HARDWARE IMPLEMENTATION OF A SENSORLESS CONTROL ALGORITHM FOR PERMANENT MAGNET SYNCHRONOUS MOTORS

A Thesis Presented

by

Gabriel Eirea

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Electrical Engineering

Northeastern University

Boston, Massachusetts

July 23, 2001

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Hardware implementation of a sensorless control algorithm for Permanent Magnet Synchronous Motors.

Author: Gabriel Eirea.

Department: Electrical and Computer Engineering.

Approved for Thesis Requirement of the Master of Science Degree:

Thesis Advisor: Prof. Aleksandar M. Stanković	Date
Thesis Co-Advisor: Prof. Gilead Tadmor	Date
Thesis Reader: Prof. Miriam Leeser	Date
Thesis Reader: Prof. Brad Lehman	Date
Department Chair: Prof. Fabrizio Lombardi	Date

Graduate School Notified of Acceptance:

Director of the Graduate School: Prof. Yaman Yener	Date

"Did I hinder you much on the road toward your goal?"

"Hinder me! Oh Goldmund, no one furthered me as much as you did.

You created difficulties for me, but I am no enemy of difficulties.

I've learned from them, I've partly overcome them."

Hermann Hesse in *Narcissus and Goldmund*

Abstract

The position-sensorless control of AC motor drives has been an important research area during the last decade. Significant results have been obtained in laboratory experiments. However, the intensive computations required, and the complexity of the algorithms involved are serious obstacles for the implementation of these results in industrial drives. The reliability and cost reduction gained by removing the position sensor usually do not justify the introduction of significantly more powerful and expensive processors.

In this thesis, we take one particular sensorless control algorithm for Permanent Magnet Synchronous Motors, which presents an attractive numerical structure and good performance in the laboratory, and implement it in a low-cost custom designed board. The board design is based on a 16-bit fixed-point Digital Signal Processor (DSP) and a low-cost Field Programmable Gate Array (FPGA), which work in parallel to perform the signal acquisition, position and speed estimation, controller computation, and Pulse Width Modulation (PWM) generation. The introduction of programmable logic in the circuit provides an opportunity to relieve the processor from certain time-consuming tasks, liberating resources to perform the heaviest computations. The FPGA was programmed in VHDL, an industry standard language, which concedes the possibility of easily converting the design into an ASIC. To program the DSP, finite word-length effects of the fixed-point operations were addressed.

The board was built and tested in the laboratory. Experimental results are presented, showing a satisfactory performance of this implementation over a wide range of rotor speeds and torque loads.

Acknowledgments

I had the opportunity to study this Master's program at Northeastern University thanks to a lot of people and organizations to whom I am very grateful. I came to the US under a Fulbright scholarship administered by LASPAU; I would like to thank the Fulbright Commission in Uruguay, the Fulbright Program, LASPAU and all their staff. Special thanks to Renee Hahn, my Program Advisor at LASPAU, for her constant support.

I am very grateful to my home institution, *Instituto de Ingeniería Eléctrica* of *Universidad de la República*, and looking forward to going back to Uruguay and sharing my experience with my colleagues and students.

Thanks to Northeastern University for providing a stimulating atmosphere and infrastructure for my research and studies.

I wish to express my sincere gratitude to Prof. Aleksandar Stanković for his unconditional support and guidance, and Prof. Gilead Tadmor for his assistance and confidence in my work. I would like to thank also the other members of my Committee: Prof. Miriam Leeser and Prof. Brad Lehman for their valuable comments.

Thanks to Prof. Miriam Leeser for providing Altera's development software and chips, thru the Altera University Program, and to Dr. Paul Kettle from Analog Devices for providing the development software and emulator for the DSP.

Special thanks to Vladan Petrović for his friendship and advice. Since my work builds completely on his doctoral dissertation, I required his assistance many times and always found a patient and generous response. Also thanks to my colleagues and friends at Northeastern for all the good moments we spent together.

Finally, I would like to express my gratitude and love to my family. Thanks to uncle Alberto and aunt Edith for their support. To my parents Luis and Cristina, and my wife Sônia for their love and encouragement. This thesis would not have been possible without you.

Gabriel Eirea
Boston, Massachusetts
July, 2001

Contents

Chapter 1

Introduction

In the industrial and academic community there is an increasing interest in position-sensorless operation of AC motor drives, motivated by the desire to reduce the system cost and improve its reliability. The position sensors are in general fragile mechanical parts, which must be installed and aligned carefully. Their cost represents a significant part of the overall system cost, but also their presence complicates the in-field maintenance, during which the sensor often breaks or gets misaligned. Therefore, the elimination of the need for a position sensor is regarded as a major advantage in the industrial community. Even when the system has a position sensor, the algorithms developed for sensorless operation can provide useful information for diagnostics, initialization and/or resolution improvement of the position feedback.

The research and development in this area have yielded significant results over the last few years. There are basically two approaches reported in the literature. The first is based on the estimation of the motor back-emf, and the extraction of position information from this signal [1, 2, 3, 4, 5]. The main drawback of these methods is

that their performance at low speed seriously deteriorates, since the back-emf value vanishes close to standstill. The other approach relies on the dependence of motor inductances on the rotor position, due to magnetic saliency. In this approach, an auxiliary signal is usually injected in the motor, and the response of the electrical subsystem to this signal is used to estimate the rotor position [6, 7, 8, 9, 10, 11].

Although there are relevant results reported, there is an important gap between the laboratory experiments and the industry applications. Usually the position estimation algorithms require heavy computations which cannot be carried out by the low-cost processors used in industry. In this thesis, we concentrate in the sensorless algorithm for Permanent Magnet Synchronous Motors described in [12], which we will call Petrović's algorithm, and present a hardware implementation which targets a low-cost architecture. Our contribution is to demonstrate in practice that it is possible to implement the algorithm using a 16 bit fixed-point DSP, an FPGA and two A/D converters. A digital board was designed and constructed for this specific task. The algorithm was converted to a fixed-point version and coded into the DSP. The modified PWM used in this algorithm was programmed in the FPGA, together with the A/D converters control and auxiliary functions.

The use of FPGAs to generate PWM signals has been reported in two early works [13, 14]. The former presents a complete space vector PWM generator in one FPGA, with programmable PWM switching frequency, deadtime, and frequency, amplitude and phase of the stator voltage vector. The latter presents the design of three PWM blocks for an FPGA: a space vector block, a random PWM block and a deadtime block. A more recent work [15] also presents the design of three types of space vector PWM generator: the alternating zero vector sequence, the symmetric sequence

and the bus clamped sequence. Other papers reporting the use of FPGA for PWM generation and control of switched reluctance motor drives, ac-voltage regularion systems, and a wheelchair system are [16], [17] and [18], respectively.

This thesis continues in Chapter 2 with an overview of the characteristics of a PMSM and a description of Petrović's algorithm. In Chapter 3 we describe the proposed hardware, its design and construction. Following, in Chapter 4 we address the finite word-length effects and show simulations comparing the difference between the floating-point and the fixed-point versions of the algorithm. In Chapter 5 we describe the FPGA programming for the modified PWM generation and signal acquisition. In Chapter 6 we show the experimental results. Finally, in Chapter 7 we outline the conclusions of our work and possible improvements.

Chapter 2

The position estimation algorithm for PMSM

In this chapter, we describe briefly the characteristics of a PMSM and present mathematical models suitable for position estimation (in the $\alpha\beta$ axis) and control (in the dq axis). Following, we describe the position estimation and control algorithms that will be used in this work.

2.1 Permanent Magnet Synchronous Motors (PMSM)

2.1.1 PMSM description

Permanent Magnet (PM) motors use magnets attached to the rotor to produce the air gap magnetic flux, while the stator holds a set of current-carrying conductors. The interaction between the magnetic flux and the currents in the stator produce torque [19].

Since the air gap magnetic flux is generated without external excitation, PM motors are very efficient. They also achieve high values of power density and torque-to-inertia ratio, which makes them attractive for applications that need a fast dynamic response. It is generally accepted that PM motors will not challenge induction motors in the general-purpose variable-speed drive market, specially in power ranges over 50kW. However, there is a wide range of applications where PM motors are good candidates, like servo actuators, commercial-residential applications and electric vehicles.

The magnets can be mounted on the surface of the rotor or buried inside it. In the first case, the magnets can be projecting outside of the surface of the rotor, with an air space between the adjacent magnets, or inset into the rotor, with an iron tooth filling the space between adjacent magnets. Since the permeability of the magnet is similar to that of the air, the projecting type has an uniform (and rather large) air gap between the rotor and the stator, resulting in constant phase inductances during a rotation. On the other hand, the inset type presents salient rotor poles and consequently the inductances depend on the rotor position. Finally, the buried magnets are more difficult to construct, but they have the advantage of mechanical robustness and a smaller air gap, while presenting a salient rotor like the inset type.

There are two types of PM motors: (1) synchronous or sinusoidal, and (2) switched or trapezoidal. The former is designed with stator windings that are distributed over multiple slots in order to approximate a sinusoidal distribution. The latter has stator windings which are concentrated into narrow belts. As a consequence, the back-EMF waveforms generated are sinusoidal in the first case and trapezoidal in the second [20]. The trapezoidal PM motor is also called brushless DC motor because it has almost identical back-EMF-to-speed and torque-to-current relationships as the DC motor.

On the other hand, the sinusoidal type, called PM synchronous motor (PMSM), has more complex characteristics and requires a more elaborate controller.

In order to produce torque, the excitation has to be precisely synchronized with the rotor frequency and phase (i.e., speed and position). Therefore, the rotor's absolute angular position has to be measured and fed back to the controller. In the case of the PMSM, a PWM-controlled inverter is used to generate sinusoidal excitation waveforms, with the proper amplitude, frequency and phase (self-synchronization).

The most common method of measuring the rotor position is to mount an absolute or relative angular position sensor on the rotor shaft. An alternative method is to obtain rotor position information directly from current and voltage's waveforms (sensorless method), which is the approach addressed in this work.

2.1.2 PMSM model in the $\alpha\beta$ axes

The $\alpha\beta$ frame reference is also called the stationary frame, because it relates to the voltages and currents as seen from the stator of the motor. In this frame, the PMSM model is [21]

$$\begin{aligned} \mathbf{v} &= \mathbf{R}\,\mathbf{i} + \mathbf{L}\,\frac{d\mathbf{i}}{dt} + \omega\,\frac{d\lambda_r}{d\theta} \\ J\,\frac{d\omega}{dt} &= \tau_m - B\,\omega - \tau_l \\ \frac{d\theta}{dt} &= \omega \end{aligned} \tag{2.1}$$

where $\mathbf{v} = [v_\alpha, v_\beta]^T$ is the vector of stator phase voltages, $\mathbf{i} = [i_\alpha, i_\beta]^T$ is the vector of stator phase currents, and $\lambda_\mathbf{r} = [\lambda_{r\alpha}, \lambda_{r\beta}]^T$ is the vector of stator phase fluxes due to

the rotor field. The resistance and inductance matrices can be written as

$$\mathbf{R} = \begin{bmatrix} R_s - 2\omega L_1 \sin 2\theta & 2\omega L_1 \cos 2\theta \\ 2\omega L_1 \cos 2\theta & R_s + 2\omega L_1 \sin 2\theta \end{bmatrix}$$

$$\mathbf{L} = \begin{bmatrix} L_0 + L_1 \cos 2\theta & L_1 \sin 2\theta \\ L_1 \sin 2\theta & L_0 - L_1 \cos 2\theta \end{bmatrix}$$

The load torque is τ_l and the torque produced by the motor is given by

$$\tau_m = P \cdot \left(\frac{1}{2} \mathbf{i}^T \frac{d\mathbf{L}}{d\theta} \mathbf{i} + \mathbf{i}^T \frac{d\lambda_r}{d\theta} \right)$$

where P is the number of pole pairs. The mechanical parameters of the motor J (moment of inertia) and B (friction constant) are normalized with P.

2.1.3 PMSM model in the dq axes

The dq frame reference is attached to the rotor, i.e., it rotates with it such that the angle between the α and the d axis equals to θ, the rotation angle of the rotor measured in electrical degrees. In this frame, the PMSM model is [21]

$$\begin{aligned} L_d \frac{di_d}{dt} &= \omega\ L_q\ i_q - R\ i_d + v_d \\ L_q \frac{di_q}{dt} &= -\omega\ L_d\ i_d - R\ i_q - \omega\ \Phi + v_q \\ J \frac{d\omega}{dt} &= (\Phi + \Delta L\ i_d)\ i_q - B\ \omega - \tau_l \\ \frac{d\theta}{dt} &= \omega \end{aligned} \tag{2.2}$$

where v_d and v_q are the dq voltages, i_d and i_q are the dq currents, L_d and L_q are the total dq axis inductances, $\Delta L = L_d - L_q$, and Φ is the flux due to permanent magnets.

2.2 Position estimation algorithm

As outlined in the introduction, this work builds on Petrović's PhD dissertation [12]. We refer the interested reader to that publication for a detailed description of the algorithm. In this section, we will outline the most interesting steps in its derivation as well as the final results.

This algorithm can be classified as an excitation-based one, as opposed to the back-EMF type which is also found in the literature. The idea behind this approach is to excite the motor with high-frequency currents and extract position information from the response of the motor to this injected signal, since the inductance is dependent on position due to magnetic saliency. In Petrović's algorithm, the basic idea is to take advantage of the current and voltage waveforms in one PWM period, which already contain high frequency components, to estimate the rotor position. Therefore, there is no injection of an auxiliary high frequency signal, but a utilization of the already existent PWM signals.

One of the main highlights of this algorithm is that it has a good performance in a wide range of motor speeds and load torques, even at zero or low speeds.

First, we will describe the modified PWM pattern used by this algorithm, which guarantees the non-singularity and well-conditioning of the problem. Following, we will describe the discretization of the PMSM model by analyzing the behavior of the system during one PWM period. Then, we will construct a least-squares problem that needs to be solved in order to determine the system parameters. The mechanical states observer, described at the end, generates the desired estimated rotor position, speed and acceleration.

2.2.1 PWM pattern

The actuator that applies the control signal to the motor is a voltage-sourced inverter, which consists of three pairs of electronic switches. These switches can connect each of the three motor phases to a DC voltage (VDC) or ground (GND), therefore being able to apply one out of eight possible combinations to the motor. In a Pulse Width Modulation (PWM) strategy based on the concept of voltage space vectors, each one of these combinations correspond to a vector in the $\alpha\beta$ plane: six vectors form an hexagon and the other two correspond to the origin (Fig. 2.1). The desired three-phase sinusoidal output corresponds to a circular path [22].

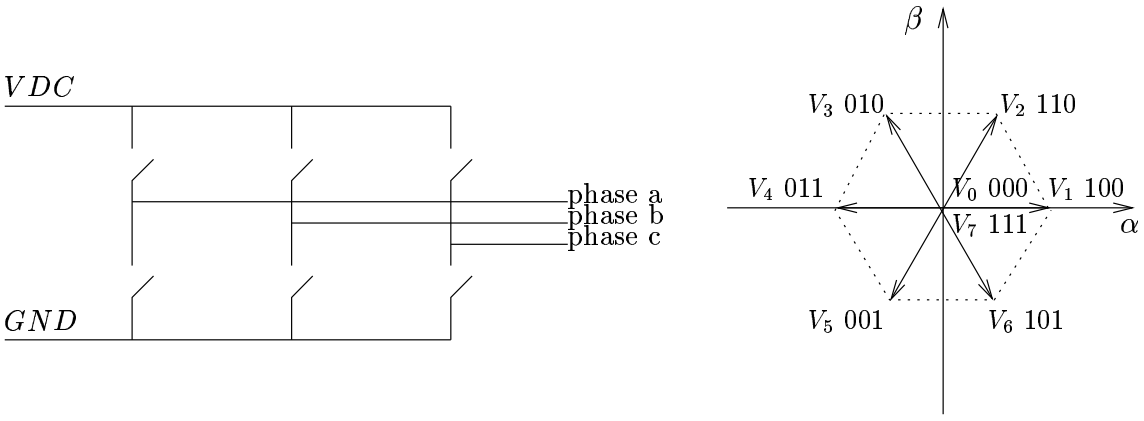


Figure 2.1: Inverter schematics and voltage vectors that can be generated (1 means that the corresponding phase a, b or c is connected to VDC, while 0 means that it is connected to GND).

The purpose of the PWM algorithm is to generate a sequence of switch combinations which, averaged in time, produce the desired voltage vector. The basic unit of time is called a PWM period and its value is constant. Each PWM period is subdivided in N subintervals during which a switch combination is held constant in

the inverter. The duration of these subintervals (duty ratios), as well as the switch combination in each one of them, are adjusted so that the average over the PWM period is the desired voltage output. High frequency components are usually filtered by the electrical dynamics of the load.

One of the frequently used patterns for producing the desired voltage vector is known as symmetric or centered PWM. This pattern uses three inverter states per PWM period: the two inverter vectors adjacent to the desired output (called lead and lag vectors) and a zero vector. These three vectors are arranged in the following sequence (Fig. 2.2): first, the zero vector is set for half of its corresponding time, then, the lag vector for half of its time, third, the lead vector for its complete time, then, the lag vector again for half of the time, and finally, the zero vector for half of the time. In this way, there are only four switchings per PWM period, since the zero vector is constant from one period to the next.

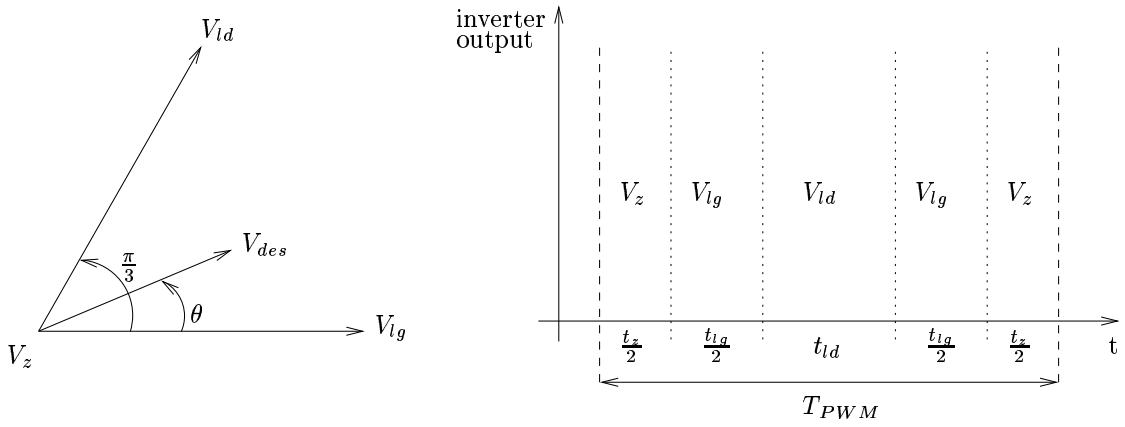


Figure 2.2: Symmetric PWM: the desired vector is obtained by combining the zero, lag and lead vectors.

In Petrović's algorithm the PWM pattern has 6 subintervals ($N = 6$) and the switching combination is always the same: each one of the six non-zero vectors

($V_1 ... V_6$) are generated sequentially (Fig. 2.3). The duty ratio of the subintervals are computed as

$$\begin{aligned} \zeta_1 &= \frac{1}{6} + \frac{5\sqrt{3}}{12}\frac{v_\alpha}{v_{max}} - \frac{7}{12}\frac{v_\beta}{v_{max}} \\ \zeta_2 &= \frac{1}{6} - \frac{\sqrt{3}}{12}\frac{v_\alpha}{v_{max}} + \frac{11}{12}\frac{v_\beta}{v_{max}} \\ \zeta_{3,4,5,6} &= \frac{1}{6} - \frac{\sqrt{3}}{12}\frac{v_\alpha}{v_{max}} - \frac{1}{12}\frac{v_\beta}{v_{max}} \end{aligned} \tag{2.3}$$

where the duty ratios are defined as $\zeta_i = \frac{t_i}{T_{PWM}}$, ζ_1 is the duty ratio of the subinterval corresponding to the lag vector, ζ_2 corresponds to the lead vector, and $\zeta_{3,4,5,6}$ corresponds to the other vectors. In fact, this pattern is very similar to the symmetric PWM, the only difference is that the zero vector is substituted by the sum of all six non-zero vectors, with a duty ratio of $\frac{1}{6}$ each. This pattern guarantees that the duty ratios are bounded from below.

