transitions at low loads. The transformer turns ratio was the selected to ensure voltage matching between the transformation stage and buffer/cycloconverter stages, as is detailed in Section 2.2.5.

The resonant inductance of the circuit includes both the discrete inductor and the leakage inductance of the transformer, totaling 6.76 $\mu$H. The resonant capacitance was selected such that its impedance was less than half of the impedance of inductance at the minimum frequency range (100 kHz). This resulted in a value of 0.6 $\mu$F, and placed the resonant frequency at 79 kHz.

Table 3.3: The power level and weighting values to calculate the CEC efficiency result.

| **Normalized Power** | 1.0 | 0.75 | 0.50 | 0.30 | 0.20 | 0.10 |
|---|---|---|---|---|---|---|
| **Weighting Factor** | 0.05 | 0.53 | 0.21 | 0.12 | 0.05 | 0.04 |

In determining the control parameters for the prototype converter, the direct search method outlined in Section 2.3 was implemented, including the path-finding methods defined in Section 2.5.

## 3.3 Testing Methodology

Efficiency for an inverter can be defined as power-out divided by power-in. The inverter efficiency will vary with ambient temperature, DC input voltage, and the average power delivery. As a result, defining inverter efficiency leaves a large space for interpretation. The California Energy Commission has created a "weighted" inverter test procedure [ ] in an attempt to distill these variations into a single number. This weighted inverter efficiency is known as the CEC inverter efficiency, which the parameters are shown in Table 3.3.

### 3.3.1 Static Measurements

To evaluate specifically defined operating conditions, such as discrete points over the line cycle, requires steady-state dc-dc operation. In this case, all three ports of the converter will be either sourcing or sinking power for the converter, therefore each block (including the buffer) are connected to a dc power supply having a parallel-connected ballast resistance, and series connected common-mode choke as shown in Fig. 3.3.

The three power supplies and associated ballast resistance used for the PV, Line, and Buffer are: KLP-150-16-1.2k with $50\,\Omega$, KLP-300-8-1.2k with $100\,\Omega$, and a KLP-600-4-1.2k with $250\,\Omega$ respectively. The ballast resistance for each power supply is implemented

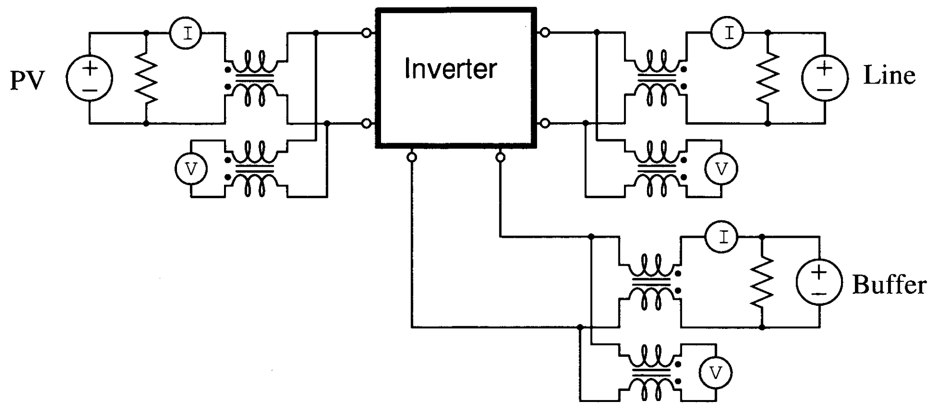


Figure 3.3: Illustration of the static dc-dc measurement setup. Each of the three ports must be supplied and measured, as the buffer-block will either continuously sink or source power at a single operating point.

using three parallel resistors with ratios of 1:2:2 which can be connected in combinations to obtain alternative resistances if necessary. The common-mode chokes are used to reduce the displacement-current flowing to ground through the capacitance of the power supplies and meters. The common-mode inductance used for all three voltage measurements is 60 mH, 30 mH for the Buffer and Line power supplies, and 5 mH for the PV power supply.

Measuring voltage and current for all three converter ports is implemented with a Keithley 2701 DMM, with a Model 7700 multiplexer module. When the converter is operated in steady-state at fixed dc operating points, the meter performs multiple high-resolution sequential readings for the three voltage and three current values, which are then used to calculate the power loss and efficiency results.

### 3.3.2 Dynamic Measurements

In contrast to the step-by-step static measurement method, the dynamic measurement method in Fig. 3.4 operates the converter with a continuous sinusoidal output voltage. Operation with the ac output voltage no longer requires the power supply connected to

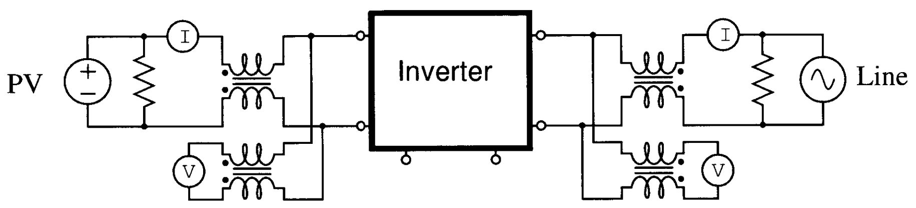


Figure 3.4: Illustration of the dynamic dc-ac measurement setup. Only the input and output ports are supplied and measured, as the buffer-block state-of-charge is managed by the system control.

the Buffer, as operation of the converter under ac conditions ideal results in no net power transfer from the buffer.

The ac-line source comes from an HP/Aglient 6834B three-phase line simulator, of which a single phase is connected. The same ballast resistances, common-mode chokes, and power supply for the PV port as the static measurements in Section 3.3.1 are used. To measure the current and voltages for the dynamic operation, four synchronized Agilent 34410a DMMs sampling at 10 kHz are used. The waveform captures were preformed using a Tektronix 4054B oscilloscope, with P5250 high-voltage differential probes and TCP202 current probes.

## 3.4 Experimental Results

The operating point solutions for the static measurements were selected based on the simple metric of minimizing the resonant current, which is highly correlated with the converter losses. Three captured waveforms for a 32 V, 100 W input are shown in Figs. 3.5, 3.6, and 3.7 for a instantaneous line voltage of 0, 170, and 340 V respectively. These waveform captures clearly show the phase shift of the switching waveforms in each case, and the clean zero-voltage switching transitions for all devices.

The efficiency results for the 5 power levels (the 10% power level is not included) are given in Fig. 3.8 for 25, 32, and 40 V input voltages. The static testing was performed at

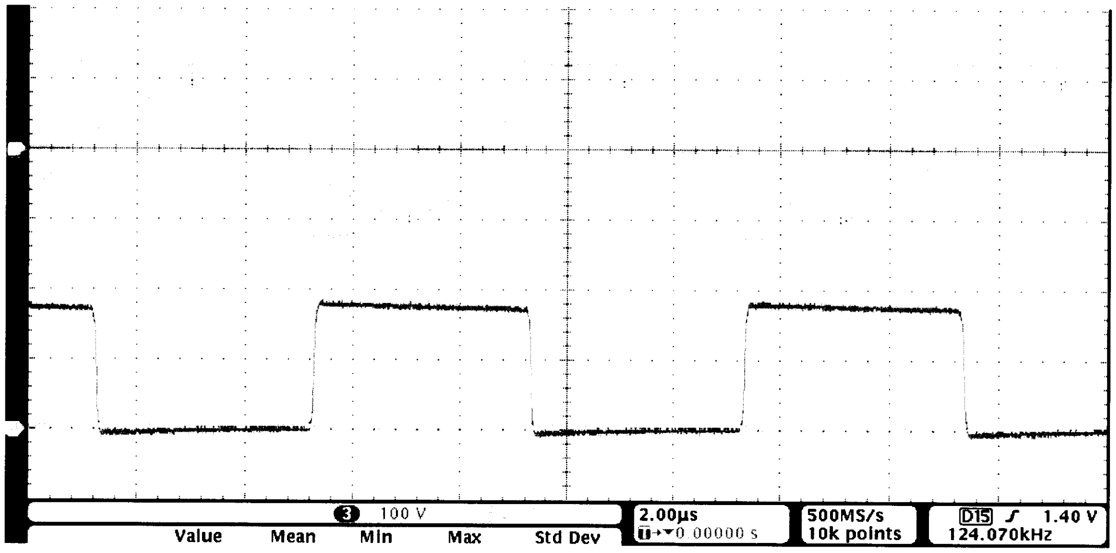


Figure 3.5: Operational prototype waveforms for $P_{avg}$=100 W, $V_{PV}$=32 V, $V_{buf}$=170 V, and $V_{line}$=0 V. CH1: Full Bridge, CH2: Resonant Current, CH3: Buffer-Block, CH4: Cycloconverter.

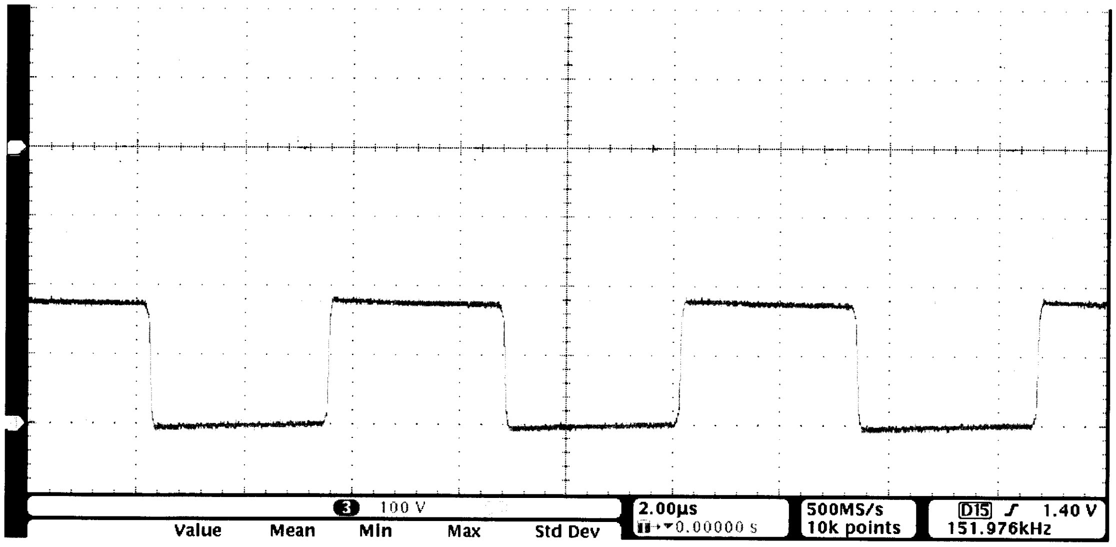


Figure 3.6: Operational prototype waveforms for $P_{avg}$=100 W, $V_{PV}$=32 V, $V_{buf}$=170 V, and $V_{line}$=170 V. CH1: Full Bridge, CH2: Resonant Current, CH3: Buffer-Block, CH4: Cycloconverter.

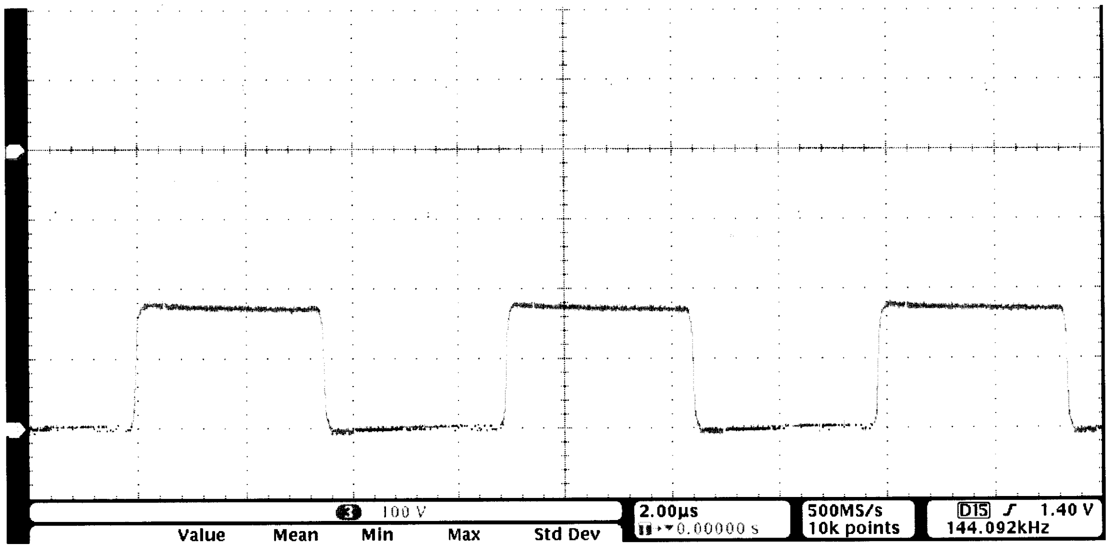
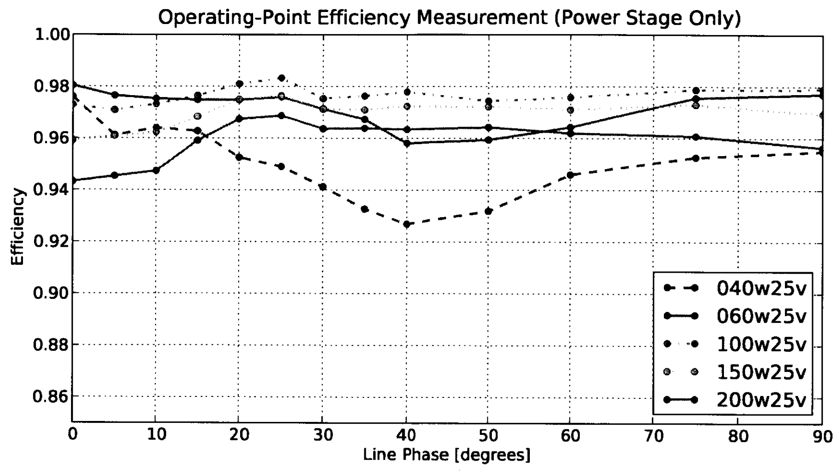
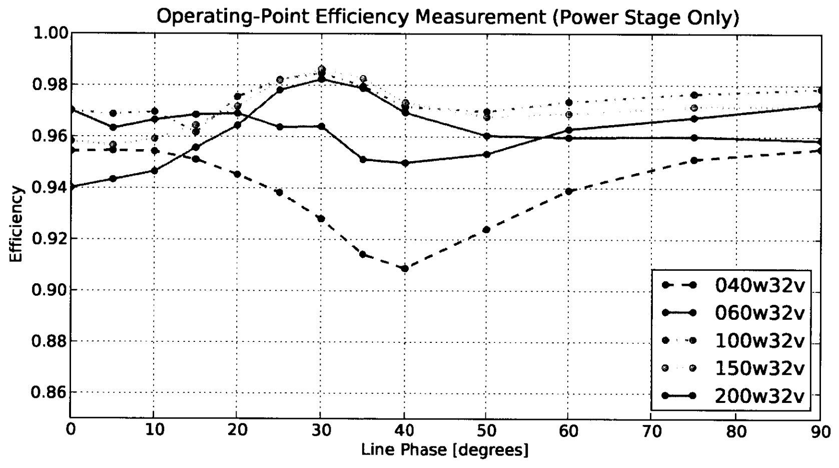


Figure 3.7: Operational prototype waveforms for $P_{avg}$=100 W, $V_{PV}$=32 V, $V_{buf}$=170 V, and $V_{line}$=340 V. CH1: Full Bridge, CH2: Resonant Current, CH3: Buffer-Block, CH4: Cycloconverter.

14 points over a quarter line-cycle. When these results are weighted according to Table 3.3, a CEC efficiency of 96.6% is calculated, minus the impact of gate-drive and digital control losses which were not accounted for. The gate-drive losses are highly frequency dependent, varying from 0.4 W at 100 kHz to 1.8 W at 500 kHz, which impacts the low power levels the most according to Fig. 3.9— weighted over the CEC measurements, this comes out to an approximately 0.75-1.25% additional loss in efficiency. The operating point parameters of the converter for each tested point are shown in Figs. 3.8– 3.11 for $f_{sw}$, $\delta_A$, and $(\theta_B, \theta_C)$ respectively. The frequency variation clearly shows an increase at lower power levels and higher input voltages, whereas the duty cycle and phase-shift parameters

Fig. 3.12 shows the power transfer for the three ports of the converter as plotted over a quarter line-cycle, along with the ideal target curves. The results illustrate the constant power from the PV port, sinusoidal power output to the line, and the bi-directional sinusoidal power transfer in the buffer-block. This verifies, at least for stepped static-operating modes, the proper operation of the topology with the predicted operating point solutions.





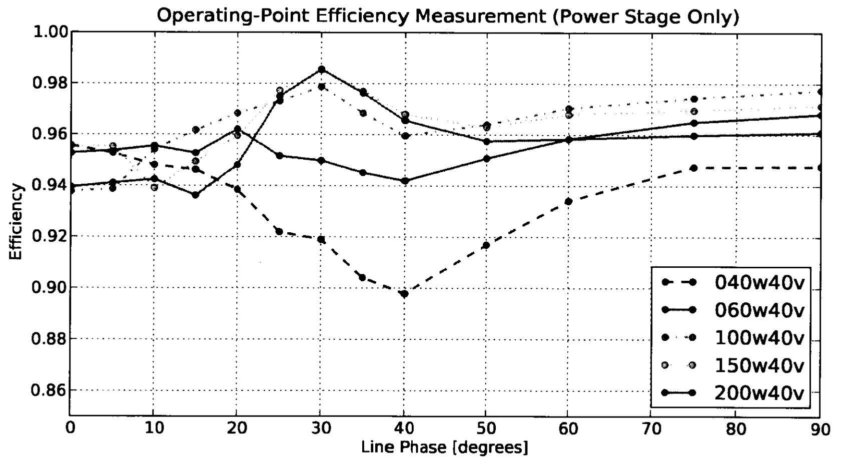


Figure 3.8: Static dc-dc efficiency measurements for input voltages of 25 V (top), 32 V (middle), and 40 V (bottom), for 14 steps over a quarter line-cycle, and five power levels.

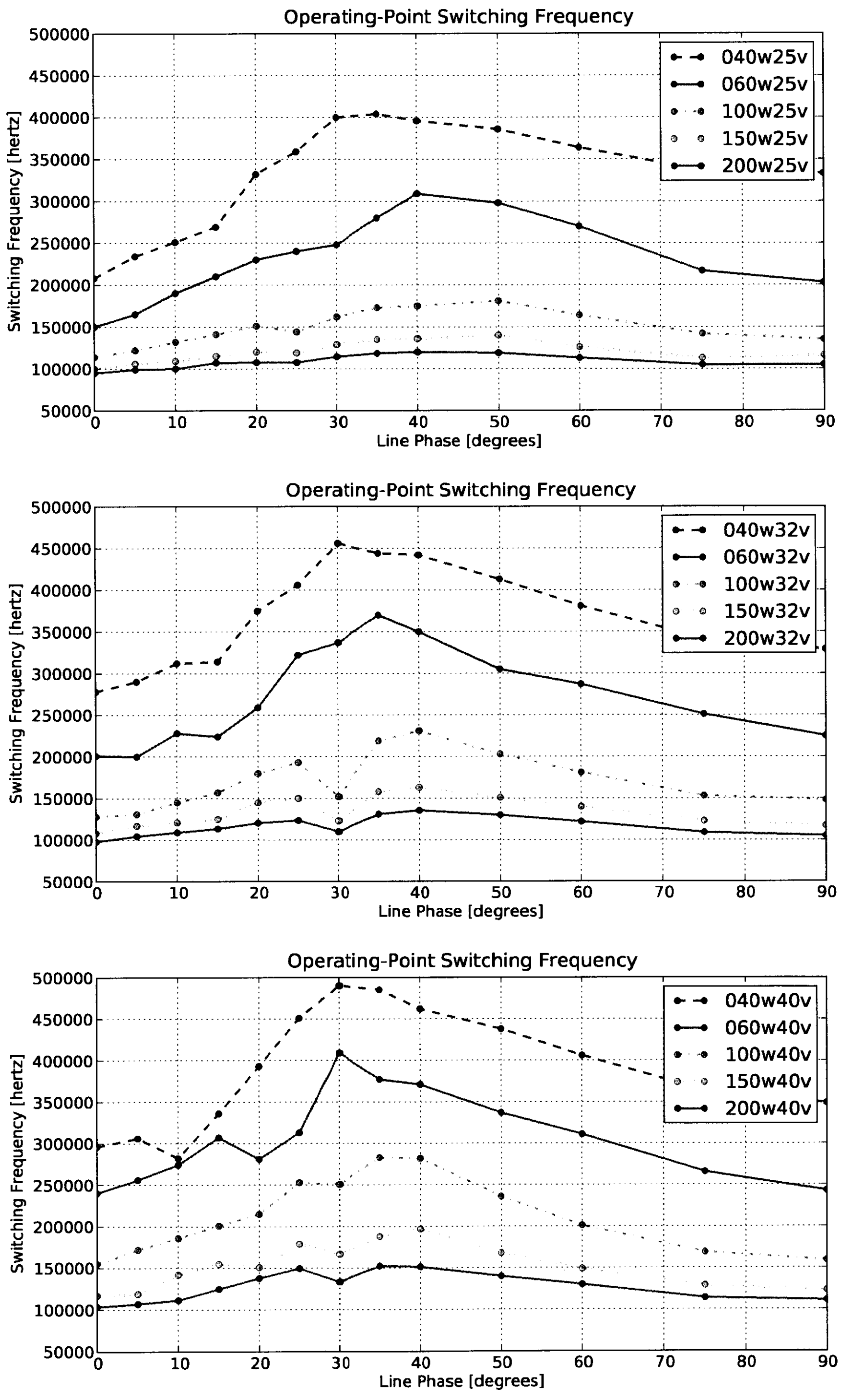
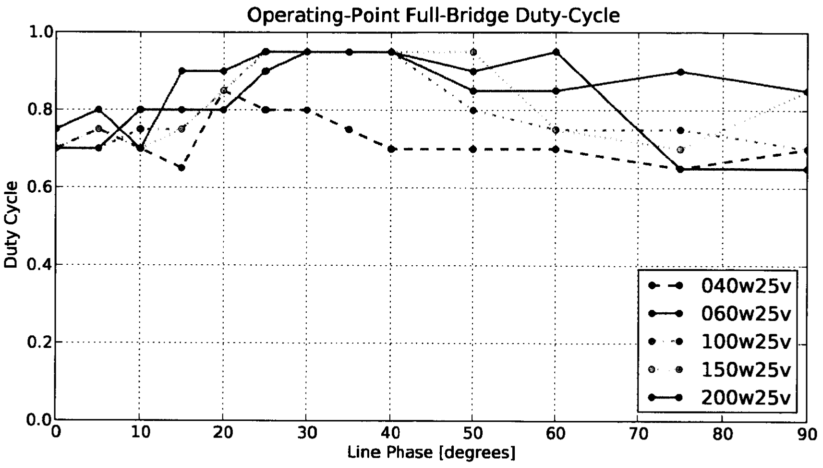
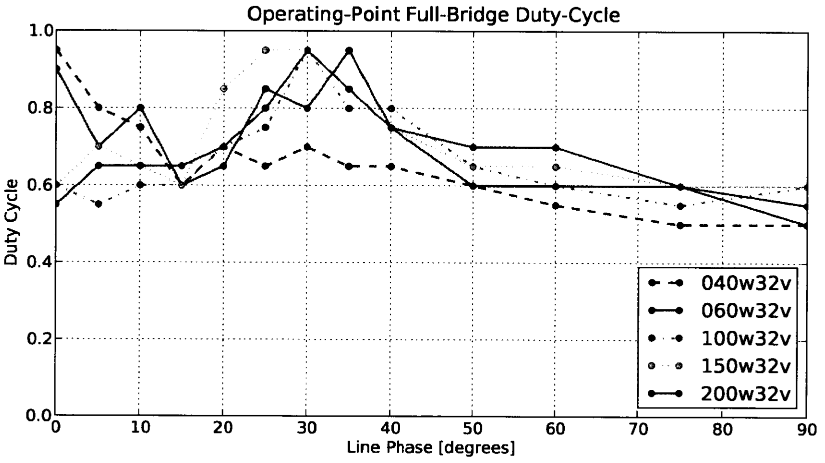


Figure 3.9: Switching frequency for the static dc-dc operating points from Fig. 3.8. The plots correspond to the input voltages of 25 V (top), 32 V (middle), and 40 V (bottom), for 14 steps over a quarter line-cycle, and five power levels.





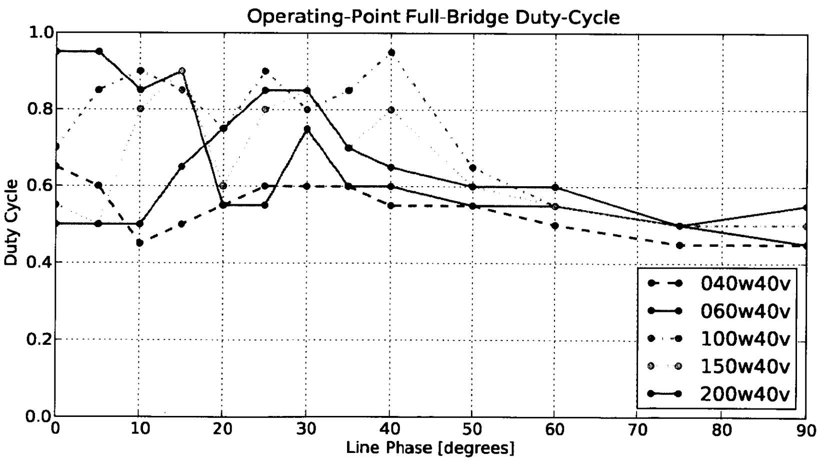


Figure 3.10: Full-bridge inverter duty-cycle for the static dc-dc operating points from Fig. 3.8. The plots correspond to the input voltages of 25 V (top), 32 V (middle), and 40 V (bottom), for 14 steps over a quarter line-cycle, and five power levels.

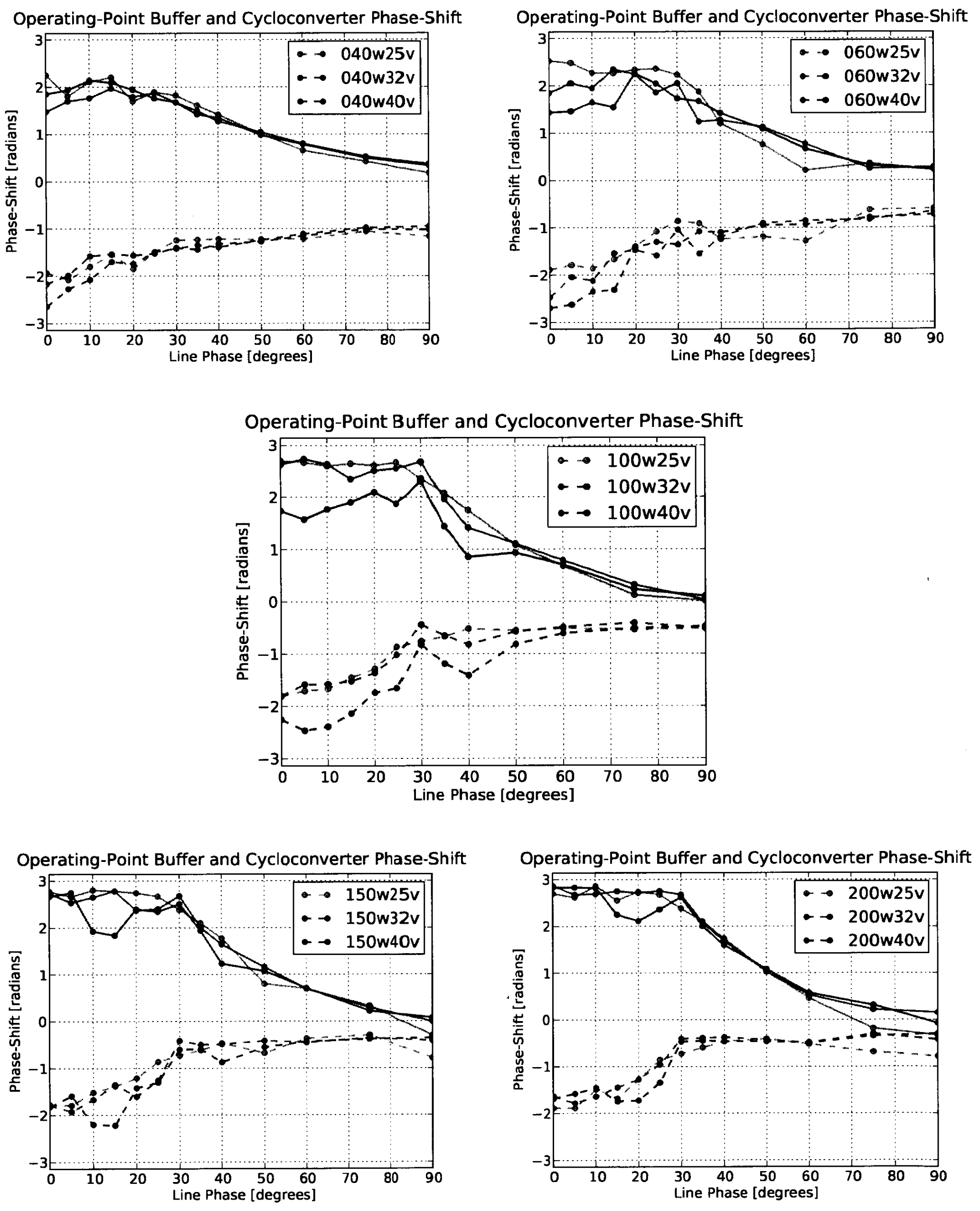


Figure 3.11: Buffer-block and cycloconverter phase-shift values for the static dc-dc operating points from Fig. 3.8. The plots correspond to the average power levels of 40 W, 60 W, 100 W, 150 W, and 200 W, for 14 steps over a quarter line-cycle, and three input voltages. In each plot, the cycloconverter datasets are represented by the dashed line, while the buffer-block datasets are solid lines.

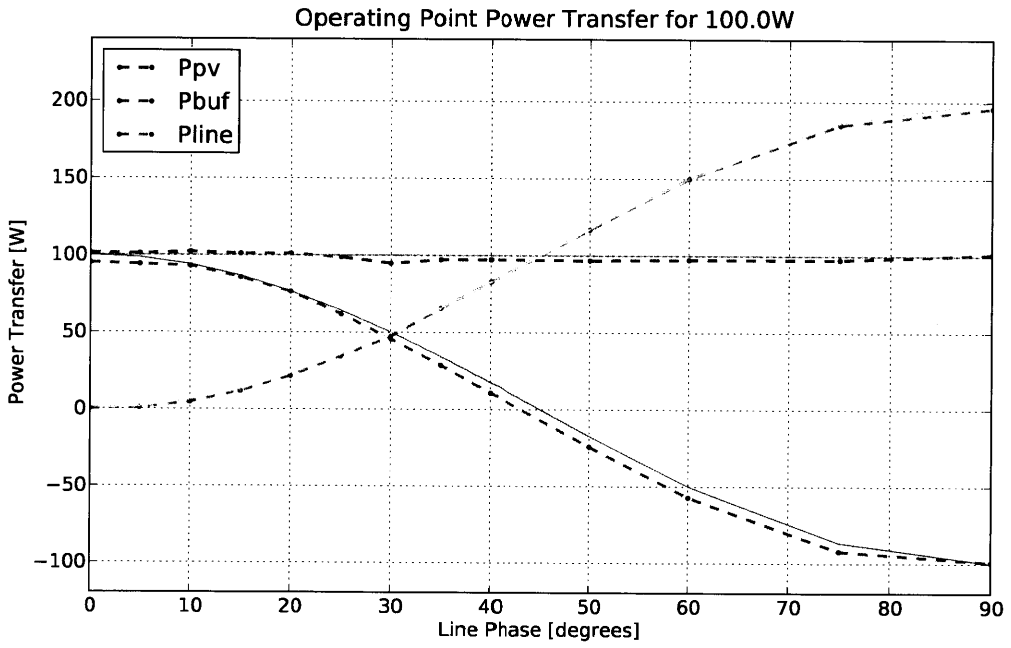


Figure 3.12: Constant input power (blue,cyan) — Sinusoidal output power (red,yellow) — Bi-directional buffer-block power transfer (green,magenta)

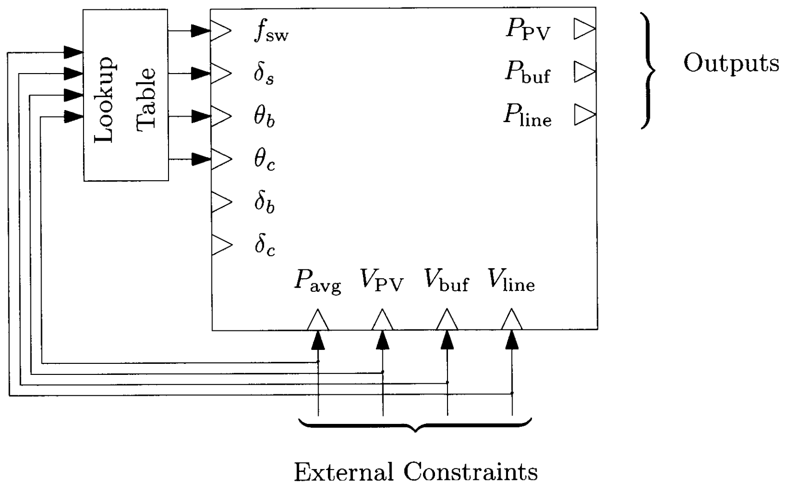


Figure 3.13: A multiple-input multiple-output model for the converter control, with a feed-forward lookup table.

To eliminate the use of the auxiliary power supply attached to the buffer-block, and demonstrate stand-alone dc-ac conversion, the path-finding method described in Section 2.5 is used to populate a lookup table. The terminal parameters are measured in real-time and used to select the appropriate control parameters for the converter from the precomputed solutions, as shown in Fig. 3.13.

The captured switching waveforms of the converter running into an ac line are shown in Fig. 3.14 for 10 cycles. The scope screen-capture is of the full sampled waveform data, of which Fig. 3.15 shows a zoomed area at an ac-line zero-crossing. Both waveform captures show the repeating resonant current envelope, and full bi-polar ac operation, and clean zero-crossing behavior. In this case, the buffer-block capacitance was intentionally undersized to $94\,\mu$F, to verify the ac-power transfer through the buffer by inspecting the buffer voltage.

The voltage and current waveform measurements for the input and output of the converter are shown in Fig. 3.16. The ideal results would be a constant input voltage and current with a sinusoidal output current in phase with the output voltage, however the measured results deviate from this somewhat for two primary reasons.