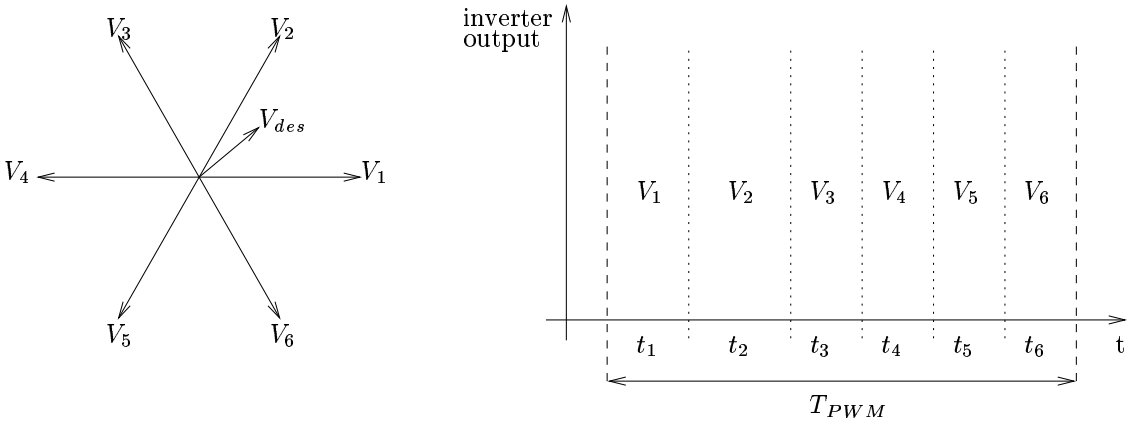


Figure 2.3: Modified PWM: the desired vector is obtained by combining all non-zero vectors (in this case, the desired vector is in sector 1).

2.2.2 Model discretization

We recall the PMSM model in the $\alpha\beta$ axis, which can be written as

$$\mathbf{v} = \mathbf{R}\mathbf{i} + \mathbf{L}\frac{d\mathbf{i}}{dt} + \omega\frac{d\lambda_r}{d\theta} \tag{2.4}$$

As described in section 2.2.1, a PWM period consists of N subintervals. In each one of them, the voltage vector $\mathbf{v}$ supplied to the motor is constant. The currents are therefore exponential, but due to the time constants of the electrical subsystem, they can be regarded as almost linear during the relatively short time duration of a subinterval. Assuming that the current changes are linear, it is only necessary to sample them at the subinterval boundaries to have a complete description of its behavior. This idea is illustrated in Fig. 2.4, where we also introduce the notation that will be used in the next equations.

Under the assumptions of linear currents and constant mechanical variables during each subinterval, we can write

$$\mathbf{v}_n = \mathbf{R}\frac{\mathbf{i}_{n+1} + \mathbf{i}_n}{2} + \mathbf{L}\frac{\mathbf{i}_{n+1} - \mathbf{i}_n}{t_{n+1} - t_n} + \omega\frac{d\lambda_r}{d\theta} \tag{2.5}$$

for $n = 1, 2, ...N$. The parameters that bare information on speed and position are contained in matrices $\mathbf{R}$ and $\mathbf{L}$, while the vectors $\mathbf{v}_n$ are known and the vectors $\mathbf{i}_n$ can be measured. The back-EMF term is a nuisance parameter which can be eliminated by subtraction of equations from two subsequent subintervals, assuming again that mechanical parameters do not change from one subinterval to the next. The resulting equations are

$$\mathbf{v}_{n+1} - \mathbf{v}_n = \mathbf{L}\left(\frac{\mathbf{i}_{n+2} - \mathbf{i}_{n+1}}{t_{n+2} - t_{n+1}} - \frac{\mathbf{i}_{n+1} - \mathbf{i}_n}{t_{n+1} - t_n}\right) + \mathbf{R}\left(\frac{\mathbf{i}_{n+2} - \mathbf{i}_n}{2}\right) \tag{2.6}$$

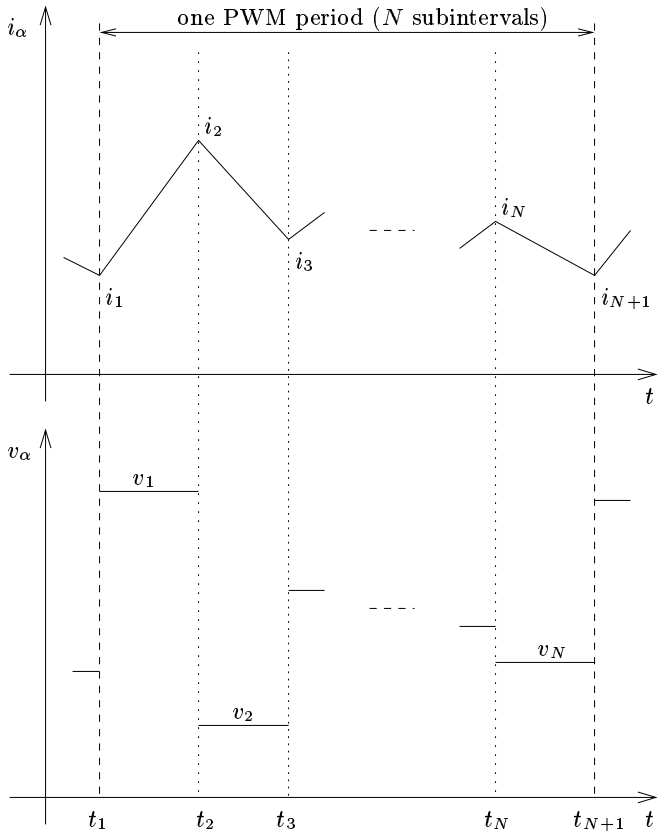


Figure 2.4: Current and voltage waveforms in one PWM period.

for $n = 1, 2, ...N - 1$.

An extensive analysis in the original work concludes that the effect of the parameters contained in the matrix $\mathbf{R}$ (i.e., R_s and ω) in the motor behavior at the PWM frequency is negligible. This result is not surprising because it is strongly related to the assumption that the currents are linear during a subinterval. As a consequence, we can neglect the resistance term in (2.6) and reduce the problem to the simpler form

$$\mathbf{w}_n = \mathbf{L}\mathbf{x}_n \tag{2.7}$$

where

$$\begin{aligned} \mathbf{w}_n &= \mathbf{v}_{n+1} - \mathbf{v}_n \\ \mathbf{x}_n &= \left(\frac{\mathbf{i}_{n+2} - \mathbf{i}_{n+1}}{t_{n+2} - t_{n+1}} - \frac{\mathbf{i}_{n+1} - \mathbf{i}_n}{t_{n+1} - t_n} \right) \end{aligned}$$

2.2.3 Parameter estimation

Before we construct the least-squares problem, we will rewrite (2.7) in a more convenient way. Observing this equation, we can see that $\mathbf{w}_n$ contains voltage values which are known *a priori*, while $\mathbf{x}_n$ contains current values which have to be measured in real time. As it was shown in the original work, it is more convenient to have the unknown matrix, which contains the system parameters, multiplied by the already known voltage values. Therefore, the equation is rewritten as

$$\mathbf{x}_n = \mathbf{L}^{-1}\mathbf{w}_n \tag{2.8}$$

where

$$\mathbf{L}^{-1} = \frac{1}{L_0^2 - L_1^2} \begin{bmatrix} L_0 - L_1\cos(2\theta) & -L_1\sin(2\theta) \\ -L_1\sin(2\theta) & L_0 + L_1\cos(2\theta) \end{bmatrix}$$

At this point, we have to select a convenient parametrization. Because parameters L_0, L_1 and θ are unknown, one possibility is to choose directly these quantities. However, it is clear that the resulting least-squares problem will be non linear. In addition, it was shown in the original work that the problem will not be well conditioned numerically. For those reasons, the selected parameter vector is

$$\mathbf{q} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \end{bmatrix} = \frac{1}{L_0^2 - L_1^2} \begin{bmatrix} L_0 \\ -L_1\cos(2\theta) \\ -L_1\sin(2\theta) \end{bmatrix} \tag{2.9}$$

This parametrization has the advantage of generating a linear and well conditioned least-squares problem.

The next step is to rewrite (2.8) as a function of the parameter vector $\mathbf{q}$. We recall here that the vectors $\mathbf{x}_n$ and $\mathbf{w}_n$ have both an α and a β component, so we can write

$$\mathbf{x}_n = \begin{bmatrix} x_{n\alpha} \\ x_{n\beta} \end{bmatrix} = \begin{bmatrix} q_0 + q_1 & q_2 \\ q_2 & q_0 - q_1 \end{bmatrix} \begin{bmatrix} w_{n\alpha} \\ w_{n\beta} \end{bmatrix} = \\ = \begin{bmatrix} w_{n\alpha} & w_{n\alpha} & w_{n\beta} \\ w_{n\beta} & -w_{n\beta} & w_{n\alpha} \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \end{bmatrix} = \mathbf{W}_n \mathbf{q} \tag{2.10}$$

for $n = 1, 2, ...N - 1$.

We can stack the $N-1$ equations to finally form our least-squares problem: given $\mathbf{x}$ and $\mathbf{W}$, find a solution to the equation

$$\mathbf{x} = \mathbf{W}\mathbf{q} \tag{2.11}$$

Since a convenient choice of N and $\mathbf{W}$ is made by elaborating the PWM pattern (see 2.2.1), we already know that the system (2.11) is over-determined. The solution that satisfies

$$\hat{\mathbf{q}} = \arg\min \|\mathbf{W}\mathbf{q} - \mathbf{x}\|^2 \tag{2.12}$$

is

$$\hat{\mathbf{q}} = \left(\mathbf{W}^T\mathbf{W}\right)^{-1} \mathbf{W}^T\mathbf{x} = \mathbf{W}_{pi}\ \mathbf{x} \tag{2.13}$$

In section 2.2.1 we described the PWM pattern used in this algorithm and showed that the voltage values used in each subinterval are always the same. This important

property permits an off-line computation of the matrix $\mathbf{W}_{pi}$ and therefore the estimation algorithm will be reduced to a multiplication of a constant matrix by a vector which depends on measured current values and the durations of the subintervals. It should also be noted that, since the duty ratios are bounded from below, there are no numerical problems when we divide current differences by subinterval durations and construct the vector $\mathbf{x}$.

2.2.4 Mechanical states observer

Once the parameter vector $\mathbf{q}$ has been estimated, the rotor position could be computed using inverse trigonometric functions. However, this approach has two drawbacks: first, the noise which is present in the parameter estimates is not filtered, and second, the inverse trigonometric function computation is time consuming.

To overcome these drawbacks, an observer of the mechanical variables is introduced. This approach filters the noise in the estimates and requires a few computations. Additionally, it provides an estimate of all mechanical states (not only position, but also speed and acceleration). The dynamics of this observer are much faster than the dynamics of the mechanical loop controller, so it can be neglected while designing the control loop.

The observer has the following dynamics

$$\dot{\hat{\alpha}} = \gamma_3 \epsilon \tag{2.14}$$

$$\dot{\hat{\omega}} = \hat{\alpha} + \gamma_2 \epsilon \tag{2.15}$$

$$\dot{\hat{\theta}} = \hat{\omega} + \gamma_1 \epsilon \tag{2.16}$$

where $\hat{\theta}$, $\hat{\omega}$ and $\hat{\alpha}$ are the estimates of the rotor position, speed and acceleration,

respectively, γ_i are design parameters, and ϵ is a non-linear observation error that is used to drive variable estimates to their true value. The error is derived from inductance parameter estimates as

$$\epsilon = \mathbf{q}_2 \cos(2\hat{\theta}) - \mathbf{q}_1 \sin(2\hat{\theta}) \approx L_1 \sin(2\tilde{\theta}) \tag{2.17}$$

where $\tilde{\theta} = \theta - \hat{\theta}$.

The incremental error system becomes non-linear with stable equilibria at $\tilde{\theta} = n\pi$. The linearized system around $\tilde{\theta} = 0$ is an autonomous linear system whose poles can be set arbitrarily by a proper choice of parameters γ_i. The pole placement is governed by a trade-off between a fast transient response and noise filtering.

2.3 Control algorithm

This work is focused in the position estimation problem. However, in order to be able to operate the motor and run simulations and experiments, we need to close the loop and implement a controller.

The control algorithm is developed in the dq axis, and consists of nested-loop PI controllers that use position and speed estimates as feedback, as seen in Fig. 2.5. There is one outer speed loop whose aim is to track the reference speed, and one inner current loop which keeps $i_d = 0$ and tracks the desired i_q provided by the speed controller.

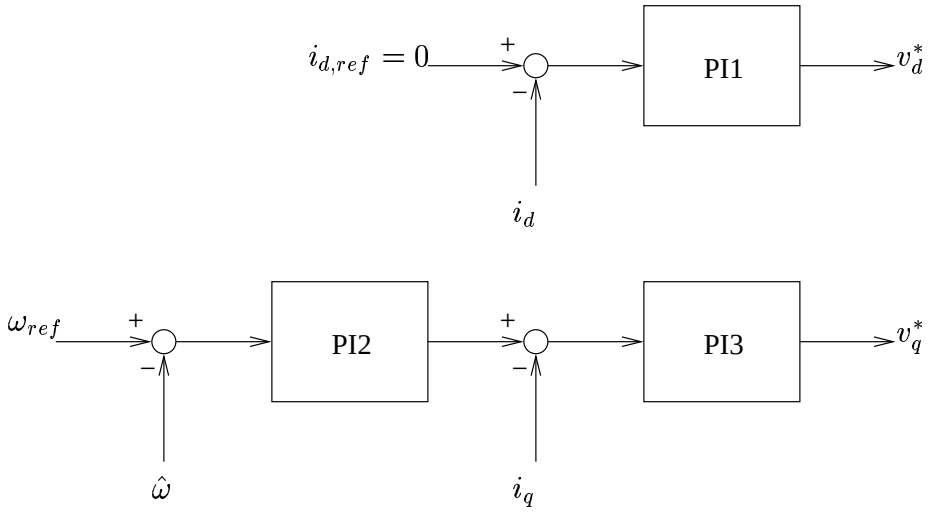


Figure 2.5: Block diagram of the controller.

2.4 Summary

In the previous sections, we described the algorithm that will be implemented in a low-cost hardware platform.

This algorithm requires the input of measured currents i_α and i_β. The only accessible points of the motor are the three phase lines a, b and c, so we can measure currents i_a, i_b and i_c. In fact, only two of these measurements are needed because there is no neutral connection in the motor, so we can derive the $\alpha\beta$ components from only two phase measurements, e.g., i_a and i_b.

Based on these current measurements, the position estimation algorithm computes $\hat{\theta}$, which is used in the $\alpha\beta - dq$ transformation, and $\hat{\omega}$, which is fed to the input of the controller.

At the output of the controller we have the desired $\alpha\beta$ voltages. The PWM block generates the appropriate switching signals which are fed to the inverter, who in turn will produce the desired voltages in the motor phase lines a, b and c.

The complete system is illustrated in Fig. 2.6.

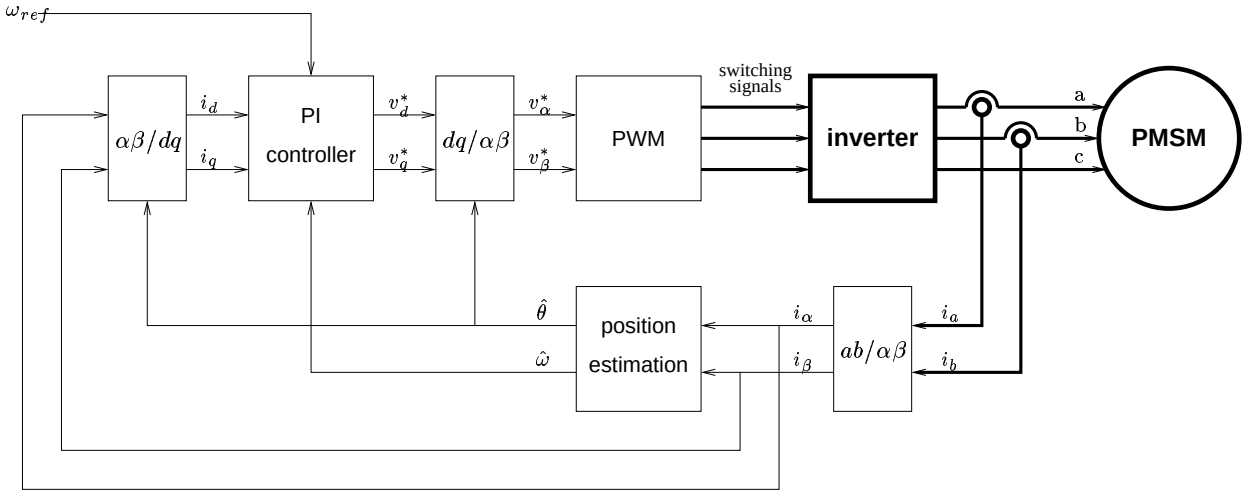


Figure 2.6: Block diagram of the complete system, hardware parts are drawn in bold.

Chapter 3

Hardware design

In this chapter we describe the hardware that was designed and built in order to implement the algorithm described in the previous chapter.

First, we outline the basic ideas behind the conception of this design, in particular the partition of the implementation into two main stages: a computational stage, which is implemented in a general purpose fixed-point Digital Signal Processor (DSP), and a state-machine stage, which is implemented in a Field Programmable Gate Array (FPGA). Second, we describe the details of the signal acquisition, comprising the analog part of the board. Following, we address the selection and electrical considerations of the two main chips in the board: the FPGA and the DSP. Next, we describe the issues involved in the Printed Circuit Board (PCB) design and the construction of the prototype. Finally, we summarize the main concepts introduced in this chapter.

3.1 General description

The sensorless control of Permanent Magnet Synchronous Motors requires additional computations for the estimation of the shaft position. Also, depending on the algorithm used, it can require specific characteristics on the PWM generation and current measurements. For these reasons, the implementation of sensorless control in general leads to high-cost solutions, jeopardizing the industrial interest in this field.

Our challenge was to find a low-cost implementation of a sensorless control algorithm by performing a careful design which combines a general purpose fixed-point DSP and an FPGA. The introduction of programmable logic in the design is very convenient because it is a powerful and very flexible tool that can free the processor from several time-consuming tasks.

The algorithm described in the previous chapter is already optimized so that the numerical computations are kept to a minimum. However, the PWM generation is completely non-standard and the sampling of the currents must be synchronized with the switchings. If these tasks were to be carried out by the DSP, a complex and time-consuming interrupt scheme would have to be programmed. Our approach frees the processor from anything but the numerical computations, using only one interrupt per PWM period for synchronization.

The PWM generation and the signal acquisition are performed by the FPGA. The DSP is interrupted at the end of a PWM period and has access to the current measurements of each one of the subintervals, already stored in the FPGA, so that it can compute the estimated position and control outputs. Once the computations are finished, the desired time for each subinterval is transferred to the FPGA, which

generates the switching signals for the inverter on the next PWM period.

Therefore, the architecture of the hardware implementation can be described by three basic blocks (see Fig. 3.1): Signal Acquisition (consisting of analog signal conditioning and analog-to-digital conversion), FPGA and DSP. These blocks are described in the following subsections.

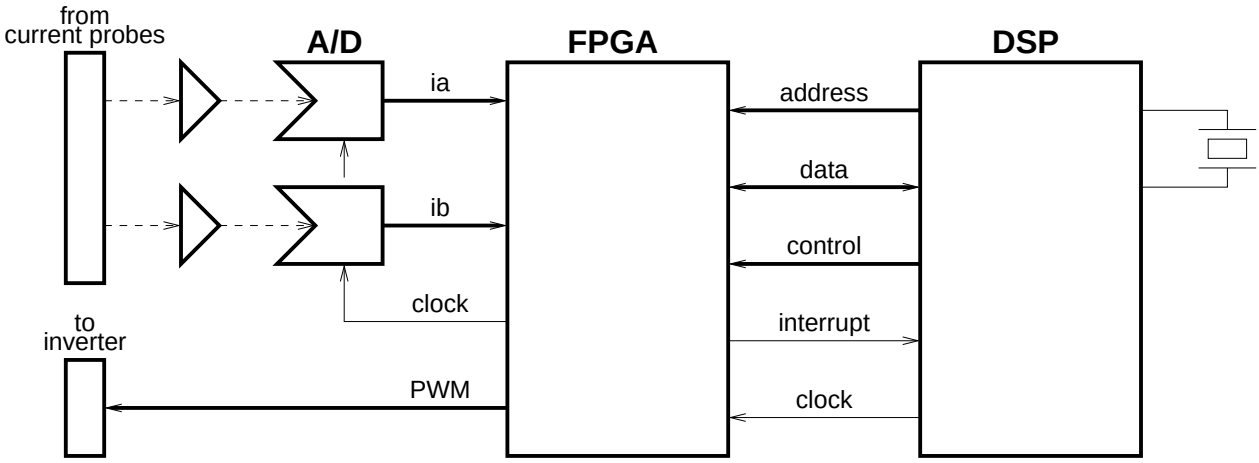


Figure 3.1: Board architecture (simplified). Dashed lines are analog signals, solid lines are digital signals and bold solid lines are digital buses.

It is important to notice that the clock signal is generated in the DSP, by using a 10MHz crystal. The internal clock of the DSP is 20MHz, and this signal is transferred to the FPGA. Inside the FPGA, this clock is divided by 2 to generate the internal clock of the FPGA and to control the analog-to-digital converters. The synchronization of the system is achieved by a periodic interruption generated by the FPGA at the end of every PWM period.

3.2 Signal acquisition

The main components in the signal acquisition stage are the analog-to-digital converters (ADC). We wanted to have good precision in the current measurements, because we deal with current differences of less than $100mA$, therefore we looked for a resolution of about $1mA$. Since the dynamic range of the phase currents is $\pm 12A$, we concluded that it was necessary to have at least 14 bits in order to avoid introducing a significant quantization error.

We have selected the AD9240, from Analog Devices. This is a 14-bit, 10 MSPS, monolithic ADC with an on-chip voltage reference. It uses a single clock signal to control its pipelined conversion architecture. The digital output is presented in straight binary format with the 14 bits in parallel [23].

Since the overall cost of the system is an important issue addressed in this work, we have to point out that a less expensive ADC can be used in a final product if we use less bits and a slower conversion rate. However, we consider that this ADC is a good choice for a prototype because it permits to study the effect of different resolutions and to explore the possibility of sampling the currents at different time instants in the PWM period, even oversampling and averaging to reduce noise.

The current probes provide a current output i_{cp} proportional to the phase current i, such that $i_{cp} = \frac{2}{1000}i$. A 200Ω resistor converts this current to a voltage v_{cp}, which is the input of our board (see Appendix E for details of the current probe design). Since the phase current has a limit of $\pm 12A$, the voltage v_{cp} is limited by $\pm 12A \frac{2}{1000} 200\Omega = \pm 4.8V$.

The AD9240 works with a single $+5V$ power supply, therefore we need a signal

conditioning stage to adapt the input v_{cp} to a suitable range for the ADC. We set the voltage reference $VREF$ to $+2.5V$, the middle-point of the power supply, to have a symmetric span. The conversion range is therefore $\pm 2.5V$. As a consequence, we need to reduce the input v_{cp} to fit into this range, introducing a gain of $1/2$ in the signal conditioning stage.

The AD9240 has two analog inputs, $VINA$ and $VINB$, which operate as a differential input, i.e., the analog voltage to be converted is $VINA - VINB$. This implies that one of the analog inputs must be kept constant, equal to $VREF$, while the other is fed by the desired signal plus a DC component equal to $VREF$. In this way both inputs are in the range of the power supply (0 to $+5V$) and the differential value cancels the DC component.

A detailed schematic of the signal conditioning stage and the ADC configuration is shown in Fig. 3.2. The design follows the guidelines found in the datasheets of the AD9240.

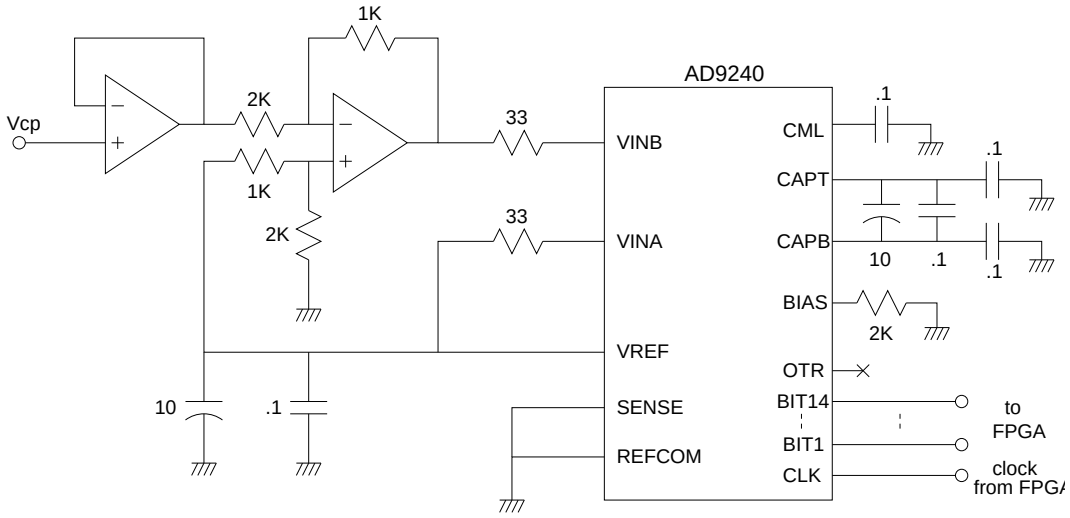


Figure 3.2: Schematics of the ADC configuration.

As stated above, the input v_{cp} has a dynamic range of $\pm 4.8V$. The input $VINB$ is equal to $-\frac{1}{2}v_{cp} + VREF$ and therefore it spans from $+0.1V$ to $+4.9V$, this is inside the power supply range. The differential input is $VINA - VINB = \frac{1}{2}v_{cp}$ as desired.

We use the operational amplifiers AD8042, also from Analog Devices, because of their high speed, fast settling time and low distortion [24]. Due to the dynamic range of the input, a symmetric $\pm 5V$ power supply is needed, and since the AD8042 is a rail-to-rail amplifier, it is guaranteed that no saturation of the signal will happen.

The input from the current probes is available at one 8-pin header connector (J1 in the schematic). For debugging purposes, we included another connector in parallel (J2) so that we could read current values with another instrument.

3.3 The FPGA

The selection of the FPGA chip to be used depends mainly on the size of the digital circuit that is needed, i.e., the tasks that the FPGA will perform. The basic tasks that we considered important to be implemented in the FPGA are the PWM generation and the synchronization of the data acquisition. The larger the FPGA, more computational work can be done, and therefore more processor cycles can be freed from the DSP.

We selected EPF6016, from Altera. This FPGA belongs to the FLEX 6000 family, a low-cost alternative to high-volume gate arrays designs. The EPF6016 contains 16,000 gates arranged in 1,320 Logical Elements. We selected the 144-pin TQFP package, which is the more convenient for manual soldering. This package offers a total of 117 user I/O pins [25].

The connections between the FPGA and the other components on the board are straightforward:

- Two 14-bit inputs and one clock output are used to interface with the ADCs.
- A 6-bit output carries the PWM switching signals for the inverter (one signal for each IGBT gate), thru a DB15 female connector (J3 in the schematic).
- A 16-bit data bus, a 4-bit address bus, control signals, clock and interrupt request are used to interface with the DSP.
- A reset signal, shared with the DSP, generated by a traditional RC net plus a pushbutton (J12 in the schematic).
- A 16-bit output is used to inform the estimated position to a host, thru a 2x10 header connector (J4 in the schematic).
- Four bits are used as test points for debugging purposes (J7 thru J10 in the schematic).
- An 8-bit interface with the host, thru a ByteBlaster connector (J6 in the schematic) is used to configure the FPGA.

Several $.1\mu F$ capacitors are arranged close to the VCC and GND pins to supply high frequency current peaks to the chip and reduce the interference with the rest of the circuit.

The configuration of the FPGA is made via the ByteBlaster cable from a PC host [26]. However in a stand-alone application we would need a serial EPROM in the circuit to perform the configuration after power-up.

3.4 The DSP

Our design focuses on a low-cost hardware platform, so we concentrated on a general purpose, 16-bit fixed-point DSP.

The chip selected is the ADSP-2181, from Analog Devices. It is based on the ADSP-2100 family architecture, plus several on-chip peripherals like memory, serial ports, timer, programmable I/O, DMA and power-down control [27]. Most of these peripherals are not needed for this work; however they can be used for additional tasks in a final application, like human-machine interface or integration to an automation system.

The ADSP-2181 can work with an internal clock of up to 40MHz, but we use half of this capability. To set an internal clock of 20MHz, we connect a 10MHz fundamental frequency crystal (X3 in the schematic) between pins XTAL and CLKIN.

In order to load the program from the host and to debug the system, we use an In-Circuit Emulator (EZ-ICE), which is connected to the DSP thru a special set of pins called ICE-Port. These pins are wired to a 14-pin header connector (J5 in the schematic) [28].

The booting method selected is IDMA Booting, since it is the most appropriate for the EZ-ICE operation, therefore the pin MMAP is connected to GND and BMODE to VCC. Jumpers were introduced so that we could test other booting methods.

Two inputs were provided so that we could have real-time interaction, like steps in the speed reference. We used the flag I/O pins PF1 and PF2 because they are very simple to interface. One of them was connected to a switch and the other to a pushbutton (J16 and J15 in the schematic respectively).

Since we have the EZ-ICE for our prototype, no other input or output possibilities were used. In an industrial implementation we would use an EPROM to boot the processor, and a serial port to receive the inputs (speed reference at least) and maybe additional communication.

3.5 Construction of the prototype

The packaging options available for the main components of the board (i.e., ADCs, FPGA and DSP) are surface-mount type; in the case of the FPGA, the pin-to-pin distance is as low as .020in (20 mils), while the DSP and ADCs have 50 mils. One possibility was to use sockets and wire-wrap all the circuit. However, we decided to use a more robust solution and design a Printed Circuit Board (PCB).

The high density of pins and connections forced us to select a 4-layer board, with the inner layers reserved as power and ground layers, while the top and bottom are used for signal connections. Signal traces had a width of 8 mils if connected to the surface-mount components, and 10 mils otherwise, with the exception of the -5V power applied to the operational amplifiers which has wider traces.

The PCB, with a size of 6 by 3.5 inches and around 350 holes, was manufactured by an Internet-based company, specialized in prototype boards. The components were soldered manually with the aid of magnifiers. The critical issue was to avoid thermal and mechanical stress in the components, while achieving a perfect positioning.

The system is completed with a $\pm 5V$ power supply and the following cables:

- ByteBlaster cable connected to the parallel port of the PC, to download the FPGA configuration.

- EZ-ICE cable connected to the serial port of the PC, to download the DSP program and debug the system.
- a twisted pair cable to connect to the inverter board, to receive the current probes signals.
- a flat cable to connect the PWM signals to the inverter board.

Additionally, for debugging purposes, two cables are used to send current probes signals and estimated position to the PC host, via a Dspace acquisition board.

In Fig. 3.3 we show a picture of the prototype board.

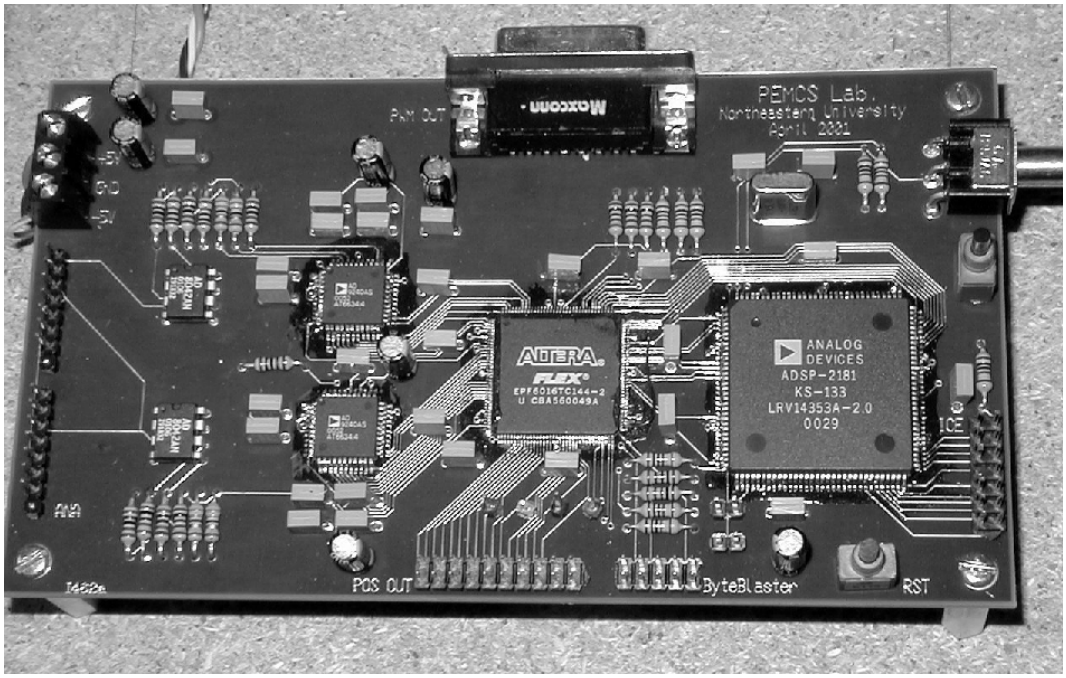


Figure 3.3: Prototype board (size=$6 \times 3.5''$).

3.6 Summary

We have described in this chapter a general architecture and the electrical details of the board designed and built to implement the sensorless control of a PMSM. The complete schematic of the circuit and a plot of the PCB layers are shown in Appendix A.

We believe that this architecture is suitable for a more cost-efficient implementation on an industrial scale, if we integrate the FPGA design and the core of the DSP into one single chip. This is not difficult to achieve, because the FPGA design is described completely in VHDL, and it can be converted into an ASIC by using standard software. The DSP requirements can be reduced by pruning the peripherals that are not used.

Chapter 4

Word-length effects

In this chapter we present the issues involved in the implementation of the sensorless control algorithm in a fixed-point DSP. Word-length effects are studied and proper scaling factors are introduced to avoid overflow while achieving a reasonable numerical precision. The results are simulated and compared with a floating-point version of the same algorithm.

4.1 Overview of the algorithm and implementation issues

The algorithm presented in Chapter 2 consists of the following sequence of operations:

1. read phase currents from the input
2. convert phase currents to the $\alpha\beta$ frame
3. construct vector $\mathbf{x}$
4. compute vector $\mathbf{q}$

5. update mechanical states observer
6. average currents over the PWM period
7. convert average currents to the dq frame
8. compute controller output (voltages)
9. convert voltages back to the $\alpha\beta$ frame
10. find sector and rotate desired vector to sector 1
11. compute subinterval times (counter values)
12. write counter values to the output

The algorithm is discrete in nature, since the basic time unit of the control output is the PWM period. It makes no sense to have a controller working at a shorter period because inputs would not be available, and the output cannot actuate faster. The natural choice for the controller is to select a sampling period equal to the PWM period or an integer multiple of it. In this work, we selected the controller (and observer) period equal to the PWM period.

For an efficient implementation in a fixed-point processor, all values are represented in fractional format. This format is convenient because all multiplications yield to valid results. The main drawback is that the less significant bits are lost and precision can degrade on each multiplication if the operands are not close to 1.

The fractional representation we use is called "1.15 format" because it uses 1 bit for the integer part (sign) and 15 for the fractional part [29]. This format makes use of a two's-complement representation, being able to represent values in the range $(-1, 1 - 2^{-15})$ with a precision of 2^{-15}. When two values are multiplied, the result can be expressed in 1.31 format, but the 16 less-significant bits have to be discarded to express it again in 1.15 format.

So far, we can identify the main issues in the implementation of the algorithm as:

- try to keep all values in the range $(-1, 1 - 2^{-15})$, but not too close to zero; the absolute value of the constant coefficients should be in the range (0.5,1) (whenever possible).

- make sure that the final result of a sequence of additions/subtractions is still in the valid range; sometimes it would be necessary to scale down the operands before the operation.

4.2 Scaling

The scaling of the inputs, outputs, state variables and coefficients has to be performed carefully so that the numerical precision is preserved [30]. Due to the limited dynamic range of the fixed-point representation, there is always a trade-off between avoiding overflow and minimizing quantization effects.

In the following sections, we describe the transformation of the algorithm into a format suitable for the ADSP2100 family of 16-bit fixed-point processors. The complete assembler code is included in Appendix C.

We have adopted the notation $\bar{x}$ to indicate the fractional (scaled) value of x, i.e., $\bar{x} = \frac{x}{x_{max}}$ where x_{max} is the maximum absolute value that x can take without producing overflow.

4.2.1 Scaling the inputs

The phase currents are scaled several times in their way to the ADCs. The sensors introduce a scaling factor of $\frac{2}{1000}$. Next, a resistor converts the current into a voltage with a factor of 200Ω. Finally, a factor of $\frac{1}{2}$ is introduced in the signal conditioning stage. As a result, the ADC input range of $\pm 2.5V$ is equivalent to a phase current range of $\pm 12.5A$. Since the ADCs output has 14 bits, there are two spare bits that can be used to find a convenient scaling. An examination of the operations carried out with phase current values will provide us information on how to use these spare bits.

The currents have to be converted to the $\alpha\beta$ frame, according to the transformation

$$\begin{aligned} i_\alpha &= \sqrt{\frac{3}{2}}\, i_a \\ i_\beta &= \frac{1}{\sqrt{2}}\, i_a + \sqrt{2}\, i_b \end{aligned} \tag{4.1}$$

In a worst-case scenario, the maximum value of i_α is $\sqrt{\frac{3}{2}} \approx 1.22$, and the maximum value of i_β is $\frac{1}{\sqrt{2}} + \sqrt{2} \approx 2.12$ times the maximum value of the phase currents. The latter result leads to the conclusion that the $\alpha\beta$ currents need 2 bits more than the phase currents. This means that, if the phase currents can take a maximum absolute value of $12.5A$, the $\alpha\beta$ currents have to be allowed a maximum absolute value of $4 \times 12.5A = 50A$, i.e., $i_{\alpha\beta,max} = 50A$.

Going back to the transformation (4.1), we observe that some coefficients are greater than 1. We need to divide the coefficients by 2 to make them fit into the

fractional range, so we write

$$\begin{aligned} i_\alpha &= \left[\frac{\sqrt{\frac{3}{2}}}{2}\right][2i_a] \\ i_\beta &= \left[\frac{1}{2\sqrt{2}}\right][2i_a] + \left[\frac{\sqrt{2}}{2}\right][2i_b] \end{aligned} \tag{4.2}$$

To convert to fractional notation, we divide by $i_{\alpha\beta,max}$

$$\begin{aligned} \bar{i_\alpha} &= \left[\frac{\sqrt{\frac{3}{2}}}{2}\right]\left[2\frac{i_a}{i_{\alpha\beta,max}}\right] \\ \bar{i_\beta} &= \left[\frac{1}{2\sqrt{2}}\right]\left[2\frac{i_a}{i_{\alpha\beta,max}}\right] + \left[\frac{\sqrt{2}}{2}\right]\left[2\frac{i_b}{i_{\alpha\beta,max}}\right] \end{aligned} \tag{4.3}$$

Finally, by defining $i_{ab,max} = \frac{i_{\alpha\beta,max}}{2} = 25A$, we can write the transformation as

$$\begin{aligned} \bar{i_\alpha} &= \left[\frac{\sqrt{\frac{3}{2}}}{2}\right]\bar{i_a} \\ \bar{i_\beta} &= \left[\frac{1}{2\sqrt{2}}\right]\bar{i_a} + \left[\frac{\sqrt{2}}{2}\right]\bar{i_b} \end{aligned} \tag{4.4}$$

Now each factor is fractional and the sum is guaranteed not to overflow.

We can obtain $\bar{i_a}$ and $\bar{i_b}$ from the 14-bit ADCs outputs with a few operations. First, the MSB is inverted to convert to two's-complement notation (according to the datasheets). This value interpreted in 1.13 format is in the range $(-1, 1-2^{-13})$ and corresponds to currents in the range $\pm 12.5A$. We add one bit on the left, using sign extension. The result interpreted in 1.14 format has the correct scaling, so finally one zero is added on the right to obtain the 1.15 format representation. All these operations are performed automatically in the FPGA (see Chapter 5), so the DSP will read the value $\bar{i_a}$ in the correct format and scaling.

The multiplications involved in transformation (4.4) can be performed in the multiplier-accumulator unit (MAC). The multiplication of two 1.15 numbers has a

result in 1.31 format. No overflow will happen in the addition because of the selected scaling. The results should be extracted from the MAC by discarding the 16 less-significant bits.

4.2.2 Constructing vector x

We recall that

$$x_n = \frac{i_{n+2} - i_{n+1}}{t_{n+2} - t_{n+1}} - \frac{i_{n+1} - i_n}{t_{n+1} - t_n} \tag{4.5}$$

where the α and β subindices have been omitted for simplification.

We can distinguish three operations involved:

1. subtract consecutive currents
2. divide difference of currents by subinterval durations to find the slope values
3. subtract consecutive slope values

The subtraction of currents does not represent any overflow potential because the ripple in the currents is much less than the maximum absolute value. The potential problem here is the loose of too many significant bits, but there is nothing we can do since the number of bits of the ADCs is fixed. Therefore, these subtractions can be performed in the arithmetic-logic unit (ALU) with no additional considerations. We call $d\bar{i}_n$ the result and notice that the scaling factor is the same as the $\alpha\beta$ currents, i.e., $i_{\alpha\beta,max}$.