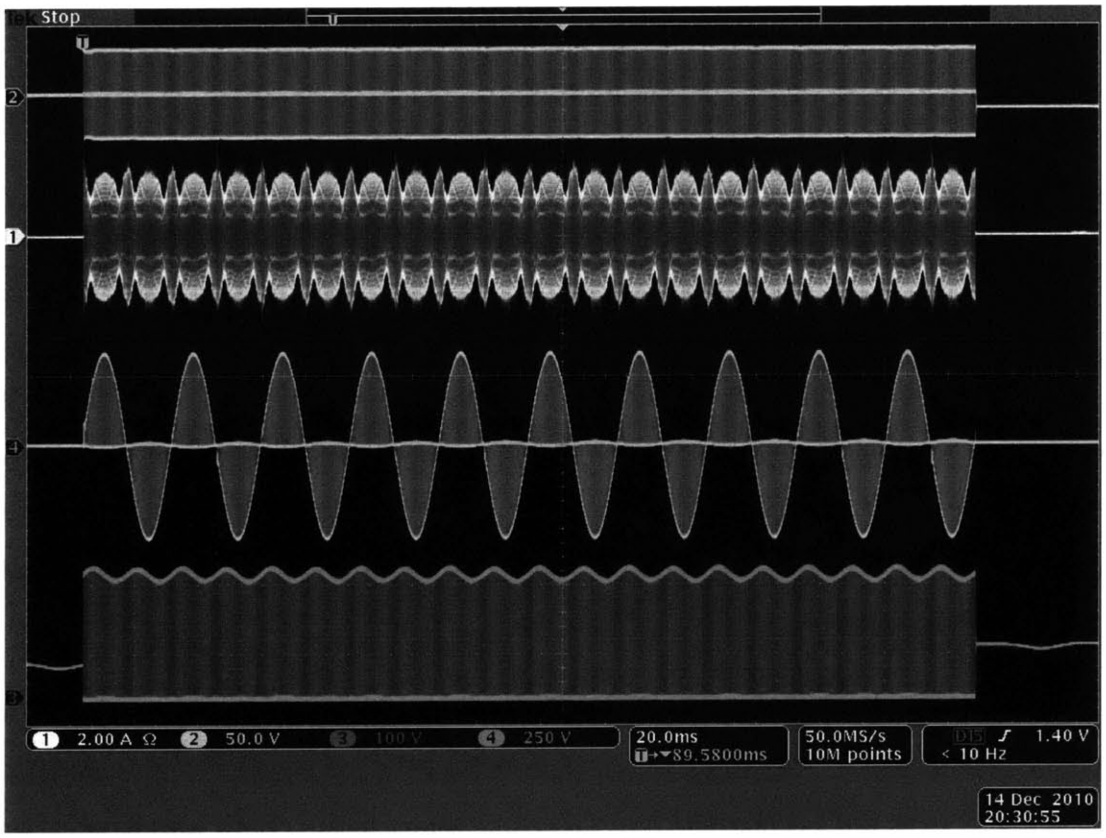


Figure 3.14: Overview of switching waveforms for the series-buffer-block prototype converter running into an ac-line, with $P_{avg}$=100 W, $V_{PV}$=32 V, $V_{buf}\approx$170 V. The waveforms are CH1: Resonant Current, CH2: Full Bridge, CH3: Buffer-Block, CH4: Cycloconverter.

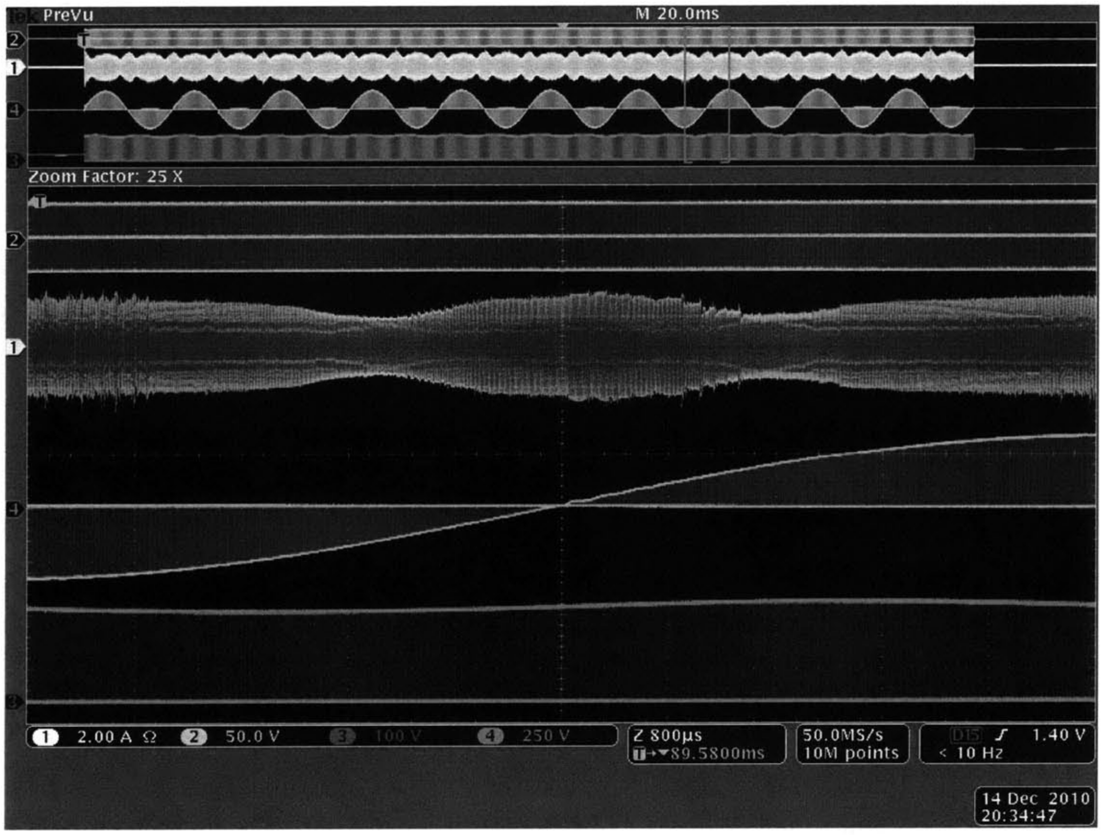


Figure 3.15: Enlargement of a zero-crossing from the waveform in Fig. 3.14, for the series-buffer-block prototype converter running into an ac-line, with $P_{avg}$=100 W, $V_{PV}$=32 V, $V_{buf}\approx$170 V. The waveforms are CH1: Resonant Current, CH2: Full Bridge, CH3: Buffer-Block, CH4: Cycloconverter.

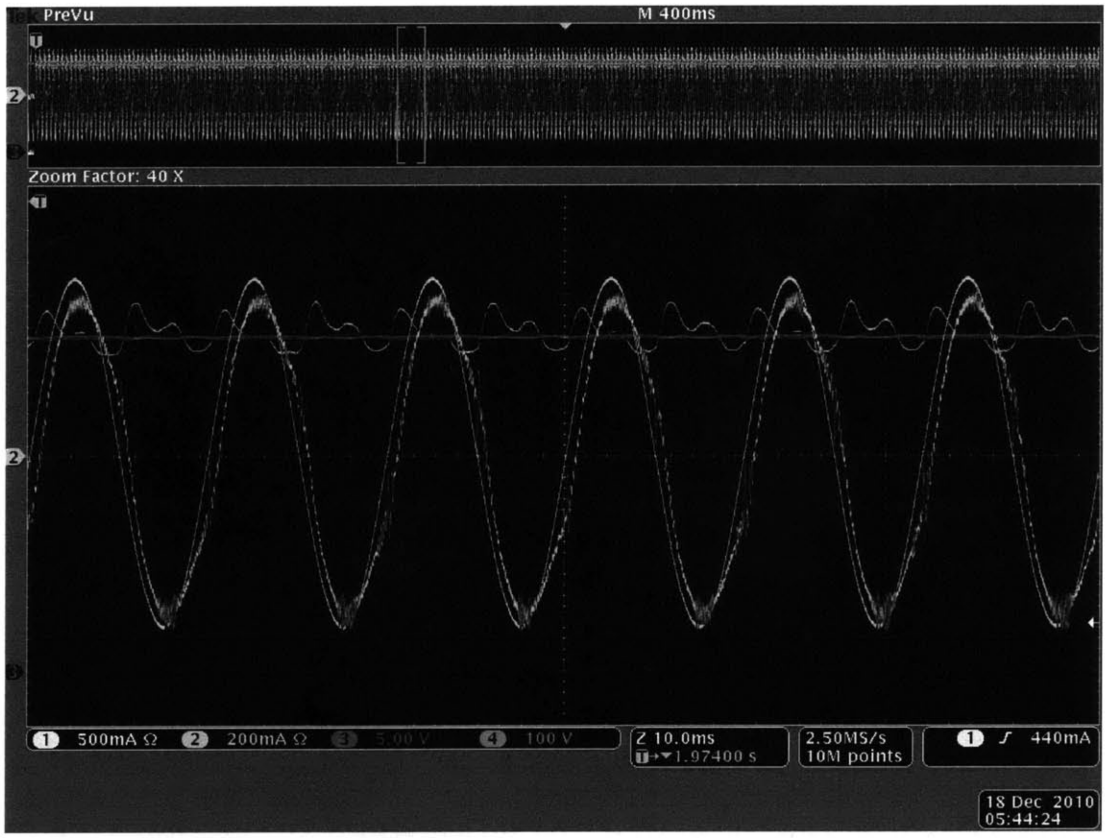


Figure 3.16: The input and output, voltage and current, waveforms for the series-buffer-block prototype converter running into an ac-line, with $P_{avg}$=100 W, $V_{PV}$=32 V, $V_{buf}$≈170 V, $V_{ac}$=240 $V_{rms}$. The waveforms are CH1: Input Current, CH2: Output Current, CH3: Input Voltage, CH4: Output Voltage.

First, the operating point solutions were initially calculated with an assumption of a constant dc voltage on the buffer capacitor, but as seen in Fig. 3.14, a voltage ripple was permitted. This is only a small contributor to the input current ripple, as the variation will *slightly* skew the expected resonant current shape, and thus the current transferred. This contribution could be nearly eliminated with the addition of a larger buffer capacitor.

The second, and more significant impact on the non-ideal behavior, comes from the delay associated with the analog-to-digital measurement process and the update time of the converter operating parameters. The A/D and parameter update latency result in the converter reacting with a non-negligible delay, and delivering a phase shifted current to the line. This phase shift between the current and voltage results in reactive power flow to the line, which is ultimately reflected back as input power ripple. It is worth noting that the precomputed lookup table could be compensated to account for this delay, removing this a source of error.

## 3.5 Conclusion

In this chapter, the operation of series buffer-block topology is clearly demonstrated, as is the efficacy of the converter model and associated parameter solution methods. The CEC efficiency demonstrated for the static measurements exceeds 96.5% for the power stage, and approximately 95.5% overall; the ac-connected test demonstrated 95.3% efficiency, all inclusive, for the input power and voltage tested.

For the results presented in this chapter, it should be noted that the operating point solutions used for the converter testing (both static, and dynamic), were calculated using the converter models, simulation techniques, and path finding algorithms presented in Chapter 2. There was no attempt to tune or hand-select any of the operating parameters, and it should be expected that small refinements to the parameter selection process, or the inclusion of non-ideal components in the converter models, should readily improve the upon these already successful results.

*Chapter 4*

# *Multi-Phase Grid Interface*

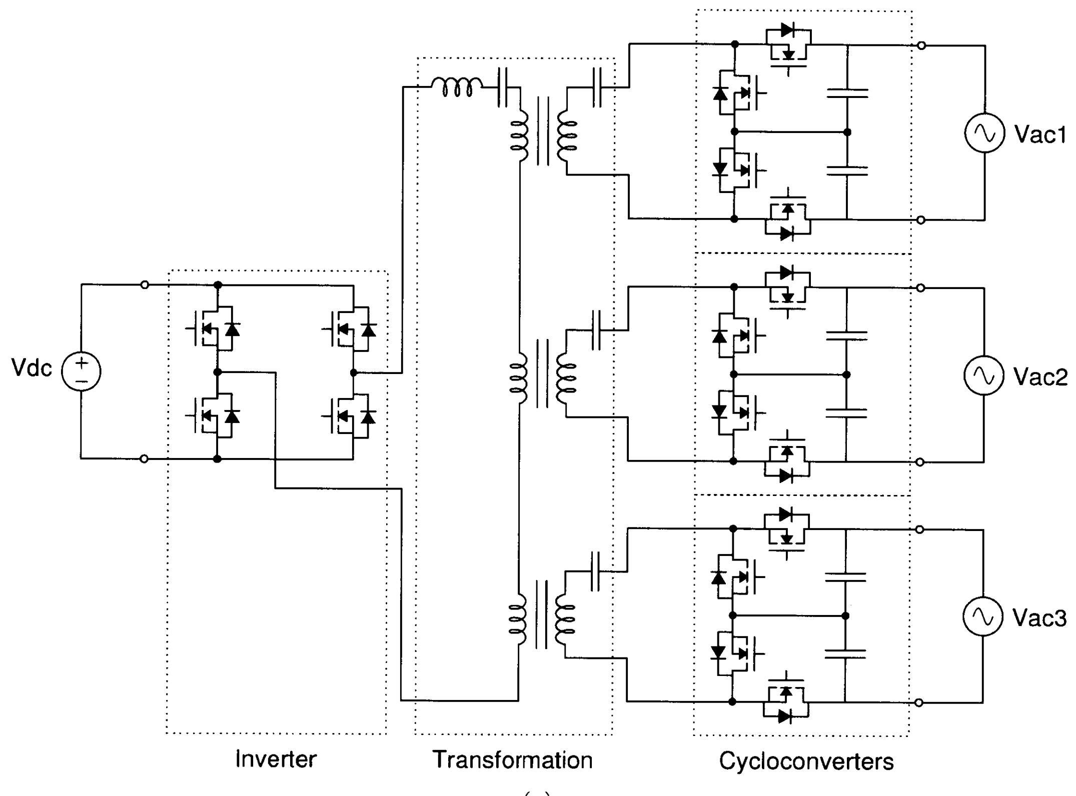
## 4.1 Introduction

Interfacing to a three-phase system bypasses the issues of energy storage in the power converter. Even though each phase individually has a sinusoidal power transfer requirement, the sum of all three phases results in a constant power transfer. This effectively eliminates the need for the large energy storage buffer needed in the converters designed for single phase systems, as outlined in Chapter 1. For this reason, a three phase inverter is a substantial improvement over three single-phase inverters in a system where this configuration can be supported.

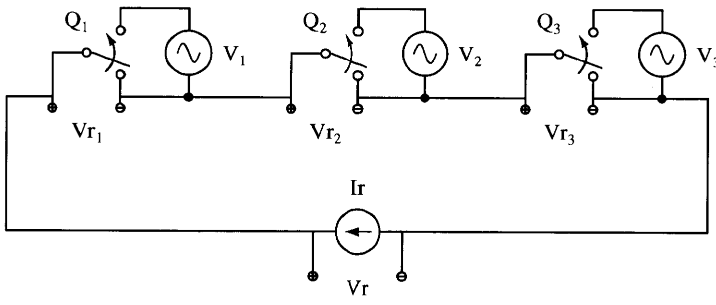
Even though the energy storage buffer can be eliminated, the basic approach of the series-buffer-block topology finds appropriate application. The original design can be extended by the use of additional series connected blocks, with the limitation being the synthesis of the resonant current through the circuit.

## 4.2 Balanced Three-Phase Operation

An ideal switch model of a series-connected three-phase converter output stage is illustrated in Fig. 4.1b. The operation of a balanced three phase voltage distribution system relies on sinusoidal voltages with equal phase distribution over the $2\pi$ interval, giving the terminal



(a)



(b)

Figure 4.1: A three-phase output, series-connected converter illustrated using (a) an example implementation with MOSFET devices, series-resonant tank, and an individual transformer for each output phase; (b) canonical switch models for applying the methods in Chapter 2.

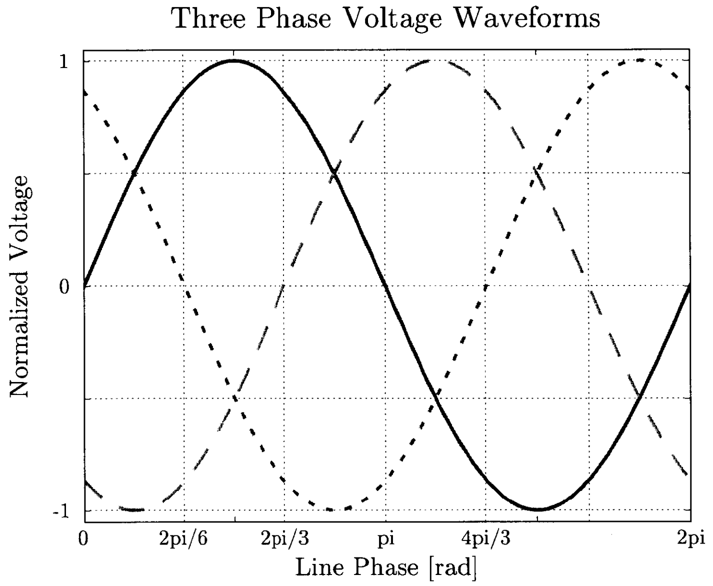


Figure 4.2: Three-phase voltage waveforms, where each sinusoid is separated by $2\pi/3$ radians.

voltage constraints

$$V_i(t) = V_{pk}\sin(\omega_l t + \phi_i) \tag{4.1}$$

for $i \in \{1, 2, 3\}$, where $\phi_1$=0, $\phi_2$=$2\pi/3$, and $\phi_3$=$4\pi/3$ for $V_1$, $V_2$, and $V_3$ respectively. Fig. 4.2 plots these three sources over a line $2\pi$ cycle. For the delivery of power at unity power factor, the local average terminal current requirements over a switching cycle are proportional to the voltages

$$I_i(t) = I_{pk}\sin(\omega_l t + \phi_i) \tag{4.2}$$

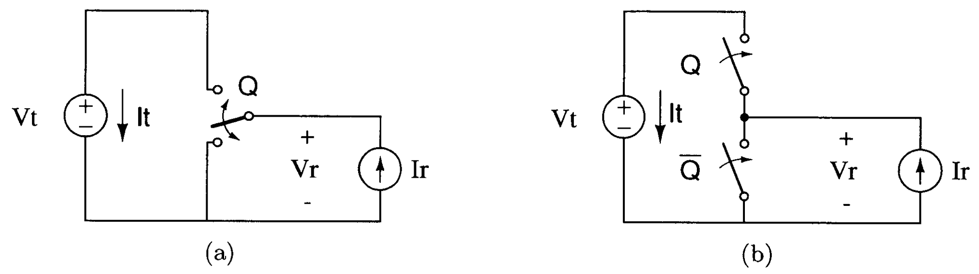


Figure 4.3: The standard switching module implemented with (a) a canonical single-pole-dual-throw switch, and (b) two complimentary single-pole-single-throw switches.

These voltage and current relationships yield the expected sinusoidal power transfer to each source

$$P_i(t) = V_i(t)I_i(t) = P_{pk}\sin^2(\omega_l t + \phi_i) \tag{4.3}$$

## 4.3 Power Modulation

The modulation of power through the blocks of the converter is accomplished by controlling the switching function of each block relative to the series resonant current. The basic circuit models for a single block is presented in Fig. 4.3, and the accompanying switching waveforms are shown in Fig. 4.4.

The power transfer for the circuit block is defined by voltage and current magnitudes, as well as two control variables: the pulse width, $\delta$, and the phase shift, $\theta$. Modulation by use of $\theta$ is called phase-shift modulation, while the use of $\delta$ is referred to as pulse-width modulation (PWM). For the analysis presented here, phase-shift control is implemented, with the assumption that $\delta$ remains a constant.

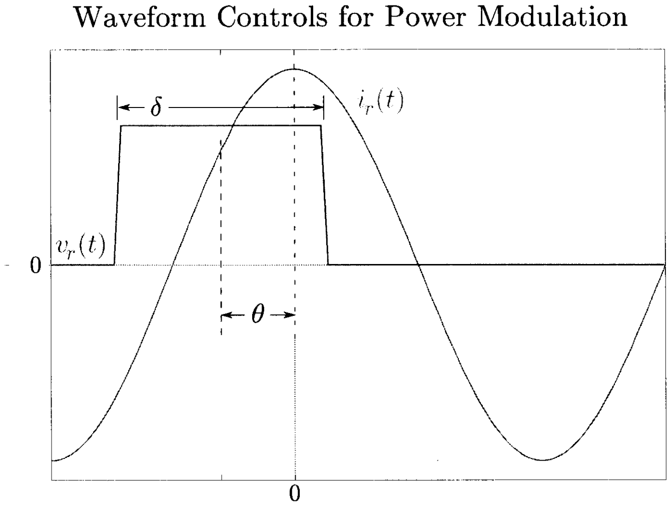


Figure 4.4: The relationship between the series-path current, $i_r(t)$, and voltage switching function, $v_r(t)$, determines the transfer of energy through the converter.

The average transfer of power *from* the current source over a switching cycle can be written as the time-averaged product of the source's applied current and voltage waveforms,

$$P_r = \frac{1}{T}\int_0^T v_r(t) i_r(t) dt \tag{4.4}$$

$$P_r = \frac{V_t I_r}{\pi}\sin(\delta/2)\cos(\theta), \tag{4.5}$$

given the parameters $\delta$ and $\theta$, expressed in radians. This solution supposes that the terminal voltage $V_t$, and resonant current magnitude $I_r$ are constant. However, if these values are permitted to vary over time, but at a rate such that they can be considered constant over a switching cycle, the average power transfer over a switching cycle can be rewritten as

$$P(t) = \frac{V_t(t) I_r(t)}{\pi}\sin(\delta(t)/2)\cos(\theta(t)). \tag{4.6}$$

The expressions for power transfer for a single source in (4.3), and the power transfer for the switching model in (4.6), can be equated to obtain an expression for the control variable

$\theta$, such that

$$V_{pk}\sin(\omega_l t+\phi_i)I_i(t) = \frac{V_{pk}\sin(\omega_l t+\phi_i)I_r(t)}{\pi}\sin(\delta(t)/2)\cos(\theta_i(t)) \tag{4.7}$$

$$I_i(t) = \frac{I_r(t)}{\pi}\sin(\delta(t)/2)\cos(\theta_i(t)) \tag{4.8}$$

$$\pm\cos^{-1}\left(\frac{\pi I_i(t)}{I_r(t)\sin(\delta(t)/2)}\right) = \theta_i(t), \tag{4.9}$$

where the sign of angle determines if the voltage switching waveform leads or lags the current — ultimately resulting in either a hard-switching or soft-switching condition. The desired sign can be derived from the sign of the terminal voltage, $V_i(t)$; zero-voltage switching is obtained for

$$\operatorname{sgn}(\theta_i(t)) = \operatorname{sgn}(-V_i(t)) \tag{4.10}$$

giving the proper solution for $\theta_i(t)$ to be

$$\theta_i(t) = \operatorname{sgn}(-V_i(t))\cos^{-1}\left(\frac{\pi I_i(t)}{I_r(t)\sin(\delta(t)/2)}\right) \tag{4.11}$$

## 4.4 Effective Load Impedance

The waveforms in Fig. 4.4 can also be expressed and analyzed in terms of the phasors (with the implicit $e^{j\omega_l t}$ omitted)

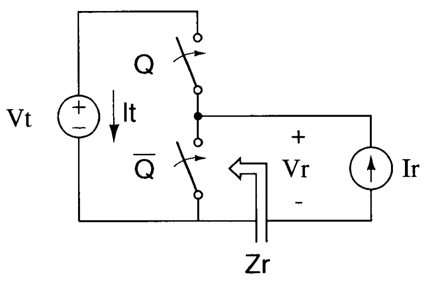


Figure 4.5: The input impedance notation of the canonical switching module.

$$\bar{I}_r = I_r \tag{4.12}$$

$$\bar{V}_r = V_t \frac{2}{\pi} \sin(\delta/2) e^{j\theta}. \tag{4.13}$$

With phasors defined, an equivalent impedance for each block can be generated, as indicated in Fig. 4.5. The impedance driven by the current source, is written simply as

$$Z_r = \frac{\bar{V}_r}{\bar{I}_r} \tag{4.14}$$

$$Z_r = \frac{V_t}{I_r} \frac{2}{\pi} \sin(\delta/2) e^{j\theta}, \tag{4.15}$$

which indicates that the impedance is a variable magnitude complex load, based on the selection of control variables $\delta$ and $\theta$. Additionally, the variables in (4.15) can be extended using time varying phasors to give the time varying impedance

$$Z_r(t) = \frac{V_t(t)}{I_r(t)} \frac{2}{\pi} \sin(\delta(t)/2) e^{j\theta(t)}. \tag{4.16}$$

To be clear, the expressions are "time dependant impedance", which is the impedance at the switching frequency as it varies in time (slowly) over a line cycle. Consequently, this does not represent an inappropriate admixture of time and frequency domains as may be suspected in expressing Z as a function of t. When written in terms of the an unknown line-interface voltage source $V_i(t)$, and the solution for the corresponding $\theta_i(t)$, the result in (4.16) can be expressed as

$$Z_i(t) = \frac{V_i(t)}{I_r(t)}\frac{2}{\pi}\sin(\delta_i(t)/2)\left[\cos(\theta_i(t)) + j\sin(\theta_i(t))\right] \tag{4.17}$$

$$Z_i(t) = \frac{V_i(t)}{I_r(t)}\frac{2}{\pi}\sin(\delta(t)/2)\left[\frac{\pi I_i(t)}{\sin(\delta/2)I_r(t)} - j\,\mathrm{sgn}(V_i(t))\sqrt{1-\left(\frac{\pi I_i(t)}{\sin(\delta/2)I_r(t)}\right)^2}\,\right], \tag{4.18}$$

resulting in an expression for the input impedance for the single block $i$. The effective impedance seen by the resonant current source in Fig. 4.11b, is the sum of the input impedances of the three series-connected blocks.

To find an expression for the sum of the three impedances over a line cycle, the expression must be broken up into multiple domains to account for the non-algebraic nature of the sgn() function in each expression. The sub-domains are delineated by the zero-crossings of the voltage sources. Additionally, the three-phase waveform sum is periodic over $[0, \pi/3]$, so redundant sections can be avoided, leaving this single domain with even symmetry.

The equivalent impedance of the three series-connected blocks is

$$\begin{aligned} Z(t) =& \frac{V_1(t)}{I_r(t)}\frac{2}{\pi}\sin(\delta(t)/2)\left[\frac{\pi I_1(t)}{\sin(\delta/2)I_r(t)} - j\sqrt{1-\left(\frac{\pi I_1(t)}{\sin(\delta/2)I_r(t)}\right)^2}\right] \\ &+ \frac{V_2(t)}{I_r(t)}\frac{2}{\pi}\sin(\delta(t)/2)\left[\frac{\pi I_2(t)}{\sin(\delta/2)I_r(t)} + j\sqrt{1-\left(\frac{\pi I_2(t)}{\sin(\delta/2)I_r(t)}\right)^2}\right] \\ &+ \frac{V_3(t)}{I_r(t)}\frac{2}{\pi}\sin(\delta(t)/2)\left[\frac{\pi I_3(t)}{\sin(\delta/2)I_r(t)} - j\sqrt{1-\left(\frac{\pi I_3(t)}{\sin(\delta/2)I_r(t)}\right)^2}\right], \end{aligned} \tag{4.19}$$

which can be simplified by using only phase-shift modulation ($\delta{=}\pi$), and assuming a constant magnitude for the resonant current envelope, $I_r(t){=}\pi I_{pk}$. This simplification gives

$$\begin{aligned} Z(t) =& \frac{V_{pk}\sin(\omega_l t)}{\pi I_{pk}}\frac{2}{\pi}\left[\sin(\omega_l t) - j\sqrt{1-\sin^2(\omega_l t)}\right] \\ &+ \frac{V_{pk}\sin\left(\omega_l t + \frac{2\pi}{3}\right)}{\pi I_{pk}}\frac{2}{\pi}\left[\sin\left(\omega_l t + \frac{2\pi}{3}\right) + j\sqrt{1-\sin^2\left(\omega_l t + \frac{2\pi}{3}\right)}\right] \\ &+ \frac{V_{pk}\sin\left(\omega_l t + \frac{4\pi}{3}\right)}{\pi I_{pk}}\frac{2}{\pi}\left[\sin\left(\omega_l t + \frac{4\pi}{3}\right) - j\sqrt{1-\sin^2\left(\omega_l t + \frac{4\pi}{3}\right)}\right]. \end{aligned} \tag{4.20}$$

Of greatest interest are the real and imaginary components, which are orthogonal and can be treated independently. First, the real part gives

$$\mathrm{Re}\{Z(t)\} = \frac{V_{pk}}{I_{pk}}\frac{2}{\pi^2}\left[\sin^2(\omega_l t) + \sin^2\left(\omega_l t + \frac{2\pi}{3}\right) + \sin^2\left(\omega_l t + \frac{4\pi}{3}\right)\right] \tag{4.21}$$

$$\mathrm{Re}\{Z(t)\} = \frac{V_{pk}}{I_{pk}}\frac{3}{\pi^2}, \tag{4.22}$$

where the summation terms were reduced through traditional three-phase trigonometric identities [ ]. This constant valued result is notable, although unsurprising, as this is a byproduct of the constant-power aspect of balanced three-phase systems. The investigation of the imaginary part of the impedance in (4.20), begins by applying the substitutions $\cos x = \pm\sqrt{1-\sin^2 x}$, and $\sin(2x) = 2\sin(x)\cos(x)$, which restricts the valid domain of $\omega_l t$ to $[0, \pi/6]$. These substitutions yield the much simpler result of

$$\mathrm{Im}\left\{Z(t)\right\} = \frac{V_{pk}}{I_{pk}}\frac{2}{\pi^2}\left[-\frac{1}{2}\sin(2\omega_l t) + \frac{1}{2}\sin\left(2\omega_l t + 2\frac{2\pi}{3}\right) - \frac{1}{2}\sin\left(2\omega_l t + 2\frac{4\pi}{3}\right)\right], \quad (4.23)$$

which can be factored further (with the same limited domain) to

$$\mathrm{Im}\left\{Z(t)\right\} = -\frac{V_{pk}}{I_{pk}}\frac{2}{\pi^2}\sin\left(2\left[\omega_l t + \frac{\pi}{6}\right]\right), \quad (4.24)$$

which is periodic about $\pi/6$ with even symmetry. The calculated real and reactive impedances of (4.22) and (4.24) are plotted in Fig. 4.6. The ratio of 1.5:1 for real to reactive impedance is very favorable when contrasted with the ratio of 1:1 (down to as low as 0.5:1) for the single-phase converter described in Sections 2.2.4 and 2.2.5.

## 4.5 Simulation Results

The circuit simulation schematic in Fig. 4.7 is constructed to implement the phase-shift modulation solution in (4.11) for the three phase converter. The sub-circuit X1, which provides the switching control signals for each block, is implemented as a "soft" digital controller, described by the following C code:

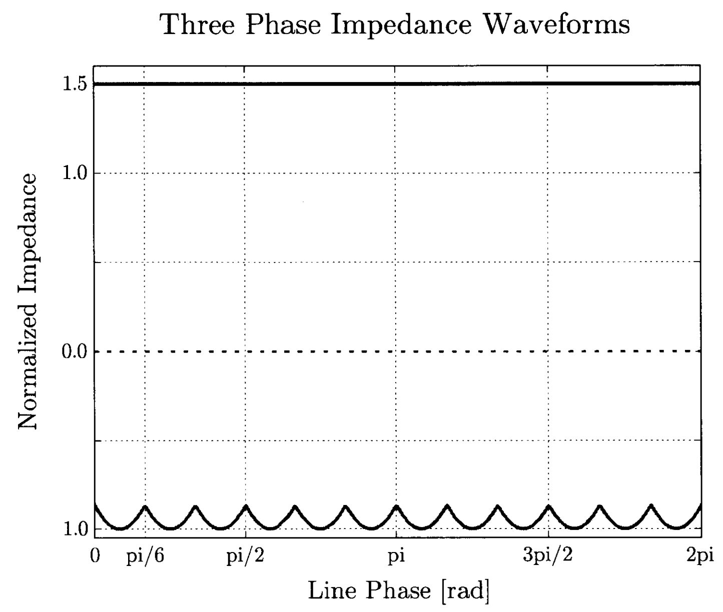


Figure 4.6: Real and reactive impedance waveforms for a three-phase-output series-connected inverter-cycloconverter system.

```
vlogic = 5;

vp1 = sin(wline*t - 0);
vp2 = sin(wline*t - 120);
vp3 = sin(wline*t - 240);

vt1 = (vp1>0) ? acos(vp1) : -acos(vp1);
vt2 = (vp2>0) ? acos(vp2) : -acos(vp2);
vt3 = (vp3>0) ? acos(vp3) : -acos(vp3);

vg1 = ( sin(wsw*t - vt1) )>0 ) ? vlogic : 0;
vg2 = ( sin(wsw*t - vt2) )>0 ) ? vlogic : 0;
vg3 = ( sin(wsw*t - vt3) )>0 ) ? vlogic : 0;
```

The resulting waveforms from the simulation are shown in Fig. 4.8, with the individual traces of the voltages and impedances labeled. Comparing these simulation waveforms with those of the calculated results in Figs. 4.2 and 4.5 indicates that the relative magnitudes and periodicity match, validating the derivation. The sinusoidal power transfer constraint in (4.3) is verified by inspecting a single phase of the three phases in the circuit, also displayed in Fig. 4.8.

## 4.6 Conclusion

This chapter introduced an extension of the series-buffer topology to a balanced three-phase system. The power modulation and system control are derived using the same methodology as the single phase system in Chapter 2, and confirmed through the use of a simulation model.

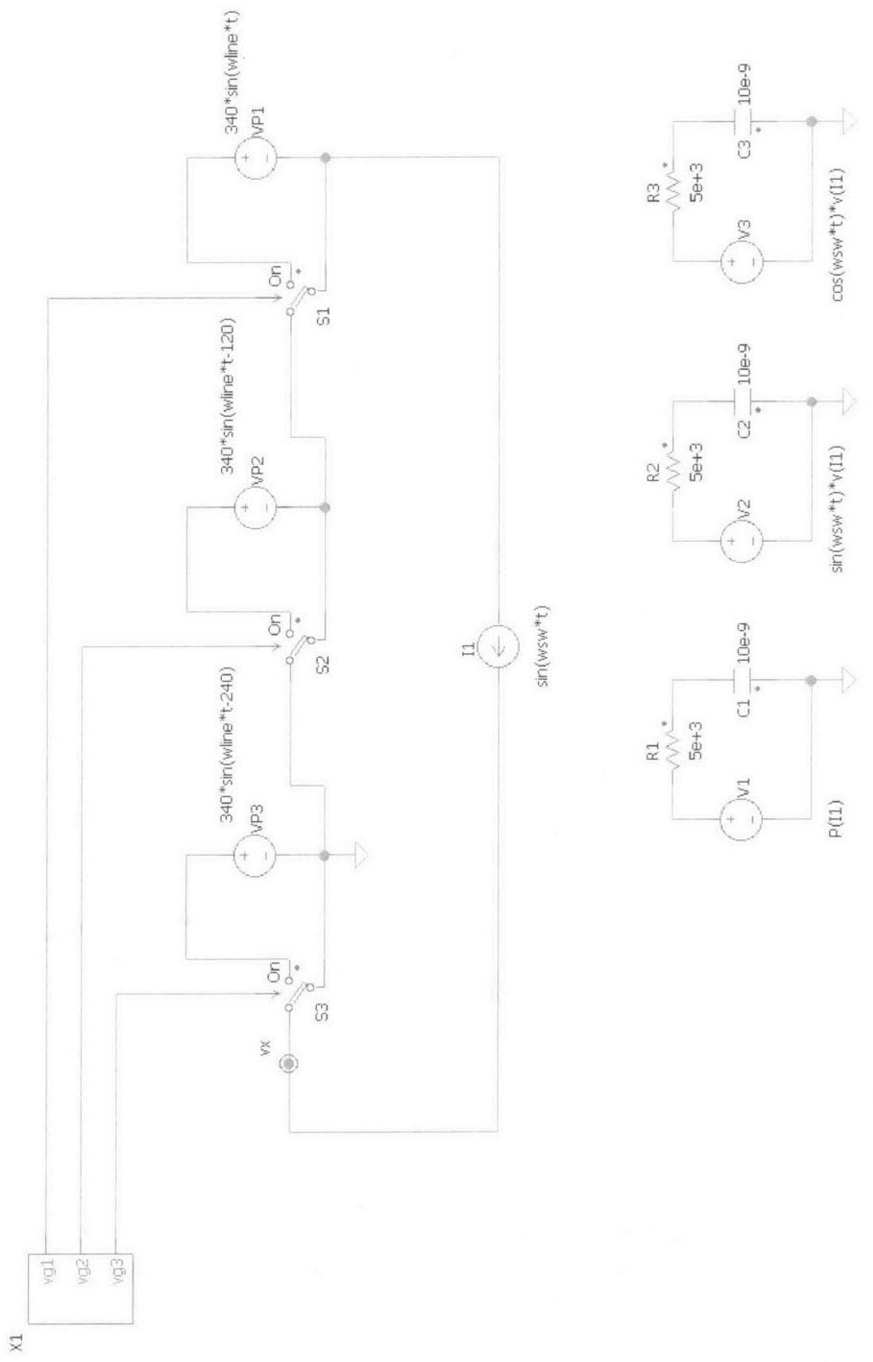
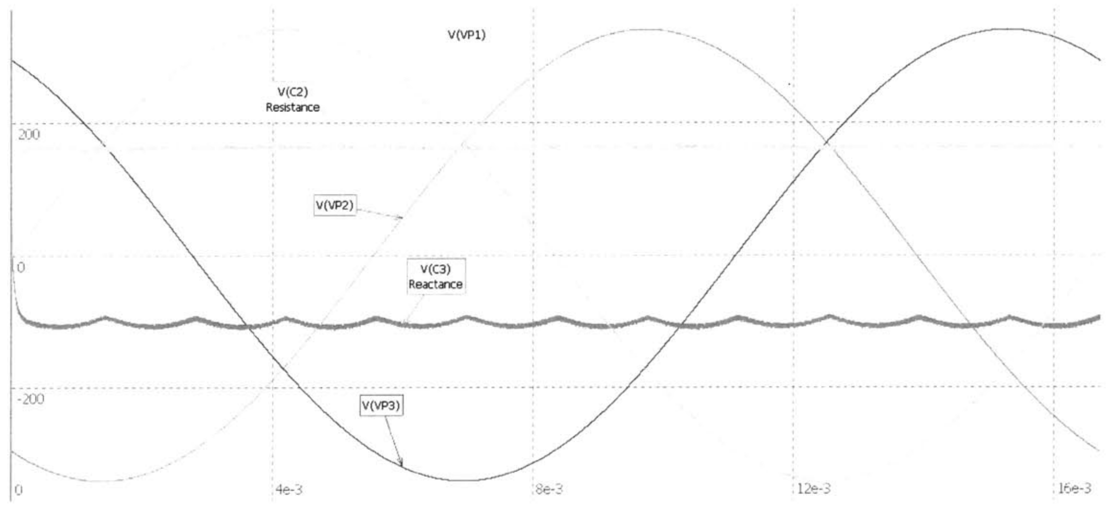
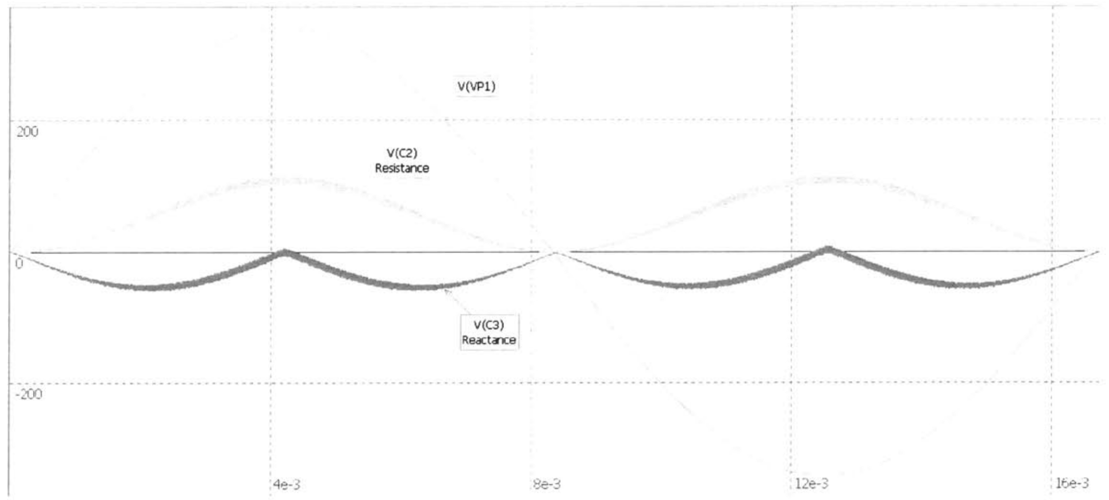


Figure 4.7: Schematic of the circuit used to simulate the three-phase load impedance.



(a)



(b)

Figure 4.8: The resulting voltage and impedance waveforms for the circuit simulation of Fig. 4.7, for (a) all three phases and (b) a single phase of the circuit.

*Chapter 5*

# *Thesis Conclusions*

## 5.1 Summary

The primary objective of this thesis was to develop a grid-tied power converter, with the goal of improving present-day efficiency, and provide a path for future improvements. The dc-ac inverter application was chosen due to the continued interest and rapid development in photovoltaic systems. In particular, the emergence of micro-inverters in the commercial space has illustrated the demand for high-efficiency and reliability of power converters in a small form factor.

The content of this thesis has provided tools and techniques for the development of a single-phase ultra-high efficiency grid-connected power converter, which

- provides independent control of the power buffering, allowing a reduced energy storage requirement,
- is stable and controllable over wide input and output voltages and output power levels,
- has bi-directional power transfer capability, including reactive power transfer,
- is scalable with future semiconductor device technology, voltage levels, and power requirements.

## 5.2 Future Work

There are a number of areas in this thesis where additional work and refinement can be identified, however there are four areas which I would have enjoyed spending additional time.

1. Evaluation of an alternative implementation of the high-frequency inverter and transformation stages, which may reduce the large operating frequency range or improve efficiency.

2. Further operational investigation into the reactive power compensation ability of the converter, particularly in regard to the robustness of the grid zero-crossing behavior and output distortion characteristics.

3. Parametric modeling of the path-finding solutions of the predictive control, particularly as the inclusion of additional dimensions precludes the use of a single, large, lookup-table.

4. Development of closed-loop control strategy for the converter, which is tolerant of measurement inaccuracies and component variations.

It is my hope that some of these areas may be approached at some point, either as an extension of the work already completed, or independently (perhaps unknowingly) by others in the future.

*Appendix A*

# *Converter Schematics, Bill-of-Materials, and PCB Artwork*

## A.1 Bill of Materials

| **Identifier** | **Description** |
|---|---|
| C13, C20, C22, C36, C37 | X7R Ceramic, 0603 |
| C27, C28, C32, C77, C98 | C-POL, E7,5-18 |
| C17, C23, C26, C41, C95, C116, C117, C118 | Panasonic D |
| C64 | C225-108X168-SMD |
| C29, C42, C43, C44, C45, C46, C47, C54, C66, C100, C101, C104, C105, C106, C107, C108, C109, C110, C111, C112, C113, C114, C115 | X7R Ceramic, 1206 |
| C18, C21, C24, C25, C34, C35, C39, C48, C49, C50, C51, C52, C94, C99 | X7R Ceramic, 1210 |
| C9, C10, C11, C55, C56, C57, C58, C59, C69, C70, C71, C72, C73, C74, C75, C78, C79, C81, C82, C83, C96 | X7R Ceramic, 1812 |
| C19 | Panasonic G |
| C86 | 0.1$\mu$F, X76, 0603 |
| C40, C87, C88 | 0.1$\mu$F, X7R, 0805 |
| C1, C5, C12, C30, C60 | 0.22$\mu$F, X7R, 0603 |
| C3, C7, C38, C62 | 0.47$\mu$F, X7R, 0603 |
| C4, C8, C33, C63, C65, C76, C97 | 2.2$\mu$F, X7R, 0603 |
| C2, C6, C14, C15, C31, C61, C67, C68 | 2.2$\mu$F, X7R, 0805 |
| D1, D2, D4, D5, D20 | DIODE123 |
| D9, D11 | DIODE323 |
| U6, U7, U9 | FAN7390M |
| JP4 | JUMPER |
| JP1, JP2, JP3, JP5, JP6, JP7 | PINHD-2X5 |
| R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17 | Resistor, 0603 |
| R22 | 1.0k$\Omega$, 0603 |
| L2 | Coilcraft LLPS6235 |

| **Identifier** | **Description** |
|---|---|
| D3 | 11V Zener, SOD323 |
| IC1, IC2, IC3, IC4, IC5 | ADUM3210 |
| CM1, CM4, CM5 | Common-Mode Choke |
| LED1, LED2, LED3, LED4, LED5, LED6, LED7 | LED, 0603 |
| U10 | LNK302 |
| U1, U4 | LTC4444 |
| X1, X2, X5 | MSTBV2 |
| X4, X6, X7 | MSTBV4 |
| Q1, Q2, Q3, Q4, Q15, Q17 | NFET-3 |
| Q6, Q9, Q12, Q13 | NFET-8 |
| U$1 | OPTO1SOP4 |
| X3, X8 | RM14-FULL |
| U2, U3, U5, U8 | TPS715 |
| TP1, TP3, TP4, TP5, TP7, TP8 | TPTHL |
| TP6, TP9 | TPTHS |
| DCDC1, DCDC2, DCDC3, DCDC4 | VBSD1SIP |
| C53 | 1206 |

Table A.1: [[TODO: ***Bill of Materials***]]

| Part Designator(s) | Description | Manufacturer | Part Number |
|---|---|---|---|
| X1 | User-Defined Part | MyCompany | A1234-67 |
| X1 | User-Defined Part | MyCompany | A1234-67 |

## A.2 Converter Main-Board

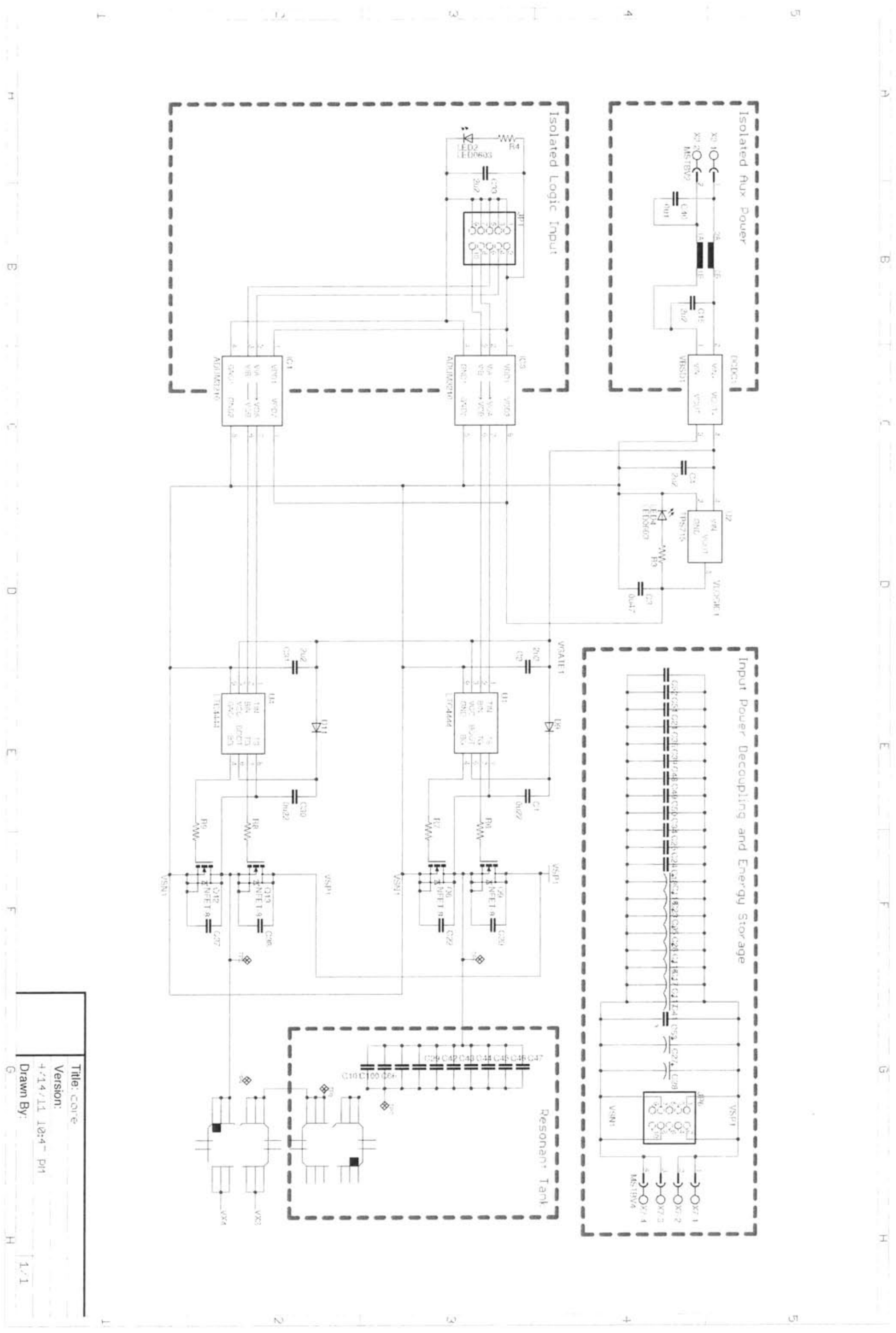


Figure A.1: Schematic of the converter (1/3).

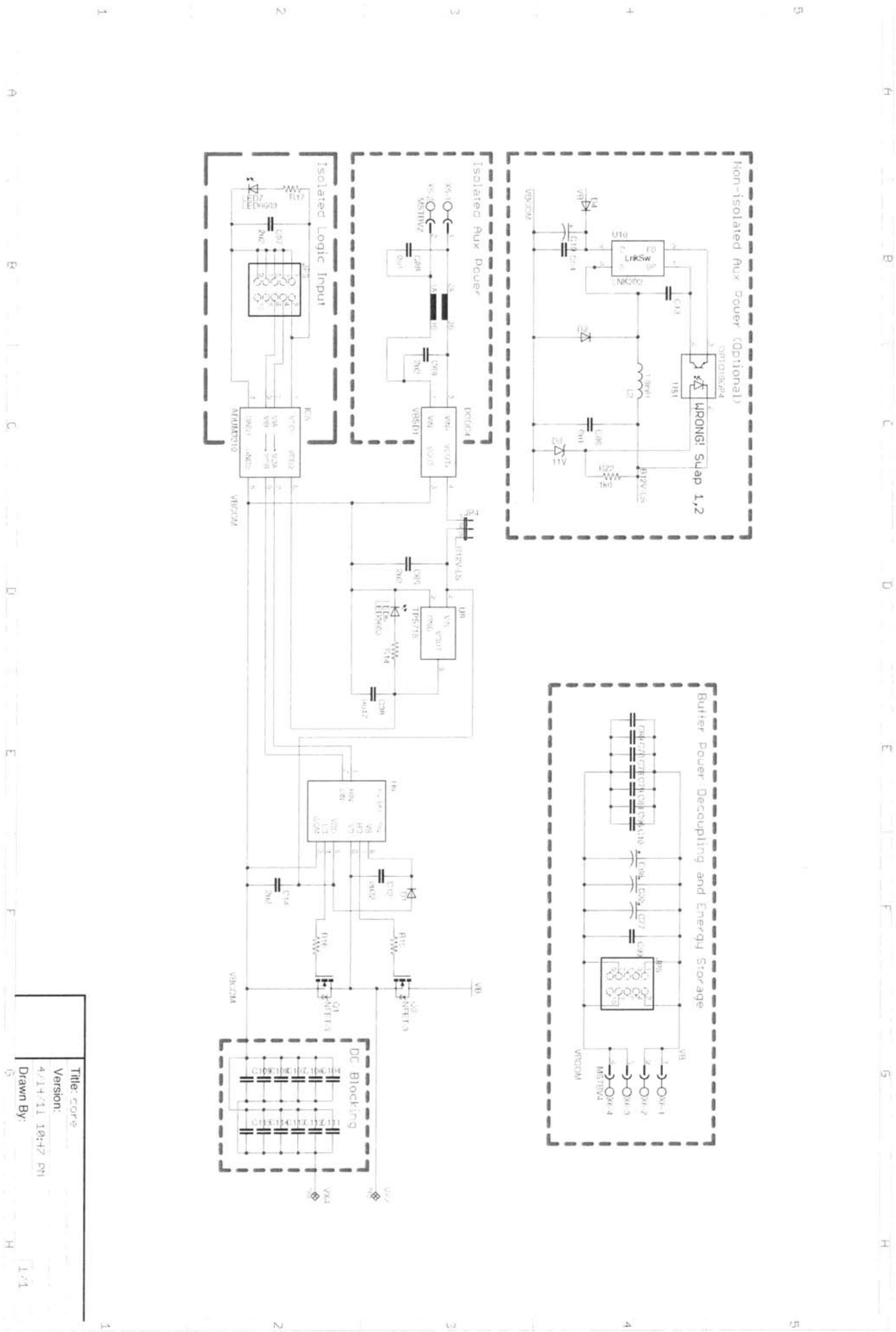


Figure A.2: Schematic of the converter (2/3).

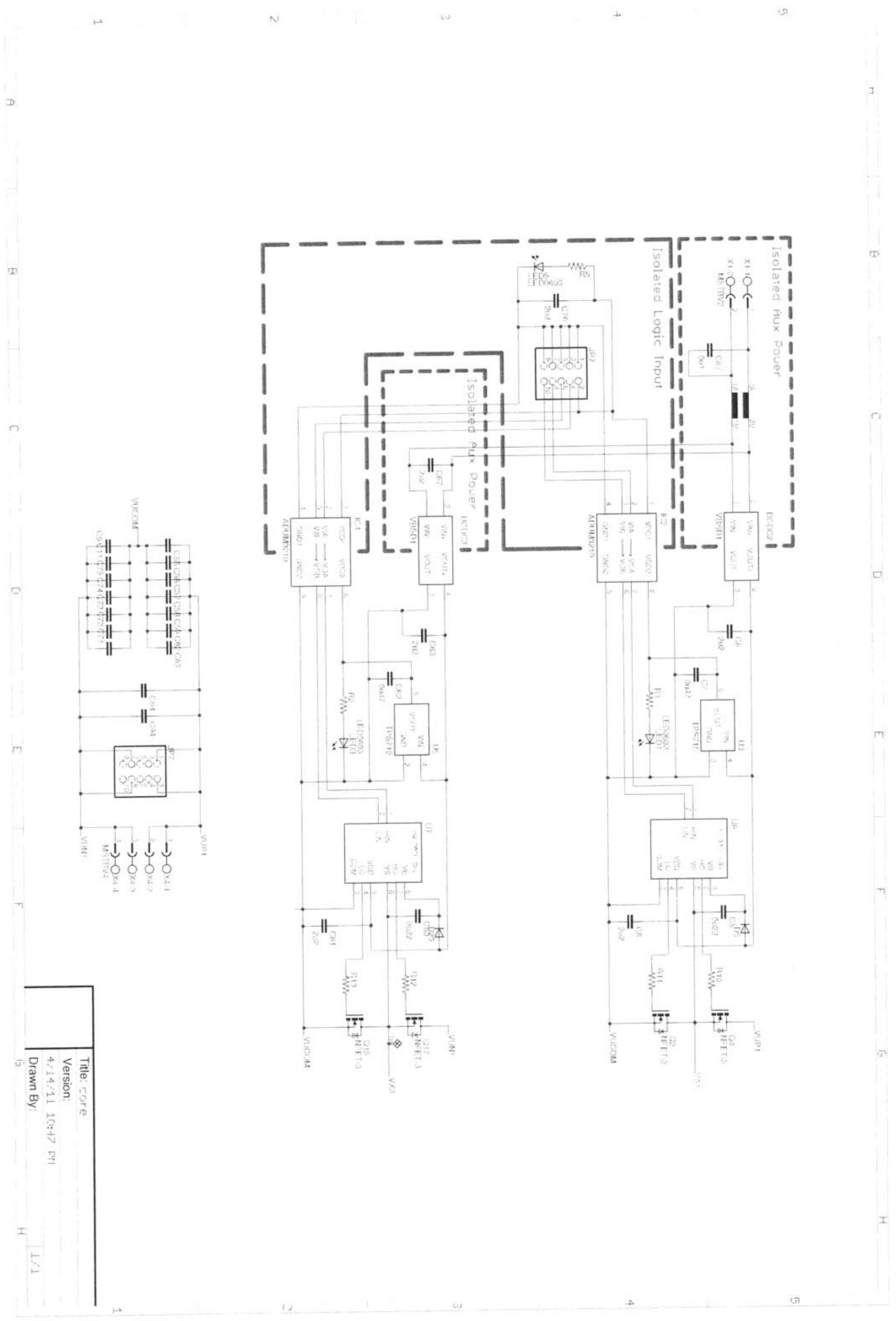


Figure A.3: Schematic of the converter (3/3).

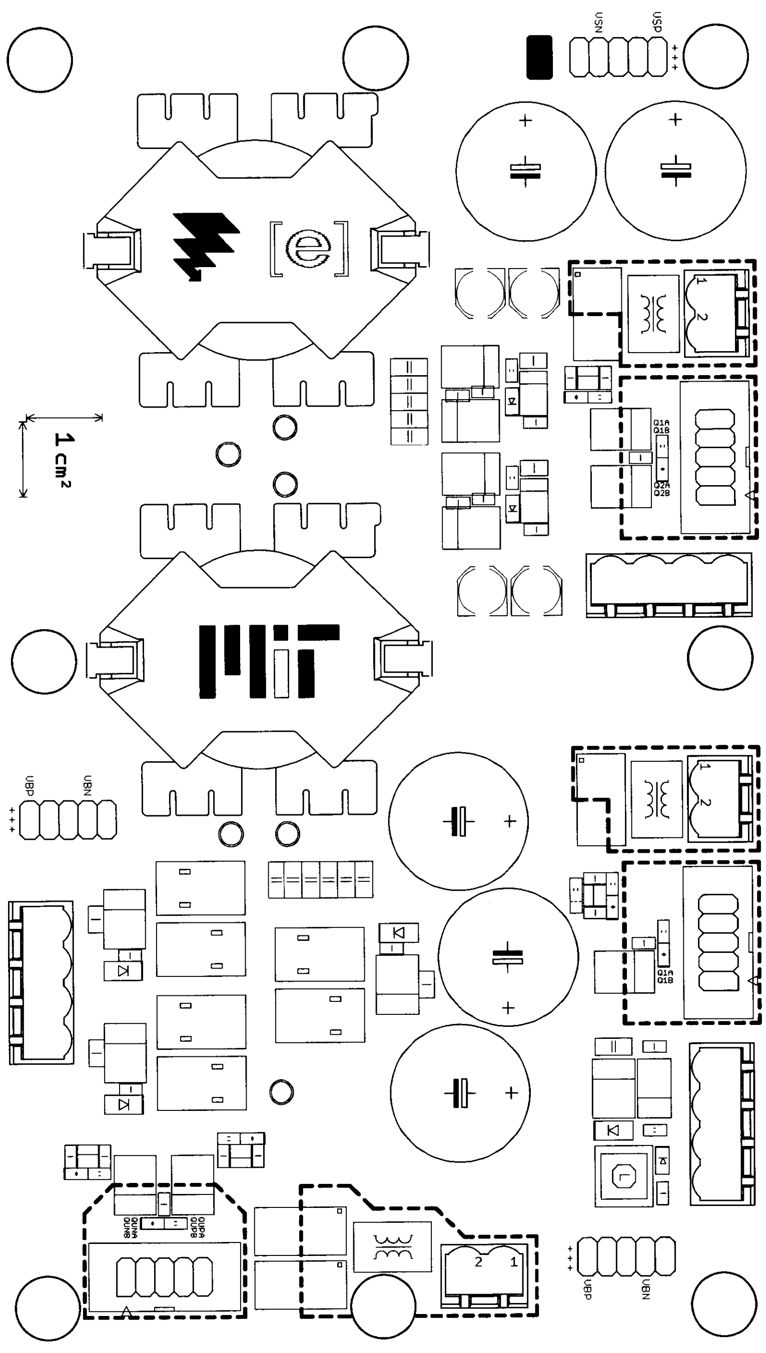


Figure A.4: Top silk screen layer.

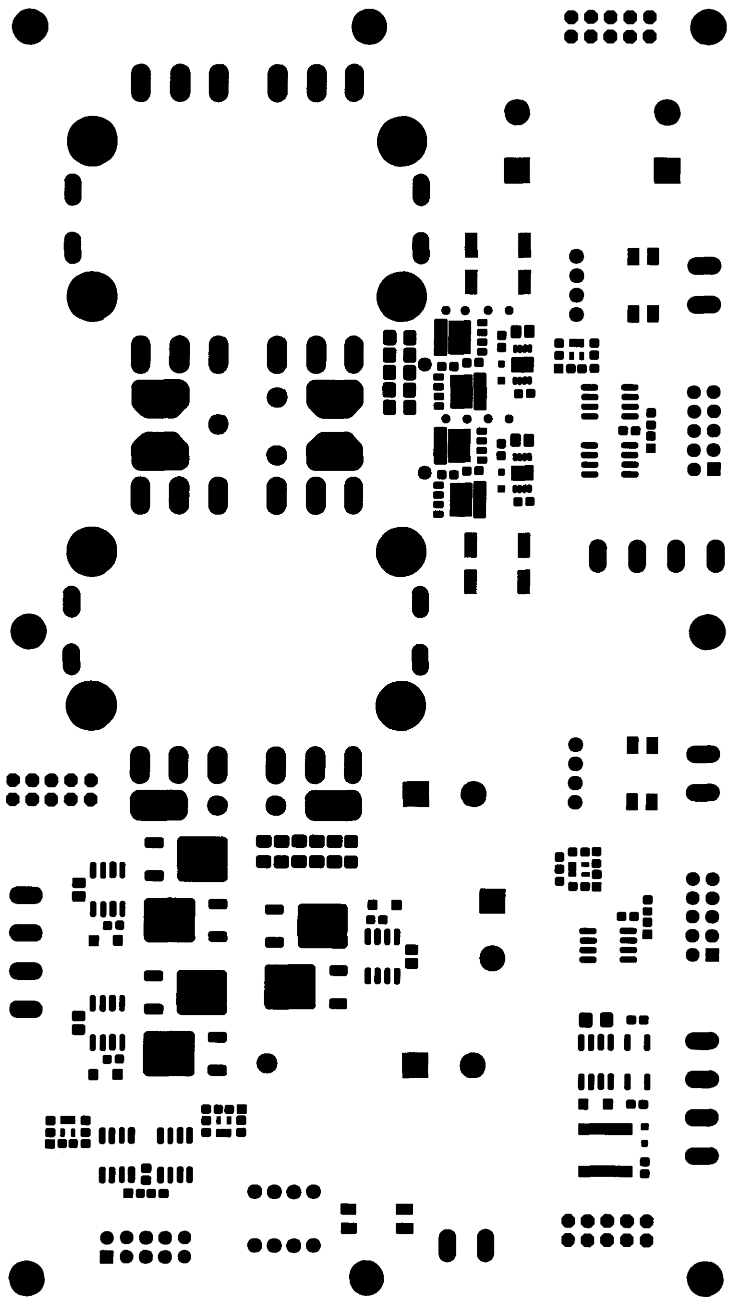


Figure A.5: Top solder mask layer.

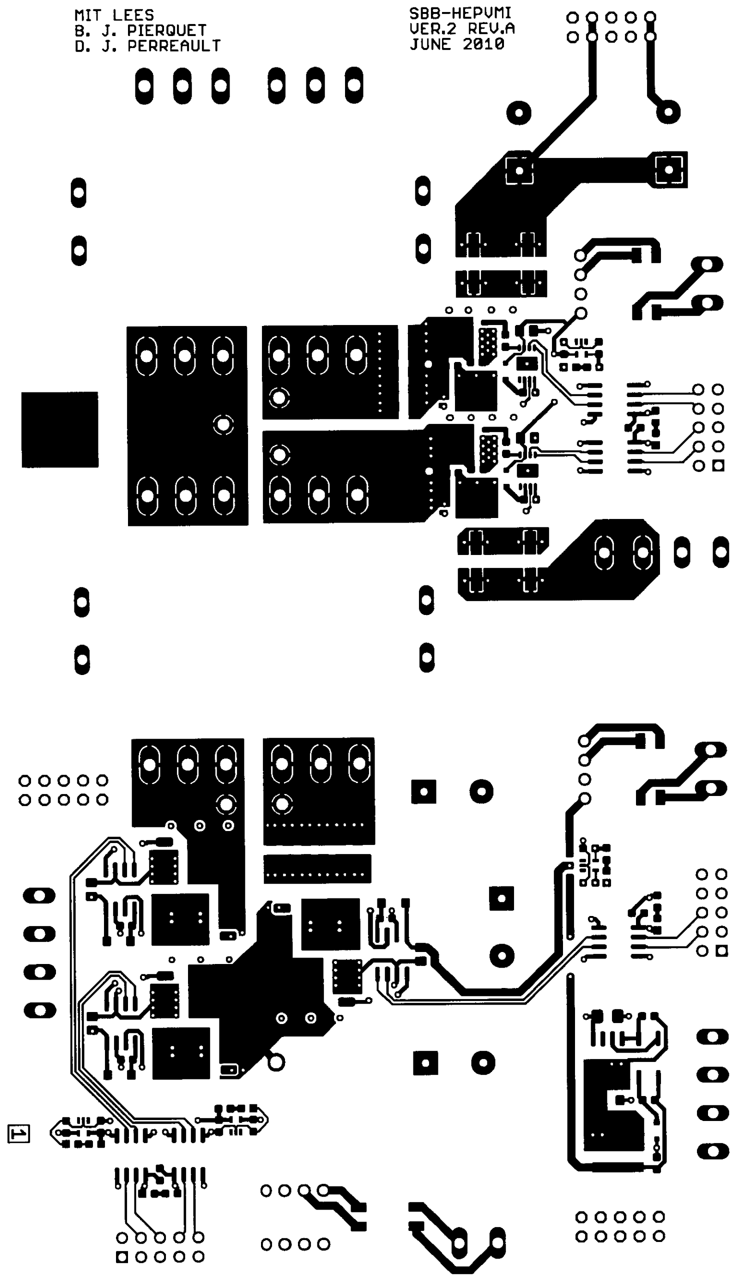


Figure A.6: Top copper (1) layer.

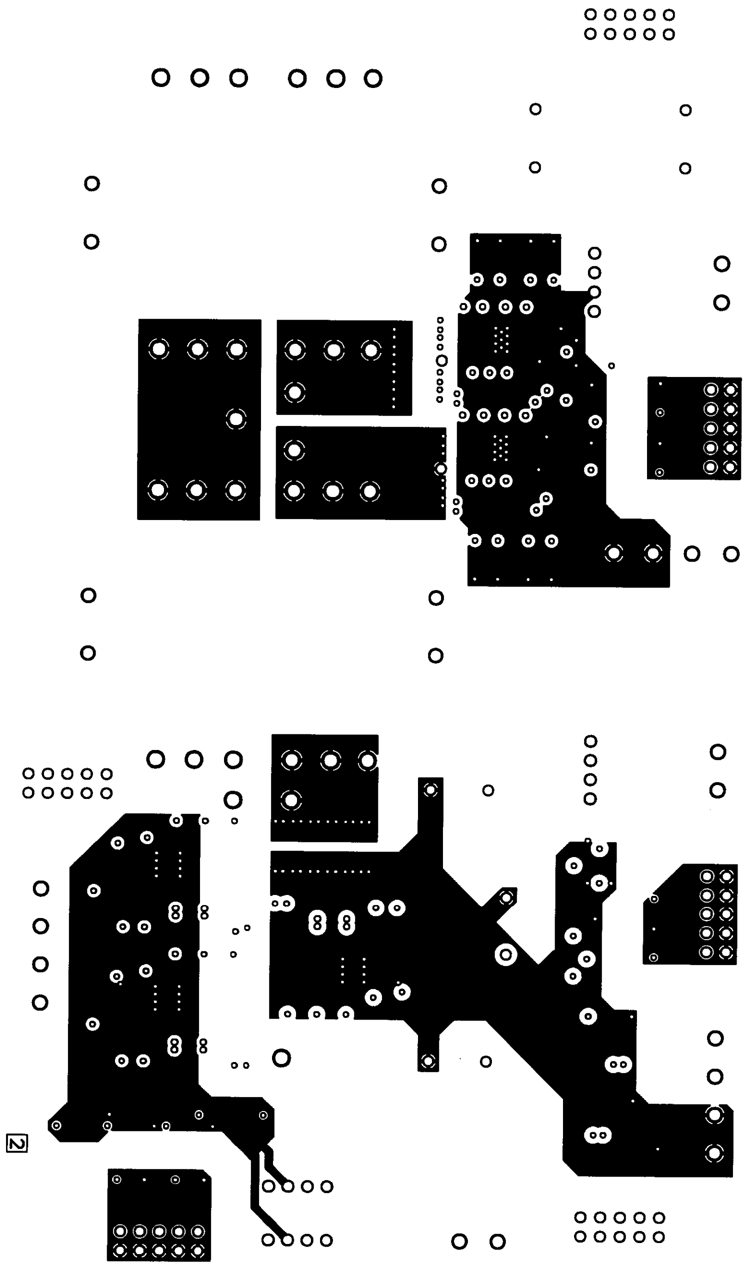


Figure A.7: Top inner copper (2) layer.

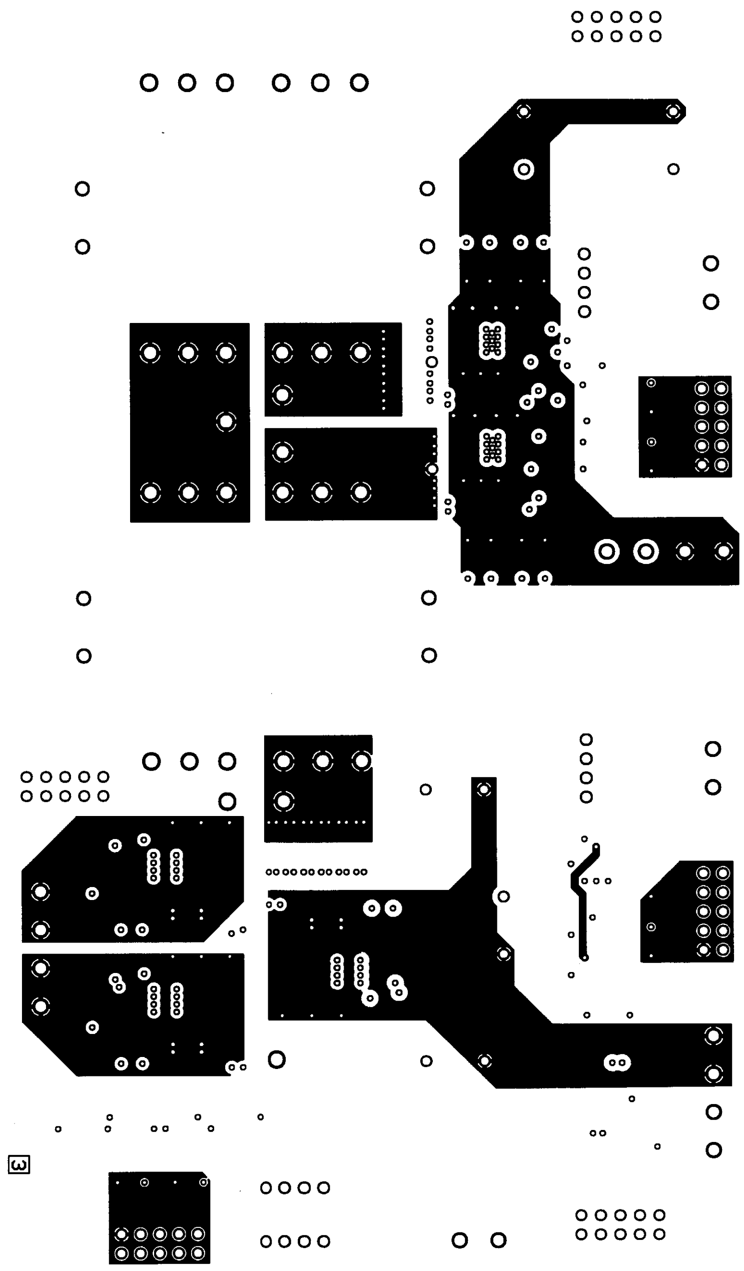


Figure A.8: Bottom inner copper (3) layer.