The next step is to divide the difference of currents by the subinterval durations. These time values are expressed in counter values as integers, therefore the time

expressed in seconds is

$$dt_n = T_{CK}\ \bar{dt}_n \tag{4.6}$$

where T_{CK} is the counter clock period (i.e., 100ns), and $\bar{dt}_n$ is the counter value in integer format.

In order to study the most convenient way of performing the division, first, we estimate the maximum absolute value of the result. Roughly speaking $\frac{di}{dt} \approx \frac{1}{L}V$. Since $V_{max} \approx 1.2V_{BUS} \approx 240V$ and $L_{min} \approx 3mHy$, then we conclude that $\left(\frac{di}{dt}\right)_{max} \approx 8 \times 10^4 A/s$.

On the other hand, the slope value, which we will call $didt_n$, can be computed as

$$didt_n = \frac{di_n}{dt_n} = \frac{i_{\alpha\beta,max}\ \bar{di}_n}{T_{CK}\ \bar{dt}_n} = 5 \times 10^8 A/s\ \frac{\bar{di}_n}{\bar{dt}_n} \tag{4.7}$$

and since we know the dynamic range of $didt_n$, we can conclude that the division $\frac{\bar{di}_n}{\bar{dt}_n}$ will have a dynamic range of $\pm\frac{8\times10^4}{5\times10^8} = \pm 1.6 \times 10^{-4}$.

Clearly, we cannot represent this result efficiently in 1.15 format. However, we have the advantage that the dividend can be expressed with 32 bits, so we can prescale the value $\bar{di}_n$ so that the result has a dynamic range close to $(-1, 1)$. For this reason, we compute $\bar{didt}_n = \frac{2^{12}\ \bar{di}_n}{\bar{dt}_n}$; now, the result will have a dynamic range of ± 0.66, which is very convenient for a representation in 1.15 format.

The scaling of the output can be found as follows:

$$\bar{didt}_n = \frac{2^{12}\ \bar{di}_n}{\bar{dt}_n} = \frac{2^{12}\ T_{CK}}{i_{\alpha\beta,max}}\frac{di_n}{dt_n} = 8.19 \times 10^{-6} s/A\ \frac{di_n}{dt_n} \tag{4.8}$$

therefore, $didt_{max} = \frac{1}{8.19\times10^{-6}s/A} = 1.22 \times 10^5 A/s$.

To conclude with the division, we have to specify how will we load the operands in the dividend and divisor registers. The latter is loaded as an integer value (i.e.,

in 16.0 format), while the former is loaded in a 32-bit register in 16.16 format. To convert the value $\bar{di}_n$ in 1.15 format to the value $2^{12}\ \bar{di}_n$ in 16.16 format, we have to shift the value to the left 13 bits (12 because of the scaling and one more because we have 16 instead of 15 bits in the fractional part). The 13 bits inserted on the right are zero, while the 3 bits inserted on the left to complete the 32-bit register are a sign extension.

The result in the division is stored in a 16-bit register, and given the format of the operands, it will be expressed in 1.15 format.

After the division, the elements of vector $\mathbf{x}$ are finally computed as the subtraction of subsequent slope values $didt_n$. We know that the dynamic range of the scaled slope values is ± 0.66 so there is a potential overflow in the subtraction operation, under the worst case. For this reason, we divide by two the operands before the subtraction, leading to

$$\bar{x_n} = \frac{1}{2}\ \bar{didt}_{n+1} - \frac{1}{2}\ \bar{didt}_n \tag{4.9}$$

which implies that the scaling factor of the elements of $\mathbf{x}$ is given by $x_{max} = 2\ didt_{max} = 2.44 \times 10^5 A/s$.

One last comment about vector $\mathbf{x}$: in this section we have omitted the subindices α and β, however we must specify the order in which the elements of $\mathbf{x}$ are stacked.

The vector is constructed as

$$\mathbf{x} = \begin{bmatrix} x_{\alpha,1} \\ x_{\beta,1} \\ x_{\alpha,2} \\ x_{\beta,2} \\ \vdots \\ x_{\alpha,5} \\ x_{\beta,5} \end{bmatrix} \tag{4.10}$$

where $x_{\alpha,n} = di_\alpha dt_{n+1} - di_\alpha dt_n$ and $x_{\beta,n} = di_\beta dt_{n+1} - di_\beta dt_n$.

4.2.3 Computing estimated parameters

In 2.2.3 we showed that the parameter estimation problem can be reduced to the matrix-vector multiplication

$$\hat{\mathbf{q}} = \mathbf{W}_{pi}\ \mathbf{x} \tag{4.11}$$

where the matrix $\mathbf{W}_{pi} = \left(\mathbf{W}^T\mathbf{W}\right)^{-1}\mathbf{W}^T$ can be computed *a priori* from the $\alpha\beta$ voltage values that are output in each subinterval, as shown in Table 4.1.

subinterval	1	2	3	4	5	6
v_α	$\frac{2V}{\sqrt{6}}$	$\frac{V}{\sqrt{6}}$	$-\frac{V}{\sqrt{6}}$	$-\frac{2V}{\sqrt{6}}$	$-\frac{V}{\sqrt{6}}$	$\frac{V}{\sqrt{6}}$
v_β	0	$\frac{V}{\sqrt{2}}$	$\frac{V}{\sqrt{2}}$	0	$-\frac{V}{\sqrt{2}}$	$-\frac{V}{\sqrt{2}}$

Table 4.1: Inverter $\alpha\beta$ voltages in each subinterval.

We recall that

$$\mathbf{W} = \begin{bmatrix} w_{\alpha,1} & w_{\alpha,1} & w_{\beta,1} \\ w_{\beta,1} & -w_{\beta,1} & w_{\alpha,1} \\ & \vdots & \\ w_{\alpha,5} & w_{\alpha,5} & w_{\beta,5} \\ w_{\beta,5} & -w_{\beta,5} & w_{\alpha,5} \end{bmatrix} \tag{4.12}$$

where $w_{\alpha,n} = v_{\alpha,n+1} - v_{\alpha,n}$ and $w_{\beta,n} = v_{\beta,n+1} - v_{\beta,n}$. Therefore,

$$\mathbf{W} = V \cdot \begin{bmatrix} -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} \\ -\frac{2}{\sqrt{6}} & -\frac{2}{\sqrt{6}} & 0 \\ 0 & 0 & -\frac{2}{\sqrt{6}} \\ -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} \\ \frac{2}{\sqrt{6}} & \frac{2}{\sqrt{6}} & 0 \\ 0 & 0 & \frac{2}{\sqrt{6}} \end{bmatrix} \tag{4.13}$$

and

$$\mathbf{W}_{pi} = \frac{1}{V} \cdot \begin{bmatrix} -\frac{\sqrt{6}}{32} & \frac{5\sqrt{2}}{32} & -\frac{3\sqrt{6}}{32} & -\frac{\sqrt{2}}{32} & -\frac{2\sqrt{6}}{32} & -\frac{6\sqrt{2}}{32} & \frac{\sqrt{6}}{32} & -\frac{5\sqrt{2}}{32} & \frac{3\sqrt{6}}{32} & \frac{\sqrt{2}}{32} \\ -\frac{15\sqrt{6}}{320} & -\frac{53\sqrt{2}}{320} & -\frac{29\sqrt{6}}{320} & \frac{\sqrt{2}}{320} & -\frac{14\sqrt{6}}{320} & \frac{54\sqrt{2}}{320} & \frac{15\sqrt{6}}{320} & \frac{53\sqrt{2}}{320} & \frac{29\sqrt{6}}{320} & -\frac{\sqrt{2}}{320} \\ \frac{45\sqrt{2}}{320} & -\frac{11\sqrt{6}}{320} & -\frac{9\sqrt{2}}{320} & -\frac{33\sqrt{6}}{320} & -\frac{54\sqrt{2}}{320} & -\frac{22\sqrt{6}}{320} & -\frac{45\sqrt{2}}{320} & \frac{11\sqrt{6}}{320} & \frac{9\sqrt{2}}{320} & \frac{33\sqrt{6}}{320} \end{bmatrix} \tag{4.14}$$

where $V = V_{BUS} = 200V$.

We observe that $\max |\{\mathbf{W}_{pi}\}_{i,j}| = 1.33 \times 10^{-3}$, so we define w_{max} equal to this

value and $\bar{\mathbf{W}}_{pi} = \frac{1}{w_{max}} \mathbf{W}_{pi}$. Therefore, we can write

$$\mathbf{q} = \mathbf{W}_{pi}\ \mathbf{x} = w_{max}\ x_{max}\ \bar{\mathbf{W}}_{pi}\ \bar{\mathbf{x}} \approx 324\ \bar{\mathbf{W}}_{pi}\ \bar{\mathbf{x}} \tag{4.15}$$

This equation implies 10 multiply-accumulate operations for each element of $\mathbf{q}$, so there can be an overflow. However, a study on the characteristics of the motor shows that the elements q_2 and q_3 are always below 50, this is much less than 324 so we conclude that the operation will not overflow. The element q_1 is not considered because it is not necessary to compute it for the observer, therefore it is discarded.

So, if we define $q_{max} = 324$, then $\bar{\mathbf{q}} = \frac{\mathbf{q}}{q_{max}} = \bar{\mathbf{W}}_{pi}\ \bar{\mathbf{x}}$. The computation of the elements of $\bar{\mathbf{q}}$ can be performed with 10 consecutive MAC operations and the overflow is guaranteed not to occur. Moreover, there are a couple of spare bits that can be used to change the scaling and improve the resolution, as we will show in the next section.

4.2.4 Observer

The observer involves the computation of the acceleration, speed and position estimates ($\hat{\alpha}$, $\hat{\omega}$ and $\hat{\theta}$ respectively), based on the observer error ϵ, and the computation of an updated observer error, based on the parameter and position estimates ($\mathbf{q}$ and $\hat{\theta}$ respectively).

For convenience, we will start by studying the error update

$$\epsilon = q_3\ \cos 2\hat{\theta} - q_2\ \sin 2\hat{\theta} \tag{4.16}$$

In order to improve the precision, we will define $\epsilon_{max} = \frac{q_{max}}{2} = 162$, so we can use the values $2\bar{q_2}$ and $2\bar{q_3}$, which have a better resolution, extracting them from the

MAC result by shifting the result to the left 1 bit. Therefore

$$\bar{\epsilon} = \frac{\epsilon}{\epsilon_{max}} = 2\bar{q_3}\ \cos 2\hat{\theta} - 2\bar{q_2}\ \sin 2\hat{\theta} \tag{4.17}$$

The sin and cos functions are computed using a fifth order polynomial approximation of the sin function on the first quadrant. All values can be converted to fit this range by basic trigonometric identities. The subroutine was extracted from [29].

The first equation of the observer is

$$\dot{\hat{\alpha}} = \gamma_3\ \epsilon \tag{4.18}$$

which can be discretized as

$$\hat{\alpha} := \hat{\alpha} + T_{PWM}\ \gamma_3\ \epsilon \tag{4.19}$$

We estimate the maximum value of the acceleration from the peak torque as $\alpha_{max} \approx \frac{T_{L,peak}}{H} \approx 10,000 rad/s$. Therefore

$$\bar{\hat{\alpha}} = \frac{\hat{\alpha}}{\alpha_{max}} := \bar{\hat{\alpha}} + \frac{T_{PWM}\ \gamma_3\ \epsilon_{max}}{\alpha_{max}}\ \bar{\epsilon} \tag{4.20}$$

We define the new coefficient $\bar{\gamma_3} = \frac{T_{PWM}\ \gamma_3\ \epsilon_{max}}{\alpha_{max}}$.

We follow a similar procedure for the speed and we conclude that

$$\bar{\hat{\omega}} := \bar{\hat{\omega}} + \frac{T_{PWM}\ \alpha_{max}}{\omega_{max}}\ \bar{\hat{\alpha}} + \frac{T_{PWM}\ \gamma_2\ \epsilon_{max}}{\omega_{max}}\ \bar{\epsilon} \tag{4.21}$$

where $\omega_{max} = 4,500 rpm = 1,800 rad/s$ and we define $ac_{coeff} = \frac{T_{PWM}\ \alpha_{max}}{\omega_{max}}$, and $\bar{\gamma_2} = \frac{T_{PWM}\ \gamma_2\ \epsilon_{max}}{\omega_{max}}$.

Finally, a similar expression is obtained for the position

$$\bar{\hat{\theta}} := \bar{\hat{\theta}} + \frac{T_{PWM}\ \omega_{max}}{\theta_{max}}\ \bar{\hat{\omega}} + \frac{T_{PWM}\ \gamma_1\ \epsilon_{max}}{\theta_{max}}\ \bar{\epsilon} \tag{4.22}$$

where $\theta_{max} = \pi$ and we define $sp_{coeff} = \frac{T_{PWM}\ \omega_{max}}{\theta_{max}}$, and $\bar{\gamma_1} = \frac{T_{PWM}\ \gamma_1\ \epsilon_{max}}{\theta_{max}}$. In this case, we will allow the register overflow, since in two's-complement it will provide the natural wraparound of the position value.

As described in 2.2.4, the values of γ_1, γ_2 and γ_3 are selected so that the poles of the linearized observer guarantee a fast response while not amplifying the noise. One possible set of values is $\gamma_1 = 9.45$, $\gamma_2 = 2,295$ and $\gamma_3 = 202,500$, which define poles at -300 and $-200 \pm j100$.

4.2.5 Controller

To compute the controller, we first find the average of the currents in the PWM period, then we convert the $\alpha\beta$ values to the dq frame, next we compute the desired dq voltages, and finally we convert these values back to the $\alpha\beta$ frame.

Average the currents

The average of the currents over one PWM period is

$$i_{av} = \sum_{k=1}^{6} \frac{i_{k-1} + i_k}{2} \frac{dt_k}{T_{PWM}} = \frac{i_{\alpha\beta,max}\ T_{CK}\ 2^{15}}{T_{PWM}} \sum_{k=1}^{6} \left(\frac{\bar{i_{k-1}}}{2} + \frac{\bar{i_k}}{2} \right) \frac{\bar{dt_k}}{2^{15}} \tag{4.23}$$

where the sum $\frac{\bar{i_{k-1}}}{2} + \frac{\bar{i_k}}{2}$ will not overflow and $\frac{\bar{dt_k}}{2^{15}}$ is the counter value of each subinterval represented in fractional format.

The sum will have at least two spare bits, because the value of $\bar{dt_k}$ is always less than 2^{10}. Therefore, to improve the resolution we define $i_{av,max} = \frac{i_{\alpha\beta,max}\ T_{CK}\ 2^{15}}{4\ T_{PWM}}$ so we can write

$$\bar{i_{av}} = \frac{i_{av}}{i_{av,max}} = 4 \sum_{k=1}^{6} \left(\frac{\bar{i_{k-1}}}{2} + \frac{\bar{i_k}}{2} \right) \frac{\bar{dt_k}}{2^{15}} \tag{4.24}$$

After a series of six MAC operations, the result can be extracted by shifting the accumulator two bits to the left. We note that,given these definitions, $i_{av,max} = 204.8A$ which means that there are still a few spare bits that could be exploited if more precision is needed.

Convert to dq frame

We define $i_{dq,max} = i_{av,max} = 204.8$ so the transformation is straightforward

$$\begin{aligned} \bar{i_d} &= \bar{i_{\alpha,av}} \cos\hat{\theta} + \bar{i_{\beta,av}} \sin\hat{\theta} \\ \bar{i_q} &= -\bar{i_{\alpha,av}} \sin\hat{\theta} + \bar{i_{\beta,av}} \cos\hat{\theta} \end{aligned} \tag{4.25}$$

Clearly, there will not be an overflow because of the already big value of $i_{dq,max}$.

Again, the sin and cos functions are computed using the subroutine introduced in section 4.2.4.

Compute controller outputs

The controller has three PI (proportional-integral) blocks, each one of them synthesizing the transfer function $k\frac{s+a}{s}$. A discrete version of these blocks uses one variable as the memory of the integrator, and can be expressed as

$$\begin{aligned} u &= x_{ref} - x \\ int &:= int + T_{PWM} \cdot u \\ y &= k \cdot u + k \cdot a \cdot int \end{aligned} \tag{4.26}$$

where x is the input, u is the intermediate error, int is the integrator variable, and y is the output.

As we can see, in addition to selecting the appropriate input and output scaling, we need to select the scaling for the integrator variable and make sure that no overflow will happen. We will select int_{max} such that the scaling of $k \cdot a \cdot int$ is the same scaling of the output y. We are assuming that the integral of the error will not be such that it will drive the output to its maximum value. In case this happens, we are adding also a saturation in the integrator so that, even if the error is big for a long time, the integrator variable will reach its maximum and stay there until the error changes its sign.

The controller coefficients are selected to obtain a good performance of the closed-loop system. One possible set of values, implementing the PI controllers transfer function $k_i \frac{s+a_i}{s}$ for $i = 1, 2, 3$ (see Fig. 2.5), is $k_1 = 4.4300$, $a_1 = 304.74$, $k_2 = 0.0362$, $a_2 = 9.6983$, $k_3 = 2.6300$ and $a_3 = 136.88$.

The block PI1, whose input is the current i_d, will become

$$\begin{aligned} \bar{u_1} &= \bar{i_{d,ref}} - \bar{i_d} \\ \bar{int_1} &:= \bar{int_1} + \frac{i_{dq,max}\ T_{PWM}}{int_{1,max}}\ \bar{u_1} \\ \bar{v_{dc}} &= \frac{k_1\ i_{dq,max}}{v_{dq,max}}\ \bar{u_1} + \frac{k_1\ a_1\ int_{1,max}}{v_{dq,max}}\ \bar{int_1} \end{aligned} \tag{4.27}$$

where $v_{dq,max} = 2V_{BUS} = 400V$ and $int_{1,max} = \frac{v_{dq,max}}{k_1\ a_1} = 0.2963$. The coefficient of $\bar{int_1}$ in the output equation equals to 1 because of the selection of its scaling.

The block PI2 can be written as

$$\begin{aligned} \bar{u_2} &= \bar{\omega_{ref}} - \bar{\hat{\omega}} \\ \bar{int_2} &:= \bar{int_2} + \frac{\omega_{max}\ T_{PWM}}{int_{2,max}}\ \bar{u_2} \\ \bar{i_{q,ref}} &= \frac{k_2\ \omega_{max}}{i_{dq,max}}\ \bar{u_2} + \frac{k_2\ a_2\ int_{2,max}}{i_{dq,max}}\ \bar{int_2} \end{aligned} \tag{4.28}$$

In this case, if we select $int_{2,max}$ as before, we would have a very small coefficient $\frac{\omega_{max}\ T_{PWM}}{int_{2,max}} \approx 6 \times 10^{-4}$ and this would generate steady-state errors due to the low precision of the coefficient. Therefore, we select a smaller value $int_{2,max} = \frac{i_{dq,max}}{4\ k_2\ a_2} = 145.84$ so that the coefficient becomes 2.468×10^{-3} and the precision is improved. The output of this block is the reference for the i_q block; to avoid problems we introduce a saturation on this output.

Finally, the block PI3 is

$$\begin{aligned} \bar{u_3} &= \bar{i_{q,ref}} - \bar{i_q} \\ \bar{int_3} &:= \bar{int_3} + \frac{i_{dq,max}\ T_{PWM}}{int_{3,max}}\ \bar{u_3} \\ \bar{v_{qc}} &= \frac{k_3\ i_{dq,max}}{v_{dq,max}}\ \bar{u_3} + \frac{k_3\ a_3\ int_{3,max}}{v_{dq,max}}\ \bar{int_3} \end{aligned} \tag{4.29}$$

where the values are computed in the same way as in PI1.

Some of the coefficients can take values slightly greater than one. To be able to perform a multiplication, we use only the fractional part of the coefficient and then add as many times as the integer part, e.g., $2.3x = 0.3x + x + x$.

Convert back to $\alpha\beta$ frame

The transformation is the inverse of (4.25) and can be written as

$$\begin{aligned} \bar{v_\alpha} &= \bar{v_{dc}}\ \cos\hat{\theta}_c - \bar{v_{qc}}\ \sin\hat{\theta}_c \\ \bar{v_\beta} &= \bar{v_{dc}}\ \sin\hat{\theta}_c + \bar{v_{qc}}\ \cos\hat{\theta}_c \end{aligned} \tag{4.30}$$

where we use the same scaling in both voltage frames, i.e., $v_{\alpha\beta,max} = v_{dq,max} = 400V$.