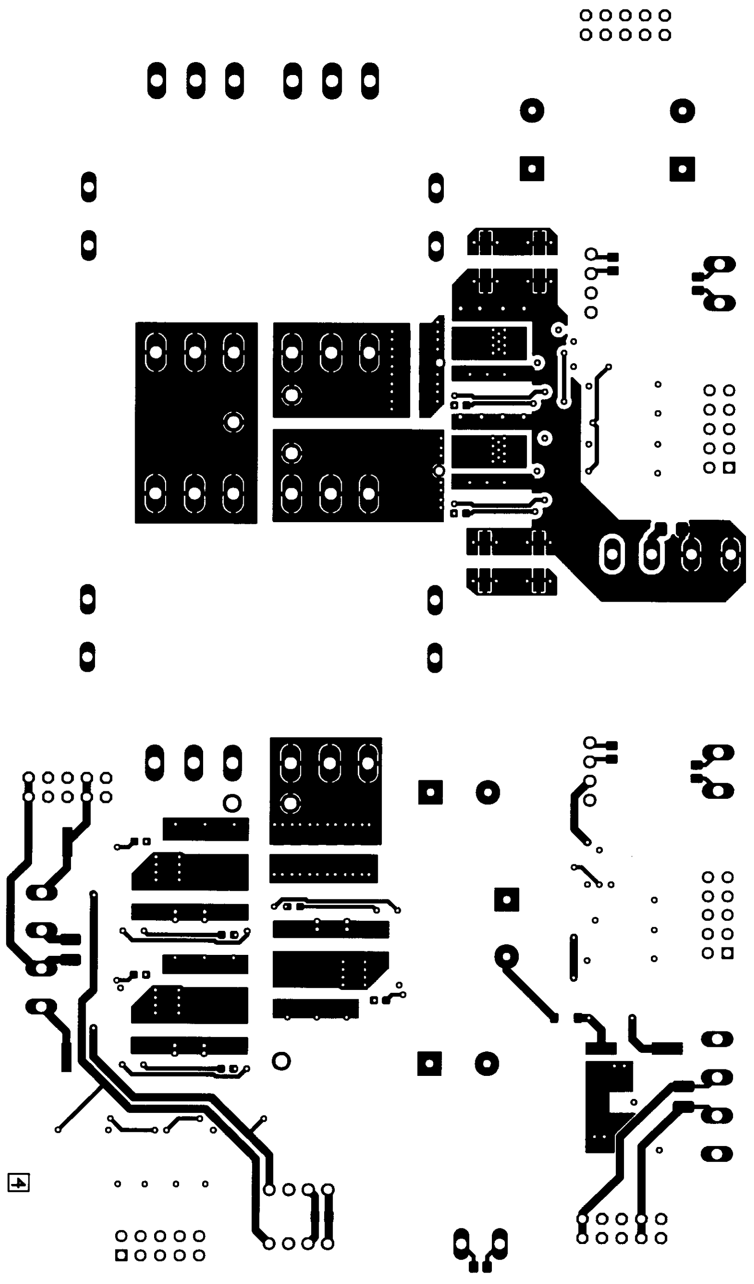


Figure A.9: Bottom copper (4) layer.

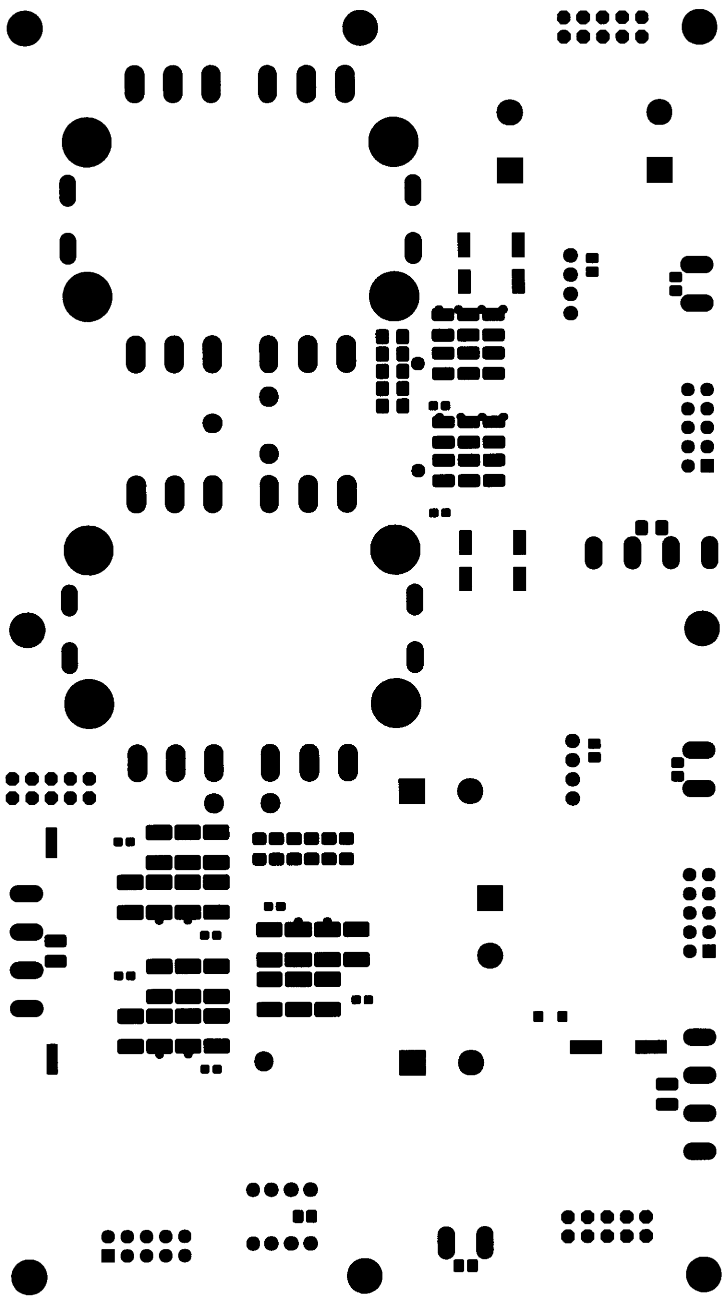


Figure A.10: Bottom solder mask layer

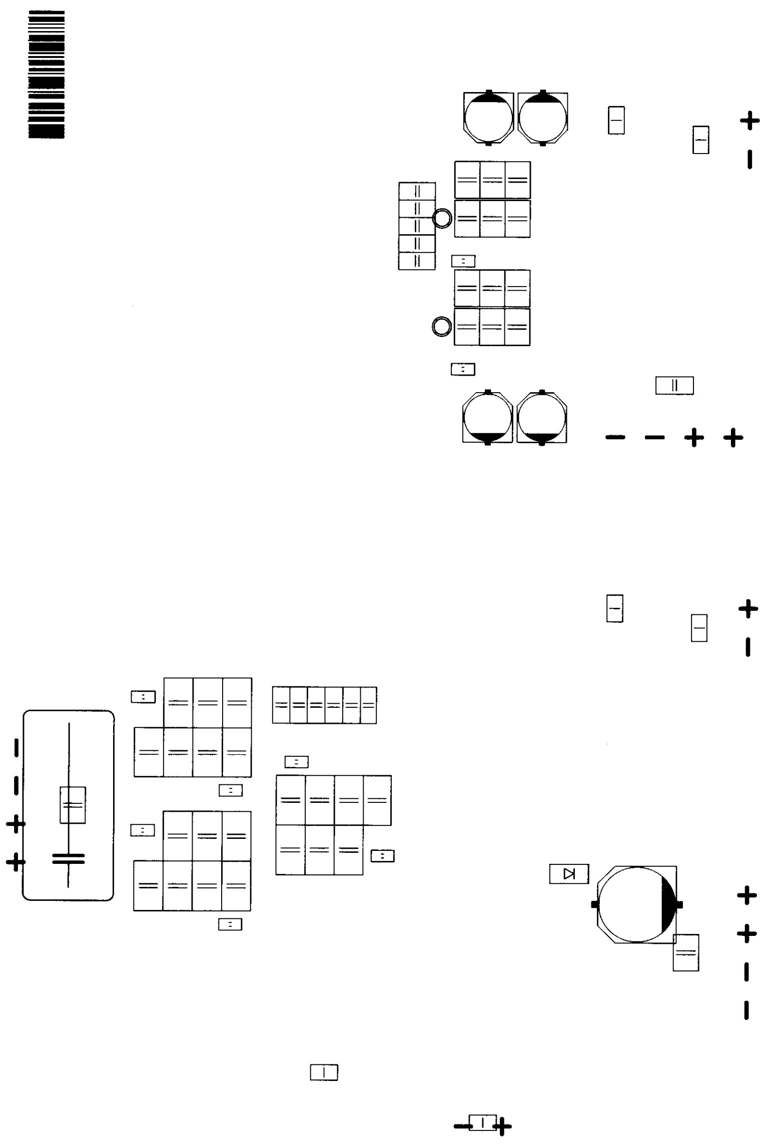


Figure A.11: Bottom silk screen layer.

## A.3 Isolated Voltage Measurement

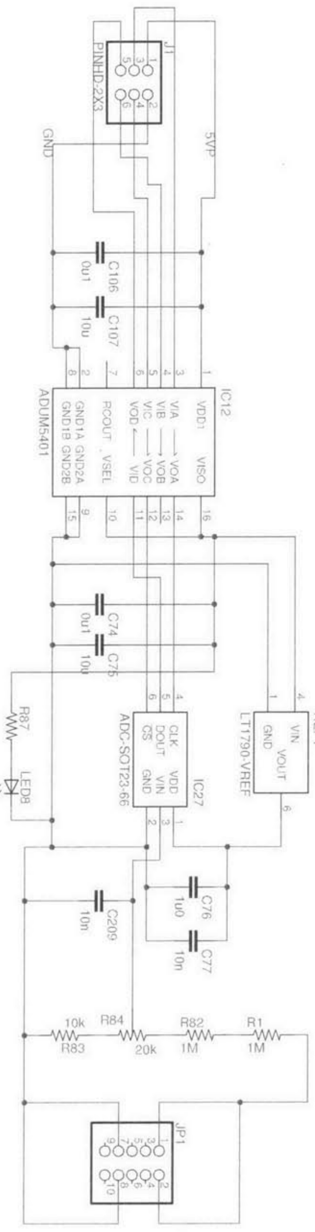


Figure A.12: Schematic of the isolated voltage measurement plug-in board.

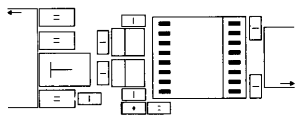


Figure A.13: Top silk screen layer.

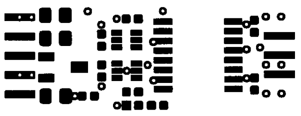


Figure A.14: Top solder mask layer.

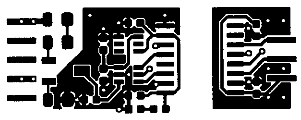


Figure A.15: Top copper (1) layer.

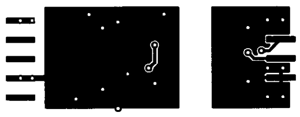


Figure A.16: Bottom copper (2) layer.

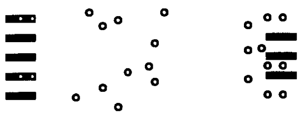


Figure A.17: Bottom solder mask layer.

# *Appendix B* *Simulation Code*

## B.1 Converter Simulation

### B.1.1 `hepvmi/compensation.py`

```python
from math import pi
import numpy as np

# Return the argument that is numerically in the middle
def limit(a,b,c):
    l = [a,b,c]
    l.sort()
    return l[1]

def calculate_deadtime(origop, s1qoss_func=None, s2qoss_func=None, uqoss_func=None,
     bqoss_func=None, mindt=0e-9):
    op = dict(origop)

    zero = lambda v: 0.0
    if not s1qoss_func: s1qoss_func = zero
    if not s2qoss_func: s2qoss_func = zero
    if not uqoss_func: uqoss_func = zero
    if not bqoss_func: bqoss_func = zero

    # Calculate the fraction of the charge required to what's available, which
    # is also the "bias", which indicate the shift of the rise/fall
    # of the waveform from the ideal center location
    biass1_lh = limit(1,0, (s1qoss_func(op["vi"]/op["N"])/op["N"]) / abs(op["zvs_qs1"][0]) )
    biass2_lh = limit(1,0, (s2qoss_func(op["vi"]/op["N"])/op["N"]) / abs(op["zvs_qs2"][0]) )
    biasu_lh = limit(1,0, ( uqoss_func(op["vl"]) ) / abs(op["zvs_qu" ][0]) )
    biasb_lh = limit(1,0, ( bqoss_func(op["vc"]) ) / abs(op["zvs_qb" ][0]) )

    biass1_hl = limit(1,0, (s1qoss_func(op["vi"]/op["N"])/op["N"]) / abs(op["zvs_qs1"][1]) )
    biass2_hl = limit(1,0, (s2qoss_func(op["vi"]/op["N"])/op["N"]) / abs(op["zvs_qs2"][1]) )
    biasu_hl = limit(1,0, ( uqoss_func(op["vl"]) ) / abs(op["zvs_qu" ][1]) )
    biasb_hl = limit(1,0, ( bqoss_func(op["vc"]) ) / abs(op["zvs_qb" ][1]) )

    #print uqoss_func(op["vl"]), bqoss_func(op["vc"])

    # Correct the sign on the bias to indicate if the transition is shifted
    # towards the rise or fall signal
    # A negative bias value indicates that the first half of the transition
    # takes longer than the second half
    biass1_lh = biass1_lh * np.sign(op["zvs_qs1"][0]) * np.sign(op["zvs_ts1"][0]-pi/2)
    biass2_lh = biass2_lh * np.sign(op["zvs_qs2"][0]) * np.sign(op["zvs_ts2"][0]-pi/2)
    biasu_lh = biasu_lh * np.sign(op["zvs_qu" ][0]) * np.sign(op["zvs_tu" ][0]-pi/2)
    biasb_lh = biasb_lh * np.sign(op["zvs_qb" ][0]) * np.sign(op["zvs_tb" ][0]-pi/2)

    biass1_hl = biass1_hl * np.sign(op["zvs_qs1"][1]) * np.sign(op["zvs_ts1"][1]-pi/2)
    biass2_hl = biass2_hl * np.sign(op["zvs_qs2"][1]) * np.sign(op["zvs_ts2"][1]-pi/2)
    biasu_hl = biasu_hl * np.sign(op["zvs_qu" ][1]) * np.sign(op["zvs_tu" ][1]-pi/2)
    biasb_hl = biasb_hl * np.sign(op["zvs_qb" ][1]) * np.sign(op["zvs_tb" ][1]-pi/2)

    #print biass1_lh, biass2_lh, biasu_lh, biasb_lh
    #print biass1_hl, biass2_hl, biasu_hl, biasb_hl

    # Use the fraction of charge to determine the corresponding percent transition time
    mint = 2*pi*mindt*op['fsw']
    as1_lh = max(mint, op["zvs_ts1"][0]*biass1_lh)
    as2_lh = max(mint, op["zvs_ts2"][0]*biass2_lh)
    au_lh = max(mint, op["zvs_tu" ][0]*biasu_lh)
    ab_lh = max(mint, op["zvs_tb" ][0]*biasb_lh)

    as1_hl = max(mint, op["zvs_ts1"][1]*biass1_hl)
    as2_hl = max(mint, op["zvs_ts2"][1]*biass2_hl)
    au_hl = max(mint, op["zvs_tu" ][1]*biasu_hl)
    ab_hl = max(mint, op["zvs_tb" ][1]*biasb_hl)

    #print as1_lh, as2_lh, au_lh, ab_lh
    #print as1_hl, as2_hl, au_hl, ab_hl

    # Calculate the dead time for the falling and rising signals (in that
    # order) for both the low/high and high/low transitions
    op["as1_lh"] = ( as1_lh*(0.5-0.25*biass1_lh), as1_lh*(0.5+0.25*biass1_lh) )
    op["as2_lh"] = ( as2_lh*(0.5-0.25*biass2_lh), as2_lh*(0.5+0.25*biass2_lh) )
    op["au_lh" ] = ( au_lh *(0.5-0.25*biasu_lh ), au_lh *(0.5+0.25*biasu_lh ) )
    op["ab_lh" ] = ( ab_lh *(0.5-0.25*biasb_lh ), ab_lh *(0.5+0.25*biasb_lh ) )

    op["as1_hl"] = ( as1_hl*(0.5-0.25*biass1_hl), as1_hl*(0.5+0.25*biass1_hl) )
    op["as2_hl"] = ( as2_hl*(0.5-0.25*biass2_hl), as2_hl*(0.5+0.25*biass2_hl) )
    op["au_hl" ] = ( au_hl *(0.5-0.25*biasu_hl ), au_hl *(0.5+0.25*biasu_hl ) )
    op["ab_hl" ] = ( ab_hl *(0.5-0.25*biasb_hl ), ab_hl *(0.5+0.25*biasb_hl ) )

    return op

def calculate_lag(origop, delays=None):
    op = dict(origop)

    if not delays:
        delays = {}
        delays["com"] = 0.0
        delays["s1"] = 0.0
        delays["s2"] = 0.0
        delays["u"] = 0.0
        delays["b"] = 0.0

    op["lag_s1"] = (delays["com"]+delays["s1"]) * 2*pi*op["fsw"]
    op["lag_s2"] = (delays["com"]+delays["s2"]) * 2*pi*op["fsw"]
    op["lag_u" ] = (delays["com"]+delays["u" ]) * 2*pi*op["fsw"]
    op["lag_b" ] = (delays["com"]+delays["b" ]) * 2*pi*op["fsw"]

    return op
```

### B.1.2 hepvmi/contourintersection.py

```
'''
Created on Dec 6, 2009

@author: pierquet
'''
import numpy as np
import matplotlib.pyplot as plt

def find_clusters(vlist, thresh):
# vlist = list(vlist)

    # Clusters are found by taking the difference between the sorted points
    # in each dimension
    # The locations where the difference between the x or y locations is
    # greater than thresh are assumed to be locations where a new cluster
    # has begun/end. This requires the addition of the first and last
    # locations to indicate the start of the first, and end of the last.

    vlist.sort()
    vdiff = np.diff(vlist) # take diff of x coords
    #print "Vdiff:", vdiff
    vclusterloc = ( abs(vdiff)>thresh ).nonzero()[0] # check for threshold
    vclusterloc = np.concatenate(([0], vclusterloc+1, [len(vdiff)+1]))

    # Generate a list of ranges for the clusters
    clusterlist = []
    for cl in range(len(vclusterloc)-1):
        clusterpoints = vlist[vclusterloc[cl]:vclusterloc[cl+1]]
        #print "Points in this cluster: ", len(clusterpoints)
        pmin = np.min(clusterpoints)
        pmax = np.max(clusterpoints)
        ## print "Cluster bounded by: ", pmin,pmax
        clusterlist.append([pmin,pmax])

    return clusterlist

def intersection_by_contour(xyval1, zval1, xyval2, zval2):
    # For both the desired source and line power, create a new contour plot
    # with the only contour line being the power. The (x,y) values for the
    # paths are extracted for all line segments
    ax = plt.Figure().gca()
    c = ax.contour(xyval1, [zval1])
    p = c.collections[0].get_paths()
    xy1 = np.array([]).reshape((0,2))
    for i in range(len(p)):
        xy1 = np.concatenate( (xy1,p[i].vertices),0)

    c = ax.contour(xyval2, [zval2])
    p = c.collections[0].get_paths()
    xy2 = np.array([]).reshape((0,2))
    for i in range(len(p)):
        xy2 = np.concatenate( (xy2,p[i].vertices),0)

    # The (x,y) pairs are in the [0,domN] domain, but not necessarily
    # integers. Each value is bounded to integers to ensure that an
    # intersection point generates at least one common value.
    # These sets are then intersected to find the common (x,y) pairs;
    # there may be many of them around each intersection point due to the
    # integer bounding
    xy1_set = set( (tuple(x) for x in list(np.floor(xy1))) ) | set( (tuple(x) for x in list(
        np.ceil(xy1))) )
    xy2_set = set( (tuple(x) for x in list(np.floor(xy2))) ) | set( (tuple(x) for x in list(
        np.ceil(xy2))) )
    xy_set = xy1_set & xy2_set

    #print "contour xy1 set: ", xy1
    #print "Contour xy_set: ", xy_set
    return xy_set

def bitmap_contour_points(xyval, zval):
    A = (xyval>zval)
    dA0 = np.diff(A, axis=0)
    dA1 = np.diff(A, axis=1)
    B=np.zeros(A.shape)
    B[:-1,:]+=dA0
    B[ 1:,:]+=dA0
    B[:,:-1]+=dA1
    B[:, 1:]+=dA1
    B = np.transpose(B) # required for some reason to match the original contour version
    xypoints = np.transpose(B.nonzero())

    return xypoints

def intersection_by_bitmap(pxy1, zval1, pxy2, zval2):
    xy1 = bitmap_contour_points(pxy1, zval1)
    #print "bitmap xy1 set: ", xy1
    xy2 = bitmap_contour_points(pxy2, zval2)

    xy1_set = set( (tuple(x) for x in xy1.tolist()) )
    xy2_set = set( (tuple(x) for x in xy2.tolist()) )
    xy_set = xy1_set & xy2_set

    #print "Threshold xy_set: ", xy_set
    return xy_set

def estimate_contour_intersection(xy1, z1, xy2, z2, thresh=10):
    #xy_set = intersection_by_contour(xy1, z1, xy2, z2)
    xy_set = intersection_by_bitmap(xy1, z1, xy2, z2)
    #print "Set differences: ", (xy_set - xy_set2)

    # If there's no intersections, bail out now
    if (len(xy_set)==0):
        return np.array([]).reshape(0,2)

    # Convert the resulting set of tuples to a list of lists, for simplicity
    xy_list = [list(i) for i in xy_set]

    # Assemble the set of possible xy clusters, checking them against the
    # original xylist values to eliminate false intersections that occur
    # from unintentional x,y overlaps
    xclusters = find_clusters([xy[0] for xy in xy_list], thresh)
    #print "x Clusters Found:", xclusters
    yclusters = find_clusters([xy[1] for xy in xy_list], thresh)
    #print "y Clusters Found:", yclusters

    xyclusters = []
    for xc in xclusters:
        for yc in yclusters:
            valid = False
            for x,y in xy_list:
                if (xc[0]<=x<=xc[1]) and (yc[0]<=y<=yc[1]):
                    valid = True
            if (valid==True):
                xyclusters.append(xc+yc)
            else:
                #print "Removed Invalid Cluster"
                pass

    return xyclusters
```

### B.1.3 hepvmi/converteroperation.py

```
import numpy as np
import scipy.sparse as sparse
import scipy.linalg as linalg
import pylab
#from scipy import sparse.linalg
import math
from math import pi

import fpgacore as fpga

class ConverterWaveforms:
    """"""
    def __init__(self, converter, ideal):
        self.timevar = "Time"
        self.converter = converter
        self.waves = {}
        steps = converter.pwmcore.counter.terminal_count
        Tout = 1/converter.pwmcore.counter.out_freq
        dt = 1/converter.pwmcore.counter.count_freq
        self.waves[self.timevar] = np.linspace(0, Tout - dt, steps)

        chans = converter.pwmcore.channels.keys()
        for ch in chans:
            self.waves[ch] = self.createWaveform(converter.pwmcore.channels[ch].get_on_val(),
                                             converter.pwmcore.channels[ch].get_off_val(),
                                             ideal=ideal)

        self.waves["vs1"] = (self.waves["i1B"])*converter.oppoint.params["vi"]
        self.waves["vs2"] = (self.waves["i2B"])*converter.oppoint.params["vi"]
        self.waves["vs"] = (self.waves["i2B"]-self.waves["i1B"])*converter.oppoint.params["vi"]
        self.waves["vb"] = self.waves["bbB"]*converter.oppoint.params["vc"]
        self.waves["vu"] = self.waves["cpB"]*converter.oppoint.params["vl"]
        self.waves["vt"] = self.waves["vs"] + self.waves["vb"] - self.waves["vu"]
        self.generateCurrentWaveform()

    def createWaveform(self, onval, offval, ideal):
        if ideal:
            wave = np.zeros(self.waves[self.timevar].shape)
            start = min(onval, offval)
            stop = max(onval, offval)
            wave[start:stop] = 1
            if(onval<=offval):
                # If on came before off, we did the right thing
                pass
            else:
                # Off came before on, so we have to invert the waveform
                wave = (wave==0)*1.0
        else:
            wave = []
            if (onval<=offval):
                wave = np.zeros(self.waves[self.timevar].shape)
                wave[onval-2] = 0.05
                wave[onval-1] = 0.20
                wave[onval ] = 0.80
                wave[onval+1] = 0.95
                wave[onval+2:offval-2] = 1
                wave[offval-2] = 0.95
                wave[offval-1] = 0.80
                wave[offval % len(wave)] = 0.20
                wave[(offval+1) % len(wave)] = 0.05
            else:
                wave = np.ones(self.waves[self.timevar].shape)
                wave[offval-2] = 0.95
                wave[offval-1] = 0.80
                wave[offval ] = 0.20
                wave[offval+1] = 0.05
                wave[offval+2:onval-2] = 0
                wave[onval-2] = 0.05
                wave[onval-1] = 0.20
                wave[onval % len(wave)] = 0.80
                wave[(onval+1) % len(wave)] = 0.95

        return wave

    def plot(self, *names):
        if len(names)==0:
            names = self.waves.keys()
            names.remove(self.timevar)
        for n in names:
            pylab.plot(self.waves[self.timevar],self.waves[n])
        pylab.grid(True)
        pylab.draw()
        pylab.show()

#scipy.sparse.linalg.spsolve(A, b, permc_spec=2)
    def generateCurrentWaveform(self):
        steps = int(self.converter.pwmcore.counter.terminal_count)
        dt = 1/self.converter.pwmcore.counter.count_freq
        L = self.converter.L
        C = self.converter.C
        R = self.converter.R

        vals = [1]*2*steps
        rows = range(steps) + [steps]*steps
        cols = [steps]*steps + range(steps)
        A = sparse.csr_matrix( (vals,(rows,cols)))

        k = L/(dt**2)
        Al = sparse.lil_diags([[k]*(steps-1)+[0],
                               [-2*k]*steps+[0],
                               [k]*(steps-1)+[0],
                               [k]+[0]*steps, [k]+[0]*steps],
                              [-1,0,1,-(steps-1),steps-1], (steps+1, steps+1)).tocsr()

        #
        k = R/dt
        Ar = sparse.lil_diags([[-k]*(steps)+[0],
                                 [k]*(steps-1)+[0],
                                 [k,0]],
                                [0,1,-(steps-1)], (steps+1, steps+1)).tocsr()

        #
        k = 1/C
        Ac = sparse.lil_diags([[k]*steps+[0]], [0], (steps+1, steps+1)).tocsr()

        #
        A = A + Al + Ar + Ac

        # vt = vs + vb - vu
        vt = self.waves["vt"]
        #vt = vt-np.average(vt) # remove dc offset
        b = np.zeros(steps+1)
        b[:-1] = np.diff(np.concatenate( (vt, [vt[0]]) )) / dt

        # extend solution to include sum(i_k)=0
        sparse.linalg.use_solver(useUmfpack=False)
        x = sparse.linalg.spsolve(A,b)
        #x = x - np.average(x) # enforce zero dc current

        self.waves["is"] = x[:-1]

    def generateCurrentWaveform_old(self):
        steps = self.converter.pwmcore.counter.terminal_count
        dt = 1/self.converter.pwmcore.counter.count_freq
        L = self.converter.L
        C = self.converter.C
        R = self.converter.R

        A = np.ones((steps+1,steps+1))
        #A[:,-1] = np.zeros(steps+1)
        A[-1,-1] = 0
        b = np.zeros(steps+1)

        Al = ( np.diag(np.ones(steps-1),k=1) - 1*np.diag(np.ones(steps),k=0) )
        Al[-1,0] = 1
        Al = Al * L/(dt)
        #
        Ar = ( np.diag(np.ones(steps-1),k=1) )
        Ar[-1,0] = 1
        Ar = Ar * R
        #
        Ac = ( np.tril(np.ones((steps,steps)),k=1) )
        Ac[-1,0] = 2
        Ac = Ac * dt/C
        #
        A[:-1,:-1] = Al + Ar + Ac


        # vt = vs + vb - vu
        vt = self.waves["vt"]
        #vt = vt-np.average(vt) # remove dc offset
        b[:-1] = vt

        # extend solution to include sum(i_k)=0

        x = linalg.solve(A,b)
        #x = x - np.average(x) # enforce zero dc current

        self.waves["is"] = x[:-1]

class OperatingPoint:
    ''''''
    def __init__(self, name="", valdict=None):
        self.name = name

        self.target = {}
        self.target["pin"] = 0
        self.target["pout"] = 0
        self.target["pbuf"] = 0

        self.params = {}
        self.params["fsw"] = 1
        # Inverter Parameters
        self.params["vs"] = 0
        self.params["ds"] = 0
        # Buffer-Block Parameters
        self.params["vc"] = 0
        self.params["tb"] = 0
        self.params["db"] = 0
        # Cycloconverter Parameters
        self.params["vl"] = 0
        self.params["tu"] = 0
        self.params["du"] = 0

        if (valdict!=None):
            self.params.update(valdict)

    def generate_model(self, fclock=100e6):
        clk = self.params.has_key('pwmclock') and self.params['pwmclock'] or fclock
        self.model = ConverterModel(self, fclock=100e6)

    def generate_waveforms(self, ideal=True):
        self.waveforms = ConverterWaveforms(self.model, ideal=ideal)

class ConverterModel:
    """"""
    def __init__(self, op, fclock):
        self.L = 1.0
        self.C = 1.0
        self.R = 1.0
        self.L = op.params["L"]
        self.C = op.params["C"]
        self.R = op.params["R"]
        self.oppoint = op
        self.pwmcore = fpga.PWMModule(fpga.TimerCounter(fclock))

        i1A = self.pwmcore.add_channel("i1A", 5)
        i1B = self.pwmcore.add_channel("i1B", 6)
        i2A = self.pwmcore.add_channel("i2A", 3)
        i2B = self.pwmcore.add_channel("i2B", 4)

        bbA = self.pwmcore.add_channel("bbA", 1)
        bbB = self.pwmcore.add_channel("bbB", 2)

        cpA = self.pwmcore.add_channel("cpA", 7)
        cpB = self.pwmcore.add_channel("cpB", 8)
        cnA = self.pwmcore.add_channel("cnA", 9)
        cnB = self.pwmcore.add_channel("cnB", 10)

        self.pwmcore.counter.set_output_freq(op.params["fsw"])
        bbB.theta = op.params["tb"]
        bbA.theta = op.params["tb"] + pi
        cpB.theta = op.params["tu"]
        cpA.theta = op.params["tu"] + pi

        i2_theta = -(1.0-op.params["ds"])/4 * 2*pi
        i1_theta = -(1.0+op.params["ds"])/4 * 2*pi

        i2B.theta = i2_theta
        i2A.theta = i2_theta + pi
        i1B.theta = i1_theta
        i1A.theta = i1_theta + pi

        # Don't generate any dead-time for the waveforms

        # Set all on time values to 0.5
        for (name, ch) in self.pwmcore.channels.iteritems():
            ch.beta = 0.5

def cshift(l, offset):
    offset %= len(l)
    return np.concatenate((l[-offset:], l[:-offset]))

def gen_oppoint(linepos=0, vlpk=0, pavg=0, theta=0):
    '''Returns a skeleton operating point with the input and output
       power targets, along with the grid voltage'''
    dic = {}
    dic["tl"] = linepos
    dic["vl"] = vlpk*math.sin(linepos)
    dic["pu"] = pavg*(math.cos(theta) - math.cos(2*linepos+theta) )
    dic["ps"] = -pavg
    dic["pb"] = -dic["pu"]-dic["ps"]
    return dic

def simop(op=None, opp=None, linewidth=2):
    """a"""
    if (op!=None):
        opp = OperatingPoint(valdict=op)
    opp.generate_model()
    opp.generate_waveforms(ideal=False)
    convmodel = opp.model
    convwaves = opp.waveforms

    longtime = convwaves.waves["Time"]
    longtime = np.concatenate((longtime,longtime+longtime[-1]+longtime[1]))

    vt = convwaves.waves["vt"]
    longvt = np.concatenate((vt,vt))
    vs = convwaves.waves["vs"]
    longvs = np.concatenate((vs,vs))
    vb = convwaves.waves["vb"]
    longvb = np.concatenate((vb,vb))
    vu = convwaves.waves["vu"]
    longvu = np.concatenate((vu,vu))
    iss = convwaves.waves["is"]
    longiss = np.concatenate((iss,iss))

    pylab.plot(longtime, longvs, linewidth=linewidth)
    pylab.plot(longtime, longvb, linewidth=linewidth)
    pylab.plot(longtime, longvu, linewidth=linewidth)
    pylab.plot(longtime, longiss*100, linewidth=linewidth)
    pylab.grid(True)

    ni = iss.shape[0]
    ps = sum(vs*(-iss))/ni
    pb = sum(vb*(-iss))/ni
    pu = sum(vu*( iss))/ni
    pt = sum(vt*( iss))/ni

    print "ps: %.4f, pb: %.4f, pu: %.4f, err: %.4f" % (ps, pb, pu, pt)
    return opp
```

### B.1.4 hepvmi/fpgacore.py

```
import math

class TimerCounter:
    """Describes the primary timer/counter operation for compare channels"""
    def __init__(self, frequency):
        self.count_freq = frequency
        self.out_freq = 0
        self.terminal_count = 0

    def set_output_freq(self, ofreq):
        self.out_freq = ofreq
        self.terminal_count = math.floor(self.count_freq / self.out_freq)

class PWMChannel:
    """ Contains parameterized description of a single cosine ref PWM Channel

    N : maximum counter steps (0:255 = 256)
    theta : phase shift of the waveform cos(wt+theta) [0:2*pi]
    beta : the on time of the waveform, given zero dead-time [0:1]
    alphar : dead-time (delay) for the turn-on edge of the waveform [0:1]
    alphaf : dead-time (advance) for the turn-off edge of the waveform [0:1]
    """

    def __init__(self, channel, counter):
        self.channel = channel
        self.counter = counter
        self.theta = 0
        self.beta = 0
        self.alphar = 0
        self.alphaf = 0
        self.defstate = 0

    def get_on_val(self):
        self.onval = self.counter.terminal_count * \
                (-self.theta/(2*math.pi) - (self.beta/2 - self.alphar))
        return round(self.onval % self.counter.terminal_count)

    def get_off_val(self):
        self.offval = self.counter.terminal_count * \
                (-self.theta/(2*math.pi) + (self.beta/2 - self.alphaf))
        return round(self.offval % self.counter.terminal_count)

    def get_default(self):
        return self.defstate

class PWMModule:
    """Encapsulates a TimerCounter and multiple associated PWMChannels"""
    def __init__(self, counter=TimerCounter(0)):
        self.channels = {}
        self.counter = counter

    def add_channel(self, key, num):
        self.chan = PWMChannel(num, self.counter)
        self.channels[key] = self.chan
        return self.chan

    def generate_program(self, disablebefore=True, runafter=True):
        cmds = []
        if disablebefore:
            # Disable timer/counter
            cmds.append("w0000000")

        # Set the reset value for the counter
        cmds.append("w001%04X" % self.counter.terminal_count)

        # Process commands for each channel
        for (name, ch) in self.channels.iteritems():
            # Set the 'off' compare value
            cmds.append("w%02X1%04X" % ( ch.channel, ch.get_off_val() ))
            # Set the 'on' compare value
            cmds.append("w%02X2%04X" % ( ch.channel, ch.get_on_val() ))
            # Enable the channel
            if (ch.defstate==1):
                cmds.append("w%02X0%04X" % ( ch.channel, 3 ))
            else:
                cmds.append("w%02X0%04X" % ( ch.channel, 1 ))
        if runafter:
            # Enable the main counter
            cmds.append("w0000001")

        return cmds
```