At this point we introduce a correction to compensate for the delay in the controller. We know that in the dq frame the values are almost constant in steady state,

however, when we convert them to the $\alpha\beta$ frame they become more sensitive to delays. The average currents used in the controller, as well as the estimated position, correspond to the middle-point of one PWM period, say period k. The observer and controller computations are done during the next period $k+1$, and the output is fed to the system during the following period $k+2$, achieving the desired averaged voltages in the middle of it. Therefore there is a delay of 2 T_{PWM} between inputs and outputs. It is correct to convert averaged currents from the $\alpha\beta$ to the dq frame using the value $\hat{\theta}$ because they correspond to the same point in time, but it is not correct to convert the desired voltages from dq to $\alpha\beta$ with that position value because they correspond to a delayed point in time.

For this reason, we use in transformation (4.30) the angle $\hat{\theta}_c$ which is a corrected version of $\hat{\theta}$, according to the prediction

$$\begin{aligned} \hat{\theta}_c &= \hat{\theta} + 2\ T_{PWM}\ \hat{\omega} \\ \Rightarrow \bar{\hat{\theta}}_c &= \bar{\hat{\theta}} + \frac{2\ T_{PWM}\ \omega_{max}}{\theta_{max}}\ \bar{\hat{\omega}} \end{aligned} \tag{4.31}$$

4.2.6 Counter values

Once we have the desired voltages in the $\alpha\beta$ frame, we need to compute the duty ratios (or equivalently, counter values) for each vector in the PWM period, according to (2.3). One way of achieving this is to find first the sector which the desired vector belongs to, so we can identify the lead and lag vectors. Then we can rotate the desired vector by increments of $\frac{\pi}{3}$ to the first sector and compute the duty ratios there. Finally, we assign the computed duty ratios to the correspondent vectors according to the sector number.

To identify the sector, we first identify the quadrant and then we find out the sector by comparing $\frac{v_\beta}{v_\alpha}$ with $\tan\frac{\pi}{3} = \sqrt{3}$. Once the sector has been identified, the vector is rotated to the first sector, where computations will take place.

The desired voltage is limited to the capabilities of the PWM generation. Ideally, the magnitude of the vector should be limited to $\frac{V_{BUS}}{\sqrt{2}}$, but this would imply the computation of a square-root which is numerically intensive. Rather, we will limit the vector so that it does not cross the straight line defined by the maximum voltages in the lead and lag vectors. This line is defined as $v_\beta = \sqrt{3}\left(\frac{V_{BUS}}{\sqrt{2}} - v_\alpha\right)$. If v_β is greater than this value, then the vector is scaled down by a factor $\frac{1}{v_\alpha + \frac{v_\beta}{\sqrt{3}}}$. This limitation strategy has the drawback that it reduces the voltage that can be delivered by the inverter without distortion from $\frac{1}{\sqrt{2}}V_{BUS}$ to $\sqrt{\frac{3}{8}}V_{BUS}$ (from 70.7% to 61.1% approximately). On the other hand, the implementation only requires a couple of multiplications instead of a square-root computation. In a future industrial implementation, the issue of computing the magnitude of the desired vector by means of a square-root operation should be addressed in order to extend the voltage limit. Two approaches could be taken: on one hand, we could implement the operation in the DSP if there are enough free program cycles; on the other hand, the operation could be implemented in the FPGA if more logic cells are available.

Finally, to obtain the counter values we note that

$$\begin{aligned} \frac{\sqrt{3}}{12\ v_{max}}\ v_\alpha &= \frac{1}{\sqrt{6}}\ \bar{v_\alpha} \\ \frac{1}{12\ v_{max}}\ v_\beta &= \frac{\sqrt{2}}{6}\ \bar{v_\beta} \end{aligned} \tag{4.32}$$

given that $v_{max} = \frac{V_{BUS}}{\sqrt{2}}$.

By adding and subtracting terms like (4.32) to the value $\frac{1}{6}$ all duty ratios are

computed. The counter values are obtained by converting the duty ratios to integers, i.e., taking the same register like an integer representation instead of fractional.

4.3 Simulations

The algorithm described above was implemented in a floating-point and a fixed-point version. The Matlab files are listed in Appendix B.

In Fig. 4.1 we show the simulation of a speed step from 0 to $500 rad/s$ (electrical), with a constant torque load of $1 Nm$. We can see that the fixed-point version has a larger overshot and a longer stabilization time. This is likely caused by the quantization errors introduced in the process, which lead to a different controller implementation. In Fig. 4.2 we show the error in the position estimation for both cases. In the fixed-point version, the error is slightly larger during the transient.

4.4 DSP code

Based on this description, the assembler code for the ADSP-2181 processor was developed. In Appendix C we show the assembler code and the Linker Description File, needed to assemble and link the program using the development software VisualDSP.

The algorithm runs in an interrupt service routine, while there is a main loop which only waits for the interrupt request. After a reset, the initialization of the variables is performed, and the program waits until a switch is changed of position to write in the FPGA the command that starts to run the PWM generation and the interrupt requests.

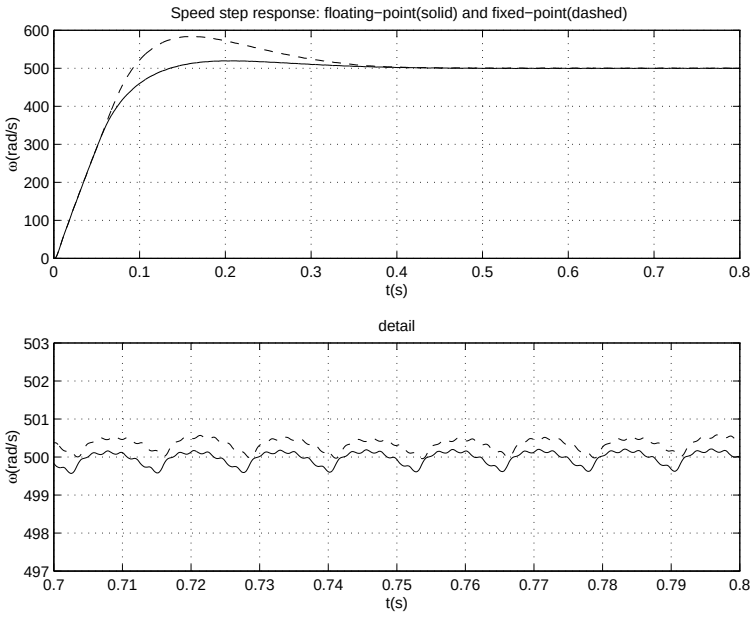


Figure 4.1: Simulation of the speed step response, from 0 to $500 rad/s$ with a constant torque of $1 Nm$. Floating-point version is shown in solid and fixed-point version is shown in dashed lines.

The total running time of the interrupt routine is $1,113$ cycles, equivalent to $55.65 \mu s$, while the PWM period is $200 \mu s$. There is enough time to add other tasks in the processor, like serial communication with a host to receive a dynamic speed reference and probably exchange other information. It is important to note that the processor clock frequency ($20 MHz$), as well as the PWM frequency ($5 kHz$), can be changed to obtain other results, but this would imply minor changes in the board design and in some constants.

4.5 Summary

The issues involved in the implementation of a fixed-point version of the sensorless algorithm were described in this chapter. All inputs, outputs, variables and constants

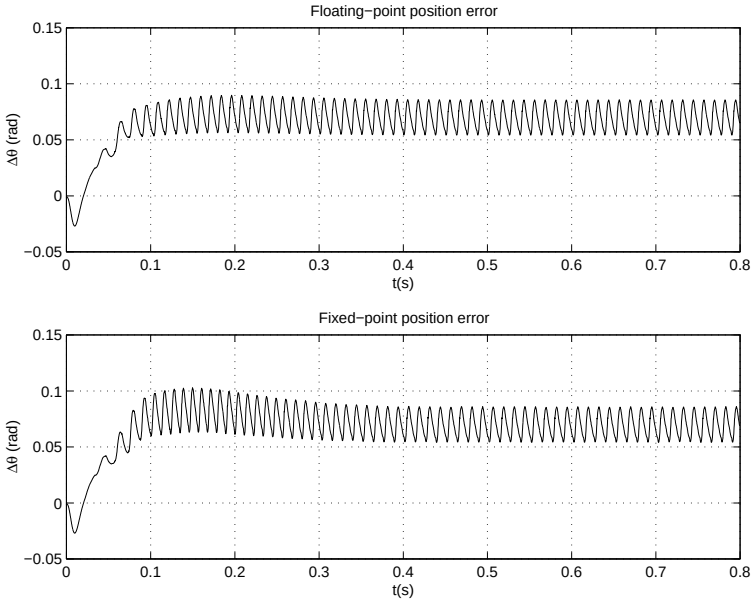


Figure 4.2: Simulation of the speed step response, position estimate error.

were scaled to a fractional representation, and all operations were studied to prevent overflow and to maximize the precision of the results. Simulations show that the fixed-point version of the algorithm differs from the floating-point version, but its performance is still satisfactory.

The algorithm was coded in assembler for the ADSP-2181 processor. With a clock frequency of $20MHz$ and a PWM frequency of $5kHz$, the algorithm fits into one PWM period and there is still time remaining to introduce other tasks, like serial communication or other application-specific tasks.

Chapter 5

PWM implementation using FPGA

In this chapter we describe the design of the PWM generation circuit and auxiliary tasks, which was implemented in an FPGA. The complete design was written in VHDL [31], an industry standard hardware description language, so that it can be easily implemented in different platforms (e.g., different FPGA families, ASICs). The VHDL files are listed in Appendix D.

5.1 General description

The main task performed by the FPGA is to generate the PWM signals, based on the duty ratios received from the DSP, and at the end of each PWM period it has to generate an interrupt request signal. Additionally, it has to store the samples of the currents at the end of each subinterval, and output them to the DSP when requested.

The design was partitioned in two blocks (Fig. 5.1). The **regs** block contains the internal registers which store the PWM vectors, the duration of each subinterval in the current and next PWM period, and the samples from the ADCs at the end of each subinterval. The **pwmcntr** block provides the exact timing for each subinterval and keeps track of the number of subintervals in a PWM period.

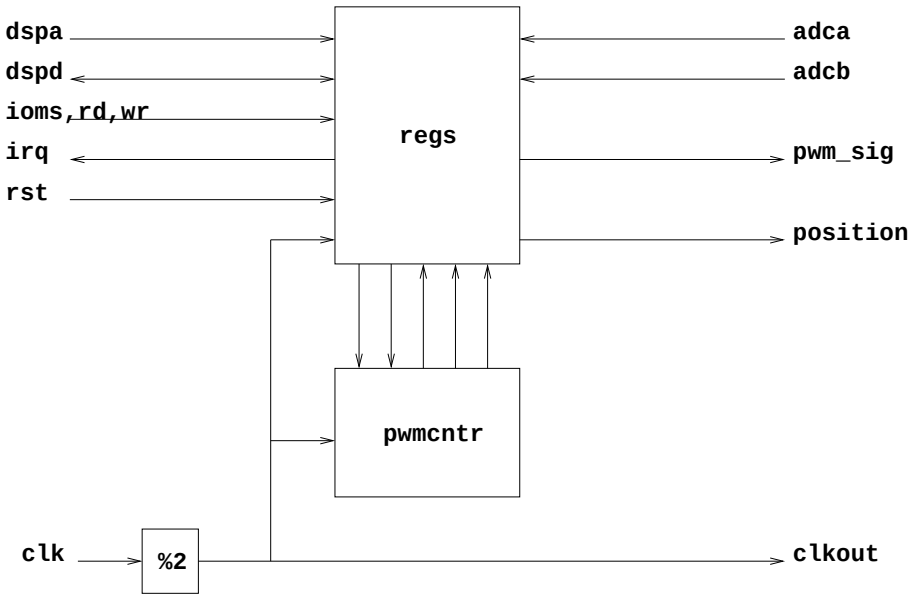


Figure 5.1: Block diagram of the FPGA design.

The interface between these blocks is very simple: **regs** commands **pwmcntr** by driving its reset signal and providing the value to be loaded in the counter for the next subinterval. The outputs of **pwmcntr** are the step (subinterval) number and two signals to indicate the end of a step and the end of a PWM period.

The communication with the DSP is performed by the **regs** block, whose registers are seen as I/O ports from the processor. For that purpose, the data and address buses, as well as control signals, are used to decode the I/O read or write operations. An interrupt request signal is issued at the end of every PWM period in order to

synchronize the system.

The clock signal generated by the DSP is divided by two to obtain a $10MHz$ clock, which is used to synchronize the digital circuit and the data acquisition.

5.2 PWM counter block

The PWM counter is implemented in the `pwmcntr.vhd` file. It consists of a single synchronous process. The counter `cnt` decreases at each clock cycle; when it reaches zero this means that a step has ended and therefore a new value is loaded to begin the count of the next step, the step number is incremented, and a `new_step` signal is issued.

The counter is used to generate the time duration of each subinterval, and also the duration of the deadband. The deadband (or deadtime) is a short time interval (in our case, $2\mu s$) inserted between two different switching combinations of the inverter to wait for a complete cease of conduction of one switch, before turning on its complementary [22]. As a consequence, instead of 6 subintervals in a PWM period, we have 12 since each subinterval is preceded by a deadband time. For this reason, we use the term *step* instead of *subinterval*. When the 12 steps have elapsed, the step counter is reset and a `new_period` signal is issued.

There is a synchronous reset signal which initializes all variables. This reset signal is commanded by the **regs** block when the PWM generation is stopped.

In Fig. 5.2 we present the results of a simulation of this block, showing the first step after a reset, and the `new_step` signal generation. In Fig. 5.3 we show what happens when the last step is finished, in particular the generation of the `new_period`

signal.

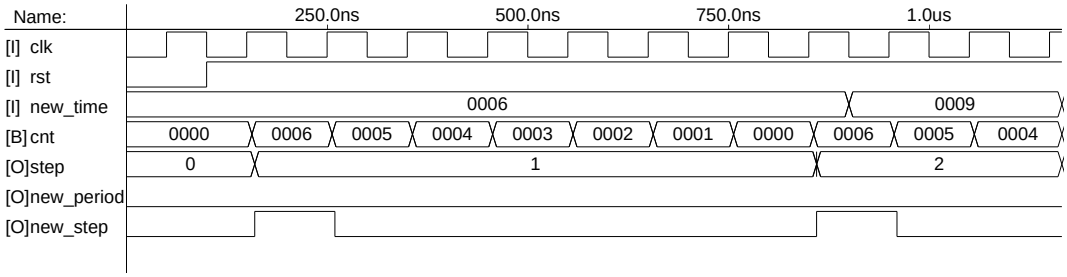


Figure 5.2: Simulation of the PWM counter block: reset and first step.

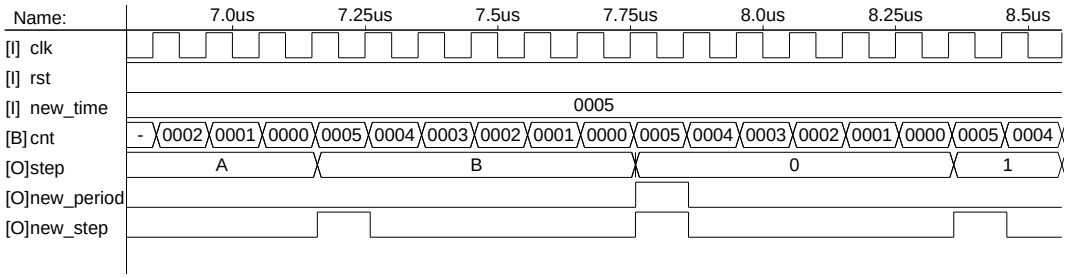


Figure 5.3: Simulation of the PWM counter block: last step.

5.3 Registers block

The registers block is implemented in the `regs.vhd` file. The main functions of this block are the interface with the DSP, the signal acquisition and the generation of the PWM signals according to the step number.

In Fig. 5.4 we show the registers inside this block. There is one bank of registers to store the time duration of each subinterval plus the deadband of the current PWM period (`tt1,...,tt6,ttdb`), and one bank for the time values of the next period (`t1,...,t6,tdb`). There is one bank of registers to store the PWM vectors to be used in each subinterval (`v1,...v6`). Register `sta` stores the status of the system

(stop or running), register `p` stores the estimated position of the shaft, and register `pwm_out` stores the actual PWM signals that are being sent to the inverter. Finally, the current measurements at the end of each subinterval are stored temporarily in one bank (`tadca1,tadcb1,...,tadca6,tadcb6`) and at the end of the PWM period copied to another bank (`adca1,adcb1,..., adca6,adcb6`).

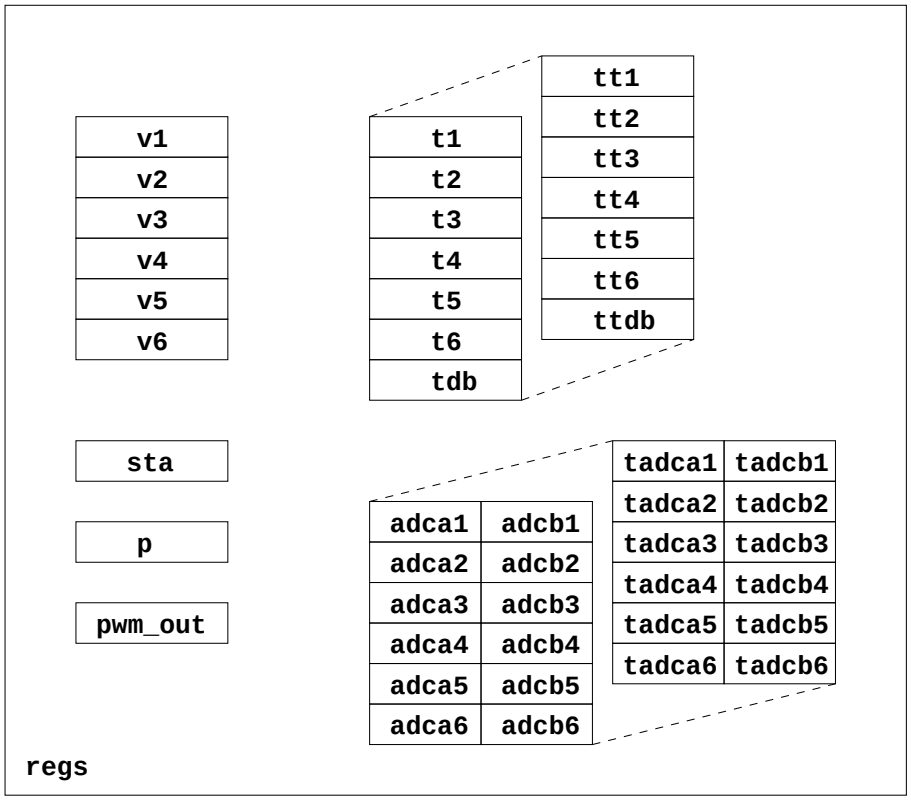


Figure 5.4: Internal registers.

5.3.1 Interface with the DSP

The internal registers of the FPGA are mapped in the I/O space of the DSP by decoding the appropriate control signals. The `ioms_n` signal is asserted by the DSP

when an I/O access is being done; the signals `wr_n` and `rd_n` distinguish a write from a read operation.

The write operation is implemented in a process whose sensitive list includes `wrs = wr_n OR ioms_n`. On the rising edge of this signal, an internal register is loaded depending on the address. The vectors can only be loaded if the status is 0 (stop); this means that, for security reasons, the vectors can't be changed while they are being output to the inverter. In a normal operation they would be loaded only once at the beginning and never changed. The time duration of the subintervals can be loaded any time on the temporal registers so that the PWM period that is currently running is not affected. At the end of the period, the temporal registers are copied to the actual ones, which are used to generate the next PWM period. Finally, the status and position registers can be written at any time.

The read operation is implemented with a tristate buffer controlled by the signal `srden = rd_n OR ioms_n`. When this signal is 0, the value `srdata` is connected to data bus; otherwise a high impedance output is forced. The values that can be red on the `srdata` signal are the ADC values (sampled at the end of each subinterval of the previous PWM period) or the status, depending on the address bus value. The ADC values are transformed in the following way: the MSB is inverted to achieve a two's-complement notation and the 14 bits are shifted one position to the left for appropriate scaling (see 4.2.1). The addresses of the registers are summarized in Table 5.1. Only the four less significant bits of the address bus are used in the decodification.

The interrupt request signal `irq_n` is the inverse of the `new_period` signal. In this way, at the end of a PWM period `irq_n` will go to zero for one clock cycle; this is enough because the interruption that we are using in the DSP is edge sensitive.

address	write	read
0h	v1	adca1
1h	v2	adca2
2h	v3	adca3
3h	v4	adca4
4h	v5	adca5
5h	v6	adca6
6h	sta	sta
7h	p	—
8h	t1	adcb1
9h	t2	adcb2
Ah	t3	adcb3
Bh	t4	adcb4
Ch	t5	adcb5
Dh	t6	adcb6
Eh	tdb	—
Fh	—	—

Table 5.1: Register addresses.