### B.1.5 hepvmi/idealzvs.py

```
from converteroperation import OperatingPoint
from parameterestimation import estimate_currents
import numpy as np
from math import pi

def rms(data, axis=0):
    return np.sqrt(np.mean(data ** 2, axis))

def minmagnitude(*l):
    minp = np.min(l[l>=0])
    minn = np.max(l[l<=0])
    if abs(minp)<abs(minn):
        return minp
    else:
        return minn

def calculate_zvsmargins(origop):
    op = dict(origop)

    opp = OperatingPoint(valdict=op)
    opp.generate_model()
    opp.generate_waveforms(ideal=True)
    waves = opp.waveforms.waves

    vs1 = waves["vs1"]
    vs2 = waves["vs2"]
    vu = waves["vu"]
    vb = waves["vb"]
    iss = waves["is"]

    #
    # Determine if the switching transitions occur under the proper
    # current directions
    # -1==nonzvs, 1=zvs
    s2_cross = np.sign(-iss)*np.diff( np.sign( np.concatenate(([vs2[-1]],vs2)) ))
    s1_cross = np.sign( iss)*np.diff( np.sign( np.concatenate(([vs1[-1]],vs1)) ))
    u_cross = np.sign( iss)*np.diff( np.sign( np.concatenate(([vu[ -1]],vu )) ))
    b_cross = np.sign(-iss)*np.diff( np.sign( np.concatenate(([vb[ -1]],vb )) ))
    ii_cross = np.diff( np.sign( np.concatenate(([iss[-1]],iss)) ))

    s2_cross = s2_cross[s2_cross.nonzero()]
    s1_cross = s1_cross[s1_cross.nonzero()]
    u_cross = u_cross[ u_cross.nonzero()]
    b_cross = b_cross[ b_cross.nonzero()]
    ii_cross = ii_cross[ ii_cross.nonzero()]

    # Valid operation is when there are two zvs transitions
    valid = 1
    if len(ii_cross>0)!=2: valid = 0
    if sum(s2_cross>0)!=2: valid = 0
    if sum(s1_cross>0)!=2: valid = 0
    if sum( b_cross>0)!=2: valid = 0
    if sum( u_cross>0)!=2: valid = 0

    # If the transitions are valid, then calculate the total charge transferred
    # between the current crossing and switching time, the time for the
    # half-charge transfer, along with the the delay between the current and
    # switching
    if valid:
        ######
        # Positive and negative values of resonant current
        issn, issp = ( (iss<0)*iss, (iss>0)*iss )
        dtheta = 2*pi/len(iss)
        # Zero and Non-zero vectors of the voltages
        vs1z, vs2z, vuz, vbz = ( (vs1==0), (vs2==0), (vu==0), (vb==0) )
        vs1nz, vs2nz, vunz, vbnz = ( (vs1!=0), (vs2!=0), (vu!=0), (vb!=0) )

        ######
        # The current at which the ideal switching waveform occurs is recorded
        # in the case that it is needed moving forward for current-dependent
        # device turn-on/off times
        dvs1 = np.diff( np.sign( np.concatenate(([vs1[-1]],vs1)) ))
        dvs2 = np.diff( np.sign( np.concatenate(([vs2[-1]],vs2)) ))
        dvu = np.diff( np.sign( np.concatenate(([vu[-1]] ,vu )) ))
        dvb = np.diff( np.sign( np.concatenate(([vb[-1]] ,vb )) ))

        is1_ab = abs(iss[(dvs1>0).nonzero()[0]])[0]
        is2_ab = abs(iss[(dvs2>0).nonzero()[0]])[0]
        iu_ab = abs(iss[(dvu >0).nonzero()[0]])[0]
        ib_ab = abs(iss[(dvb >0).nonzero()[0]])[0]

        is1_ba = abs(iss[(dvs1<0).nonzero()[0]])[0]
        is2_ba = abs(iss[(dvs2<0).nonzero()[0]])[0]
        iu_ba = abs(iss[(dvu <0).nonzero()[0]])[0]
        ib_ba = abs(iss[(dvb <0).nonzero()[0]])[0]

        op.update({"zvs_is1":(is1_ab,is1_ba),
                   "zvs_is2":(is2_ab,is2_ba),
                   "zvs_iu" :(iu_ab ,iu_ba ),
                   "zvs_ib" :(ib_ab ,ib_ba )})

        ######
        # "Time" is expressed as the phase shift from the reference of 0 for the
        # zero crossing of the current
        # "Time" (radians) available for commutation, low->high
        ts1_ab = minmagnitude( -len(np.nonzero(vs1z *issp)[0]), len(np.nonzero(vs1nz*issp)
            [0]) ) * dtheta
        ts2_ab = minmagnitude( -len(np.nonzero(vs2z *issn)[0]), len(np.nonzero(vs2nz*issn)
            [0]) ) * dtheta
        tu_ab = minmagnitude( -len(np.nonzero(vuz *issp)[0]), len(np.nonzero(vunz *issp)[0])
             ) * dtheta
        tb_ab = minmagnitude( -len(np.nonzero(vbz *issn)[0]), len(np.nonzero(vbnz *issn)[0])
             ) * dtheta

        # "Time" (radians) available for commutation, high->low
        ts1_ba = minmagnitude( -len(np.nonzero(vs1nz*issn)[0]), len(np.nonzero(vs1z *issn)
            [0]) ) * dtheta
        ts2_ba = minmagnitude( -len(np.nonzero(vs2nz*issp)[0]), len(np.nonzero(vs2z *issp)
            [0]) ) * dtheta
        tu_ba = minmagnitude( -len(np.nonzero(vunz *issn)[0]), len(np.nonzero(vuz *issn)[0])
             ) * dtheta
        tb_ba = minmagnitude( -len(np.nonzero(vbnz *issp)[0]), len(np.nonzero(vbz *issp)[0])
             ) * dtheta

        op.update({"zvs_ts1":(ts1_ab,ts1_ba),
                  "zvs_ts2":(ts2_ab,ts2_ba),
                  "zvs_tu": (tu_ab, tu_ba),
                  "zvs_tb": (tb_ab, tb_ba)})

        ######
        # The available charge is found by integrating the current over the
        # time when the switch is off (low-high) or on (high-low). In the case
        # where the increase in the amount of charge is slowing (current is
        # decreasing, negative di/dt), a negative value is given to represent
        # this.
        dt = 1/float(opp.model.pwmcore.counter.count_freq)
        # "Charge" transfer for low->high transition
        qs1_ab = minmagnitude( abs(np.sum(vs1z *issp)), -abs(np.sum(vs1nz*issp)) ) * dt
        qs2_ab = minmagnitude( abs(np.sum(vs2z *issn)), -abs(np.sum(vs2nz*issn)) ) * dt
        qu_ab = minmagnitude( abs(np.sum(vuz *issp)), -abs(np.sum(vunz *issp)) ) * dt
        qb_ab = minmagnitude( abs(np.sum(vbz *issn)), -abs(np.sum(vbnz *issn)) ) * dt
        # "Charge" transfer for high->low transition
        qs1_ba = minmagnitude( abs(np.sum(vs1nz*issn)), -abs(np.sum(vs1z *issn)) ) * dt
        qs2_ba = minmagnitude( abs(np.sum(vs2nz*issp)), -abs(np.sum(vs2z *issp)) ) * dt
        qu_ba = minmagnitude( abs(np.sum(vunz *issn)), -abs(np.sum(vuz *issn)) ) * dt
        qb_ba = minmagnitude( abs(np.sum(vbnz *issp)), -abs(np.sum(vbz *issp)) ) * dt

        # Convert to actual value in Coulombs

        op.update({"zvs_qs1":(qs1_ab,qs1_ba),
                  "zvs_qs2":(qs2_ab,qs2_ba),
                  "zvs_qu": (qu_ab, qu_ba),
                  "zvs_qb": (qb_ab, qb_ba)})

        ######
        # Calculate current rms and interpolated "phase"
        iipk = np.max(iss)
        iirms = rms(iss)
        op.update({"iipk":iipk})
        op.update({"iirms":iirms})
        # Get the positive going zero crossing index
        iizx = np.diff( np.sign(np.concatenate( ([iss[-1]], iss) ) ) )
        iizx = np.transpose(iizx.nonzero())
        iizxp = iizx[iss[iizx]>0]
        # Calculate the coarse offset in the vector
        tibase = iizxp-1
        # Interpolate (linear) to get fractional offset increment
        x1, x2 = 0.0, 1.0
        y1, y2 = iss[iizxp-1], iss[iizxp]
        m = (y1-y2)/(x1-x2)
        b = y1
        tiinc = -b/m
        # Combine coarse offset and fractional estimation, and convert
        # it to radians
        tibase = tibase + tiinc
        ti = -float(tibase)/(len(iss)-1) * 2*pi - pi/2
        if ti<-pi: ti = ti+2*pi
        if ti> pi: ti = ti-2*pi
        op.update({"ti":ti})

    else:
        op = None

    return op


def iszvs_estimate(op, thresh=0.31416, modifyop=False):
    '''Quickly estimates whether an operating condition meets
       ZVS constraints, using the impedance approximation.'''
    valid = True
    pow = estimate_currents(op)
    ii = np.abs(pow[3])
    ti = np.angle(pow[3])

    ts2 = -pi/2*(1-op["ds"])
    ts1 = -pi/2*(1+op["ds"])
    tb = op["tb"]
    tu = op["tu"]

    if ((ti-ts2) > thresh): valid = False
    if ((ti-ts1) < -thresh): valid = False
    if ((ti-tb ) > thresh): valid = False
    if ((ti-tu ) < -thresh): valid = False

    if modifyop:
        op['iipk'] = abs(pow[3])
        op['iirms'] = abs(pow[4])

    return valid
```

### B.1.6 hepvmi/__init__.py

```
```

### B.1.7 hepvmi/optimize.py

```
from math import pi
import numpy as np
import hepvmi.parameterestimation as paramest

############################

def minimize_powererror(op0, printstatus=False, fullreturn=False, maxiter=50, ftol=1e-3,
        weights=[1,0,1]):
    powerfunc = paramest.calculate_currents
    ps = op0['ps']
    pu = op0['pu']
    pb = -op0['ps']-pu
    powErrFunc = lambda p: weights[0]*abs(p[0]-ps) + \
                           weights[1]*abs(p[1]-pb) + \
                           weights[2]*abs(p[2]-pu)
    objFunc = lambda o: powErrFunc([ii*v for (ii,v) in zip(powerfunc(o),[op0['vi'],op0['vc
        '],op0['vl']])] )

    dxy = op0['fsw']/50e6*2*pi

    ## For the first step chioce, we evaluate the current and all adjacent steps
    Ax = np.array([[-1, 0, 1],[-1, 0, 1],[-1, 0, 1]]) #np.array([-1,0,1]*3).reshape(3,3)
    Ay = np.array([[-1, -1, -1],[ 0, 0, 0],[ 1, 1, 1]]) #Ax.transpose()
    # reshape the array to a list for ease of use
    Ab = Ax.reshape(1,9)[0]
    Au = Ay.reshape(1,9)[0]

    # make a copy of the starting point
    bestop = dict(op0)
    orig = 4

    if printstatus: print "Rounding angles to nearest multiple of resolution: %f" % (dxy,)
    bestop['tb'] = round(bestop['tb']/dxy)*dxy
    bestop['tu'] = round(bestop['tu']/dxy)*dxy

    pows = powerfunc(bestop)
    bestobj = powErrFunc(pows)
    if printstatus: print "Ideal powers:", [ps,pb,pu]
    if printstatus: print "Current powers:", pows[0:3]
    if printstatus: print "Starting at: %.4f" % (bestobj,)

    finished = 0
    iter = 0

    while True:
        # Create the list of steps to evaluate
        opup = [dict({'tb':atb*dxy+bestop['tb'], 'tu':atu*dxy+bestop['tu']}) for (atb,atu)
            in zip(Ab,Au)]
        # Evaluate each step
        evalops = [dict(bestop.items()+dop.items()) for dop in opup]
        objvals = [objFunc(o) for o in evalops]

        # Make a copy of the step that gives the lowest objective
        curridx = np.argmin(objvals,0)
        currobj = objvals[curridx]
        currop = evalops[curridx]

        # Test for stopping conditions
        if currobj >= bestobj: finished = 1 # Finished normally
        if abs(currobj-bestobj) < ftol: finished = 2 # No more steps below tolerance
        if iter > maxiter: finished = 3 # Exceeded iterations
        if abs(currobj)>np.max(np.abs([ps,pb,pu]))/2: finished = 4 # Initial error is too
            high

        if finished:
            if printstatus: print "Finished (",finished,")."
            if fullreturn:
                # return (op,{'retval':finished,'iter':iter})
                return (bestop, bestobj, finished, iter)
            else:
                return bestop
        else:
            dab = Ab[curridx]
            dau = Au[curridx]
            if printstatus: print "(%d) Improved objective %.4f->%.4f ... [tb: %f (%+.4f),
                tu: %f (%+.4f)]" % (iter, bestobj,currobj, bestop['tb'],dab*dxy, bestop['tu
                '],dau*dxy)
            iter = iter + 1
            # Create new list of steps to evaluate based on the most recent step
            if dab==0:
                Au = [dau, dau, dau]
                Ab = [-1,0,1]
            elif dau==0:
                Ab = [dab, dab, dab]
                Au = [-1,0,1]
            else:
                Ab = [ dab, dab, dab, 0 , -dab]
                Au = [-dau, 0, dau, dau, dau]
                #Ab = [ dab, dab, 0 ]
                #Au = [ 0, dau, dau ]

            Ab = np.array(Ab)
            Au = np.array(Au)
            bestop =dict(currop)
            bestobj=currobj
```

### B.1.8 hepvmi/parameterestimation.py

```
#!/usr/bin/env python

from math import pi
from math import sqrt
import math
import cmath
import numpy as np
import scipy.optimize as optim
import matplotlib.pyplot as plt

from converteroperation import *
from contourintersection import *

def rms(data, axis=0):
    return np.sqrt(np.mean(data ** 2, axis))

def estimate_currents(vars):
    fsw = vars["fsw"]
    ds = vars["ds"]
    tb = vars["tb"]
    tu = vars["tu"]
    vi = vars["vi"]
    vc = vars["vc"]
    vl = vars["vl"]
    R = vars["R"]
    L = vars["L"]
    C = vars["C"]

    w = 2*pi*fsw
    # Resonant Tank Impedance
    zt = 1j*w*L + 1/(1j*w*C) + R
    # Generate Voltages
    db = du = 0.5
    vs = 4/pi*(vi )*np.sin(pi*ds/2) * np.exp(1j* 0)
    vb = 4/pi*(vc/2)*np.sin(pi*db ) * np.exp(1j*tb)
    vu = 4/pi*(vl/2)*np.sin(pi*du ) * np.exp(1j*tu)
    vt = vs+vb-vu
    # Generate Current
    ii = 1/(zt)*vt

    # Define Constraint Equations
    ias = (1/2.)*( vs*(-ii.conjugate()) ).real/(vi)
    iab = (1/2.)*( vb*(-ii.conjugate()) ).real/(vc)
    iau = (1/2.)*( vu*( ii.conjugate()) ).real/(vl)

    # Alternative Formulation, equivalent results
    #z = cmath.polar(zt)[0]
    #tz = cmath.polar(zt)[1]
    #vi = 4/pi*vi *np.sin(pi*ds/2)
    #vc = 4/pi*vc/2*np.sin(pi*db )
    #vl = 4/pi*vl/2*np.sin(pi*du )
    #ps = (1/2.)* vi*(np.cos(tz-tu )*vl-np.cos(tz )*vi-np.cos(tz-tb )*vc)/(z)
    #pu = (1/2.)*-vl*(np.cos(tz )*vl-np.cos(tz+tu)*vi-np.cos(tz+tu-tb)*vc)/(z)
    #pb = (1/2.)* vc*(np.cos(tz-tu+tb)*vl-np.cos(tz+tb)*vi-np.cos(tz )*vc)/(z)

    return [ias, iab, iau, ii, ii/sqrt(2)]

def calculate_currents(vars):
    opp = OperatingPoint(valdict=vars)
    opp.generate_model()
    opp.generate_waveforms()
    convmodel = opp.model
    convwaves = opp.waveforms

    vt = convwaves.waves["vt"]
    vvs = convwaves.waves["vs"]/opp.params['vi']
    vvb = convwaves.waves["vb"]/opp.params['vc']
    vvu = convwaves.waves["vu"]/opp.params['vl']
    iss = convwaves.waves["is"]

    ni = iss.shape[0]
    ias = sum(vvs*(-iss))/ni
    iab = sum(vvb*(-iss))/ni
    iau = sum(vvu*( iss))/ni
    pt = sum(vt*( iss))/ni


    ## Get the positive going zero crossing index
    iizx = np.diff( np.concatenate( ([iss[-1]], iss) ) )
    iizx = np.transpose(iizx.nonzero())
    if len(iizx)>2:
        iipk = np.NAN
        iirms = np.NAN
    else:
        iipk = np.max(iss)
        iirms = rms(iss)

    # else:
    # ### Calculate current rms and interpolated "phase"
    # iiv= iss
    # ii = rms(iiv)

    # iizxp = iizx[iiv[iizx]>0]
    # # Calculate the coarse offset in the vector
    # tibase = iizxp-1
    # # Interpolate (linear) to get fractional offset increment
    # x1, x2 = 0.0, 1.0
    # y1, y2 = iiv[iizxp-1], iiv[iizxp]
    # m = (y1-y2)/(x1-x2)
    # b = y1
    # tiinc = -b/m
    # # Combine coarse offset and fractional estimation, and convert
    # # it to radians
    # tibase = tibase + tiinc
    # ti = -float(tibase)/(len(iiv)-1) * 2*pi

    #iii = ii*math.sqrt(2)*cmath.exp(1j*ti)
    return [ias, iab, iau, iipk, iirms]

def calculate_current_array(vars, debugprint=False):
    # ## "calculated power" equivalent of "estimated power" with matrix angles
    # ## for
    tba = vars['tb']
    tua = vars['tu']

    op = dict(vars)
    ias = np.zeros(tba.shape)
    iab = np.zeros(tba.shape)
    iau = np.zeros(tba.shape)
    iipk = np.zeros(tba.shape)
    iirms = np.zeros(tba.shape)
    if debugprint: print ps.shape
    for x in xrange(tba.shape[0]):
        for y in xrange(tba.shape[1]):
            if debugprint: print "p",
            op.update({"tb":tba[x,y], "tu":tua[x,y]})
            ias[x,y], iab[x,y], iau[x,y], iipk[x,y], iirms[x,y] = calculate_currents(op)

    return [ias, iab, iau, iipk, iirms]

def remap_domain(values=np.array([]), dom=[0,0], npoints=1):
    return values*(dom[1]-dom[0])/float(npoints)+dom[0]

def estimate_angles_intersection(opvars, N=101, thresh=10, domain=[-pi,pi,-pi,pi], samples
     =[], recurse=True, recurse_res=0.01, use_waveforms=False, debugprint=False):
## print "entering estimate_angles:", N, thresh, domain, samples, recurse, recurse_res

    if (len(samples)!=2):
        samples = [N, N]

    Nx = samples[0]
    Ny = samples[1]
    domainx = [domain[0],domain[1]]
    domainy = [domain[2],domain[3]]

    # Setup the domain to brute force
    domx = np.linspace(domainx[0], domainx[1],Nx)
    domy = np.linspace(domainy[0], domainy[1],Ny)
    t0a = np.outer(np.ones(domy.shape), domx) #x
    t1a = np.outer(domy, np.ones(domx.shape)) #y

    # Duplicate the operating params to assign the search domain
    # Estimate the resulting power; returns matrices with full result
    op = opvars
    op.update({"tu":t1a, "tb":t0a})

    if debugprint: print "Mapping domain"
    if use_waveforms:
        if debugprint: print "Using PWL waveforms"
        ias, iab, iau, iipk, iirms = calculate_current_array(op)
    else:
        if debugprint: print "Using sinusoidal approximation"
        ias, iab, iau, iipk, iirms = estimate_currents(op)

    # Find the intersection estimates: returns a list of bounds for each
    if debugprint: print "Estimating Contour Intersection"
    xyclusters = estimate_contour_intersection(iab*op['vc'], op["pb"], iau*op['vl'], op["pu
        "], thresh=thresh)

    # Now step through each of the clusters and see if the resolution
    # requirement has been met, or if we're recursing without progress
    # * this could be put into the above nested loop, but brought out for clarity
    if debugprint: print "Processing %d clusters" % len(xyclusters)
    isectsdom = []
    for cl in xyclusters:
        # Calculate the uncertainty in the original domain
        pminx, pmaxx = remap_domain(np.array((cl[0]-1,cl[1]+1)),domainx, Nx)
        pminy, pmaxy = remap_domain(np.array((cl[2]-1,cl[3]+1)),domainy, Ny)
        reserrx = abs(pminx-pmaxx)
        reserry = abs(pminy-pmaxy)
        reserr = max(reserrx, reserry)

        # Set a new domain that bounds the cluster, and check it against
        # the previous domain to ensure that we're making progress
        newdomain = [pminx, pmaxx, pminy, pmaxy]
        domdelta = abs(np.array(domain)-np.array(newdomain))
        domdelta = max(domdelta)

        # If either of the errors is greater than the requested resolution,
        # and recursion is requested, shrink the search domain around this
        # new domain and try again
        #print "Reserr:",reserr, "Domdelta:", domdelta, "Recurse:", recurse
        if (reserr>recurse_res) and (domdelta>recurse_res) and (recurse==True):
            if debugprint: print "Recursing (resx=%.2e, resy=%.2e, delta=%.2e)" % (reserrx,
                reserry, domdelta)
            isects = estimate_angles_intersection(opvars, domain=newdomain, samples=[Nx,Ny],
                                            recurse=recurse, recurse_res=recurse_res,
                                            use_waveforms=use_waveforms, debugprint=
                                                debugprint)
            for i in isects:
                isectsdom.append(list(i))
        else:
            # Accept the uncertainty of of this intersection, and take the
            # point at the center of the intersection bounds
            isectdom = [ (pminx+pmaxx)/2.0, (pminy+pmaxy)/2.0 ]
            isectsdom.append(isectdom)
            if debugprint: print "Found solution:", isectdom
            #print "Found Solution:", isectdom
    # With all clusters investigated and intersects collected, convert to an
    # array and return. If the result list was empty, return a zero length
    # 2d array
    isectsdom = np.array(isectsdom)
    if min(isectsdom.shape)==0: isectsdom = np.zeros((0,2))
    return isectsdom

def estimate_angles_intersection_wrap(x, debugprint, use_waveforms):
    #print "**Estimating Angle"
    return estimate_angles_intersection(x, domain=[-pi,pi,-pi,pi], N=21,
                                        recurse_res=0.02, recurse=True,
                                        debugprint=debugprint, use_waveforms=use_waveforms)

def get_angles(op, usewaves=False, debugprint=False, printstatus=True):
    finalops = []

    #if statusprint: print ".", #dict(op).setdefault('serial',0),
    if printstatus: print "\n** initial op\n", op
    #Estimate the angles through intersection method
    # Returns a list of arrays
    #-------------------------------- print "Initial intersection estimation..."
    isolns = estimate_angles_intersection_wrap(dict(op), debugprint=debugprint,
        use_waveforms=usewaves)
    if printstatus: print "** initial solns: ", [np.array(x) for x in isolns]

    #---------------------------- print "Creating full operating point list..."
    # Put the all estimated solutions into complete operating points
    allops = []
    for sol in isolns:
        newop = dict(op)
        newop.update({"tb": sol[0], "tu": sol[1]})
        allops.append(newop)

    return allops
```

## B.1.9 extract-ops.py

```
import operator
import os.path
from math import pi
import hepvmi.idealzvs as zvs
import hepvmi.compensation as comp
import mosfet_parameters as mosdev
import pickle
import loadsave
import sys

```

```
saveAsPickle = True
saveAsOps = False

fb_mosfet = mosdev.BSC042NE7NS3
bb_mosfet = mosdev.STx13NM60
cc_mosfet = mosdev.STx13NM60

delays = {}
delays["com"] = 40e-9
delays["s1"] = 80e-9
delays["s2"] = 80e-9
delays["u"] = 180e-9
delays["b"] = 180e-9


def isgoodzvs2(op):
    # old function, but semi-valid, used as comparison agains isgoodzvs for sanity
    s1c, s2c = 2e-9/36, 2e-9/36
    s1qv_func = lambda v: s1c*v
    s2qv_func = lambda v: s2c*v
    uqv_func = lambda v: (v<50) and (3.043e-9*v**0.585) or ( (1.026e-9*(v-50)**1.375)/v +
        1.5e-6/50 )
    bqv_func = lambda v: uqv_func(v)

    qs1 = s1qv_func(op["vi"]/op['N'])
    qs2 = s2qv_func(op["vi"]/op['N'])
    qu = uqv_func(op["vl"])
    qb = bqv_func(op["vc"])

    goodzvs = True
    goodzvs = goodzvs and ( qs1 < abs(min(op["zvs_qs1"])) )
    goodzvs = goodzvs and ( qs2 < abs(min(op["zvs_qs2"])) )
    goodzvs = goodzvs and ( qu < abs(min(op["zvs_qu" ])) )
    goodzvs = goodzvs and ( qb < abs(min(op["zvs_qb" ])) )

    return goodzvs

def isgoodzvs(op):
    qs1 = 2*mosdev.get_output_charge( device=fb_mosfet, voltage=op["vi"]/op['N'] ) / op["N"]
    qs2 = 2*mosdev.get_output_charge( device=fb_mosfet, voltage=op["vi"]/op['N'] ) / op["N"]
    qu = 2*mosdev.get_output_charge( device=cc_mosfet, voltage=op["vl"] )
    qb = 2*mosdev.get_output_charge( device=bb_mosfet, voltage=op["vc"] )

    goodzvs = True
    goodzvs = goodzvs and ( qs1 < abs(min(op["zvs_qs1"])) )
    goodzvs = goodzvs and ( qs2 < abs(min(op["zvs_qs2"])) )
    goodzvs = goodzvs and ( qu < abs(min(op["zvs_qu" ])) )
    goodzvs = goodzvs and ( qb < abs(min(op["zvs_qb" ])) )

    return goodzvs


#---------------------------------------------------------------------------
```

```


# Load the data
#zvsops = pickle.load(file('zvsops.pickle'))
if len(sys.argv) == 2:
    filename = sys.argv[1]
else:
    sys.exit(2)

# define the voltage-charge functions
qfuncs = {}
qfuncs["s1qoss_func"] = lambda v: 2*mosdev.calculate_qoss( device=mosdev.BSC042NE7NS3,
     voltage=v)
qfuncs["s2qoss_func"] = lambda v: 2*mosdev.calculate_qoss( device=mosdev.BSC042NE7NS3,
     voltage=v)
qfuncs[ "uqoss_func"] = lambda v: 1.75*mosdev.get_output_charge( device=mosdev.STx13NM60,
     voltage=v)
qfuncs[ "bqoss_func"] = lambda v: 1.75*mosdev.get_output_charge( device=mosdev.STx13NM60,
     voltage=v)

#zvsops = pickle.load(file('200w32v15d0.pickle'))
zvsops = pickle.load(file(filename))
#zvsops = [zvs.calculate_zvsmargins(o) for o in zvsops if o]
zvsops = [comp.calculate_deadtime(o, mindt=30e-9, **qfuncs) for o in zvsops if o]
zvsops = [comp.calculate_lag(o,delays=delays) for o in zvsops if o]


badzvs = [x for x in zvsops if not isgoodzvs(x)]
goodzvs = [x for x in zvsops if isgoodzvs(x)]

# not necessary, but useful for comparison with bad/good zvs
badzvs2 = [x for x in zvsops if not isgoodzvs2(x)]
goodzvs2 = [x for x in zvsops if isgoodzvs2(x)]


#sort the good ops by the quality factor, and select the top few
goodzvs.sort(key=lambda x: x['Q'], reverse=False)
bestQops = []#goodzvs[:20]
goodzvs.sort(key=lambda x: x['iirms'], reverse=False)
bestIops = goodzvs[:50]

#write out the top oppoints to separate files, ready to be used
fname = os.path.split(filename)[1]
if saveAsPickle:
    pickle.dump(bestQops+bestIops,open(fname[:11]+"_x.pickle" , "w" ))

if saveAsOps:
    for n,o in enumerate(bestQops+bestIops):
        loadsave.savevar(filename[:10]+str(n)+'.op', o)
```

### B.1.10 gen_sweeplist.py

```

def gen_listpermutations(varlist, sweepvarlist, n):
    '''
    Takes two lists of tuples (from a dict().items() for example),
    and then creates a generator that iterates over all possible
    combinations of the sweepvarlist entries.

    Example:
     Args: varlist=[('a',1)],
           sweepvarlist=[ ('b',xrange(2)), ('c', xrange(2))]
     Generator result: [[('a', 1), ('b', 0), ('c', 0)],
                        [('a', 1), ('b', 0), ('c', 1)],
                        [('a', 1), ('b', 1), ('c', 0)],
                        [('a', 1), ('b', 1), ('c', 1)]]
    '''
    # Check end-of-recursion for when we're done augmenting the list
    try:
        sweepvar, sweepvals = sweepvarlist[0]
    except IndexError, ie:
        yield varlist
    else:
        # Ensure that sweepvals is an iterable
        try:
            iter(sweepvals)
        except TypeError, te:
            sweepvals = [sweepvals]
        # For each value for the var, augment the list, and dive down
        # recursively
        for val in sweepvals:
            newvarlist = varlist + [(sweepvar,val)] + [('serial',n)]
            for x in gen_listpermutations(newvarlist, sweepvarlist[1:],n):
                n += 1
                yield x+[('serial',n)]


def opp_expander(op):
    listperms = gen_listpermutations(op.items(),op.items(),0)
    return [ dict(perm) for perm in listperms ]
```

### B.1.11 loadsave.py

```
import json

def loadvar(file):
    return json.load(open(file,'r'))

def savevar(file, var):
    json.dump(var, open(file,'w'), indent=1)
```

```

def loadsave(file, var=None):
    if var==None:
        return loadvar(file)
    else:
        savevar(file, var)
```

### B.1.12 mosfet_parameters.py

```
# MOSFET capacitance parameters specified in pF (1e-12) units

STx13NM60 = {}
STx13NM60['_name'] = "STx13NM60"
STx13NM60['_desc'] = ""
STx13NM60['c_oss'] = [5500, 3800, 1500, 950, 400, 61, 55, 49, 44, 38, 33, 32, 31, 30 ]
STx13NM60['v_oss'] = [0, 1, 10, 20, 30, 40, 60, 80, 100, 200,300, 400, 500, 600]

IPP60R250CP = {}
IPP60R250CP['_name'] = "IPP60R250CP"
IPP60R250CP['_desc'] = ""
IPP60R250CP['c_oss'] = [8000, 2500, 1500, 1000, 120, 75, 58, 50, 45, 42, 40, 39 ]
IPP60R250CP['v_oss'] = [0, 12.5, 25, 37.5, 50, 75, 100, 125, 150, 200, 350, 500]

BSC042NE7NS3 = {}
BSC042NE7NS3['_name'] = "BSC042NE7NS3"
BSC042NE7NS3['_desc'] = ""
BSC042NE7NS3['c_oss'] = [3500, 2000, 1700, 1400, 950, 700, 550, 500]
BSC042NE7NS3['v_oss'] = [0, 5, 10, 15, 30, 45, 60, 75 ]

PSMN8R5 = {}
PSMN8R5['_name'] = "PSMN8R5"
PSMN8R5['_desc'] = ""
PSMN8R5['c_oss'] = [2000, 1100, 580, 430, 350, 310, 280, 260]
PSMN8R5['v_oss'] = [0, 1, 5, 10, 20, 30, 40, 50 ]

PSMN5R5 = {}
PSMN5R5['_name'] = "PSMN8R5"
PSMN5R5['_desc'] = ""
PSMN5R5['c_oss'] = [2900, 1900, 830, 650, 510, 440, 410, 400]
PSMN5R5['v_oss'] = [0, 1, 5, 10, 20, 30, 40, 50 ]

#----------------------------------------------------------------------------
def calculate_qoss(device=None, C=None, V=None, voltage=0):
    return get_output_charge(device=device, C=C, V=V, voltage=voltage)

def get_output_charge(device=None, C=None, V=None, voltage=0):
    # If the device is specified, pull out the parameters
    if device:
        C = device['c_oss']
        V = device['v_oss']

    # Check to see if cap and volt vectors were passed
    if (C==None) or (V==None):
        return -1
    # Check if
    if (voltage>V[-1]): return -1


    ## Integrate the complete segments first
    i = 0
    Q = 0
    while (voltage>V[i+1]):
        # Trapezoidal Integration for single step
        Qtmp1 = (V[i+1]-V[i])*min(C[i+1],C[i]) # rectangular base
        Qtmp2 = 0.5*(V[i+1]-V[i])*abs(C[i+1]-C[i]) # triangle top
        Q = Q+Qtmp1+Qtmp2
        i = i+1

    ## integrate remaining fractional portion
    # perform interpolation
    m = (C[i+1]-C[i]) / float(V[i+1]-V[i])
    b = C[i]
    Vend = voltage
    Cend = m*(voltage-V[i])+b

    Qtmp1 = (Vend-V[i])*min(Cend,C[i]) # rectangular base
    Qtmp2 = 0.5*(Vend-V[i])*abs(Cend-C[i]) # triangle top
    Q = Q+Qtmp1+Qtmp2

    return Q*1e-12
```

### B.1.13 sweep-main.py

```

import sys
import time
import pickle
import numpy as np
import hepvmi.parameterestimation as paramest
import hepvmi.converteroperation as conv
import hepvmi.idealzvs as zvs
import hepvmi.optimize as optim
from gen_sweeplist import *
from math import pi

#from hepvmi.converteroperation import OperatingPoint
#from hepvmi.operatingpoints import simop

def get_angles_wrap(x):
    try:
        #print x['serial'], "",
        rval = paramest.get_angles(x, usewaves=False, debugprint=False, printstatus=False)
        return rval
    except:
        print "************* EXCEPTION *************"
        print te
  print " ****************************"
  print x
        print "************************************"
        return []

def fmin_wrap(x):
    return optim.minimize_powererror(x,fullreturn=True,printstatus=False)

#------------------------------------------------------------------------------
#------------------------------------------------------------------------------
#------------------------------------------------------------------------------
def usage():
    sys.stderr.write("\n")
    sys.stderr.write("Usage: <thisscript.py> [file]\n")
    sys.stderr.write("\n")
    sys.stderr.write("[file] defines the variables for the sweep, using python syntax.\n")
    sys.stderr.write("An example [file] contents would be:\n")
    sys.stderr.write("\n")

    sys.stderr.write(" outputfilename = '200w32v45d0.pickle'\n")
    sys.stderr.write(" N = 6.05\n")
    sys.stderr.write(" op = {\n")
    sys.stderr.write(" 'C': 23.93e-09,\n")
    sys.stderr.write(" 'L': 169.2e-6,\n")
    sys.stderr.write(" 'R': 1.5,\n")
    sys.stderr.write(" 'N': N,\n")
    sys.stderr.write(" 'ds': 0.8,\n")
    sys.stderr.write(" 'vc': 170.0,\n")
    sys.stderr.write(" 'vi': np.array([32])*N,\n")
    sys.stderr.write(" 'fsw': np.arange(100e3, 500e3, 5e3),\n")
    sys.stderr.write(" 'ds': np.arange(0.6, 1.0, .05),\n")
    sys.stderr.write(" 'tl': np.array([45])*pi/180,\n")
    sys.stderr.write(" 'pavg': np.array([200]) }\n")
    sys.stderr.write("\n")
    sys.exit(2)


if __name__ == "__main__":
    USESMP = True
    SAVE_RESULTS = True
    SAVE_WITH_PICKLE = True
    printstatus = True
    printdebug = False

    REDUCE_SET_USING_ZVS = False
    REDUCE_SET_USING_Q = False
    MAXIMUM_Q = 10
    CONVERGE_USING_PWL = False
    CALCULATE_ZVS_MARGINS = False

    #sys.setrecursionlimit(50000)

    if len(sys.argv) != 2:
        usage()
    else:
        outputfilename = None
        op = None
        # the execfile loads the variables from the file
        execfile(sys.argv[1])
        if (not outputfilename) or (not op):
            usage()

    # outputfilename = "200w32v45d0.pickle"
    # op = \
    #  {'C': 23.93e-09,
    #  'L': 169.2e-6,
    #  'R': 1.5,
    #  'N': 6.0,
    #  'ds': 0.8,
    #  'vc': 170.0,
    #  'vi': np.array([32])*6.05,
    #  'fsw': np.arange(100e3, 500e3, 5e3),
    #  'ds': np.arange(0.6, 1.0, .05),
    #  'tl': np.array([45])*pi/180,#np.arange(1e-12,91,10)*pi/180,
    #  'pavg': np.array([200])
    #  }

    numops = 1
    for v in op.values():
        try:
            numops = numops * len(v)
        except TypeError, te:
            pass
    if printstatus: print "%d Operating points will be constructed" % numops


#----------------------------------------------------------------------------
#----------------------------------------------------------------------------
    if USESMP:
        from multiprocessing import Pool, cpu_count
        cpus = cpu_count()
        p = Pool(processes=cpus)
        if printstatus: print "Using SMP with %d CPUs" % cpus
        mymap = lambda x,y: p.map(x,y,chunksize=numops/128+1)
    else:
        mymap = map
#----------------------------------------------------------------------------
    timefirststart = time.time()
#----------------------------------------------------------------------------