In Fig. 5.5 we show a simulation where the DSP reads the ADC values from internal registers of the FPGA. One wait state had to be added so that the data is ready on the rising edge of the read signal.

5.3.2 PWM signals generation

The PWM signals are generated from the vector registers and the step number. The even steps correspond to deadband intervals, during which a logical AND of the previous and the next vectors is output to the inverter; this operation guarantees that those switches that change from one subinterval to the next, during the deadband will be forced to an OFF state. The PWM signal is therefore selected according to Table 5.2.

To achieve the desired duration for each step, the output `new_time` is used to

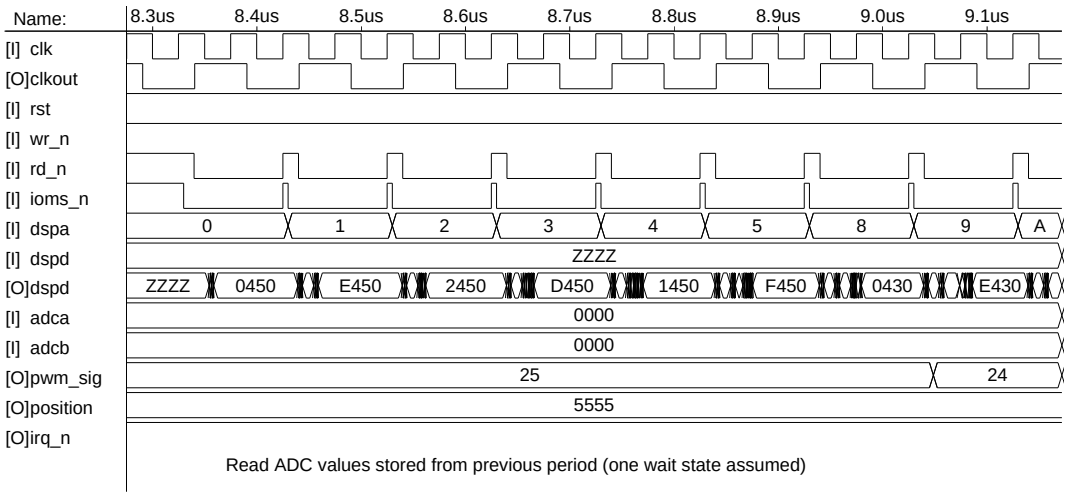


Figure 5.5: Simulation of the registers block: DSP reading ADC values.

communicate the block **pwmcntr** which is the counter value to be loaded for the next step. Therefore, in step 0 we output the register corresponding to subinterval 1, in step 1 the register corresponding to deadband, and so on.

In Fig. 5.6 we show a simulation where the DSP writes a command to start the PWM generation. After one period delay, a vector corresponding to the deadband is output for one period, and next the first vector is output for the corresponding time.

5.3.3 Signal acquisition

The signal acquisition belongs to the synchronous process. When the `new_step` signal is received, the step value has been already increased. Therefore when step is 2, it corresponds to the end of the first subinterval; when step is 4, to the end of the second subinterval, and so on. When step is 0, it means that the last subinterval has finished and then all temporary registers are copied to the definitive bank, where they will be accessible to the DSP during the next PWM period.

step	subinterval	PWM vector
0	DB	v6 AND v1
1	1	v1
2	DB	v1 AND v2
3	2	v2
4	DB	v2 AND v3
5	3	v3
6	DB	v3 AND v4
7	4	v4
8	DB	v4 AND v5
9	5	v5
10	DB	v5 AND v6
11	6	v6

Table 5.2: PWM signal generation according to the step number

Because of the latency in the ADCs, the value that is latched in the registers corresponds to the analog value of the current four clock periods before (i.e., $.4\mu s$).

5.4 Practical considerations

The files were compiled for an Altera EPF6016 chip (package TC144 and speed grade -3), using the software Max+PlusII. The total of logic cells and I/O pins used are around 75% of the chip capacity.

The design was extensively simulated to verify its correctness. Some of those simulations were shown in the previous sections.

One of the big advantages of programmable logic is its flexibility in the pin assignment, which make it possible to simplify the PCB design. For this reason, the pins were assigned according to the pin distribution of the other chips and their position in the PCB with respect to the FPGA.

There are four things that can be noticed in the VHDL code and deserve an

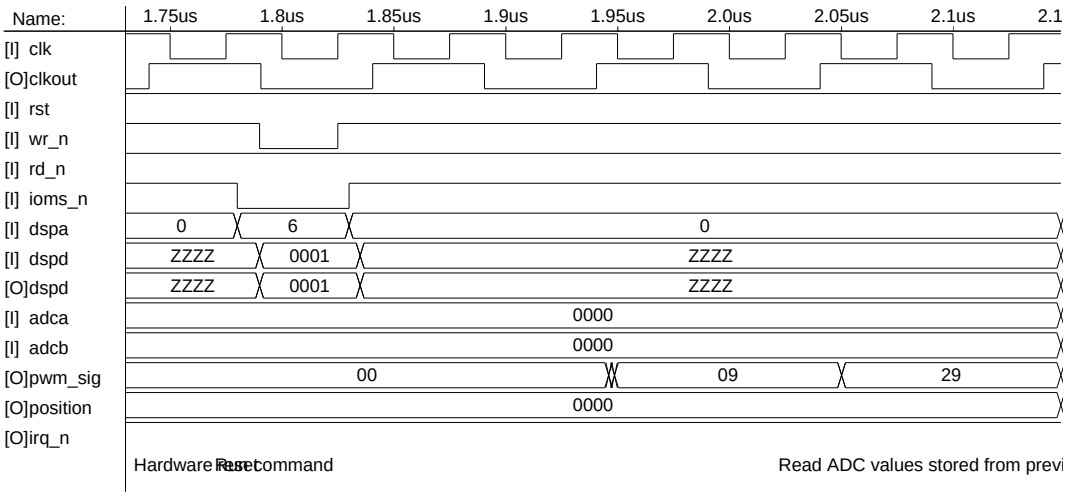


Figure 5.6: Simulation of the registers block: start of the PWM generation.

explanation:

- four outputs (test points `tp`) are used for debugging purposes, so that we could have access to signals buried inside the FPGA.

- the bits in the inputs from the ADCs had to be reversed because there was an error in the design of the PCB: bit 1 in the ADCs is the most significant bit (MSB) and it was assumed to be the least significant (LSB); therefore the inputs `adca` and `adcb` were inverted into signals `adcar` and `adcbr` respectively.

- four additional inputs (`inp75,inp79,inp80,inp81`) were added to the design because of an accidental shortcut of corresponding pins during manual soldering; these pins have to be declared as inputs to avoid electrical problems.

- also because of an accident during the soldering, pin 107 of the chip was broken, so we had to assign signal `dspd14` to another I/O pin (#74) and connect it to the proper PCB trace with a cable.

5.5 Summary

We presented the design of the PWM generation and data acquisition, which was written in VHDL, an industry standard hardware description language. Although in this prototype the design was implemented in an Altera FLEX 6000 family chip, this design can be easily implemented in other platforms, like other FPGA chips or ASICs.

The interface with the DSP was designed to be as simple as possible. The PWM generation is robust in the sense that it can't be corrupted in the middle of a PWM period. However, it is the responsibility of the DSP programmer to load the correct vectors and counter values.

With a bigger FPGA, some improvements can be made. On one hand, basic computations can be performed, like the subtraction of subsequent current measurements, to free some program cycles from the DSP. On the other hand, the performance of the PWM generation can be made more robust if the DSP writes the desired voltage output and the FPGA itself computes the counter values needed to generate them.

Chapter 6

Experimental results

6.1 Experimental setup

The prototype board was introduced in a laboratory test bed consisting of a PM motor, an inverter with its power supply, and a host PC.

The motor used is a Kollmorgen Goldline XT, model MT306B, with a rated power of 2HP (1.48kW), maximum speed 4,600rpm, rated torque 3.32Nm, and 4 pole pairs. It is mechanically coupled to a torquemeter and a hysteretic load cell. It also has a digital encoder to measure the shaft position (for performance evaluation purposes).

The inverter, which is described in Appendix E, is built around a Powerex (Mitsubishi) PM15CJ060 intelligent power module, consisting of six IGBTs and gate drivers. It also contains three closed-loop Hall effect current sensors to measure the phase currents. The input signals are isolated from the power module with optocouplers. The DC power is provided by a programmable power supply HP6575A from Hewlett Packard, with a maximum of 200V/11A.

The Personal Computer (PC) runs a Windows NT Operating System. The software Max+PlusII from Altera is used to compile and download the FPGA configuration (via a ByteBlasterMV cable connected to the parallel port), while the software VisualDSP from Analog Devices is used to compile and download the DSP program (via an EZ-ICE2181 emulator, connected to the serial port). A controller DSP board (DS1103 from Dspace) registers the values of phase currents, actual and estimated position, via A/D converters, encoder interface and digital I/O respectively. This registered values were used for debugging purposes and to generate the plots shown in the next section.

In Fig. 6.1 we show a picture of the experimental setup in the laboratory.

Figure 6.1: Experimental setup. From left to right: motor, CPU, prototype board, monitor, inverter and DC power supply.

The procedure followed to perform the experiments is as follows:

1. turn on the PC
2. turn on the prototype board
3. turn on the emulator
4. turn on the inverter gate drivers
5. turn on the DC power supply and set 200V/3A (or more current if needed)
6. run Max+PlusII and configure the FPGA
7. run ControlDesk and read shaft position (note: the ByteBlasterMV cable has to be disconnected because it conflicts with the Dspace hardware key)
8. run VisualDSP, introduce initial shaft position in the DSP program, compile, download to the prototype board and run
9. change switch to enable PWM signals in the inverter
10. change switch to start algorithm in prototype board
11. read experimental data in ControlDesk
12. if needed, operate load cell
13. if needed, operate pushbutton in prototype board to change reference speed
14. change switch to disable PWM signals in the inverter when finished
15. turn off power supply
16. turn off inverter gate drivers
17. turn off emulator
18. turn off prototype board
19. shut down and turn off PC

Several experiments can be performed in one session by repeating steps 7 to 14.

6.2 Results

In this section we describe some of the experiments carried out to assess the performance of the sensorless control algorithm implementation.

In Fig. 6.2 we show the response of the system to a speed reference step. The torque was kept constant, equal to $1Nm$, ($\approx 141inoz$) while the speed reference changed abruptly from 50 to $500rad/s$ (electrical). In this experiment, we used the same controller constants as in the simulations carried out in Chapter 4. We can see the high overshoot generated by the quantization errors, as anticipated by the simulations. In Fig. 6.3 we show the position estimation error during the same experiment; as expected, during the transient the error is large, but in steady state it is significantly reduced.

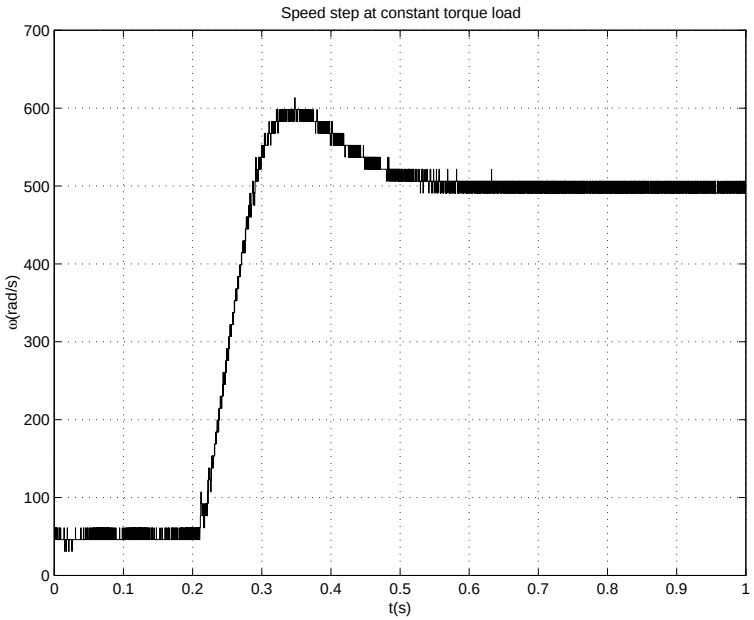


Figure 6.2: Speed step response, the speed reference jumps from 50 to $500rad/s$ (electrical) with a constant torque of $1Nm$.

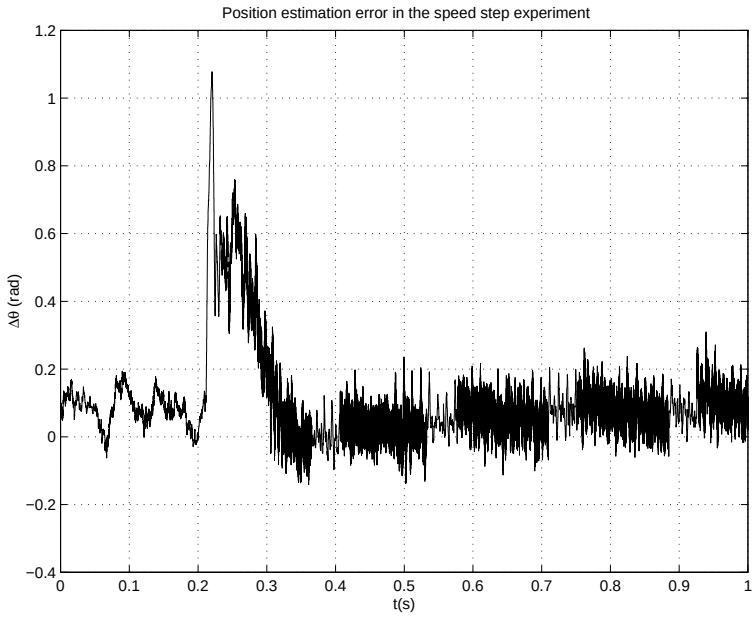


Figure 6.3: Speed step response, position estimation error in rad (electrical).

A better performance can be achieved if we use a less agressive controller. In Figs. 6.4 and 6.5 we show the results of the same experiment, with a modified controller. We can observe that the overshoot is reduced, while the stabilization time is slightly increased. The position estimation error is reduced during the transient, but in steady state it is almost the same.

In Fig. 6.6 we show the response to a load torque step, while trying to keep the speed constant. The torque changed from 0 to $1Nm$ and the speed remained constant at $200rad/s$. In Fig. 6.7 we show the position estimation error.

6.3 Summary

The algorithm has been implemented in the prototype board and its performance is satisfactory, although it is not as good as its floating-point counterpart. In steady

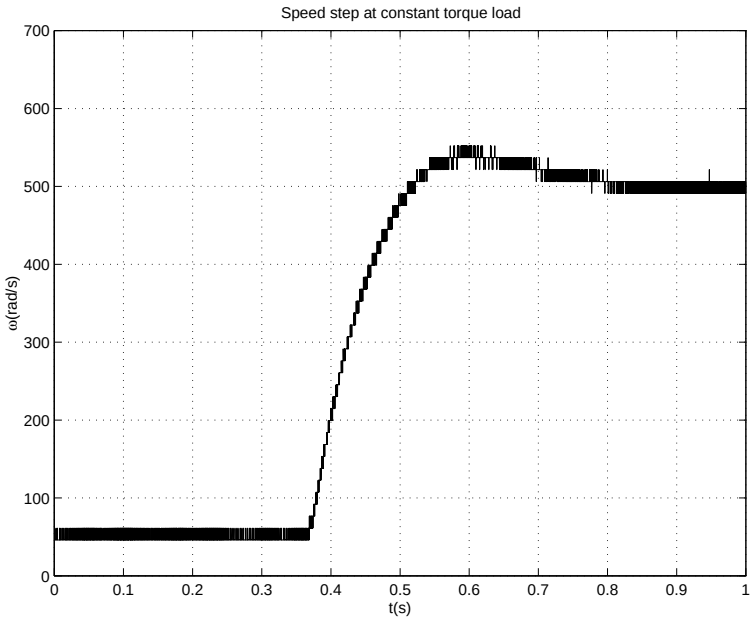


Figure 6.4: Speed step response with modified controller, the speed reference jumps from 50 to $500 rad/s$ (electrical) with a constant torque of $1 Nm$.

state, the position estimation error is about $.25 rad$ (electrical), while the error reported for the floating-point version is $.15 rad$ [12]. Also, in our implementation there is a positive offset when torque is applied, which is not present in the floating-point implementation.

Since it was not the main objective of this work to optimize the controller constants, there is still the possibility to improve the performance of this implementation. Simulations presented in chapter 4 show that the behavior of the fixed-point version of the algorithm can differ from the floating-point one, due to quantization errors. In this chapter, we showed how a modification in the controller constants can improve the performance; however, the procedure was an empirical one. Additional analysis should be made in order to find a specific set of constants for a better performance of the fixed-point version.

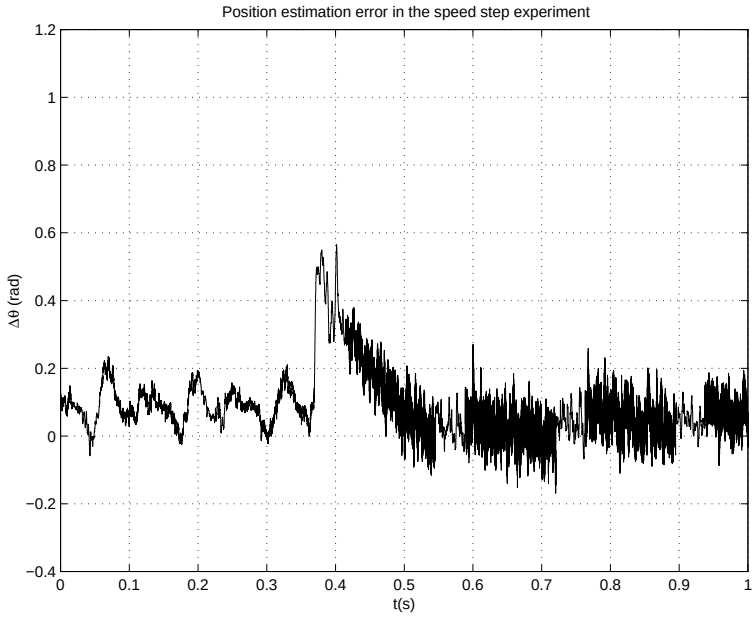


Figure 6.5: Speed step response with modified controller, position estimation error in rad (electrical).

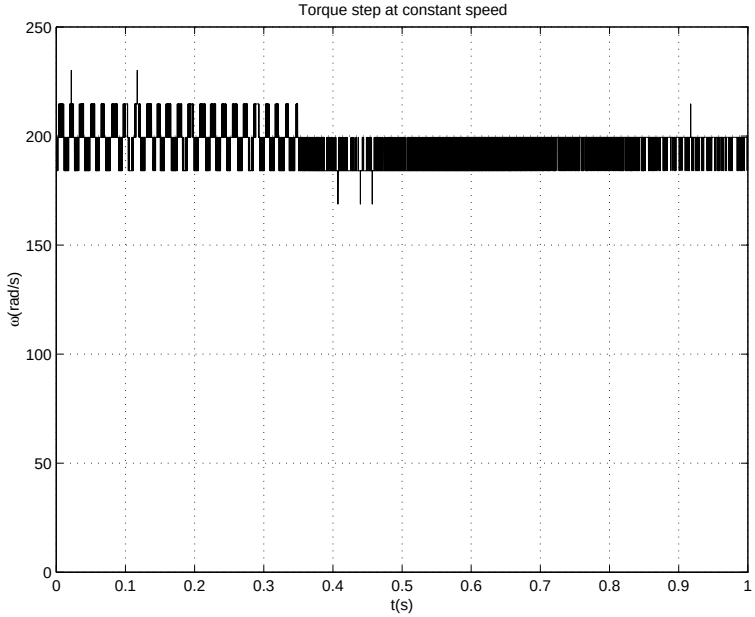


Figure 6.6: Torque step response, the load torque jumps from 0 to $1Nm$ (electrical) with a constant speed reference of $200rad/s$.