```

```
    if printstatus: print "Constructing operating points ...",
    timestart = time.time()
    allops = opp_expander(op)
    allops = [ dict( o.items()+conv.gen_oppoint(linepos=o["tl"],theta=o.get('theta',0.0),
         vlpk=340, pavg=o["pavg"]).items() )
               for o in allops ]
    if printstatus: print "Time elapsed:", (time.time()-timestart)
    if printstatus: print " ** Initial points: ", len(allops)

#--------------------------------------------------------------------------------
    if printstatus: print "Calculating Results ...",
    timestart = time.time()
    if len(allops)>0:
        allops = mymap(get_angles_wrap,allops)
    else:
        allops = []
    if printstatus: print "Time elapsed:", (time.time()-timestart)

#--------------------------------------------------------------------------------
    # Flatten the list of lists that are generated, ignoring the empty ones;
    flatten = lambda it: [ y for x in it for y in x if x ]
    #if printstatus: print "Flattening Results ...",
    #timestart = time.time()
    allops = flatten(allops)
    if printstatus: print " ** Initial solutions:", len(allops)
    #if printstatus: print "Time elapsed:", (time.time()-timestart)
    zvsops = allops

#--------------------------------------------------------------------------------
    if REDUCE_SET_USING_ZVS:
  timestart = time.time()
        if printstatus: print "Reducing solution set with ZVS approximation ...",
  if len(allops)>0:
            zvsstatus = mymap(zvs.iszvs_estimate,allops)
      zvsops = [op for (op, iszvs) in zip(allops,zvsstatus) if iszvs]
  else:
            zvsops = []

        if printstatus: print "Time elapsed:", (time.time()-timestart)
        if printstatus: print " ** Reduced solutions:", len(zvsops)

#--------------------------------------------------------------------------------
    if REDUCE_SET_USING_Q:
  timestart = time.time()
        if printstatus: print "Reducing solution set by limiting loaded Q ...",
        maxQ = MAXIMUM_Q
        if len(zvsops)>0:
            filtops = []
            for op in zvsops:
                vals = paramest.estimate_currents(op)
          iipk = abs(vals[3])
                Q = 2*pi*op['fsw']/2*op['L']*iipk**2/max([op['pu'],op['pavg']])
          if Q<maxQ:
```

```
                    filtops.append(op)
        op['Q'] = Q
          zvsops = filtops
        else:
            zvsops = []

        if printstatus: print "Time elapsed:", (time.time()-timestart)
        if printstatus: print " ** Reduced solutions:", len(zvsops)

#---------------------------------------------------------------------------------------
    if CONVERGE_USING_PWL:
        if printstatus: print "Converging Angles to PWL Solutions ...",
        timestart = time.time()
        if len(zvsops)>0:
            zvsopsfull = mymap(fmin_wrap,zvsops)
            for (op,rank,code,iter) in zvsopsfull:
                op.update({'optim_eval':rank, 'optim_cond':code, 'optim_iters':iter})
            zvsops = zip(*zvsopsfull)[0]
        else:
            zvsops = []

        if printstatus: print "Time elapsed:", (time.time()-timestart)

#---------------------------------------------------------------------------------------
    if CALCULATE_ZVS_MARGINS:
        if printstatus: print "Calculating ZVS Margins ...",
  timestart = time.time()
  if len(zvsops)>0:
            zvsops = mymap(zvs.calculate_zvsmargins, zvsops)
  else:
            zvsops = []
        zvsops = [o for o in zvsops if o]
        #for i in range(zvsops.count(None)):
        # zvsops.remove(None)

  if printstatus: print "Time elapsed:", (time.time()-timestart)
  if printstatus: print " ** Valid ZVS solutions:", len(zvsops)

#---------------------------------------------------------------------------------------
    if SAVE_RESULTS:
        timestart = time.time()
        if printstatus: print "Saving Results to: ", outputfilename
        if SAVE_WITH_PICKLE:
            if printstatus: print "Using Pickle..."
            pickle.dump(zvsops,open( outputfilename, "w" ))
        else:
            f = open(outputfilename,"w")
            for o in zvsops: f.write(o.__repr__()+'\n')
            f.close()
        if printstatus: print "Time elapsed:", (time.time()-timestart)

#---------------------------------------------------------------------------------------
    if printstatus: print "Finished:", (time.time()-timefirststart)
```

Appendix C

# Digital Control Hardware Code

## C.1 FPGA PWM Implementation

### C.1.1 clocking.v

```
////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 1995-2009 Xilinx, Inc. All rights reserved.
////////////////////////////////////////////////////////////////////////////////
// ____ ____
// / /\/ /
// /___/ \ / Vendor: Xilinx
// \ \ \/ Version : 11.1
// \ \ Application : xaw2verilog
// / / Filename : clocking.v
// /___/ /\ Timestamp : 08/31/2010 11:29:58
// \ \ / \
// \___\/\___\
//
//Command: xaw2verilog -intstyle Z:/fpga-20100830/hepvmi-pwm-controller/ipcore_dir/clocking.xaw -st clocking.v
//Design Name: clocking
//Device: xc3s700a-4fg484
//
// Module clocking
// Generated by Xilinx Architecture Wizard
// Written for synthesis tool: XST
`timescale 1ns / 1ps

module clocking(CLKIN_IN,
                RST_IN,
                CLKIN_IBUFG_OUT,
                CLK0_OUT,
                CLK2X_OUT,
                LOCKED_OUT);

    input CLKIN_IN;
    input RST_IN;
   output CLKIN_IBUFG_OUT;
   output CLK0_OUT;
   output CLK2X_OUT;
   output LOCKED_OUT;

   wire CLKFB_IN;
   wire CLKIN_IBUFG;
   wire CLK0_BUF;
   wire CLK2X_BUF;
   wire GND_BIT;

   assign GND_BIT = 0;
   assign CLKIN_IBUFG_OUT = CLKIN_IBUFG;
   assign CLK2X_OUT = CLKFB_IN;
   IBUFG CLKIN_IBUFG_INST (.I(CLKIN_IN),
                          .O(CLKIN_IBUFG));
   BUFG CLK0_BUFG_INST (.I(CLK0_BUF),
                       .O(CLK0_OUT));
   BUFG CLK2X_BUFG_INST (.I(CLK2X_BUF),
                        .O(CLKFB_IN));
   DCM_SP DCM_SP_INST (.CLKFB(CLKFB_IN),
                      .CLKIN(CLKIN_IBUFG),
                      .DSSEN(GND_BIT),
                      .PSCLK(GND_BIT),
                      .PSEN(GND_BIT),
                      .PSINCDEC(GND_BIT),
                      .RST(RST_IN),
                      .CLKDV(),
                      .CLKFX(),
                      .CLKFX180(),
                      .CLK0(CLK0_BUF),
                      .CLK2X(CLK2X_BUF),
                      .CLK2X180(),
                      .CLK90(),
                      .CLK180(),
                      .CLK270(),
                      .LOCKED(LOCKED_OUT),
                      .PSDONE(),
                      .STATUS());
   defparam DCM_SP_INST.CLK_FEEDBACK = "2X";
   defparam DCM_SP_INST.CLKDV_DIVIDE = 2.0;
   defparam DCM_SP_INST.CLKFX_DIVIDE = 1;
   defparam DCM_SP_INST.CLKFX_MULTIPLY = 4;
   defparam DCM_SP_INST.CLKIN_DIVIDE_BY_2 = "FALSE";
   defparam DCM_SP_INST.CLKIN_PERIOD = 20.000;
   defparam DCM_SP_INST.CLKOUT_PHASE_SHIFT = "NONE";
   defparam DCM_SP_INST.DESKEW_ADJUST = "SYSTEM_SYNCHRONOUS";
   defparam DCM_SP_INST.DFS_FREQUENCY_MODE = "LOW";
   defparam DCM_SP_INST.DLL_FREQUENCY_MODE = "LOW";
   defparam DCM_SP_INST.DUTY_CYCLE_CORRECTION = "TRUE";
   defparam DCM_SP_INST.FACTORY_JF = 16'hC080;
   defparam DCM_SP_INST.PHASE_SHIFT = 0;
   defparam DCM_SP_INST.STARTUP_WAIT = "FALSE";
endmodule
```

### C.1.2 counters.v

```
module counter_ce_sc(CLK, CLR, CE, Q);

parameter WIDTH = 16;

input CLK, CE, CLR;
output [WIDTH-1:0] Q;
reg [WIDTH-1:0] Q;

  always @(posedge CLK)
  begin
    if (CLR)
      Q <= 16'b0;
    else
      if (CE)
        Q <= Q + 1'b1;
    end

endmodule
```

### C.1.3 hexascii.v

```
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 15:42:18 08/04/2009
// Design Name:
// Module Name: hex2ascii
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module bin2asciihex(
    input [3:0] bin,
    output reg [7:0] ascii
    );

  // Continuous decoding of binary values into hex chars
  always @(bin)
  begin
```

```
    case (bin)
      4'h0: ascii = 8'h30;
      4'h1: ascii = 8'h31;
      4'h2: ascii = 8'h32;
      4'h3: ascii = 8'h33;
      4'h4: ascii = 8'h34;
      4'h5: ascii = 8'h35;
      4'h6: ascii = 8'h36;
      4'h7: ascii = 8'h37;
      4'h8: ascii = 8'h38;
      4'h9: ascii = 8'h39;
      4'hA: ascii = 8'h41;
      4'hB: ascii = 8'h42;
      4'hC: ascii = 8'h43;
      4'hD: ascii = 8'h44;
      4'hE: ascii = 8'h45;
      4'hF: ascii = 8'h46;
      default: ascii = 8'h3F; // "?"
    endcase
  end


endmodule

module asciihex2bin(
  input [7:0] ascii,
  output reg ishexchar,
  output reg [3:0] bin
  );

// Continuous decoding of ascii hex chars into binary values
always @(ascii)
  case (ascii)
    8'h30: bin = 4'h0;
    8'h31: bin = 4'h1;
    8'h32: bin = 4'h2;
    8'h33: bin = 4'h3;
    8'h34: bin = 4'h4;
    8'h35: bin = 4'h5;
    8'h36: bin = 4'h6;
    8'h37: bin = 4'h7;
    8'h38: bin = 4'h8;
    8'h39: bin = 4'h9;
    8'h41: bin = 4'hA;
    8'h42: bin = 4'hB;
    8'h43: bin = 4'hC;
    8'h44: bin = 4'hD;
    8'h45: bin = 4'hE;
    8'h46: bin = 4'hF;
    default: bin = 4'h0;
  endcase

  // Continuous decoding of ascii hex chars into binary values
  always @(ascii)
    case (ascii)
      8'h30: ishexchar = 1'b1;
      8'h31: ishexchar = 1'b1;
      8'h32: ishexchar = 1'b1;
      8'h33: ishexchar = 1'b1;
      8'h34: ishexchar = 1'b1;
      8'h35: ishexchar = 1'b1;
      8'h36: ishexchar = 1'b1;
      8'h37: ishexchar = 1'b1;
      8'h38: ishexchar = 1'b1;
      8'h39: ishexchar = 1'b1;
      8'h41: ishexchar = 1'b1;
      8'h42: ishexchar = 1'b1;
      8'h43: ishexchar = 1'b1;
      8'h44: ishexchar = 1'b1;
      8'h45: ishexchar = 1'b1;
      8'h46: ishexchar = 1'b1;
      default: ishexchar = 1'b0;
    endcase

endmodule
```

### C.1.4 input_protocol_decode.v

```
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10:31:18 07/30/2009
// Design Name:
// Module Name: input_protocol_decode
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module input_protocol_decode(
  input CLK,
  input CLR,
  input DIN_TICK,
  input [7:0] DIN,
  output reg CMDOUT, // low for read cmd, high for write
output reg [15:0] DOUT,
output reg [11:0] AOUT,
output reg DECODED
);

reg[3:0] state;
parameter IDLE=4'b0000,
      RWAIT=4'b0001,
           CMD=4'b1000,
         ADDR1=4'b1001,
      ADDR2=4'b1010,
      ADDR3=4'b1011,
         DATA1=4'b1100,
      DATA2=4'b1101,
      DATA3=4'b1110,
      DATA4=4'b1111;


// Continuous decoding of incoming data for CMD characters
parameter CMD_WR=1'b1;
parameter CMD_RD=1'b0;
reg iscmdchar;
reg decodedcmdchar;
always @(DIN)
begin
  case (DIN)
    8'h72: {iscmdchar,decodedcmdchar} = {1'b1,CMD_RD};
    8'h77: {iscmdchar,decodedcmdchar} = {1'b1,CMD_WR};
    default: {iscmdchar,decodedcmdchar} = {1'b0,1'b0};
  endcase
end

// Continuous decoding of incoming data for DATA/ADDR characters
wire ishexchar;
wire [3:0] decodedhexchar;
asciihex2bin input_decode(
  .ascii(DIN),
  .ishexchar(ishexchar),
  .bin(decodedhexchar)
);


// State transitions
// -> occurr only on DIN_TICK events (which are sync with CLK)
always @(posedge CLK or posedge CLR)
begin
  if (CLR == 1)
    state <= IDLE;
  else
    if (DIN_TICK)
      case(state)
        IDLE: if(iscmdchar) state <= CMD;
            else state <= IDLE;
```

```
          CMD: if(ishexchar) state <= ADDR1;
              else state <= IDLE;
          ADDR1: if(ishexchar) state <= ADDR2;
              else state <= IDLE;
          ADDR2: if(ishexchar) state <= ADDR3;
              else state <= IDLE;
          ADDR3: if(ishexchar) state <= DATA1;
              else state <= IDLE;
          DATA1: if(ishexchar) state <= DATA2;
              else state <= IDLE;
          DATA2: if(ishexchar) state <= DATA3;
              else state <= IDLE;
          DATA3: if(ishexchar) state <= DATA4;
              else state <= IDLE;
          DATA4: state <= IDLE;

          default state <= IDLE;
        endcase
      else
        case(state)
          ADDR3: if(CMDOUT==CMD_RD) state <= IDLE;
          DATA4: state <= IDLE;
          default: state <= state;
        endcase

end

// State Outputs
always @(posedge CLK)
    begin
    case(state)
      CMD: AOUT[11:8] <= decodedhexchar;
      ADDR1: AOUT[7:4] <= decodedhexchar;
      ADDR2:
        begin
          AOUT[3:0] <= decodedhexchar;
        end
      ADDR3:
        begin
          DOUT[15:12] <= decodedhexchar;
          DECODED <= (CMDOUT==CMD_RD) ? 1'b1 : 1'b0;
        end
      DATA1: DOUT[11:8] <= decodedhexchar;
      DATA2: DOUT[7:4] <= decodedhexchar;
      DATA3: DOUT[3:0] <= decodedhexchar;
      DATA4: DECODED <= 1'b1;
      default: // includes IDLE
        begin
          CMDOUT <= decodedcmdchar;
          AOUT <= AOUT;
          DOUT <= DOUT;
          DECODED <= 1'b0;
        end
      endcase
      end

/* // State Outputs
  always @(posedge CLK)
    if (state==CMD) CMDOUT <= decodedcmdchar;

  always @(state or decodedhexchar)
    if (state==ADDR1) AOUT[11:8] = decodedhexchar;
    else AOUT[11:8] = AOUT[11:8];

  always @(state or decodedhexchar)
    if (state==ADDR2) AOUT[7:4] = decodedhexchar;
    else AOUT[7:4] = AOUT[7:4];

  always @(state or decodedhexchar)
    if (state==ADDR3) AOUT[3:0] = decodedhexchar;
    else AOUT[3:0] = AOUT[3:0];

  always @(state)
    if (state==RWAIT | state==DATA4) DECODED = 1'b1;
    //else DECODED = 1'b0;

  always @(state or decodedhexchar)
    if (state==DATA1) DOUT[15:12] = decodedhexchar;
    else DOUT[15:12] = DOUT[15:12];

  always @(state or decodedhexchar)
    if (state==DATA2) DOUT[11:8] = decodedhexchar;
    else DOUT[11:8] = DOUT[11:8];

  always @(state or decodedhexchar)
    if (state==DATA3) DOUT[7:4] = decodedhexchar;
    else DOUT[7:4] = DOUT[7:4];

  always @(state or decodedhexchar)
    if (state==DATA4) DOUT[3:0] = decodedhexchar;
    else DOUT[3:0] = DOUT[3:0];
  */
endmodule
```

### C.1.5 pwm_root_controller.v

```
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 13:30:39 08/01/2009
// Design Name:
// Module Name: pwm_config_interface
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module pwm_root_controller #(
  parameter PWM_CHANNELS = 25,
  parameter CHAN_REGISTERS = 3,
  parameter PWM_CHAN_A_BITS = 5, // Num of bits to address PWM_CHANNELS+1
  parameter CHAN_REG_A_BITS = 2, // Num of bits to address channel regs, 4 max

  parameter REG_STATUS = 3'h0,
  parameter REG_CMP_OFF = 3'h1,
  parameter REG_CMP_ON = 3'h2,
  parameter BIT_EN = 4'h0,
  parameter BIT_RST = 4'hF,
  parameter BIT_DEF = 4'h1
  )
  (
  input CLK,
  input RST,
  input [11:0] AIN, //11:4 are PWM channel, 3:0 are internal PWM register
  input [15:0] DIN,
  input CMD,
  input EXECUTE,
  input EN,
  output [15:0] DOUT,
  output [PWM_CHANNELS:0] PWMOUT,
  output PWMCNT_EN
  );

  // Address breakdowns for pwm channel and internal mem
  wire [PWM_CHAN_A_BITS-1:0] a_chan;
  wire [CHAN_REG_A_BITS-1:0] a_reg;
  assign a_chan = AIN[4+PWM_CHAN_A_BITS-1:4]; // AIN[15:4] is for PWM channel
  assign a_reg = AIN[0+CHAN_REG_A_BITS-1:0]; // AIN[3:0] is for channel registers

  //////////////////
  // Start RAM definition; address registers to ensure a sync-read ram
  reg [15:0] ram[PWM_CHANNELS:0][CHAN_REGISTERS-1:0];
  reg [PWM_CHAN_A_BITS-1:0] read_a_chan;
  reg [CHAN_REG_A_BITS-1:0] read_a_reg;

  wire we;
  assign we = CMD & EXECUTE;

```

```
// RAM access
always @(posedge CLK)
  begin
    if (we) ram[a_chan][a_reg] <= DIN;
    read_a_chan <= a_chan;
    read_a_reg <= a_reg;
  end
assign DOUT = ram[read_a_chan][read_a_reg];
// End RAM definition
/////////////////

parameter REG_CNTMAX = 8'h1;
parameter REG_CLKDIV = 8'h2;
parameter CHAN_ROOT = 8'h0;
wire [15:0] PWMCNT_Q;
wire PWMCNT_TC;
//wire PWMCNT_EN;
wire [15:0] CLKDIV_Q;
wire [15:0] CLKDIV_BIT;
assign CLKDIV_BIT = ram[CHAN_ROOT][REG_CLKDIV];
assign CLKDIV_RST = ram[CHAN_ROOT][REG_CLKDIV][BIT_RST];
assign PWMCNT_TC = (PWMCNT_Q==ram[CHAN_ROOT][REG_CNTMAX]);
assign PWMCNT_EN = EN & ram[CHAN_ROOT][REG_STATUS][BIT_EN];
assign PWMCNT_RST = CLKDIV_RST | ram[CHAN_ROOT][REG_STATUS][BIT_RST];

// Clock divider
wire PWMCNT_CE;
counter_ce_sc clock_divider (
  .CLK(CLK),
  .CLR(RST|CLKDIV_RST|PWMCNT_CE),
  .CE(1'b1),
  .Q(CLKDIV_Q)
);

assign PWMCNT_CE = CLKDIV_BIT[0] ? CLKDIV_Q[CLKDIV_BIT[7:4]] : 1'b1;

// Shared counter for PWM channels
counter_ce_sc counter (
  .CLK(CLK),
  .CLR(RST|PWMCNT_TC|PWMCNT_RST),
  .CE(PWMCNT_EN&PWMCNT_CE),
  .Q(PWMCNT_Q)
);


// Output assignments and dynamic PWM module instantiation
assign PWMOUT[0] = PWMCNT_TC;
generate
  genvar i;
  for (i=1; i <= PWM_CHANNELS; i=i+1) begin : PWMCHANS
    set_reset_pwm_gen pwmch (
      .CLK(CLK),
      .EN(ram[i][REG_STATUS][BIT_EN]&PWMCNT_EN),
        .DEF_VAL(ram[i][REG_STATUS][BIT_DEF]),
        .COUNTER(PWMCNT_Q),
        .CMP_ON_IN(ram[i][REG_CMP_ON]),
        .CMP_OFF_IN(ram[i][REG_CMP_OFF]),
        .Q(PWMOUT[i]));
    end
  endgenerate

endmodule

// wire[1:0] pwmout;

// genvar i;
// generate
//  for (i=0; i < PWM_CHANNELS; i=i+1) begin : PWMCHAN
//    set_reset_pwm_gen pwm (.CLK(CLK), .EN(~RST), .VALUE_DISABLED(1'b1), .COUNTER(16'b0), .
     CMP_ON_IN(16'hFFFF), .CMP_OFF_IN(16'hFFFE), .Q(pwmout[i]));
//  end
// endgenerate

//generate
// genvar i;
// for (i = 0; i < 4; i=i+1) begin : namedblock
// assign out[i*8+7 -: 8] = in[i];
// end
//endgenerate
```

### C.1.6 serial_async_receiver.v

```
// RS-232 RX module
// (c) fpga4fun.com KNJN LLC - 2003, 2004, 2005, 2006

module serial_async_receiver(clk, RxD, RxD_data_ready, RxD_data, RxD_endofpacket, RxD_idle)
    ;
input clk, RxD;
output RxD_data_ready; // onc clock pulse when RxD_data is valid
output [7:0] RxD_data;

//parameter ClkFrequency = 25000000; // 25MHz
//parameter Baud = 115200;
parameter ClkFrequency = 100000000; // 50MHz
parameter Baud = 115200;

// We also detect if a gap occurs in the received stream of characters
// That can be useful if multiple characters are sent in burst
// so that multiple characters can be treated as a "packet"
output RxD_endofpacket; // one clock pulse, when no more data is received (RxD_idle is
    going high)
output RxD_idle; // no data is being received

```

```
// Baud generator (we use 8 times oversampling)
parameter Baud8 = Baud*8;
parameter Baud8GeneratorAccWidth = 16;
wire [Baud8GeneratorAccWidth:0] Baud8GeneratorInc = ((Baud8<<(Baud8GeneratorAccWidth-7))+(
    ClkFrequency>>8))/(ClkFrequency>>7);
reg [Baud8GeneratorAccWidth:0] Baud8GeneratorAcc;
initial Baud8GeneratorAcc = 0;
always @(posedge clk) Baud8GeneratorAcc <= Baud8GeneratorAcc[Baud8GeneratorAccWidth-1:0] +
    Baud8GeneratorInc;
wire Baud8Tick = Baud8GeneratorAcc[Baud8GeneratorAccWidth];

////////////////////////////
reg [1:0] RxD_sync_inv;
initial RxD_sync_inv = 0;
always @(posedge clk) if(Baud8Tick) RxD_sync_inv <= {RxD_sync_inv[0], ~RxD};
// we invert RxD, so that the idle becomes "0", to prevent a phantom character to be
    received at startup

reg [1:0] RxD_cnt_inv;
reg RxD_bit_inv;
initial RxD_bit_inv = 0;
initial RxD_cnt_inv = 0;

always @(posedge clk)
if(Baud8Tick)
begin
  if( RxD_sync_inv[1] && RxD_cnt_inv!=2'b11)
    RxD_cnt_inv <= RxD_cnt_inv + 2'h1;
  else
  if(~RxD_sync_inv[1] && RxD_cnt_inv!=2'b00)
    RxD_cnt_inv <= RxD_cnt_inv - 2'h1;

  if(RxD_cnt_inv==2'b00)
    RxD_bit_inv <= 1'b0;
  else
  if(RxD_cnt_inv==2'b11)
    RxD_bit_inv <= 1'b1;
end

reg [3:0] state;
reg [3:0] bit_spacing;
initial state = 0;
initial bit_spacing = 0;

// "next_bit" controls when the data sampling occurs
// depending on how noisy the RxD is, different values might work better
// with a clean connection, values from 8 to 11 work
wire next_bit = (bit_spacing==4'd10);

always @(posedge clk)
if(state==0)
  bit_spacing <= 4'b0000;
else
```

```
if(Baud8Tick)
  bit_spacing <= {bit_spacing[2:0] + 4'b0001} | {bit_spacing[3], 3'b000};

always @(posedge clk)
if(Baud8Tick)
case(state)
  4'b0000: if(RxD_bit_inv) state <= 4'b1000; // start bit found?
  4'b1000: if(next_bit) state <= 4'b1001; // bit 0
  4'b1001: if(next_bit) state <= 4'b1010; // bit 1
  4'b1010: if(next_bit) state <= 4'b1011; // bit 2
  4'b1011: if(next_bit) state <= 4'b1100; // bit 3
  4'b1100: if(next_bit) state <= 4'b1101; // bit 4
  4'b1101: if(next_bit) state <= 4'b1110; // bit 5
  4'b1110: if(next_bit) state <= 4'b1111; // bit 6
  4'b1111: if(next_bit) state <= 4'b0001; // bit 7
  4'b0001: if(next_bit) state <= 4'b0000; // stop bit
  default: state <= 4'b0000;
endcase

reg [7:0] RxD_data;
initial RxD_data = 8'hFF;
always @(posedge clk)
if(Baud8Tick && next_bit && state[3]) RxD_data <= {~RxD_bit_inv, RxD_data[7:1]};

reg RxD_data_ready, RxD_data_error;
always @(posedge clk)
begin
  RxD_data_ready <= (Baud8Tick && next_bit && state==4'b0001 && ~RxD_bit_inv); // ready
      only if the stop bit is received
  RxD_data_error <= (Baud8Tick && next_bit && state==4'b0001 && RxD_bit_inv); // error if
      the stop bit is not received
end

reg [4:0] gap_count;
initial gap_count = 0;
always @(posedge clk) if (state!=0) gap_count<=5'h00; else if(Baud8Tick & ~gap_count[4])
    gap_count <= gap_count + 5'h01;
assign RxD_idle = gap_count[4];
reg RxD_endofpacket; always @(posedge clk) RxD_endofpacket <= Baud8Tick & (gap_count==5'h0F
    );

endmodule
```

### C.1.7 serial_async_transmitter.v

```
// RS-232 TX module
// (c) fpga4fun.com KNJN LLC - 2003, 2004, 2005, 2006

//`define DEBUG // in DEBUG mode, we output one bit per clock cycle (useful for faster
    simulations)
```

```

module serial_async_transmitter(clk, TxD_start, TxD_data, TxD, TxD_busy);
input clk, TxD_start;
input [7:0] TxD_data;
output TxD, TxD_busy;

parameter ClkFrequency = 100000000; // 50MHz
parameter Baud = 115200;
parameter RegisterInputData = 1; // in RegisterInputData mode, the input doesn't have to
    stay valid while the character is been transmitted

// Baud generator
parameter BaudGeneratorAccWidth = 16;
reg [BaudGeneratorAccWidth:0] BaudGeneratorAcc;
initial BaudGeneratorAcc = 0;
`ifdef DEBUG
wire [BaudGeneratorAccWidth:0] BaudGeneratorInc = 17'h10000;
`else
wire [BaudGeneratorAccWidth:0] BaudGeneratorInc = ((Baud<<(BaudGeneratorAccWidth-4))+(
    ClkFrequency>>5))/(ClkFrequency>>4);
`endif

wire BaudTick = BaudGeneratorAcc[BaudGeneratorAccWidth];
wire TxD_busy;
always @(posedge clk) if(TxD_busy)
  BaudGeneratorAcc <= BaudGeneratorAcc[BaudGeneratorAccWidth-1:0] + BaudGeneratorInc;

// Transmitter state machine
reg [3:0] state;
initial state = 0;
wire TxD_ready = (state==0);
assign TxD_busy = ~TxD_ready;

reg [7:0] TxD_dataReg;
always @(posedge clk)
  if(TxD_ready & TxD_start) TxD_dataReg <= TxD_data;

wire [7:0] TxD_dataD = RegisterInputData ? TxD_dataReg : TxD_data;

always @(posedge clk)
case(state)
  4'b0000: if(TxD_start) state <= 4'b0001;
  4'b0001: if(BaudTick) state <= 4'b0100;
  4'b0100: if(BaudTick) state <= 4'b1000; // start
  4'b1000: if(BaudTick) state <= 4'b1001; // bit 0
  4'b1001: if(BaudTick) state <= 4'b1010; // bit 1
  4'b1010: if(BaudTick) state <= 4'b1011; // bit 2
  4'b1011: if(BaudTick) state <= 4'b1100; // bit 3
  4'b1100: if(BaudTick) state <= 4'b1101; // bit 4
  4'b1101: if(BaudTick) state <= 4'b1110; // bit 5
  4'b1110: if(BaudTick) state <= 4'b1111; // bit 6
  4'b1111: if(BaudTick) state <= 4'b0010; // bit 7
  4'b0010: if(BaudTick) state <= 4'b0011; // stop1
  4'b0011: if(BaudTick) state <= 4'b0000; // stop2
  default: if(BaudTick) state <= 4'b0000;
endcase

// Output mux
reg muxbit;
always @( * )
case(state[2:0])
  3'd0: muxbit <= TxD_dataD[0];
  3'd1: muxbit <= TxD_dataD[1];
  3'd2: muxbit <= TxD_dataD[2];
  3'd3: muxbit <= TxD_dataD[3];
  3'd4: muxbit <= TxD_dataD[4];
  3'd5: muxbit <= TxD_dataD[5];
  3'd6: muxbit <= TxD_dataD[6];
  3'd7: muxbit <= TxD_dataD[7];
endcase

// Put together the start, data and stop bits
reg TxD;
always @(posedge clk) TxD <= (state<4) | (state[3] & muxbit); // register the output to
    make it glitch free

endmodule
```

### C.1.8 serial_to_pwm.v

```
`timescale 1ns / 100ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 18:50:11 07/30/2009
// Design Name: input_protocol_decode
// Module Name: C:/fpga/serial_to_pwm/input_protocol_decode_tb.v
// Project Name: serial_to_pwm
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: input_protocol_decode
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
```

```

module serial_to_pwm(
    input CLK_IN,
    input CLR,
    input SPI_ENABLE,
    input UART_RXD,
    input SPI_SSEL,
    input SPI_MOSI,
    input SPI_SCK,
    input ENLEVEL,
    input HALT,
    output UART_TXD,
    output SPI_MISO,
    output [25:0] PWMOUT,
    output PWMSYNC,
    output HALTED_LED,
    output RXD_LED,
    output TXD_LED,
    output FPGA_INIT_B,
    output STATUS_LED,
    output DCM_LOCKED_OUT
  );
  wire EN;
  wire CLK;
  wire CLK0;
  wire count_en;
  assign STATUS_LED = count_en;
  assign FPGA_INIT_B = 1;
  assign RXD_LED = SPI_ENABLE ? (SPI_MOSI&SPI_SCK) : ~UART_RXD;
  assign TXD_LED = SPI_ENABLE ? (SPI_MISO&SPI_SCK) : ~UART_TXD;
  assign CLKOUT = CLK;

  wire deser_rx_data_valid;
  wire SPI_deser_rx_data_valid;
  wire UART_deser_rx_data_valid;

  wire [7:0] deser_rx_data;
  wire [7:0] SPI_deser_rx_data;
  wire [7:0] UART_deser_rx_data;

  assign deser_rx_data_valid = SPI_ENABLE ? SPI_deser_rx_data_valid :
      UART_deser_rx_data_valid;
  assign deser_rx_data = SPI_ENABLE ? SPI_deser_rx_data : UART_deser_rx_data;

  reg HALTED;
   assign HALTED_LED = HALTED;
  always @(posedge CLK0)
  begin
    if (CLR)
      HALTED <= 1'b0;
    else
      if (HALT)
        HALTED <= 1'b1;
```

```
 end
assign EN = ENLEVEL & ~HALTED;

assign PWMSYNC = PWMOUT[0];

 // Instantiate the serial port reciever
serial_async_receiver deserial(
  .clk(CLK),
  .RxD(UART_RXD),
  .RxD_data_ready(UART_deser_rx_data_valid),
  .RxD_data(UART_deser_rx_data),
  .RxD_endofpacket(),
  .RxD_idle()
);

// Instantiate the SPI interface
SPI_slave spi(
  .clk(CLK),
  .MOSI(SPI_MOSI),
  .SCK(SPI_SCK),
  .SSEL(SPI_SSEL),
  .RX_DRDY(SPI_deser_rx_data_valid),
  .RX_DATA(SPI_deser_rx_data)
);

// Instantiate the clock doubler (DCM)
clocking instance_name (
  .CLKIN_IN(CLK_IN),
  .CLKIN_IBUFG_OUT(),
  .CLK0_OUT(CLK0),
  .RST_IN(CLR & ~DCM_LOCKED_OUT),
  .CLK2X_OUT(CLK),
  .LOCKED_OUT(DCM_LOCKED_OUT)
);


wire instr_cmd;
wire [15:0] instr_data;
wire [11:0] instr_addr;
wire instr_valid;
// Instantiate the protcol decoder
input_protocol_decode instr_decode (
  .CLK(CLK),
  .CLR(CLR),
  .DIN_TICK(deser_rx_data_valid),
  .DIN(deser_rx_data),
  .CMDOUT(instr_cmd),
  .DOUT(instr_data),
  .AOUT(instr_addr),
  .DECODED(instr_valid)
);


```

```
wire [7:0] tx_fifo_din;
wire [15:0] readback_data;
reg [2:0] readback_nibble_state;
reg [3:0] readback_nibble;
wire[7:0] readback_nibble_encoded;
reg readback_nibble_en;
reg [3:0] readback_data1;
reg [3:0] readback_data2;
reg [3:0] readback_data3;
reg [3:0] readback_data4;

// Instantiate the PWM root controller
pwm_root_controller pwm_control (
  .CLK(CLK),
  .RST(CLR),
  .AIN(instr_addr),
  .DIN(instr_data),
  .CMD(instr_cmd),
  .EXECUTE(instr_valid),
  .EN(EN),
  .DOUT(readback_data),
  .PWMOUT(PWMOUT),
  .PWMCNT_EN(count_en)
);

//assign tx_fifo_din = readback_data[7:0];

wire [7:0] tx_data;
wire tx_fifo_empty;
wire tx_busy;
// Instantiate the serial port transmit FIFO
sync_fifo tx_fifo (
  .din(tx_fifo_din),
  .wr_en(tx_fifo_we),
  .rd_en((~tx_busy)&(~tx_fifo_empty)),
  .dout(tx_data),
  .full(),
  .empty(tx_fifo_empty),
  .clk(CLK),
  .reset(CLR)
);

//reg tx_send;
// Instantiate the transmit serializer
serial_async_transmitter serout (
  .clk(CLK),
  .TxD_start(~tx_fifo_empty),
  .TxD_data(tx_data),
  .TxD(UART_TXD),
  .TxD_busy(tx_busy)
);

// State machine-esque block to chop a 16bit number into 4 4-bit values
```

```
  // which are then translated into asciihex for serial transmission
  always @(posedge CLK)
    case(readback_nibble_state)
      3'b100: readback_nibble_state <= 3'b011;
      3'b011: readback_nibble_state <= 3'b010;
      3'b010: readback_nibble_state <= 3'b001;
      3'b001: readback_nibble_state <= 3'b000;
      default:
        if(instr_valid & (instr_cmd==1'b0))
          readback_nibble_state <= 3'b100;
        else
          readback_nibble_state <= 3'b000;
    endcase

  always @(posedge CLK)
    case(readback_nibble_state)
      3'b100:
        begin
          readback_nibble <= readback_data4;
          readback_nibble_en <= 1'b1;
        end
      3'b011: readback_nibble <= readback_data3;
      3'b010: readback_nibble <= readback_data2;
      3'b001: readback_nibble <= readback_data1;
      default:
        begin
          {readback_data4, readback_data3, readback_data2, readback_data1} <= readback_data;
          readback_nibble <= 4'h0;
          readback_nibble_en <= 1'b0;
        end
    endcase

  bin2asciihex output_encode(
    .bin(readback_nibble),
    .ascii(readback_nibble_encoded)
  );

// Enabled Local Echo
// assign tx_fifo_we = (readback_nibble_en | deser_rx_data_valid);
// assign tx_fifo_din = (readback_nibble_en) ? readback_nibble_encoded : deser_rx_data;

// Disabled Local Echo
  assign tx_fifo_we = readback_nibble_en;
  assign tx_fifo_din = readback_nibble_encoded;

endmodule
```

### C.1.9 set_reset_pwm_gen.v

```
////////////////////////////////////////////////////////////////////////////////
```

```
// Company:
// Engineer:
//
// Create Date:
// Design Name:
// Module Name: set_reset_pwm
// Project Name:
// Target Devices:
// Tool versions:
// Description: Output is a PWM generated from ON and OFF values compared
// against the counter. When the enable is brought low, the
// output state is assigned DISABLED_STATE
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module set_reset_pwm_gen(
  input CLK,
  input EN,
  input DEF_VAL,
  input [15:0] COUNTER,
  input [15:0] CMP_ON_IN,
  input [15:0] CMP_OFF_IN,
  output reg Q
  );

  wire SET_ON;
  wire SET_OFF;

  reg [15:0] CMP_ON;
  reg [15:0] CMP_OFF;

  always @(posedge CLK) //
    if (SET_OFF)
      begin
        CMP_OFF <= CMP_OFF_IN;
        if (EN) Q <= 1'b0;
        else Q <= DEF_VAL;
      end
    else
      if (SET_ON)
        begin
          CMP_ON <= CMP_ON_IN;
          if (EN) Q <= 1'b1;
          else Q <= DEF_VAL;
        end
      else
        if (~EN)
        begin
          Q <= DEF_VAL;
          CMP_OFF <= CMP_OFF_IN;
          CMP_ON <= CMP_ON_IN;
        end
      else
        Q <= Q;

  assign SET_OFF = COUNTER==CMP_OFF;
  assign SET_ON = COUNTER==CMP_ON;

endmodule
```

### C.1.10 spi_slave.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 08:32:38 08/10/2010
// Design Name:
// Module Name: spi_slave
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module SPI_slave(clk, SCK, MOSI, MISO, SSEL, RX_DRDY, RX_DATA);
input clk;

input SCK, SSEL, MOSI;
output MISO, RX_DRDY;
output [7:0] RX_DATA;

reg [7:0] RX_DATA;
reg RX_DRDY;

//// Synchronization ////
// sync SCK to the FPGA clock using a 3-bits shift register
reg [2:0] SCKr; always @(posedge clk) SCKr <= {SCKr[1:0], SCK};
wire SCK_risingedge = (SCKr[2:1]==2'b01); // now we can detect SCK rising edges
wire SCK_fallingedge = (SCKr[2:1]==2'b10); // and falling edges

// same thing for SSEL
reg [2:0] SSELr; always @(posedge clk) SSELr <= {SSELr[1:0], SSEL};
wire SSEL_active = ~SSELr[1]; // SSEL is active low
wire SSEL_startmessage = (SSELr[2:1]==2'b10); // message starts at falling edge
wire SSEL_endmessage = (SSELr[2:1]==2'b01); // message stops at rising edge

// and for MOSI
reg [1:0] MOSIr; always @(posedge clk) MOSIr <= {MOSIr[0], MOSI};
wire MOSI_data = MOSIr[1];


//// Reception ////
// we handle SPI in 8-bits format, so we need a 3 bits counter to count the bits as they
    come in
reg [2:0] bitcnt;

reg byte_received; // high when a byte has been received
reg [7:0] byte_data_received;

always @(posedge clk)
begin
  if(~SSEL_active)
    bitcnt <= 3'b000;
  else
  if(SCK_risingedge)
  begin
    bitcnt <= bitcnt + 3'b001;

    // implement a shift-left register (since we receive the data MSB first)
    byte_data_received <= {byte_data_received[6:0], MOSI_data};
  end
end

always @(posedge clk) byte_received <= SSEL_active && SCK_risingedge && (bitcnt==3'b111);
always @(posedge clk)
begin
  if(byte_received)
  begin
    RX_DRDY <= byte_received;
    RX_DATA <= byte_data_received;
  end
  else
  begin
    RX_DRDY <= byte_received;
    RX_DATA <= RX_DATA;
  end
end


//// Transmission ////
reg [7:0] byte_data_sent;

reg [7:0] cnt;
always @(posedge clk) if(SSEL_startmessage) cnt<=cnt+8'h1; // count the messages

always @(posedge clk)
if(SSEL_active)
begin
  if(SSEL_startmessage)
    byte_data_sent <= cnt; // first byte sent in a message is the message count
  else
  if(SCK_fallingedge)
  begin
    if(bitcnt==3'b000)
      byte_data_sent <= 8'h00; // after that, we send 0s
    else
      byte_data_sent <= {byte_data_sent[6:0], 1'b0};
  end
end

assign MISO = byte_data_sent[7]; // send MSB first
// we assume that there is only one slave on the SPI bus
// so we don't bother with a tri-state buffer for MISO
// otherwise we would need to tri-state MISO when SSEL is inactive

endmodule
```

### C.1.11 sync_fifo.v

```
////////////////////////////////////////////////
// Author: Deepak (28/03/2009 08:54)
// Module: fifo.v
// Project:
// Description: Synchronous FIFO
// data output (dout) is un-registered.
// Version: 1.1 (not icarus verilog compatible)
//
////////////////////////////////////////////////

module sync_fifo #(
     parameter DATA_WIDTH = 8,
     parameter DEPTH = 16,
   parameter ADDR_WIDTH = log2(DEPTH)
  )
  (
     input [DATA_WIDTH-1:0] din,
     input wr_en,
     input rd_en,
     output [DATA_WIDTH-1:0] dout,
     output reg full,
     output reg empty,

   input clk,
   input reset
);

function integer log2;
   input integer n;
   begin
      log2 = 0;
      while(2**log2 < n) begin
         log2=log2+1;
      end
   end
endfunction

reg [ADDR_WIDTH : 0] rd_ptr; // note MSB is not really address
reg [ADDR_WIDTH : 0] wr_ptr; // note MSB is not really address
wire [ADDR_WIDTH-1 : 0] wr_loc;
wire [ADDR_WIDTH-1 : 0] rd_loc;
reg [DATA_WIDTH-1 : 0] mem[DEPTH-1 : 0];

assign wr_loc = wr_ptr[ADDR_WIDTH-1 : 0];
assign rd_loc = rd_ptr[ADDR_WIDTH-1 : 0];

always @(posedge clk) begin
   if(reset) begin
      wr_ptr <= 'h0;
      rd_ptr <= 'h0;
   end // end if
   else begin
      if(wr_en & (~full))begin
         wr_ptr <= wr_ptr+1;
      end
      if(rd_en & (~empty))
         rd_ptr <= rd_ptr+1;
   end //end else
end//end always

//empty if all the bits of rd_ptr and wr_ptr are the same.
//full if all bits except the MSB are equal and MSB differes
always @(rd_ptr or wr_ptr)begin
   //default catch-alls
   empty <= 1'b0;
   full <= 1'b0;
   if(rd_ptr[ADDR_WIDTH-1:0]==wr_ptr[ADDR_WIDTH-1:0])begin
      if(rd_ptr[ADDR_WIDTH]==wr_ptr[ADDR_WIDTH])
         empty <= 1'b1;
      else
         full <= 1'b1;
   end//end if
end//end always

always @(posedge clk) begin
```

```
      if (wr_en)
         mem[wr_loc] <= din;
   end //end always

   //comment if you want a registered dout
   assign dout = rd_en ? mem[rd_loc]:'h0;
   //uncomment if you want a registered dout
   //always @(posedge clk) begin
   // if (reset)
   // dout <= 'h0;
   // else if (rd_en)
   // dout <= mem[rd_ptr];
   //end
endmodule
```

## C.2 Microcontroller Implementation

### C.2.1 dac.c

```
#include "stm32f10x_conf.h"
#include "dac.h"

void DAC_OUT_Init(void)
{

  /* Enable peripheral clocks ------------------------------------------------*/
  /* DAC Periph clock enable */
  RCC_APB1PeriphClockCmd(DAC_CLK, ENABLE);
  /* DAC GPIO(A) Periph clock enable */
  RCC_APB2PeriphClockCmd(DAC_GPIO_CLK, ENABLE);

  /* ----- Configure GPIO ----- */
  GPIO_InitTypeDef GPIO_InitStructure;

  GPIO_InitStructure.GPIO_Pin = DAC_PIN;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
  GPIO_Init(DAC_GPIO, &GPIO_InitStructure);

  /* ----- Configure SPI ----- */
  DAC_InitTypeDef DAC_InitStructure;

  DAC_InitStructure.DAC_Trigger = DAC_Trigger_Software;
  DAC_InitStructure.DAC_WaveGeneration = DAC_WaveGeneration_None;
  //DAC_InitStructure.DAC_LFSRUnmask_TriangleAmplitude = DAC_TriangleAmplitude_2047;
  DAC_InitStructure.DAC_OutputBuffer = DAC_OutputBuffer_Disable;
  DAC_Init(DAC_Channel, &DAC_InitStructure);

  /* Enable DAC Channel1: Once the DAC channel1 is enabled, PA.04 is
     automatically connected to the DAC converter. */
  DAC_Cmd(DAC_Channel, ENABLE);
}

void DAC_OUT_Write(uint16_t Data)
{
  DAC_SetChannel1Data(DAC_Align_12b_L, Data);
    DAC_SoftwareTriggerCmd(DAC_Channel_1, ENABLE);
}
```

### C.2.2 dac.h

```
#ifndef DAC_H_
#define DAC_H_

#include "stm32f10x_gpio.h"

/* Define the STM32F10x hardware depending on the used evaluation board */
#define DAC_Channel DAC_Channel_1
#define DAC_CLK RCC_APB1Periph_DAC
#define DAC_GPIO GPIOA
#define DAC_GPIO_CLK RCC_APB2Periph_GPIOA
#define DAC_PIN GPIO_Pin_4

void DAC_OUT_Write(uint16_t Data);
void DAC_OUT_Init(void);

#endif /* DAC_H_ */
```

### C.2.3 leds.c

```
/*
 * leds.c
 *
 * Created on: Jun 13, 2010
 * Author: pierquet
 */

#include "leds.h"

/**
  * @brief Configures LED GPIO.
  * @param Led: Specifies the Led to be configured.
  * This parameter can be one of following parameters:
  * @arg LED1
  * @arg LED2
  * @arg LED3
  * @arg LED4
  * @retval None
  */

void LEDInit()
{
  GPIO_InitTypeDef GPIO_InitStructure;

  /* Enable the GPIO_LED Clock */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOF, ENABLE);

  /* Configure the GPIO_LED pin */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
  GPIO_Init(GPIOF, &GPIO_InitStructure);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
  GPIO_Init(GPIOF, &GPIO_InitStructure);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
  GPIO_Init(GPIOF, &GPIO_InitStructure);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
  GPIO_Init(GPIOF, &GPIO_InitStructure);
}

void LEDOn(uint8_t n)
{
  switch (n) {
  case 0:
    GPIOF->BSRR = GPIO_Pin_6;
    break;
  case 1:
    GPIOF->BSRR = GPIO_Pin_7;
    break;
  case 2:
    GPIOF->BSRR = GPIO_Pin_8;
    break;
  case 3:
    GPIOF->BSRR = GPIO_Pin_9;
    break;
  }
}

void LEDOff(uint8_t n)
{
  switch (n) {
  case 0:
    GPIOF->BRR = GPIO_Pin_6;
    break;
  case 1:
    GPIOF->BRR = GPIO_Pin_7;
    break;
  case 2:
    GPIOF->BRR = GPIO_Pin_8;
    break;
  case 3:
    GPIOF->BRR = GPIO_Pin_9;
    break;
  }
}

void LEDToggle(uint8_t n)
{
  switch (n) {
  case 0:
    GPIOF->ODR ^= GPIO_Pin_6;
    break;
  case 1:
    GPIOF->ODR ^= GPIO_Pin_7;
    break;
  case 2:
    GPIOF->ODR ^= GPIO_Pin_8;
    break;
  case 3:
    GPIOF->ODR ^= GPIO_Pin_9;
    break;
  }
}
```

### C.2.4 leds.h

```
/*
 * leds.h
 *
 * Created on: Jun 13, 2010
 * Author: pierquet
 */

void LEDInit(void);
void LEDOn(uint8_t n);
void LEDOff(uint8_t n);
void LEDToggle(uint8_t n);
```

### C.2.5 main.c

```
/*
 *
 *
 *
 *
 *
 *
 */



#define SYSCLK_FREQ_72MHz
//#define USE_DAC
#define VERBOSE
//#define DEBUG
//#define DEBUG_VERBOSE
//#define DEBUG_ADC
//#define DEBUG_ERROR

#define VOLTAGE_SCALING 1
#define ADC_READINGS 2
#include "stm32f10x.h"
#include "usart.h"
#include "leds.h"
#include "spi_fpga.h"
#include "spi_adc.h"

#include "oplist_ops_for.h"
#include "oplist_ops_rev.h"
#include "oplist_adc_for.h"
#include "oplist_adc_rev.h"
#include "oplist_fsw_for.h"
#include "oplist_fsw_rev.h"

#define oplist_adc oplist_adc_for
#define oplist_fsw oplist_fsw_for

#ifdef USE_DAC
#include "dac.h"
#endif

void MAIN_Initialize_Peripherals(void);

void sleep(uint32_t cnt);
void MAIN_Print_Usage(void);
void MAIN_Print_Status(void);
uint32_t MAIN_Calculate_Op_Index(uint32_t);
uint8_t MAIN_Calculate_Op_Update(void);
void MAIN_ADC_ReadStore(void);
void MAIN_FPGA_Initialize(void);
void MAIN_FPGA_Program_Nonblock(void);
void MAIN_FPGA_Program_Wait(void);
void MAIN_FPGA_Enable(void);
void MAIN_FPGA_Disable(void);
void MAIN_FPGA_CC_Negative(void);
void MAIN_FPGA_CC_Positive(void);
void MAIN_FPGA_CC_Bypass(void);
void MAIN_FPGA_CC_Full(void);
```

```

uint8_t compileStamp[] = "(" __DATE__ ", " __TIME__ ")";
uint8_t authorName[] = "Brandon J. Pierquet -- pierquet@alum.mit.edu";

uint32_t len_op = 168;
uint32_t len_opcmd = 8;
uint32_t len_opstring = 176;
uint32_t len_opslist = sizeof(oplist_adc)/sizeof(*oplist_adc);

// Enable All active Channels
uint8_t enableString[] = "w0E00001w0D00001w0C00001w0B00001w1400001w1300001w1200001w1100001
    ";
uint32_t enableString_len = 64;

uint32_t opPoint_adc;
uint32_t opPoint_adc_last;
uint32_t opPoint_fsw;
uint32_t opPoint_fsw_last;
uint32_t opPoint_idx;
uint32_t opPoint_idx_last;
uint32_t opPoint_sgn;
uint32_t opPoint_sgn_last;
uint32_t adcVoltage;
uint32_t adcVoltage_pos;
uint32_t adcVoltage_neg;
uint8_t* opPoint_adr=oplist_ops_for;

/*
 *
 *
 */
void MAIN_Print_Usage(void)
{
  usartx_puts((uint8_t*) "+-----------------------------------------------------+\r\n");
  usartx_puts((uint8_t*) "HEPVMI Controller: ");
  usartx_puts((uint8_t*) compileStamp);
  usartx_puts((uint8_t*) "\r\n");
  usartx_puts((uint8_t*) "Author: ");
  usartx_puts((uint8_t*) authorName);
  usartx_puts((uint8_t*) "\r\n");
  usartx_puts((uint8_t*) "+-----------------------------------------------------+\r\n");
  usartx_puts((uint8_t*) "(i) _I_nitialize FPGA Operation\r\n");
  usartx_puts((uint8_t*) "(r) _R_ead and store voltage from SPI ADC\r\n");
  usartx_puts((uint8_t*) "(p) _P_rogram FPGA using stored voltage\r\n");
  usartx_puts((uint8_t*) "(w) _W_rite arbitrary data to FPGA\r\n");
  usartx_puts((uint8_t*) "\r\n");
  usartx_puts((uint8_t*) "(u) _U_pdate FPGA operation: (r) then (p)\r\n");
  usartx_puts((uint8_t*) "(e) _E_nable FPGA operation\r\n");
  usartx_puts((uint8_t*) "(d) _D_isable FPGA operation\r\n");
  usartx_puts((uint8_t*) "(o) _O_ne cycle operation\r\n");
  usartx_puts((uint8_t*) "(a) _A_utonomously operate\r\n");
  usartx_puts((uint8_t*) "\r\n");
}
```

```


/*
 *
 *
 */
void MAIN_Print_Status(void)
{
  usartx_puts((uint8_t*) "\r\n");
  usartx_puts((uint8_t*) "Status: \r\n");
  usartx_puts((uint8_t*) "\r\n");
}


/*
 *
 *
 */
uint32_t MAIN_Calculate_Op_Index(uint32_t voltage)
{
  uint32_t idx = 0;

#ifdef DEBUG_VERBOSE
  uint32_t up;
  uint32_t down;
  usartx_puts((uint8_t*) "\r\n * MAIN_Calculate_Op_Index *\r\n");
#endif

  /* Locate the new opPoint index based on ADC reading;
   * Implement hysteresis: The table entry is only updated if
   * the measured value steps up or down a full entry in either
   * direction.
   */
  //idx = len_opslist/2;
  idx = opPoint_idx_last;

#ifdef DEBUG_VERBOSE
  usartx_puts((uint8_t*) " Volts : ");
  uart_puti((uint32_t)voltage);
  usartx_puts((uint8_t*) "\r\n Idx : ");
  uart_puti((uint32_t)idx);
#endif

  /* If the measured voltage is less than the next step down from where
   * we're operating now, the decrement until we're one step above the
   * measured voltage
   */
  if (opPoint_idx_last>0)
    if (voltage <= oplist_adc[opPoint_idx_last-1])
    {
#ifdef DEBUG_VERBOSE
      usartx_puts((uint8_t*) "\r\n V_Idx- : ");
      uart_puti((uint32_t)oplist_adc[idx-1]);
#endif
      while ( (voltage <= oplist_adc[idx-1]) && (idx>0) )
      {
        idx--;
      }
#ifdef DEBUG_VERBOSE
      uart_puti((uint32_t)oplist_adc[idx-1]);
#endif
    }
  /* If the measured voltage is more than the next step up from where
   * we're operating now, the increment until we're one step below the
   * measured voltage
   */
  if (opPoint_idx_last<(len_opslist-1))
    if ( (voltage > oplist_adc[opPoint_idx_last+1]) )
    {
#ifdef DEBUG_VERBOSE
      usartx_puts((uint8_t*) "\r\n V_Idx+ : ");
      uart_puti((uint32_t)oplist_adc[idx+1]);
#endif
      while ( (voltage >= oplist_adc[idx+1]) && (idx<len_opslist-1) )
      {
        idx++;
      }
#ifdef DEBUG_VERBOSE
      uart_puti((uint32_t)oplist_adc[idx+1]);
#endif
    }

#ifdef DEBUG_VERBOSE
  up = oplist_adc[idx+1]-voltage;
  down = voltage-oplist_adc[idx-1];

  usartx_puts((uint8_t*) "\r\n Idx : ");
  uart_puti((uint32_t)opPoint_idx_last);
  usartx_puts((uint8_t*) " -> ");
  uart_puti((uint32_t)idx);
  usartx_puts((uint8_t*) "\r\n Up : ");
  uart_puti((uint32_t)up);
  usartx_puts((uint8_t*) "\r\n Down : ");
  uart_puti((uint32_t)down);
  usartx_puts((uint8_t*) "\r\n");
#endif

#ifdef DEBUG_VERBOSE
  usartx_puts((uint8_t*) " * ---------------------- *\r\n");
#endif

  return idx;
}

/*
 *
 *
 */
uint8_t MAIN_Calculate_Op_Update(void)
{
  uint32_t offset = 0;
  uint8_t update=0;

#ifdef DEBUG_VERBOSE
  usartx_puts((uint8_t*) "\r\n * MAIN_Calculate_Op_Update *\r\n");
#endif

  opPoint_idx_last = opPoint_idx;
  opPoint_fsw_last = opPoint_fsw;
  opPoint_adc_last = opPoint_adc;
  // OpPoint_sgn_last = updated in MAIN_ADC_ReadStore()

  opPoint_idx = MAIN_Calculate_Op_Index(adcVoltage);
  // opPoint_sgn = updated in MAIN_ADC_ReadStore()
  opPoint_fsw = oplist_fsw[opPoint_idx];
  opPoint_adc = oplist_adc[opPoint_idx];

  offset = len_opstring*opPoint_idx;

  if (opPoint_fsw<opPoint_fsw_last)
    offset += len_opcmd;

  if (opPoint_adc<opPoint_adc_last)
    opPoint_adr = oplist_ops_rev+offset;
  else
    opPoint_adr = oplist_ops_for+offset;



  if (opPoint_idx != opPoint_idx_last)
  {

#ifdef DEBUG_VERBOSE
  usartx_puts((uint8_t*) " Adc : ");
  uart_puti(adcVoltage);
  usartx_puts((uint8_t*) " => ");
  uart_puti(opPoint_adc);
  usartx_puts((uint8_t*) "\r\n");
  if (opPoint_fsw<opPoint_fsw_last)
    usartx_puts((uint8_t*) " FswCmp : Decrease\r\n");
  else
    usartx_puts((uint8_t*) " FswCmp : Increase\r\n");

  usartx_puts((uint8_t*) "\r\n Idx : ");
  uart_puti((uint32_t)opPoint_idx);
  usartx_puts((uint8_t*) "\r\n IdxAdc : ");
  uart_puti((uint32_t)opPoint_adc);
  usartx_puts((uint8_t*) "\r\n Addr : ");
  uart_puthex_addr((uint32_t)opPoint_adr);
  usartx_puts((uint8_t*) "\r\n");
#endif

#ifdef DEBUG_VERBOSE
  usartx_puts((uint8_t*) "Idx: ");
  uart_puti((uint32_t)opPoint_idx_last);
  usartx_puts((uint8_t*) "->");
  uart_puti((uint32_t)opPoint_idx);
  usartx_puts((uint8_t*) "\r\n");
#endif

    update=1;

#ifdef DEBUG_VERBOSE
  usartx_puts((uint8_t*) "\r\n String: ");
  USART_putsn(opPoint_adr, len_op);
  usartx_puts((uint8_t*) "\r\n");
#endif

  }
  else
  {
    update=0;
  }

#ifdef USE_DAC
  //DAC_OUT_Write(opPoint_fsw);
  DAC_OUT_Write((uint16_t)opPoint_fsw);
#endif

#ifdef DEBUG_VERBOSE
  usartx_puts((uint8_t*) " * ---------------- *\r\n");
#endif
  return update;
}


/*
 *
 *
 */
int main(void)
{
  uint8_t rxdat;

  MAIN_Initialize_Peripherals();
  usartx_puts((uint8_t*) "\r\n");
  LEDOn(0);

  MAIN_FPGA_Initialize();
  MAIN_Print_Usage();

  while (1)
```

```
{
  rxdat = usartx_getc();
  if (rxdat != 255)
  {
    /* Read ADC and store voltage */
    if (rxdat=='i')
    {
      MAIN_FPGA_Initialize();
      MAIN_FPGA_CC_Full();
    }

    /* Read ADC and store voltage */
    if (rxdat=='r')
    {
      MAIN_ADC_ReadStore();
      uart_puti(adcVoltage_pos);
      usartx_putc(',');
      uart_puti(adcVoltage_neg);
      usartx_puts((uint8_t*) "\r\n");
    }


    /* Program FPGA using stored voltage */
    if (rxdat=='p')
    {
      MAIN_Calculate_Op_Update();
      MAIN_FPGA_Program_Nonblock();
      MAIN_FPGA_Program_Wait();
    }


    /* Write arbitrary data to FPGA */
    if (rxdat=='w')
      usartx_puts((uint8_t*)"\r\n * Not Implemented! *\r\n");


    /* Update state: read then program */
    if (rxdat=='u')
    {
      MAIN_ADC_ReadStore();
      MAIN_Calculate_Op_Update();
      MAIN_FPGA_Program_Nonblock();
      MAIN_FPGA_Program_Wait();
    }


    /* Enable the FPGA */
    if (rxdat=='e')
    {
      MAIN_FPGA_Enable();
    }


```

```
      /* Disable the FPGA */
      if (rxdat=='d')
      {
        MAIN_FPGA_Disable();
      }


      /* One cycle auto operation */
      if (rxdat=='o')
      {
        /* Setup the initial programming */
        uint32_t CCbypassThreshold = 0;
        uint8_t exit=USART_EMPTY_READ;
        uint8_t state=0;
        uint32_t runcounter=0;
        uint32_t programcounter=0;

        MAIN_FPGA_Initialize();
        /* State machine of sorts... */
        while ( (exit==USART_EMPTY_READ) && (runcounter<=20) && (programcounter<3000))
        {
          exit = usartx_getc();
          if (state==0) // Setup Trigger
          {
            MAIN_FPGA_Disable();
            MAIN_FPGA_CC_Bypass();
            // Wake up the ADCs
            MAIN_ADC_ReadStore();
            MAIN_ADC_ReadStore();
            MAIN_ADC_ReadStore();
            MAIN_ADC_ReadStore();
            state=1;
          }
          else if (state==1) // Wait for Trigger
          {
#ifdef DEBUG
            usartx_putc('1');
#endif
            // Calculate operating point update
            MAIN_ADC_ReadStore();
            MAIN_Calculate_Op_Update();
            /*
             * If there is a transition from negative output voltage
             * to positive, trigger has occurred:
             * Program the converter and enable it
             */
#ifdef DEBUG
            usartx_putc('(');
            if (opPoint_sgn==ADC_POS) usartx_putc('+'); else usartx_putc('-');
            if (opPoint_sgn_last==ADC_POS) usartx_putc('+'); else usartx_putc('-');
            usartx_putc(')');
#endif
            if ( (opPoint_sgn==ADC_POS) && (opPoint_sgn_last==ADC_NEG) )
```

```
           //if ( (opPoint_adc<CCbypassThreshold) && (opPoint_adc>opPoint_adc_last) && ( opPoint_sgn==ADC_POS) )
           {
             state=2;
             MAIN_FPGA_Program_Nonblock();
             programcounter++;
             MAIN_ADC_ReadStore();
             MAIN_FPGA_Program_Wait();
             MAIN_FPGA_Enable();
           }
         }
         else if (state==2) // Running under threshold
         {
#ifdef DEBUG
           usartx_putc('2');
#endif
           // Next state occurs when above threshold
           if (adcVoltage>CCbypassThreshold)
           {
             state=3;
             runcounter++;
             if (opPoint_sgn==ADC_POS)
               MAIN_FPGA_CC_Positive();
             else
               MAIN_FPGA_CC_Negative();

             MAIN_ADC_ReadStore();
           }
           else
           {
             // check if the operating point needs updating, and
             // if so start sending the program.
             if (MAIN_Calculate_Op_Update())
             {
               MAIN_FPGA_Program_Nonblock();
               programcounter++;
             }
             // If the fpga is being programmed, this allows the ADC reads
             // to be pipelined, otherwise the read is done to update values
             // needed to check for a needed update
             MAIN_ADC_ReadStore();
             // Wait for the DMA engine SPI Tx if necessary
             MAIN_FPGA_Program_Wait();
           }
         }
         else if (state==3) // Normal operation above threshold
         {
#ifdef DEBUG
           usartx_putc('3');
#endif


           if (opPoint_sgn!=opPoint_sgn_last)
```

```
          {
            // Next state occurs at a change of sign
            state=2;
          }
          else if (adcVoltage<CCbypassThreshold)
          {
            // Next state occurs at a drop below threhold
            state=2;
            MAIN_FPGA_CC_Bypass();
          }
          else
          {
            // Reprogram FPGA registers if an update is needed
            if (MAIN_Calculate_Op_Update())
            {
              MAIN_FPGA_Program_Nonblock();
              programcounter++;
            }
            // Pipeline the adc reads, then wait for a DMA transfer to
            // finish, if there was one
            MAIN_ADC_ReadStore();
            MAIN_FPGA_Program_Wait();
          }
        }
      }

      MAIN_FPGA_CC_Bypass();
      MAIN_FPGA_Disable();
      MAIN_FPGA_Disable();

#ifdef VERBOSE
      usartx_puts((uint8_t*) "\r\n ** One-Cycle updates: ");
      uart_puti(programcounter);
      usartx_puts((uint8_t*) "\r\n");
#endif

    }


    /* Autonomously operate */
    if (rxdat=='a')
    {
      /* Setup the initial programming */
      MAIN_FPGA_Initialize();
      MAIN_FPGA_CC_Bypass();
      // Just loop and do nothing
      while(usartx_getc()==USART_EMPTY_READ)
      {
        MAIN_ADC_ReadStore();
        if (opPoint_sgn==ADC_POS)
          usartx_putc('+');
        else
          usartx_putc('-');
        }

        /* Reset the FPGA */
        MAIN_FPGA_Disable();
      }


      /* Print Usage */
      if ( (rxdat=='h') || (rxdat=='?') )
         MAIN_Print_Usage();


    }
  }
}


/*
 *
 *
 */
void MAIN_FPGA_Initialize(void)
{
  /* Channel Designations:
   * FB1 (a,b) -- 0E, 0D
   * FB2 (a,b) -- 0C, 0B
   * BB (a,b) -- 14, 13
   * CCP (a,b) -- 12, 11
   * CCN (a,b) -- 10, 0F
   */
#ifdef DEBUG
  usartx_puts((uint8_t*) "\r\n * MAIN_Initialize_Operation *\r\n");
#endif

  SPI_FPGA_DMA_Tx((uint8_t*)"                ", 16); // Reset FPGA
  SPI_FPGA_DMA_Tx_Wait();
  SPI_FPGA_DMA_Tx((uint8_t*)"w0000000w0000000", 16); // Reset FPGA
  SPI_FPGA_DMA_Tx_Wait();
  // Enable All active Channels
  SPI_FPGA_DMA_Tx((uint8_t*)"w0E00001w0D00001", 16); // FB 1 (a,b)
  SPI_FPGA_DMA_Tx_Wait();
  SPI_FPGA_DMA_Tx((uint8_t*)"w0C00001w0B00001", 16); // FB 2
  SPI_FPGA_DMA_Tx_Wait();
  SPI_FPGA_DMA_Tx((uint8_t*)"w1400001w1300001", 16); // BB
  SPI_FPGA_DMA_Tx_Wait();

  MAIN_FPGA_CC_Bypass();

#ifdef DEBUG
  usartx_puts((uint8_t*) " * ------------------------ *\r\n");
#endif
}

```

```

//void MAIN_FPGA_Disable_Bypass(void)
//{
// /* Set the cycloconverter to bypass resonant current,
// * and avoid shorting ouput voltage. Set full-bridge low-side
// * devices on to bypass, and BB low-side on as well
// * low side of each leg on, high side off
// */
// // Set default states to off
//#ifdef VERBOSE
// usartx_puts((uint8_t*) "\r\n * MAIN_FPGA_Disable_Bypass *\r\n");
//#endif
// SPI_FPGA_DMA_Tx((uint8_t*)"w1200002w1000002w1400002w0E00002w0C00002w0000000", 56);
// SPI_FPGA_DMA_Tx_Wait();
//#ifdef VERBOSE
// usartx_puts((uint8_t*) "\r\n * ------------------------ *\r\n");
//#endif
//}

void MAIN_FPGA_CC_Bypass(void)
{
// /* Set the cycloconverter to bypass resonant current,
// * and avoid shorting ouput voltage -- operation near zero vout
// * Turn high side of each leg off, then
// * low side of each leg on
// */
// SPI_FPGA_DMA_Tx((uint8_t*)"w1100000w0F00000", 16);
// SPI_FPGA_DMA_Tx_Wait();
// SPI_FPGA_DMA_Tx((uint8_t*)"w1200002w1000002", 16);
// SPI_FPGA_DMA_Tx_Wait();
  SPI_FPGA_DMA_Tx((uint8_t*)"w1100000w0F00000w1200002w1000002", 32);
  SPI_FPGA_DMA_Tx_Wait();
}

void MAIN_FPGA_CC_Full(void)
{
  // Enable low-side and high-side of both positive and negative sides...
  // not a good idea except for testing
  SPI_FPGA_DMA_Tx((uint8_t*)"w1200001w1100001", 16); // CC P Enable
  SPI_FPGA_DMA_Tx((uint8_t*)"w1000001w0F00001", 16); // CC N Enable
  SPI_FPGA_DMA_Tx_Wait();
}

void MAIN_FPGA_CC_Positive(void)
{
  /* Channel Designations:
   * FB1 (a,b) -- 0E, 0D
   * FB2 (a,b) -- 0C, 0B
   * BB (a,b) -- 14, 13
   * CCP (a,b) -- 12, 11
   * CCN (a,b) -- 10, 0F
   */

  // Assuming: previous mode was CP switching
  // * Set positive CC to bypass
  // 1) disable the positive high-side, default 0: w1100000
  // 2) disable the positive low-side, default 1: w1200002
  // * Set negative CC to bypass (low on, high off)
  // 1) disable the negative high-side, default 0: w0F00000
  // 2) disable the negative low-side, default 1: w1000002
  // * Set positive CC to active
  // 1) enable the positive low-side : w1200001
  // 2) enable the positive high-side : w1100001
  // * Set negative CC to full bypass (low already on, high on)
  // 1) disable the negative high-side, default 1: w0F00002

  SPI_FPGA_DMA_Tx((uint8_t*)"w1100000w1200002w0F00000w1000002w1200001w1100001w0F00002", 56)
      ;
  SPI_FPGA_DMA_Tx_Wait();
}

void MAIN_FPGA_CC_Negative(void)
{
  /* Channel Designations:
   * FB1 (a,b) -- 0E, 0D
   * FB2 (a,b) -- 0C, 0B
   * BB (a,b) -- 14, 13
   * CCP (a,b) -- 12, 11
   * CCN (a,b) -- 10, 0F
   */

  // Assuming: previous mode was CP switching
  // * Set negative CC to bypass
  // 1) disable the negative high-side, default 0: w0F00000
  // 2) disable the negative low-side, default 1: w1000002
  // * Set positive CC to bypass (low on, high off)
  // 1) disable the positive high-side, default 0: w1100000
  // 2) disable the positive low-side, default 1: w1200002
  // * Set negative CC to active
  // 1) enable the negative low-side : w1000001
  // 2) enable the negative high-side : w0F00001
  // * Set positive CC to full bypass (low already on, high on)
  // 1) disable the positive high-side, default 1: w1100002

  SPI_FPGA_DMA_Tx((uint8_t*)"w0F00000w1000002w1100000w1200002w1000001w0F00001w1100002", 56)
      ;
  SPI_FPGA_DMA_Tx_Wait();
}



/*
 *
 *
 */
void MAIN_FPGA_Program_Nonblock(void)
```

```
{
  SPI_FPGA_DMA_Tx(opPoint_adr, len_op);
}


/*
 *
 *
 */
void MAIN_FPGA_Program_Wait(void)
{

  SPI_FPGA_DMA_Tx_Wait();
}

/*
 *
 *
 */
void MAIN_FPGA_Enable(void)
{
#ifdef VERBOSE
  usartx_puts((uint8_t*) "\r\n * MAIN_FPGA_Enable *\r\n");
#endif