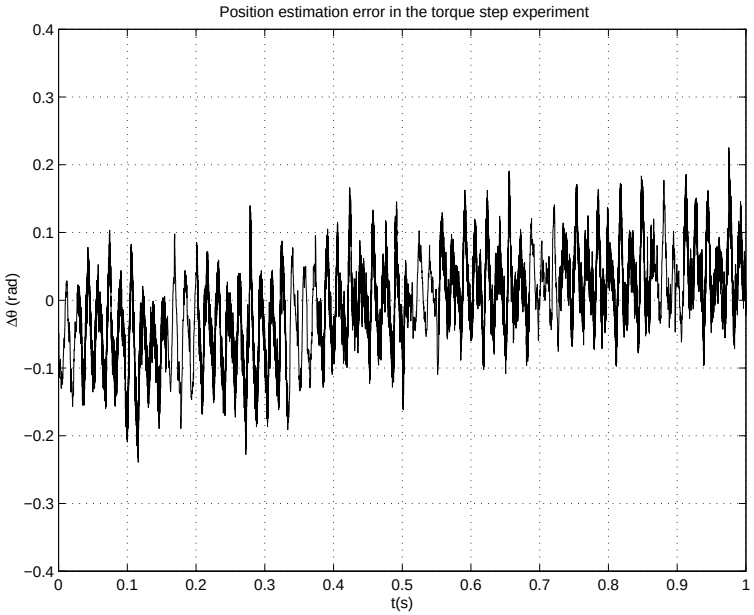


Figure 6.7: Torque step response,position estimation error in *rad* (electrical).

Chapter 7

Conclusions

In this work we have presented a low-cost hardware implementation of a sensorless control algorithm for PMSM. A custom board was designed and constructed for this purpose, based on a 16-bit fixed-point DSP and an FPGA. The main idea behind this design is to use programmable logic as a complement of the processor, so that different tasks can be implemented in parallel.

Word-length effects were studied in order to program the algorithm in the fixed-point DSP with a reasonable precision. Simulations show that the fixed-point version of the algorithm has a similar performance to that of its floating-point counterpart. However, quantization errors lead to variations in the observer and the controller which determine a modification in the response of the algorithm. Some constants should be adjusted to achieve an improved performance.

The FPGA was programmed in VHDL, an industry standard language. The PWM generation is achieved with a time resolution of $100ns$. Only basic functions were programmed in the FPGA, like PWM generation and signal acquisition, but

some of the operations performed by the DSP can be programmed also as long as the chip size permits. On one hand, the acquired data can be pre-processed, e.g., the subtraction of subsequent current measurements or the computation of the current average. On the other hand, the DSP could only output the desired $\alpha\beta$ voltages, and the duty ratios could be computed in the FPGA.

The spare time in the DSP can be used to implement serial communication to a host, to exchange data as speed reference, actual position, etc. Other background tasks could also be performed according to the specific application.

The design presented in this work is intended to target a prototype board. A final solution would need some modifications to convert it to a completely stand-alone board. An EPROM for the FPGA configuration and another for the DSP booting should be added. A startup procedure should be implemented, based on the position estimation algorithm, to find out the initial position [9, 10]. Additional communication with the user could be introduced if desired.

A more cost-effective solution could be achieved if the FPGA part is converted into an ASIC, and possibly integrated in the same chip as the DSP core. The ADCs could be substituted by others with less resolution and longer conversion time, at the expense of losing precision in the position estimation.

The prototype board built as part of this work showed a satisfactory performance in laboratory experiments, over a wide range of motor speeds and load torque. This result demonstrates that sensorless control of PMSMs can be achieved in a low-cost hardware platform if a design is performed according to the characteristics of the algorithm, using a mix of a standard DSP core and a custom designed digital circuit. The use of programmable logic provides a key to develop such a design.

Bibliography

[1] R.B. Sepe and J.H. Lang. Real-time observer-based (adaptive) control of a permanent-magnet synchronous motor without mechanical sensors. *IEEE Transactions on Industry Applications*, 28(6):1345–1352, 1992.

[2] S. Bolognani, R. Oboe, and M. Zigliotto. Sensorless full-digital PMSM drive with EKF estimation of speed and rotor position. *IEEE Transactions on Industrial Electronics*, 46(1):184–191, 1999.

[3] J.S. Kim and S.K. Sul. New approach for high-performance PMSM drives without rotational position sensors. *IEEE Transactions on Power Electronics*, 12(5):904–911, 1997.

[4] K.R. Shouse and D.G. Taylor. Sensorless velocity control of permanent magnet synchronous motors. *IEEE Transactions on Control Systems Technology*, 6(3):313–324, 1998.

[5] N. Ertugrul and P.P. Acarnley. Indirect rotor position sensing in real time for brushless permanent magnet motor drives. *IEEE Transactions on Power Electronics*, 13(4):608–616, 1998.

[6] M.J. Corley and R.D. Lorenz. Rotor position and velocity estimation for a salient-pole permanent magnet synchronous machine at standstill and high speeds. *IEEE Transactions on Industry Applications*, 34(4):784–789, 1998.

[7] P.L. Jansen and R.D. Lorenz. Transducerless position and velocity estimation in induction and salient AC machines. *IEEE Transactions on Industry Applications*, 31(2):240–247, 1995.

[8] J.M. Kim, S.J. Kang, and S.K. Sul. Vector control of interior permanent magnet synchronous motor without shaft sensor. In *Proceedings APEC'97*, volume 2, pages 743–748, Atlanta, GA, Febraury 1997.

[9] S. Östlund and M. Brokemper. Sensorless rotor-position detection from zero to rated speed for an integrated PM synchronous motor drive. *IEEE Transactions on Industry Applications*, 32(5):1158 – 1165, 1998.

[10] T. Aihara, A. Toba, T. Yanase, A. Mashimo, and K.Endo. Sensorless torque control of salient-pole synchronous motor at zero-speed operation. *IEEE Transactions on Power Electronics*, 14(1):202 – 208, 1999.

[11] M. Schroedl and P. Weinmeier. Sensorless control of reluctance machines at arbitrary operating conditions including standstill. *IEEE Transactions on Power Electronics*, 9(2):225–231, 1994.

[12] V. Petrović. *Saliency-Based Position Estimation in Permanent Magnet Synchronous Motors*. PhD thesis, Northeastern University, 2001.

[13] Y-Y. Tzou and H-J Hsu. FPGA realization of space-vector PWM control IC for three-phase PWM inverters. *IEEE Transactions on Power Electronics*, 12(6):953–963, 1997.

[14] J.A. du Toit, D.D. Bester, and J.H.R. Ensin. A DSP based controller for back to back power electronic converters with FPGA integration. In *Proceedings APEC'97*, volume 2, pages 699–705, 1997.

[15] W. Sangchai, T. Wiangtong, A. Hongyapanun, and P. Wardkean. Design and implementation of FPGA-based control IC for 3-phase PWM inverter with optimized SVM schemes. In *Proceedings APCCAS 2000*, pages 144–147, 2000.

[16] F. Blaabjerg, P.C. Kjaer, P.O. Rasmussen, and C. Cossar. Improved digital current control methods in switched reluctance motor drives. *IEEE Transactions on Power Electronics*, 14(3):563–572, 1999.

[17] S-L. Jung, M-Y. Chang, J-Y. Jyang, and Y-Y. Tzou. Design and implementation of an FPGA-based control IC for AC-voltage regulation. *IEEE Transactions on Power Electronics*, 14(3):522–532, 1999.

[18] R-X. Chen, L-G. Chen, and L. Chen. System design consideration for digital wheelchair controller. *IEEE Transactions on Industrial Electronics*, 47(4):898–907, 2000.

[19] G.R. Slemon. Electrical machines for drives. In B.K. Bose, editor, *Power Electronics and Variable Frequency Drives*, chapter 2. IEEE Press, Piscataway, NJ, 1997.

[20] T.M. Jahns. Variable frequency permanent magnet ac machines drives. In B.K. Bose, editor, *Power Electronics and Variable Frequency Drives*, chapter 6. IEEE Press, Piscataway, NJ, 1997.

[21] S.A. Nasar, I. Boldea, and L.E. Unnewehr. *Permanent Magnet, Reluctance, and Self-Synchronous Motors*. CRC Press, Boca Raton, FL, 1993.

[22] J. Holtz. Pulse width modulation for electronic power converters. In B.K. Bose, editor, *Power Electronics and Variable Frequency Drives*, chapter 2. IEEE Press, Piscataway, NJ, 1997.

[23] Analog Devices, Inc. *Complete 14-Bit, 10 MSPS Monolithic A/D Converter AD9240*, 1998. Datasheet.

[24] Analog Devices, Inc. *Dual 160MHz Rail-to-Rail Amplifier AD8042*, 1999. Datasheet.

[25] Altera Corporation. *FLEX 6000 Programmable Logic Family*, November 1999. Datasheet.

[26] Analog Devices, Inc. *Configuring APEX 20K, FLEX 10K & FLEX 6000 Devices*, May 2000. Application Note 116.

[27] Analog Devices, Inc. *DSP Microcomputer ADSP-2181*, 1998. Datasheet.

[28] Analog Devices, Inc. *Understanding 21xx/218x EZ-ICE Theory of Operation To Aid In Designing An EZ-ICE Compatible Target*, April 1998. Engineer To Engineer Note EE-34.

[29] Analog Devices Inc. *ADSP-2100 Family Manual.* Analog Devices Inc., Norwood, MA, 2000.

[30] H. Hanselmann. Implementation of digital controllers – a survey. *Automatica*, 23(1):7 – 32, 1987.

[31] IEEE. *IEEE Standard VHDL Language Reference Manual.* IEEE Press, New York, NY, 1993. IEEE Std 1076-1993.