  SPI_FPGA_DMA_Tx((uint8_t*)"w0000001w0000001", 16);
  SPI_FPGA_DMA_Tx_Wait();

#ifdef VERBOSE
  usartx_puts((uint8_t*) " * ---------------- *\r\n");
#endif
}


/*
 *
 *
 */
void MAIN_FPGA_Disable(void)
{
#ifdef VERBOSE
  usartx_puts((uint8_t*) "\r\n * MAIN_FPGA_Disable *\r\n");
#endif

  SPI_FPGA_DMA_Tx((uint8_t*)"w0000000w0000000", 16);
  SPI_FPGA_DMA_Tx_Wait();

#ifdef VERBOSE
  usartx_puts((uint8_t*) " * ----------------- *\r\n");
#endif
}

```

```
/*
 *
 *
 */
void MAIN_ADC_ReadStore(void)
{
  uint32_t adcptot = 0;
  uint32_t adcntot = 0;
  uint32_t adcp = 0;
  uint32_t adcn = 0;
  uint8_t i;

#ifdef DEBUG_VERBOSE
  usartx_puts((uint8_t*) "\r\n * MAIN_ADC_ReadStore *\r\n");
#endif

  /* Average the specified number of readings */
  for (i=0; i<ADC_READINGS; i++)
  {
    adcp = SPI_ADC_read(ADC_POS);
    adcn = SPI_ADC_read(ADC_NEG);
    if ( (adcn <= 0x3FFF) && (adcp <= 0x3FFF) )
    {
      adcptot += adcp;
      adcntot += adcn;
    }
#ifdef DEBUG_ERROR
    else
    {
      i--;
      usartx_putc('x');

    }
#endif
  }
  adcptot = adcptot >> 2; // resulting value is 10*ActualVoltage
  adcntot = adcntot >> 2; // resulting value is 10*ActualVoltage

  adcptot = adcptot*VOLTAGE_SCALING;
  adcntot = adcntot*VOLTAGE_SCALING;

  adcVoltage_pos = adcptot / ADC_READINGS;
  adcVoltage_neg = adcntot / ADC_READINGS;

  opPoint_sgn_last = opPoint_sgn;
  if (adcVoltage_pos >= adcVoltage_neg)
  {
    adcVoltage = adcVoltage_pos;
    opPoint_sgn = ADC_POS;
  }
  else
  {
    adcVoltage = adcVoltage_neg;
    opPoint_sgn = ADC_NEG;
  }
#ifdef DEBUG_ADC
  uart_puti(adcVoltage_pos);
  usartx_putc(',');
  uart_puti(adcVoltage_neg);
  usartx_putc(' ');
#endif

#ifdef DEBUG_VERBOSE
  usartx_puts((uint8_t*) "\r\n * ------------------ *\r\n");
#endif
}


/*
 *
 *
 */
void MAIN_Initialize_Peripherals(void)
{
  USARTxInit();
  usartx_puts((uint8_t*) "\r\n");
  usartx_puts((uint8_t*) "UART Init ....... ");
  USART_FlushOutput();
  usartx_puts((uint8_t*) "Done\r\n");

  usartx_puts((uint8_t*) "LED Init ........ ");
  USART_FlushOutput();
  LEDInit();
  usartx_puts((uint8_t*) "Done\r\n");

  usartx_puts((uint8_t*) "SPI Init, ADC ... ");
  USART_FlushOutput();
  SPI_ADC_Init();
  usartx_puts((uint8_t*) "Done\r\n");

  usartx_puts((uint8_t*) "SPI Init, FPGA .. ");
  USART_FlushOutput();
  SPI_FPGA_Init();
  usartx_puts((uint8_t*) "Done\r\n");

  usartx_puts((uint8_t*) "DMA Init, FPGA .. ");
  USART_FlushOutput();
  SPI_FPGA_DMA_Init();
  usartx_puts((uint8_t*) "Done\r\n");

#ifdef USE_DAC
  usartx_puts((uint8_t*) "DAC Init ........ ");
  USART_FlushOutput();
  DAC_OUT_Init();
  usartx_puts((uint8_t*) "Done\r\n");
#endif
```

```
}


/*
 *
 *
 */
void sleep(uint32_t cnt)
{
  while (cnt--);
}
```

### C.2.6 spi_adc.c

```
#include "stm32f10x_conf.h"
#include "spi_adc.h"

void SPI_ADC_Init(void)
{

  /* Enable SPI clock, then the GPIO clock */
  RCC_APB1PeriphClockCmd(SPI_ADC_CLK, ENABLE);
  RCC_APB2PeriphClockCmd(SPI_ADC_GPIO_CLK, ENABLE);


  /* ----- Configure GPIO ----- */
  GPIO_InitTypeDef GPIO_InitStructure;

  /* Configure SPI pins: SCK, MISO and MOSI */
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Pin = SPI_ADC_PIN_SCK;
  GPIO_Init(SPI_ADC_GPIO, &GPIO_InitStructure);

  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Pin = SPI_ADC_PIN_MISO;
  GPIO_Init(SPI_ADC_GPIO, &GPIO_InitStructure);

  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Pin = SPI_ADC_PIN_MOSI;
  GPIO_Init(SPI_ADC_GPIO, &GPIO_InitStructure);

  /* Configure SPI chip-select pin, POS ADC */
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Pin = SPI_ADC_PIN_NSSP;
  GPIO_Init(SPI_ADC_GPIO, &GPIO_InitStructure);

  /* Configure SPI chip-select pin, NEG ADC */
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Pin = SPI_ADC_PIN_NSSN;
  GPIO_Init(SPI_ADC_GPIO, &GPIO_InitStructure);

  /* ----- Configure SPI ----- */
  SPI_InitTypeDef SPI_InitStructure;

  SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
  SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
  SPI_InitStructure.SPI_DataSize = SPI_DataSize_16b;
  SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
  SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
  SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
  SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_8;
  SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
  SPI_InitStructure.SPI_CRCPolynomial = 7;
  SPI_Init(SPI_ADC, &SPI_InitStructure);


  /* Set CS high */
  SPI_ADC_CS_HIGH(ADC_POS);
  SPI_ADC_CS_HIGH(ADC_NEG);

  /* Enable the SPI Peripheral */
    SPI_Cmd(SPI_ADC, ENABLE);
}

uint16_t SPI_ADC_read(uint8_t adc)
{
  SPI_ADC_CS_LOW(adc);
    /*!< Loop until transmit register is empty */
    while (SPI_I2S_GetFlagStatus(SPI_ADC, SPI_I2S_FLAG_TXE) == RESET);

    /*!< Send "dummy" byte through the SPI1 peripheral */
    SPI_I2S_SendData(SPI_ADC, 0xDEAD);

    /*!< Wait to receive a byte */
    while (SPI_I2S_GetFlagStatus(SPI_ADC, SPI_I2S_FLAG_RXNE) == RESET);

    SPI_ADC_CS_HIGH(adc);

    /*!< Return the byte read from the SPI bus */
    return SPI_I2S_ReceiveData(SPI_ADC);
}

void SPI_ADC_CS_LOW(uint8_t adc)
{
  if (adc==ADC_POS)
    GPIO_ResetBits(SPI_ADC_GPIO, SPI_ADC_PIN_NSSP);
  if (adc==ADC_NEG)
    GPIO_ResetBits(SPI_ADC_GPIO, SPI_ADC_PIN_NSSN);
```

```
}

void SPI_ADC_CS_HIGH(uint8_t adc)
{
  if (adc==ADC_POS)
    GPIO_SetBits(SPI_ADC_GPIO, SPI_ADC_PIN_NSSP);
  if (adc==ADC_NEG)
    GPIO_SetBits(SPI_ADC_GPIO, SPI_ADC_PIN_NSSN);
}
```

### C.2.7 spi_adc.h

```
#ifndef SPI_ADC_H_
#define SPI_ADC_H_
#include "stm32f10x_gpio.h"

/* Define the STM32F10x hardware depending on the used evaluation board */
#define SPI_ADC SPI2
#define SPI_ADC_CLK RCC_APB1Periph_SPI2
#define SPI_ADC_GPIO GPIOB
#define SPI_ADC_GPIO_CLK RCC_APB2Periph_GPIOB
#define SPI_ADC_PIN_NSSP GPIO_Pin_0
#define SPI_ADC_PIN_NSSN GPIO_Pin_12
#define SPI_ADC_PIN_SCK GPIO_Pin_13
#define SPI_ADC_PIN_MISO GPIO_Pin_14
#define SPI_ADC_PIN_MOSI GPIO_Pin_15
#define SPI_ADC_IRQn SPI2_IRQn

#define ADC_NEG 0
#define ADC_POS 1

void SPI_ADC_CS_LOW(uint8_t adc);
void SPI_ADC_CS_HIGH(uint8_t adc);

uint16_t SPI_ADC_read(uint8_t adc);

void SPI_ADC_Init(void);

#endif /* SPI_ADC_H_ */
```

### C.2.8 spi_fpga.c

```
#include "stm32f10x_conf.h"
#include "spi_fpga.h"

void SPI_FPGA_Init(void)
{

  /* Enable SPI clock, the GPIO clock, and the DMA clock */
  RCC_APB2PeriphClockCmd(SPI_FPGA_CLK, ENABLE);
  RCC_APB2PeriphClockCmd(SPI_FPGA_GPIO_CLK, ENABLE);
  //RCC_AHBPeriphClockCmd(SPI_FPGA_DMA_CLK, ENABLE);

  /* ----- Configure GPIO ----- */
  GPIO_InitTypeDef GPIO_InitStructure;

  /* Configure SPI pins: SCK, MISO and MOSI */
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Pin = SPI_FPGA_PIN_SCK;
  GPIO_Init(SPI_FPGA_GPIO, &GPIO_InitStructure);

  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Pin = SPI_FPGA_PIN_MISO;
  GPIO_Init(SPI_FPGA_GPIO, &GPIO_InitStructure);

  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Pin = SPI_FPGA_PIN_MOSI;
  GPIO_Init(SPI_FPGA_GPIO, &GPIO_InitStructure);

  /* Configure SPI chip-select pin */
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Pin = SPI_FPGA_PIN_NSS;
  GPIO_Init(SPI_FPGA_GPIO, &GPIO_InitStructure);


  /* ----- Configure SPI ----- */
  SPI_InitTypeDef SPI_InitStructure;

  SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
  SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
  SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
  SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
  SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
  SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
  SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2;
  SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
  SPI_InitStructure.SPI_CRCPolynomial = 7;
  SPI_Init(SPI_FPGA, &SPI_InitStructure);

  /* Set CS high */
  SPI_FPGA_CS_HIGH();

  /* Enable the SPI Peripheral */
    SPI_Cmd(SPI_FPGA, ENABLE);

}

```

```
void SPI_FPGA_DMA_Init(void)
{
  RCC_AHBPeriphClockCmd(SPI_FPGA_DMA_CLK, ENABLE);

  DMA_DeInit(SPI_FPGA_Tx_DMA_Channel);

  /* ----- Configure DMA ----- */
  DMA_InitTypeDef DMA_InitStructure;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_VeryHigh;
  DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;

  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)SPI_FPGA_DR_Base;
  DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;

  /* DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t) 0x0000;
   * DMA_InitStructure.DMA_BufferSize = (uint32_t) 0x0000; */

  DMA_Init(SPI_FPGA_Tx_DMA_Channel, &DMA_InitStructure);

    SPI_I2S_DMACmd(SPI_FPGA, SPI_I2S_DMAReq_Tx, ENABLE);

}

void SPI_FPGA_DMA_Tx(uint8_t* data, uint32_t len)
{
    /* Wait for the SPI peripheral to finish */
    while (SPI_I2S_GetFlagStatus(SPI_FPGA, SPI_I2S_FLAG_TXE)==RESET);
    while (SPI_I2S_GetFlagStatus(SPI_FPGA, SPI_I2S_FLAG_BSY)==SET);
    /* Disable the SPI Peripheral */
    SPI_Cmd(SPI_FPGA, DISABLE);

    SPI_FPGA_DMA_Init();

    /* Set the transfer length (Channel Number of Data to Transfer Register) */
  SPI_FPGA_Tx_DMA_Channel->CNDTR = len;
  /* Set the base memory address (Channel Memory Address Register) */
  SPI_FPGA_Tx_DMA_Channel->CMAR = (uint32_t)data;

  /* Enable the DMA channel */
  DMA_Cmd(SPI_FPGA_Tx_DMA_Channel, ENABLE);

  /* Select the slave and start the SPI transfer */
    SPI_FPGA_CS_LOW();
    SPI_Cmd(SPI_FPGA, ENABLE);

    /* Call SPI_FPGA_DMA_Tx_Wait() to finish transfer */
}
```

```

void SPI_FPGA_DMA_Tx_Wait(void)
{
  /* Wait for the DMA transfer to complete, then disable the channel */
    while(DMA_GetFlagStatus(SPI_FPGA_Tx_DMA_FLAG)==RESET);
    DMA_Cmd(SPI_FPGA_Tx_DMA_Channel, DISABLE);
    /* Wait for the SPI peripheral to finish, then deselect slave */
    while (SPI_I2S_GetFlagStatus(SPI_FPGA, SPI_I2S_FLAG_TXE)==RESET);
    while (SPI_I2S_GetFlagStatus(SPI_FPGA, SPI_I2S_FLAG_BSY)==SET);
    SPI_FPGA_CS_HIGH();
}

void SPI_FPGA_puts(const uint8_t* data)
{
  SPI_FPGA_CS_LOW();
  while (*data) {
    while (SPI_I2S_GetFlagStatus(SPI_FPGA, SPI_I2S_FLAG_TXE)==RESET);
    SPI_I2S_SendData(SPI_FPGA, *data++);
  }
  while (SPI_I2S_GetFlagStatus(SPI_FPGA, SPI_I2S_FLAG_TXE)==RESET);
  SPI_FPGA_CS_HIGH();
}
void SPI_FPGA_puts_fast(uint8_t* data)
{
  SPI_FPGA_CS_LOW();
  while (*data) {
    SPI_FPGA->DR = *data++;
  }
  SPI_FPGA_CS_HIGH();
}

void SPI_FPGA_putc(uint8_t data)
{
  SPI_FPGA_CS_LOW();
  while (SPI_I2S_GetFlagStatus(SPI_FPGA, SPI_I2S_FLAG_TXE)==RESET);
  SPI_I2S_SendData(SPI_FPGA, data);
  while (SPI_I2S_GetFlagStatus(SPI_FPGA, SPI_I2S_FLAG_TXE)==RESET);
  SPI_FPGA_CS_HIGH();
}

void SPI_FPGA_CS_LOW(void)
{
  GPIO_ResetBits(SPI_FPGA_GPIO, SPI_FPGA_PIN_NSS);
}

void SPI_FPGA_CS_HIGH(void)
{
  GPIO_SetBits(SPI_FPGA_GPIO, SPI_FPGA_PIN_NSS);
}
```

### C.2.9 spi_fpga.h

```
#ifndef SPI_FPGA_H_
#define SPI_FPGA_H_
#include "stm32f10x_gpio.h"

/* Define the STM32F10x hardware depending on the used evaluation board */
#define SPI_FPGA SPI1
#define SPI_FPGA_CLK RCC_APB2Periph_SPI1
#define SPI_FPGA_GPIO GPIOA
#define SPI_FPGA_GPIO_CLK RCC_APB2Periph_GPIOA
#define SPI_FPGA_PIN_NSS GPIO_Pin_4
#define SPI_FPGA_PIN_SCK GPIO_Pin_5
#define SPI_FPGA_PIN_MISO GPIO_Pin_6
#define SPI_FPGA_PIN_MOSI GPIO_Pin_7
#define SPI_FPGA_IRQn SPI1_IRQn

#define SPI_FPGA_DMA DMA1
#define SPI_FPGA_DMA_CLK RCC_AHBPeriph_DMA1
#define SPI_FPGA_Tx_DMA_Channel DMA1_Channel3
#define SPI_FPGA_Tx_DMA_FLAG DMA1_FLAG_TC3
#define SPI_FPGA_DR_Base 0x4001300C

void SPI_FPGA_CS_LOW(void);
void SPI_FPGA_CS_HIGH(void);

void SPI_FPGA_Init(void);
void SPI_FPGA_puts(const uint8_t* data);
void SPI_FPGA_puts_fast(uint8_t* data);
void SPI_FPGA_putc(uint8_t data);
void SPI_FPGA_DMA_Tx(uint8_t* data, uint32_t len);
void SPI_FPGA_DMA_Tx_Wait(void);
void SPI_FPGA_DMA_Init(void);

#endif /* SPI_FPGA_H_ */
```

### C.2.10 usart.c

```
#include "usart.h"
#include "stm32f10x_usart.h"

/* Private define ------------------------------------------------------------*/

/*
 * constants and macros
 */

/* size of RX/TX buffers */
#define USARTx_RX_BUFFER_SIZE 32
#define USARTx_TX_BUFFER_SIZE 4096
```

```
#define USARTx_RX_BUFFER_MASK ( USARTx_RX_BUFFER_SIZE - 1)
#define USARTx_TX_BUFFER_MASK ( USARTx_TX_BUFFER_SIZE - 1)
#if ( USARTx_RX_BUFFER_SIZE & USARTx_RX_BUFFER_MASK )
  #error RX buffer size is not a power of 2
#endif
#if ( USARTx_TX_BUFFER_SIZE & USARTx_TX_BUFFER_MASK )
  #error TX buffer size is not a power of 2
#endif

/*
 * module global variables
 */
__IO uint8_t USARTx_TxBuf[USARTx_TX_BUFFER_SIZE];
__IO uint8_t USARTx_RxBuf[USARTx_RX_BUFFER_SIZE];
__IO uint8_t USARTx_TxHead;
__IO uint8_t USARTx_TxTail;
__IO uint8_t USARTx_RxHead;
__IO uint8_t USARTx_RxTail;
__IO uint8_t USARTx_RxErr;

char ASCIIU[] = {0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x41, 0x42, 0
     x43, 0x44, 0x45, 0x46};
char ASCIIL[] = {0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x61, 0x62, 0
     x63, 0x64, 0x65, 0x66};

/**
  * @brief This function handles USARTx global interrupt request.
  * @param None
  * @retval None
  */
void USARTx_IRQHandler(void)
{
  uint8_t newhead;
  uint8_t newtail;

  /* Rx Interrupt Needs Servicing */
  if(USART_GetITStatus(USARTx, USART_IT_RXNE) != RESET)
  {
    /* calculate next buffer head */
    newhead = ( USARTx_RxHead + 1) & USARTx_RX_BUFFER_MASK;

    if ( newhead == USARTx_RxTail ) {
      /* error: receive buffer overflow */
      USARTx_RxErr++;
    }else{
      /* update head */
      USARTx_RxHead = newhead;
      /* store the data */
    }
    // Read the data regardless of buffer overflow
    USARTx_RxBuf[newhead] = USART_ReceiveData(USARTx);
  }

  /* Tx Interrupt Needs Servicing */
  if(USART_GetITStatus(USARTx, USART_IT_TXE) != RESET)
  {
    if ( USARTx_TxHead != USARTx_TxTail) {
      /* calculate new buffer tail */
      newtail = (USARTx_TxTail + 1) & USARTx_TX_BUFFER_MASK;
      /* update tail */
      USARTx_TxTail = newtail;
      /* write the data to USART */
      USART_SendData(USARTx, USARTx_TxBuf[newtail]);
    }
    else {
      /* tx buffer empty, disable TXE interrupt */
      USART_ITConfig(USARTx, USART_IT_TXE, DISABLE);
    }
  }

}



/**
  * @brief Configures COM port.
  * @param COM: Specifies the COM port to be configured.
  * This parameter can be one of following parameters:
  * @arg COM1
  * @arg COM2
  * @param USART_InitStruct: pointer to a USART_InitTypeDef structure that
  * contains the configuration information for the specified USART peripheral.
  * @retval None
  */
void USARTxInit(void)
{
  /* USARTx configuration ------------------------------------------------------*/
  USART_InitTypeDef USART_InitStructure;
  GPIO_InitTypeDef GPIO_InitStructure;
  NVIC_InitTypeDef NVIC_InitStructure;

  /* Enable the USARTx Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = USARTx_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  /* Rx and Tx FIFO Setup */
  USARTx_TxHead = 0;
  USARTx_TxTail = 0;
  USARTx_RxHead = 0;
  USARTx_RxTail = 0;
  USARTx_RxErr = 0;

  /* USARTx configured as follow:
        - BaudRate = 9600 baud
        - Word Length = 8 Bits
        - One Stop Bit
        - No parity
        - Hardware flow control disabled (RTS and CTS signals)
        - Receive and transmit enabled
  */
  USART_InitStructure.USART_BaudRate = 115200;
  USART_InitStructure.USART_WordLength = USART_WordLength_8b;
  USART_InitStructure.USART_Parity = USART_Parity_No;
  USART_InitStructure.USART_StopBits = USART_StopBits_1;

  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
  USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

  /* begin old init func ------------------------------------------------------*/

  /* Enable GPIO clock */
  RCC_APB2PeriphClockCmd(USARTx_TX_GPIO_CLK | USARTx_RX_GPIO_CLK | RCC_APB2Periph_AFIO,
      ENABLE);
  RCC_APB1PeriphClockCmd(USARTx_CLK, ENABLE);

  /* Configure USART Tx as alternate function push-pull */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Pin = USARTx_TX_PIN;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(USARTx_TX_GPIO_PORT, &GPIO_InitStructure);

  /* Configure USART Rx as input floating */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_InitStructure.GPIO_Pin = USARTx_RX_PIN;
  GPIO_Init(USARTx_RX_GPIO_PORT, &GPIO_InitStructure);

  /* USART configuration */
  USART_Init(USARTx, &USART_InitStructure);

  /* Enable USART */
  USART_Cmd(USARTx, ENABLE);

  /* end old init function ----------------------------------------------------*/

  /* Enable the EVAL_COM1 Receive interrupt: this interrupt is generated when the
     EVAL_COM1 receive data register is not empty */
  USART_ITConfig(USARTx, USART_IT_RXNE, ENABLE);

  /* Enable the EVAL_COM1 Transmoit interrupt: this interrupt is generated when the
     EVAL_COM1 transmit data register is empty */
  /* This shouldn't be necessary, as we assume that the TX buffer is initialy
     empty at initialization, but it shouldn't hurt */
  USART_ITConfig(USARTx, USART_IT_TXE, ENABLE);
} /* USARTxInit */


```

```

void USART_FlushOutput(void) {
  while ( USARTx_TxHead != USARTx_TxTail );
}

/*************************************************************************
Function: uart_getc()
Purpose: return byte from ringbuffer
Returns: lower byte: received byte from ringbuffer
         higher byte: last receive error
**************************************************************************/
uint8_t usartx_getc(void)
{
  uint8_t newtail;
  uint8_t data;

  if ( USARTx_RxHead == USARTx_RxTail ) {
    return USART_EMPTY_READ;//UARTx_NO_DATA; /* no data available */
  }

  /* calculate new buffer tail */
  newtail = (USARTx_RxTail + 1) & USARTx_RX_BUFFER_MASK;

  /* get data from receive buffer */
  data = USARTx_RxBuf[newtail];
  /* update tail */
  USARTx_RxTail = newtail;

  return data;

}/* uart_getc */

uint8_t usartx_getc_block(void)
{
  uint8_t data;

  data = usartx_getc();
  while (data==USART_EMPTY_READ) data=usartx_getc();
  return data;

}/* uart_getc_block */
/*************************************************************************
Function: uart_putc()
Purpose: write byte to ringbuffer for transmitting via UART
Input: byte to be transmitted
Returns: none
**************************************************************************/
void usartx_putc(const uint8_t data)
{
  uint8_t newhead;

  newhead = (USARTx_TxHead + 1) & USARTx_TX_BUFFER_MASK;

  /* this is just a bad idea */
  while ( newhead == USARTx_TxTail ){
    ;/* wait for free space in buffer */
  }

  /* Store data in buffer */
  USARTx_TxBuf[newhead] = data;
  /* update buffer head */
  USARTx_TxHead = newhead;

  /* enable the TXE interrupt */
  USART_ITConfig(USARTx, USART_IT_TXE, ENABLE);

}/* uart_putc */


/*************************************************************************
Function: uart_puts()
Purpose: transmit string to UART
Input: string to be transmitted
Returns: none
**************************************************************************/
void usartx_puts(const uint8_t *s )
{
    while (*s)
      usartx_putc(*s++);

}/* uart_puts */

void USART_putsn(const uint8_t *s, uint32_t len )
{
    while (len--)
      usartx_putc(*s++ );

}/* uart_puts */


//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////



/*************************************************************************
Title: UART addon-library
Author: Martin Thomas <eversmith@heizung-thomas.de>
          http://www.siwawi.arubi.uni-kl.de/avr_projects
Software: AVR-GCC 3.3/3.4, Peter Fleury's UART-Library

DESCRIPTION:

USAGE:
    Refere to the header file uart_addon.h for a description of the routines.

```

```
****************************************************************************/


/****************************************************************************
Function: uart_puti()
Purpose: transmit integer as ASCII to UART
Input: integer value
Returns: none
****************************************************************************/
void uart_puti( const uint32_t value )
{
  uint8_t radix = 10;
  uint32_t v;
  uint8_t str[32];
  uint8_t dig[] =
      "0123456789"
          "abcdefghijklmnopqrstuvwxyz";
  uint8_t n = 0, neg = 0;
  uint8_t *p, *q;
  uint8_t c;

  if (radix == 10 && value < 0) {
      v = -value;
      neg = 1;
  } else {
    v = value;
  }
  do {
      str[n++] = dig[v%radix];
      v /= radix;
  } while (v);
  if (neg)
      str[n++] = '-';
  str[n] = '\0';

  for (p = str, q = p + (n-1); p < q; ++p, --q)
      c = *p, *p = *q, *q = c;

  usartx_puts(str);
}
/* uart_puti */

/****************************************************************************
Function: uart_puthex_nibble()
Purpose: transmit lower nibble as ASCII-hex to UART
Input: byte value
Returns: none
****************************************************************************/
void uart_puthex_nibble(const uint8_t b)
{
    /* uint8_t c = b & 0x0f; */
    /* if (c>9) c += 'A'-10; */
    /* else c += '0'; */
  usartx_putc(ASCIIU[b&0x0F]);
} /* uart_puthex_nibble */



/***********************************************************************
Function: uart_puthex_byte()
Purpose: transmit upper and lower nibble as ASCII-hex to UART
Input: byte value
Returns: none
***********************************************************************/
void uart_puthex_byte(const uint8_t b)
{
    uart_puthex_nibble(b>>4);
    uart_puthex_nibble(b);
} /* uart_puthex_byte */

void uart_puthex_addr(const uint32_t b)
{
  uart_puthex_byte((b&0xff000000)>>24);
  uart_puthex_byte((b&0x00ff0000)>>16);
  uart_puthex_byte((b&0x0000ff00)>>8);
  uart_puthex_byte((b&0x000000ff)>>0);
}
/***********************************************************************
Function: uart_putbin_byte()
Purpose: transmit byte as ASCII-bin to UART
Input: byte value
Returns: none
***********************************************************************/
void uart_putbin_byte(const uint8_t b)
{
  uint8_t i;
  for (i=7;i>=0;i--) {
    if (b & (1<<i)) {
      usartx_putc('1');
    }
    else {
      usartx_putc('0');
    }
  }
} /* uart_putbin_byte */
```

### C.2.11 usart.h

```

#define USARTx USART2
#define USARTx_CLK RCC_APB1Periph_USART2
#define USARTx_TX_PIN GPIO_Pin_2
#define USARTx_TX_GPIO_PORT GPIOA
```

```
#define USARTx_TX_GPIO_CLK RCC_APB2Periph_GPIOA
#define USARTx_RX_PIN GPIO_Pin_3
#define USARTx_RX_GPIO_PORT GPIOA
#define USARTx_RX_GPIO_CLK RCC_APB2Periph_GPIOA
#define USARTx_IRQn USART2_IRQn
#define USARTx_IRQHandler USART2_IRQHandler

#define USART_EMPTY_READ 0
void USARTx_IRQHandler(void);
void USARTxInit(void);
void USART_FlushOutput(void);
uint8_t usartx_getc(void);
uint8_t usartx_getc_block(void);
void usartx_putc(const uint8_t data);
void usartx_puts(const uint8_t *s );
void uart_puti( const uint32_t value );
void uart_puthex_nibble(const uint8_t b);
void uart_puthex_addr(const uint32_t b);
void uart_puthex_byte(const uint8_t b);
void uart_putbin_byte(const uint8_t b);
void USART_putsn(const uint8_t *s, uint32_t len );
```

# *Appendix D*
# *Bibliography*

[1] P. Krein and R. Balog, "Cost-effective hundred-year life for single-phase inverters and rectifiers in solar and led lighting applications based on minimum capacitance requirements and a ripple power port," in *Applied Power Electronics Conference and Exposition, 2009. APEC 2009. Twenty-Fourth Annual IEEE*, Feb. 2009, pp. 620 –625.

[2] C. Bush and B. Wang, "A single-phase current source solar inverter with reduced-size dc link," in *Energy Conversion Congress and Exposition, 2009. ECCE 2009. IEEE*, Sept. 2009, pp. 54 –59.

[3] T. Basso and R. DeBlasio, "IEEE 1547 series of standards: interconnection issues," *Power Electronics, IEEE Transactions on*, vol. 19, no. 5, pp. 1159–1162, 2004.

[4] R. Dugan, T. Key, G. Ball, E. Solutions, and T. Knoxville, "On standards for interconnecting distributed resources," in *Rural Electric Power Conference, 2005*, 2005, p. D2.

[5] I. Howell Jr and I. Kolodny, "IEEE recommended practice for grounding of industrial and commercial power systems," in *The Institute of Electrical and Electronics Engineers*, 1982, pp. 11–28.

[6] S. Choi, B. Lee, and P. Enjeti, "New 24-Pulse Diode Rectifier Systems for Utility Interface of High-Power AC Motor Drives," *IEEE TRANSACTIONS ON INDUSTRY APPLICATIONS*, vol. 33, no. 2, p. 531, 1997.

[7] N. Schibli, T. Nguyen, and A. Rufer, "A three-phase multilevel converter for high-power induction motors," *Power Electronics, IEEE Transactions on*, vol. 13, no. 5, pp. 978–986, 1998.

[8] L. Tolbert, F. Peng, and T. Habetler, "Multilevel converters for large electric drives," *Industry Applications, IEEE Transactions on*, vol. 35, no. 1, pp. 36–44, 1999.

[9] L. Hansen, L. Helle, F. Blaabjerg, E. Ritchie, S. Munk-Nielsen, H. Bindner, P. Sørensen, and B. Bak-Jensen, "Conceptual survey of generators and power electronics for wind turbines," 2001.

[10] S. Song, S. Kang, and N. Hahm, "Implementation and control of grid connected AC-DC-AC power converter for variable speed wind energy conversion system," in *Applied Power Electronics Conference and Exposition, 2003. APEC'03. Eighteenth Annual IEEE*, vol. 1, 2003.

[11] D. Chen, "Research on single stage combined uninterruptible ac-dc converters with high power factor," *Power Electronics Specialists Conference, 2008. PESC 2008. IEEE*, pp. 3354–3359, June 2008.

[12] J. Choi, J. Kwon, J. Jung, and B. Kwon, "High-performance online UPS using three-leg-type converter," *Industrial Electronics, IEEE Transactions on*, vol. 52, no. 3, pp. 889–897, 2005.

[13] S. Kjaer, J. Pedersen, and F. Blaabjerg, "A review of single-phase grid-connected inverters for photovoltaic modules," *Industry Applications, IEEE Transactions on*, vol. 41, no. 5, pp. 1292–1306, 2005.

[14] H. Haeberlin, "Evolution of Inverters for Grid connected PV-Systems from 1989 to 2000," *measurement*, vol. 2, p. 1.

[15] S. Kjaer, J. Pedersen, and F. Blaabjerg, "Power inverter topologies for photovoltaic modules-a review," in *Industry Applications Conference, 2002. 37th IAS Annual Meeting. Conference Record of the*, vol. 2, 2002.

[16] Y. Xue, L. Chang, S. B. Kjaer, J. Bordonau, and T. Shimizu, "Topologies of single-phase inverters for small distributed power generators: an overview," *Power Electronics, IEEE Transactions on*, vol. 19, no. 5, pp. 1305–1314, Sept. 2004.

[17] Q. Li and P. Wolfs, "A Review of the Single Phase Photovoltaic Module Integrated Converter Topologies With Three Different DC Link Configurations," *IEEE Transactions on Power Electronics*, vol. 23, no. 3, pp. 1320–1333, 2008.

[18] E. Roman, R. Alonso, P. Ibanez, S. Elorduizapatarietxe, and D. Goitia, "Intelligent PV module for grid-connected PV systems," *IEEE Transactions on Industrial Electronics*, vol. 53, no. 4, pp. 1066–1073, 2006.

[19] A. Lohner, T. Meyer, and A. Nagel, "A new panel-integratable inverter concept for grid-connected photovoltaic systems," in *Industrial Electronics, 1996. ISIE '96., Proceedings of the IEEE International Symposium on*, vol. 2, Jun 1996, pp. 827–831 vol.2.

[20] M. Kamil, "Grid-connected solar microinverter reference design using a dspic digital signal controller," Microchip Technology Inc., Tech. Rep., 2010.

[21] A. Trubitsyn, B. Pierquet, A. Hayman, G. Gemache, C. Sullivan, and D. Perreault, "High-efficiency inverter for photovoltaic applications," in *Energy Conversion Congress and Exposition, 2010. ECCE. IEEE*, 2010.

[22] P. Krein and R. Balog, "Cost-Effective Hundred-Year life for Single-Phase inverters and rectifiers in solar and LED lighting applications based on minimum capacitance requirements and a ripple power port," in *Applied Power Electronics Conference and Exposition, 2009. APEC 2009. Twenty-Fourth Annual IEEE*, 2009, pp. 620–625.

[23] C. Bush and B. Wang, "A single-phase current source solar inverter with reduced-size DC link," in *Energy Conversion Congress and Exposition, 2009. ECCE. IEEE*, 2009, pp. 54–59.

[24] C. Henze, H. Martin, and D. Parsley, "Zero-voltage switching in high frequency power converters using pulse width modulation," in *Applied Power Electronics Conference and Exposition, 1988. APEC '88. Conference Proceedings 1988., Third Annual IEEE*, Feb 1988, pp. 33–40.

[25] K. Sheng, F. Udrea, G. Amaratunga, and P. Palmer, "Unconventional Behaviour of the CoolMOS device."

[26] L. Lorenz, G. Deboy, A. Knapp, and M. Marz, "COOLMOS TM-a new milestone in high voltage power MOS," in *Power Semiconductor Devices and ICs, 1999. ISPSD'99. Proceedings., The 11th International Symposium on*, 1999, pp. 3–10.

[27] L. Lorenz, I. Zverev, A. Mittal, and J. Hancock, "CoolMOS-a new approach towards system miniaturization and energysaving," in *Industry Applications Conference, 2000. Conference Record of the 2000 IEEE*, vol. 5, 2000.

[28] J. Cooper Jr, M. Melloch, R. Singh, A. Agarwal, and J. Palmour, "Status and prospects for SiC power MOSFETs," *Electron Devices, IEEE Transactions on*, vol. 49, no. 4, pp. 658–664, 2002.

[29] B. Baliga, "Trends in power semiconductor devices," *IEEE Transactions on Electron Devices*, vol. 43, no. 10, pp. 1717–1731, 1996.

[30] S. Almer and U. Jonsson, "Dynamic phasor analysis of periodic systems," *Automatic Control, IEEE Transactions on*, vol. 54, no. 8, pp. 2007 –2012, Aug. 2009.

[31] G. Strang, *Linear Algebra and Its Applications.* Brooks Cole, February 1988.

[32] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[33] W. Bower, C. Whitaker, W. Erdman, M. Behnke, and M. Fitzgerald, "Performance test protocol for evaluating inverters used in Grid-Connected photovoltaic systems," 2004.

[34] E. A. Guillemin, *The Mathematics of Circuit Analysis.* Wiley, 1949.