Appendix A

Hardware schematics and PCB

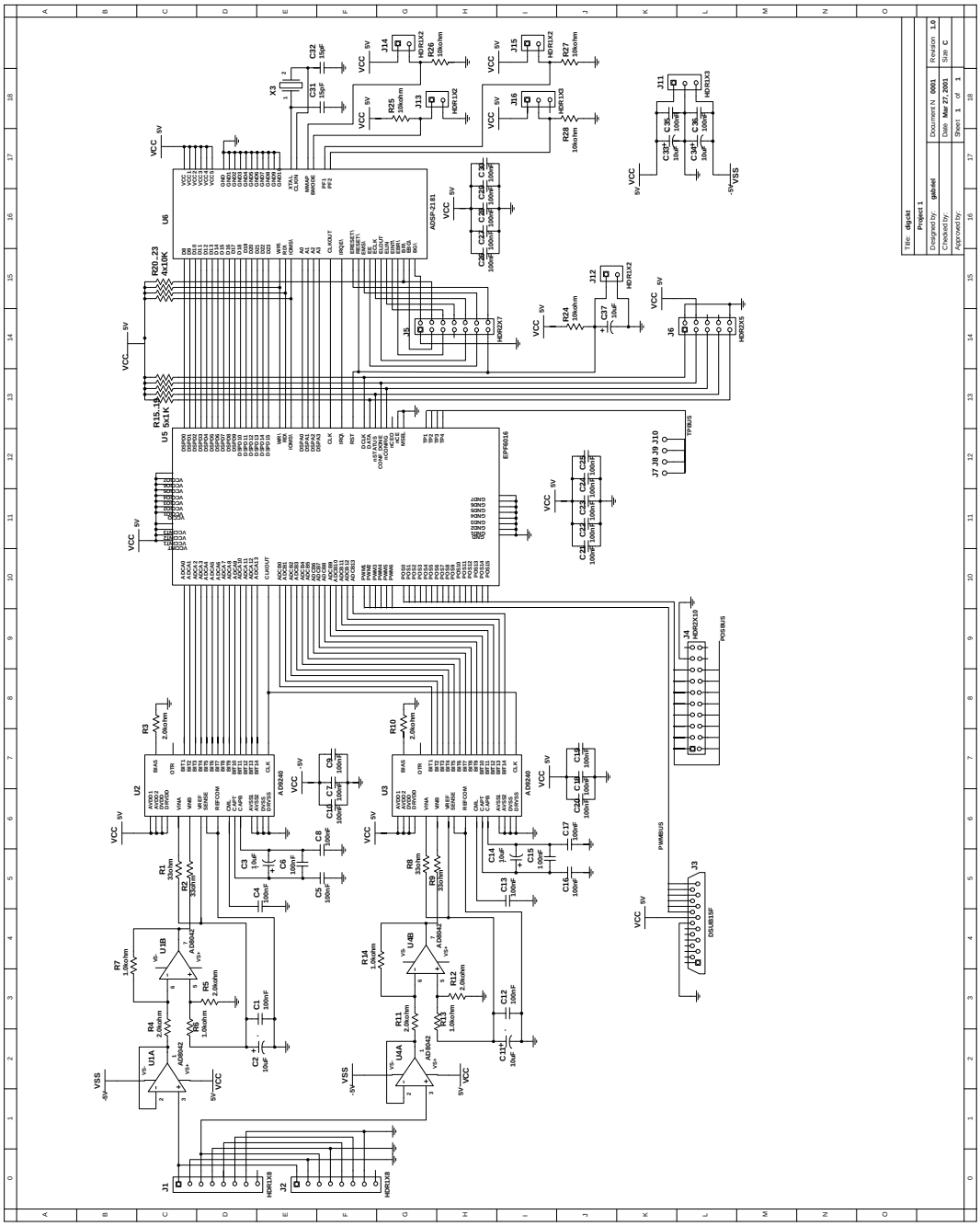


Figure A.1: Board schematics

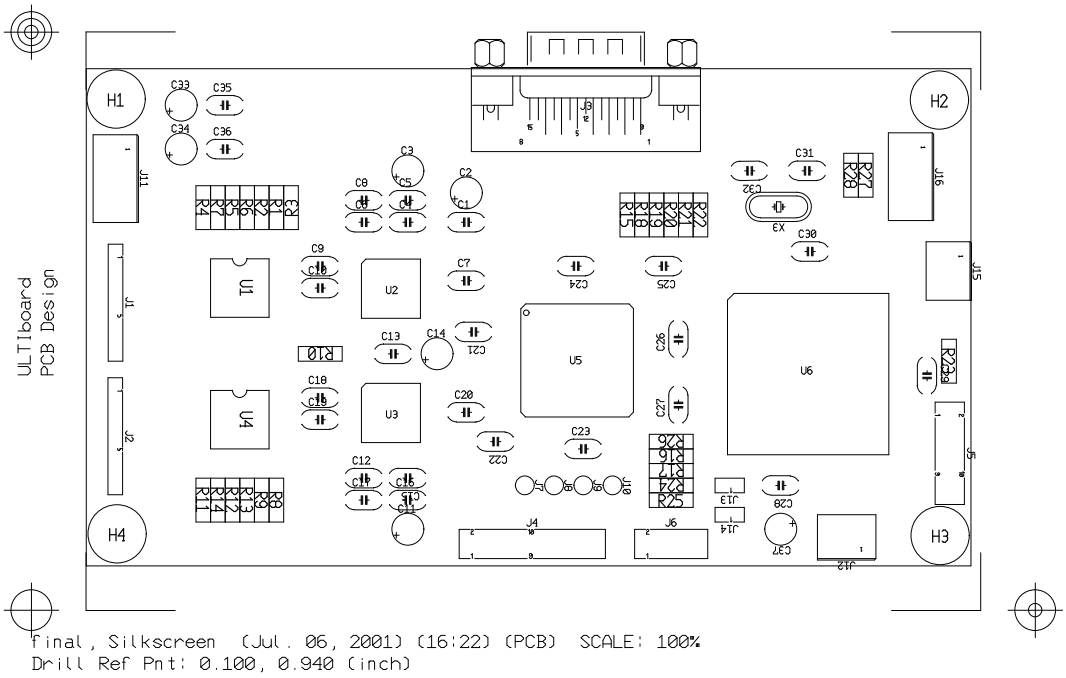


Figure A.2: PCB silkscreen

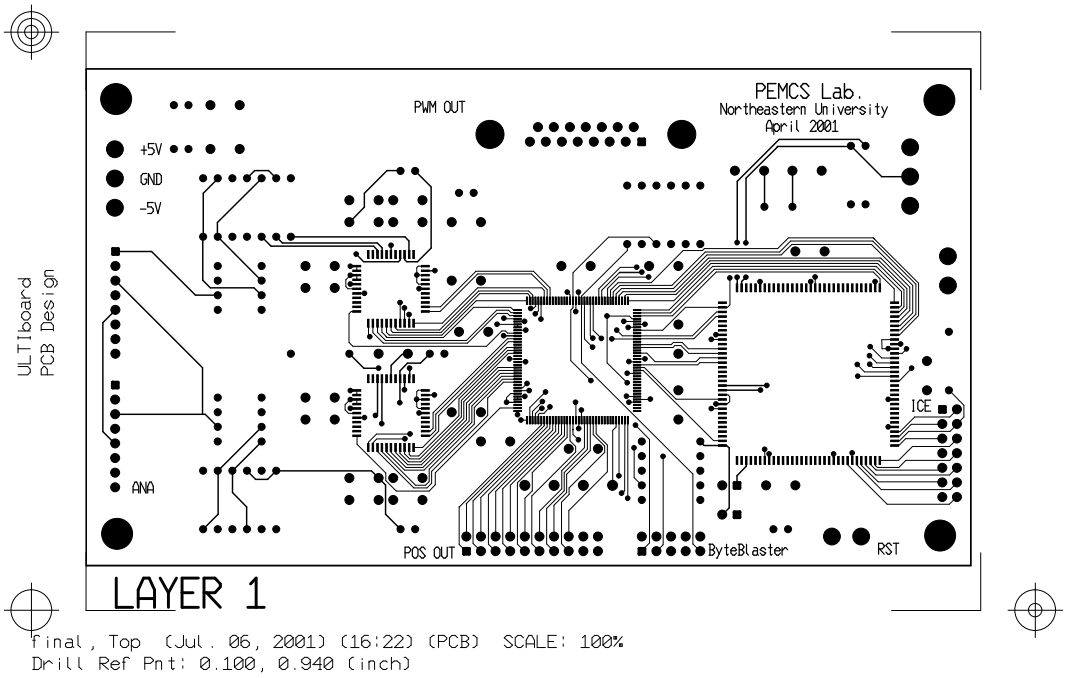


Figure A.3: PCB top layer

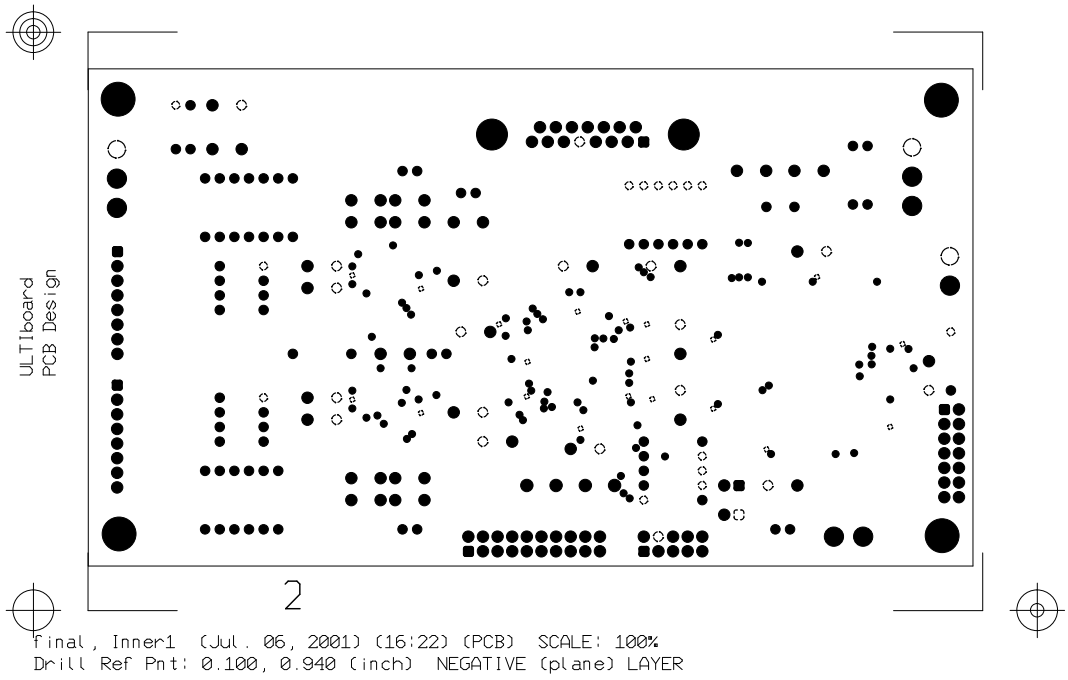


Figure A.4: PCB inner layer 1

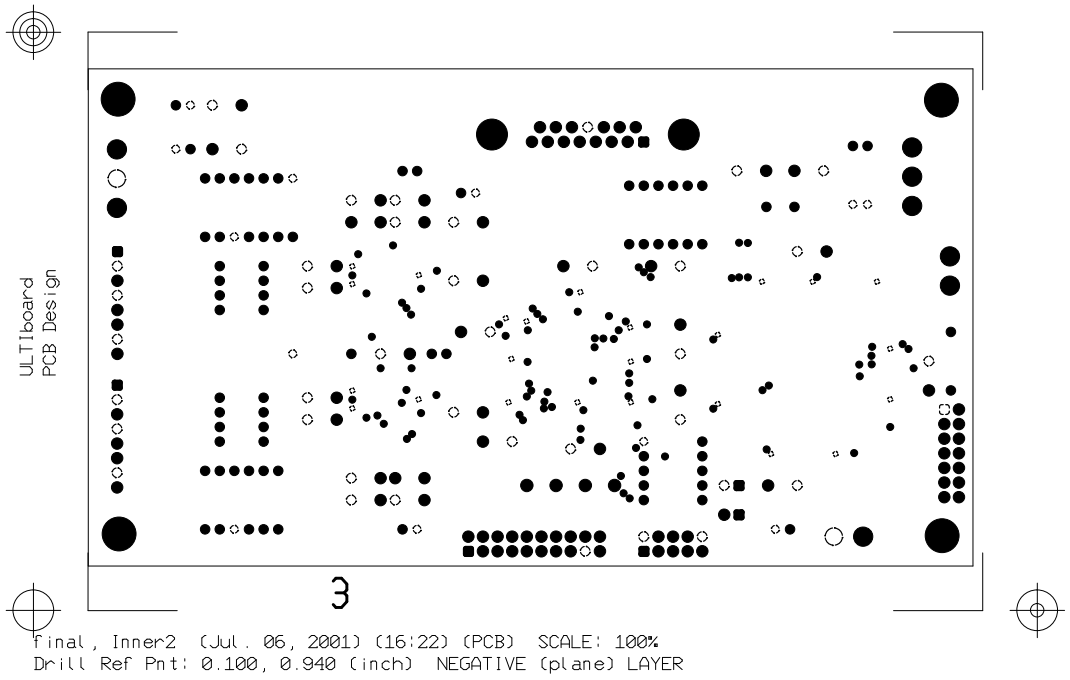


Figure A.5: PCB inner layer 2

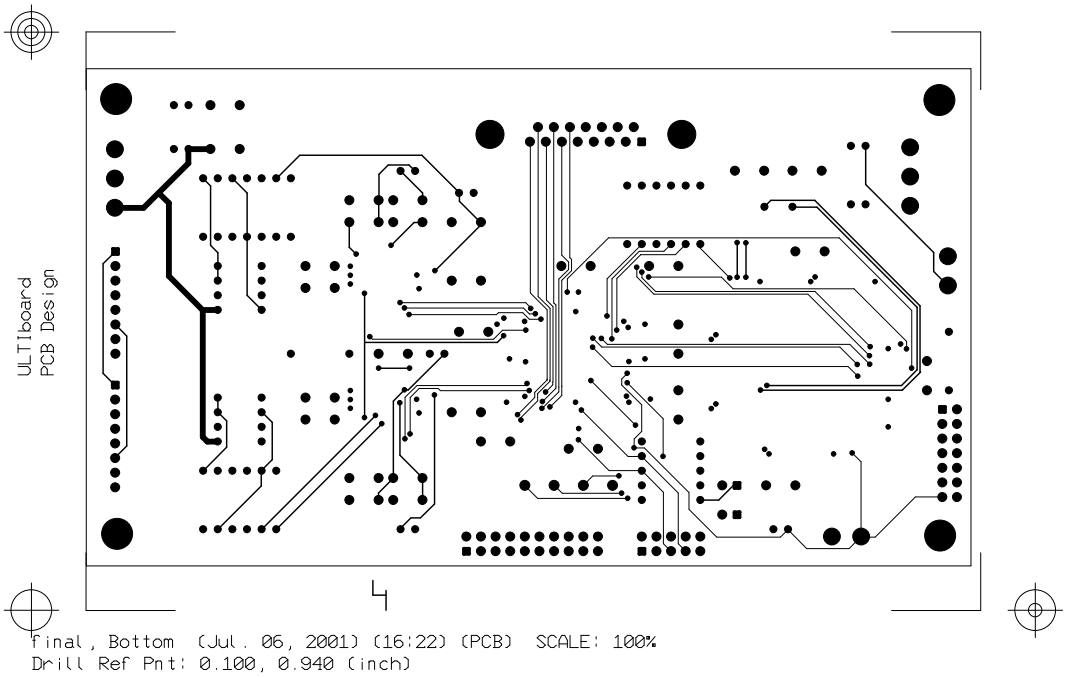


Figure A.6: PCB bottom layer

Appendix B

Matlab files for simulations

B.1 Motor model

In this section we present the motor model in the stationary frame. The function was coded in a format suitable to be used with the `ode` solvers. The constants are defined in the `constsl.m` or `constslq.m` files.

B.1.1 `stpmsm.m`

```
function xdot = stpmsm(t,x,flag,val,vbt,Tl)

% function xdot = stpmsm(t,x,flag,val,vbt,Tl)
%
% Stationary frame model of the PMSM, suited for an ODE solver.
%
% state vector:
%       [ialpha; ibeta; omega; theta]
%
% parameter inputs:
%       val: alpha axis voltage
%       vbt: beta axis voltage
%       Tl: load torque
%
% also uses constants defined in constsl.m

global pd6m pd12m pq0m pq6m pq12m
global Rsm L0m L1m
global Jm Bm P

ial = x(1);
ibt = x(2);
om = x(3);
```

```
th = x(4);

phid = pd6m*sin(6*th) + pd12m*sin(12*th);
phiq = pq0m + pq6m*cos(6*th) + pq12m*cos(12*th);
phial = phid*cos(th) - phiq*sin(th);
phibt = phid*sin(th) + phiq*cos(th);

Lialdot = val - Rsm*ial - om*(2*L1m*(-ial*sin(2*th)+ibt*cos(2*th))+phial);
Libtdot = vbt - Rsm*ibt - om*(2*L1m*(ial*cos(2*th)+ibt*sin(2*th))+phibt);

ialdot = (Lialdot*(L0m-L1m*cos(2*th))-L1m*Libtdot*sin(2*th)) / (L0m^2-L1m^2);
ibtdot = (Libtdot*(L0m+L1m*cos(2*th))-L1m*Lialdot*sin(2*th)) / (L0m^2-L1m^2);

taum = P*(L1m*((ibt^2-ial^2)*sin(2*th)+2*ial*ibt*cos(2*th))+ial*phial+ibt*phibt);

omdot = 1/Jm*(taum - Bm*om - Tl);

thdot = om;

xdot = [ialdot;ibtdot;omdot;thdot];
```

B.2 Floating-point version

The floating-point version of the algorithm uses one script file (`simsless.m`) to run the simulation, and in each PWM period it calls the sensorless algorithm `sless.m`. Constants are defined in `constsl.m`.

B.2.1 `simsless.m`

```
% simsless.m
%
% Simulation of the sensorless algorithm.

% load constants
clear
constsl;

% initialize memory values
ialfa0 = 0;
ibeta0 = 0;
ia = zeros(1,6);
ib = zeros(1,6);
t = ones(1,6)*TPWM/6;
tn = t;
pcnt = t/TCNT;
alhat = 0;
omhat = 0;
thhat = 0;
obserr = 0;
int1 = 0;
int2 = 0;
int3 = 0;

% initial conditions
ial0 = ialfa0;
ibt0 = ibeta0;
om0 = 0;
th0 = 0;
```

```
% repeat NSIM times:
%               run sless algorithm
%               run each subinterval of the PWM cycle on the PMSM model
%                       (including deadtime), collecting currents at each switching

t0 = 0;
y0 = [ial0;ibt0;om0;th0];
tout = t0;
yout = y0';
thhatout = th0;

for npwmint = 1:NSIM,

   % recall counter values computed on last PWM period
   cnt = pcnt;

   % run sensorless algorithm
   [pcnt,thtemp,valfa,vbeta] = sless(ia,ib,idref,wref);

   % first PWM subinterval
   tf = t0 + cnt(1)*TCNT;
   [ttemp,ytemp] = ode23('stpmsm',[t0 tf],y0,[],vals1,vbts1,Tl);
   nt = length(ttemp);
   tout = [tout; ttemp(2:nt)];
   yout = [yout; ytemp(2:nt,:)];
   thhatout = [thhatout; ones(nt-1,1)*thtemp];
   ia(1) = ytemp(nt,1);
   ib(1) = ytemp(nt,2);
   t0 = tf;
   y0 = ytemp(nt,:)';

   % second PWM subinterval
   tf = t0 + cnt(2)*TCNT;
   [ttemp,ytemp] = ode23('stpmsm',[t0 tf],y0,[],vals2,vbts2,Tl);
   nt = length(ttemp);
   tout = [tout; ttemp(2:nt)];
   yout = [yout; ytemp(2:nt,:)];
   thhatout = [thhatout; ones(nt-1,1)*thtemp];
   ia(2) = ytemp(nt,1);
   ib(2) = ytemp(nt,2);
   t0 = tf;
   y0 = ytemp(nt,:)';

   % third PWM subinterval
   tf = t0 + cnt(3)*TCNT;
   [ttemp,ytemp] = ode23('stpmsm',[t0 tf],y0,[],vals3,vbts3,Tl);
   nt = length(ttemp);
   tout = [tout; ttemp(2:nt)];
   yout = [yout; ytemp(2:nt,:)];
   thhatout = [thhatout; ones(nt-1,1)*thtemp];
   ia(3) = ytemp(nt,1);
   ib(3) = ytemp(nt,2);
   t0 = tf;
   y0 = ytemp(nt,:)';

   % fourth PWM subinterval
   tf = t0 + cnt(4)*TCNT;
   [ttemp,ytemp] = ode23('stpmsm',[t0 tf],y0,[],vals4,vbts4,Tl);
   nt = length(ttemp);
   tout = [tout; ttemp(2:nt)];
   yout = [yout; ytemp(2:nt,:)];
   thhatout = [thhatout; ones(nt-1,1)*thtemp];
   ia(4) = ytemp(nt,1);
   ib(4) = ytemp(nt,2);
   t0 = tf;
   y0 = ytemp(nt,:)';

   % fifth PWM subinterval
   tf = t0 + cnt(5)*TCNT;
   [ttemp,ytemp] = ode23('stpmsm',[t0 tf],y0,[],vals5,vbts5,Tl);
   nt = length(ttemp);
   tout = [tout; ttemp(2:nt)];
   yout = [yout; ytemp(2:nt,:)];
   thhatout = [thhatout; ones(nt-1,1)*thtemp];
   ia(5) = ytemp(nt,1);
   ib(5) = ytemp(nt,2);
```

```
   t0 = tf;
   y0 = ytemp(nt,:)';

   % sixth PWM subinterval
   tf = t0 + cnt(6)*TCNT;
   [ttemp,ytemp] = ode23('stpmsm',[t0 tf],y0,[],vals6,vbts6,Tl);
   nt = length(ttemp);
   tout = [tout; ttemp(2:nt)];
   yout = [yout; ytemp(2:nt,:)];
   thhatout = [thhatout; ones(nt-1,1)*thtemp];
   ia(6) = ytemp(nt,1);
   ib(6) = ytemp(nt,2);
   t0 = tf;
   y0 = ytemp(nt,:)';

end
```

B.2.2 constsl.m

```
% constsl.m
%
% Numeric constants for the sensorless algorithm.

global Wpit TPWM TCNT CPTPWM GAMMA1 GAMMA2 GAMMA3
global k1 a1 k2 a2 k3 a3 ilim vlim
global CN1OSR3 SR3O2 CN1O6 SR3O12E CN1O12E
global ialfa0 ibeta0 t tn alhat omhat thhat obserr int1 int2 int3

% simulation parameters

NSIM = 4000;    % number of PWM cycles to simulate
TPWM = 2e-4;
wref = 500;
idref = 0;
Tl = 1;

% PWM counter parameters

TCNT = 1e-7;
CPTPWM = TPWM/TCNT;

% inverter parameters

VBUS = 200;
EMAX = VBUS/sqrt(2);
TDEAD = 2e-6;
CPTDEAD = TDEAD/TCNT;
vals1 = 2/sqrt(6)*VBUS;
vbts1 = 0;
vals2 = 1/sqrt(6)*VBUS;
vbts2 = 1/sqrt(2)*VBUS;
vals3 = -1/sqrt(6)*VBUS;
vbts3 = 1/sqrt(2)*VBUS;
vals4 = -2/sqrt(6)*VBUS;
vbts4 = 0;
vals5 = -1/sqrt(6)*VBUS;
vbts5 = -1/sqrt(2)*VBUS;
vals6 = 1/sqrt(6)*VBUS;
vbts6 = -1/sqrt(2)*VBUS;

% general numeric constants

SR3OSR2 = sqrt(3/2);
SR3O2 = sqrt(3)/2;
CN1O6 = 1/6;
CN1OSR2 = 1/sqrt(2);
CN1OSR3 = 1/sqrt(3);
CN1OSR6 = 1/sqrt(6);
SR3O12E = sqrt(3)/12/EMAX;
CN1O12E = 1/12/EMAX;

% motor parameters
```

```
global pd6m pd12m pq0m pq6m pq12m
global Rsm L0m L1m
global Jm Bm P

P=4;

% Mechanical ( !! normalised with P !! ) constants:
Jm=0.5e-3;
Bm=0.5e-3;

% Motor electrical parameters:
Rsm=0.97;
L0m=7.2e-3;
L1m=-1.8e-3;
Ldm=L0m+L1m;
Lqm=L0m-L1m;

% Constant in q2 and q3 parameters (used in observer error generation)
iL1m=-L1m/(L0m^2-L1m^2);

pd6m=0.001;
pd12m=0.0008;
pq0m=0.1;
pq6m=0.005;
pq12m=0.0008;

% matrix of precomputed voltages

Wpit = [
   -sqrt(6)/32/VBUS     -15*sqrt(6)/320/VBUS     45*sqrt(2)/320/VBUS;
    5*sqrt(2)/32/VBUS   -53*sqrt(2)/320/VBUS    -11*sqrt(6)/320/VBUS;
   -3*sqrt(6)/32/VBUS   -29*sqrt(6)/320/VBUS    -9*sqrt(2)/320/VBUS;
   -sqrt(2)/32/VBUS      sqrt(2)/320/VBUS       -33*sqrt(6)/320/VBUS;
   -2*sqrt(6)/32/VBUS   -14*sqrt(6)/320/VBUS    -54*sqrt(2)/320/VBUS;
   -6*sqrt(2)/32/VBUS    54*sqrt(2)/320/VBUS    -22*sqrt(6)/320/VBUS;
    sqrt(6)/32/VBUS       15*sqrt(6)/320/VBUS   -45*sqrt(2)/320/VBUS;
   -5*sqrt(2)/32/VBUS    53*sqrt(2)/320/VBUS     11*sqrt(6)/320/VBUS;
    3*sqrt(6)/32/VBUS    29*sqrt(6)/320/VBUS     9*sqrt(2)/320/VBUS;
    sqrt(2)/32/VBUS      -sqrt(2)/320/VBUS       33*sqrt(6)/320/VBUS];


%%% Observer constants
L0=7.2e-3;
L1=-1.8e-3;

% Constant in q2 and q3 parameters
iL1=-L1/(L0^2-L1^2);

% Desired observer poles
r1=200; r2=300; w1=100;
p1=-r1+i*w1; p2=-r2; p3=-r1-i*w1;

% Observer constants
GAMMA1 = -real(p1+p2+p3)/2/iL1;
GAMMA2 = real(p1*p2+p1*p3+p2*p3)/2/iL1;
GAMMA3 = -real(p1*p2*p3)/2/iL1;

clear L0 L1 iL1 r1 r2 w1 p1 p2 p3

%%% Controller constants

% Current and voltage limits
ilim=12;
vlim=sqrt(3)/2*VBUS/sqrt(2);

% PI controller constants
r1=15*(1+0.5*i); r2=15*(1-0.5*i);
k2=(Jm*(r1+r2)-Bm)/P/pq0m;
a2=r1*r2/(r1+r2-Bm/Jm);
r1=500; r2=500;
k1=Ldm*(r1+r2)-Rsm;
a1=r1*r2/(r1+r2-Rsm/Ldm);
r1=200; r2=200;
k3=Lqm*(r1+r2)-Rsm;
a3=r1*r2/(r1+r2-Rsm/Lqm);
```

```
clear r1 r2
```

B.2.3 sless.m

```
function [cnt,thhat,valfa,vbeta] = sless(ia,ib,idref,wref)

% function [cnt,thhat] = sless(ia,ib,idref,wref)
%
% Performs one round of the sensorless observer and control algorithm.
%
% inputs:
%                       ia[1..6]:       current measurements
%                       ib[1..6]:       idem
%                       idref, wref:    reference values for the controller
%
% memory:
%                       ialfa0: last current measurement from previous period
%                       ibeta0: idem
%                       t[1..6]:        time intervals computed two periods before (and used now)
%                       tn[1..6]:       time intervals computed last period (to be used next period)
%                       alhat, omhat, thhat, obserr:    observer variables
%                       int1, int2, int3:       integrators for the PI blocks
%
% outputs:
%                       cnt[1..6]:      count values for the PWM generation
%                       thhat:  estimated position
%
% ge, March 2001

% ------------------------------------------------
% Numeric constants loaded previously from constsl.m
% ------------------------------------------------
global Wpit TPWM TCNT CPTPWM GAMMA1 GAMMA2 GAMMA3
global k1 a1 k2 a2 k3 a3 ilim vlim
global CN1OSR3 SR3O2 CN1O6 SR3O12E CN1O12E
global ialfa0 ibeta0 t tn alhat omhat thhat obserr int1 int2 int3

% ------------------------------------------------
% Observer
% ------------------------------------------------

% scale current?

% convert currents to alfa-beta frame
for i=1:6,
   % modified for simulation only!!!!
        % ialfa(i) = SR3OSR2*ia(i);
        % ibeta(i) = CN1OSR2*(2*ib(i)+ia(i));
        ialfa(i) = ia(i);
        ibeta(i) = ib(i);
end

% construct vector x
x(1) = (ialfa(2)-ialfa(1))/t(2) - (ialfa(1)-ialfa0)/t(1);
x(2) = (ibeta(2)-ibeta(1))/t(2) - (ibeta(1)-ibeta0)/t(1);
x(3) = (ialfa(3)-ialfa(2))/t(3) - (ialfa(2)-ialfa(1))/t(2);
x(4) = (ibeta(3)-ibeta(2))/t(3) - (ibeta(2)-ibeta(1))/t(2);
x(5) = (ialfa(4)-ialfa(3))/t(4) - (ialfa(3)-ialfa(2))/t(3);
x(6) = (ibeta(4)-ibeta(3))/t(4) - (ibeta(3)-ibeta(2))/t(3);
x(7) = (ialfa(5)-ialfa(4))/t(5) - (ialfa(4)-ialfa(3))/t(4);
x(8) = (ibeta(5)-ibeta(4))/t(5) - (ibeta(4)-ibeta(3))/t(4);
x(9) = (ialfa(6)-ialfa(5))/t(6) - (ialfa(5)-ialfa(4))/t(5);
x(10) = (ibeta(6)-ibeta(5))/t(6) - (ibeta(5)-ibeta(4))/t(5);

% find q = Wpi*x = Wpit'*x
for j=1:3,
   q(j) = 0;
   for i=1:10,
      q(j) = q(j) + Wpit(i,j)*x(i);
   end
```

```
end

% observer update
alhat = alhat + TPWM*GAMMA3*obserr;
omhat = omhat + TPWM*(alhat+GAMMA2*obserr);
thhat = thhat + TPWM*(omhat+GAMMA1*obserr);

% error update
obserr = q(3)*cos(2*thhat) - q(2)*sin(2*thhat);

% modulus
thhat = mod(thhat,2*pi);

% ------------------------------------------------
% Controller
% ------------------------------------------------

% average current measurements
ialav = (ialfa0+ialfa(1))*t(1);
ibtav = (ibeta0+ibeta(1))*t(1);
for i=2:6,
   ialav = ialav + (ialfa(i-1)+ialfa(i))*t(i);
   ibtav = ibtav + (ibeta(i-1)+ibeta(i))*t(i);
end
ialav = ialav/2/TPWM;
ibtav = ibtav/2/TPWM;

% convert to d-q frame
sinth = sin(thhat);
costh = cos(thhat);
id = ialav*costh + ibtav*sinth;
iq = -ialav*sinth + ibtav*costh;

% id control block (PI1)
u1 = idref-id;
int1 = int1 + u1*TPWM;
if (abs(int1)>vlim),
   int1 = sign(int1)*vlim;
end
vdc = k1*(u1+a1*int1);

% w control block (PI2)
u2 = wref-omhat;
int2 = int2 + u2*TPWM;
if (abs(int2)>ilim),
   int2 = sign(int2)*ilim;
end
iqref = k2*(u2+a2*int2);
if (abs(iqref)>ilim),
   iqref = sign(iqref)*ilim;
end

% iq control block (PI3)
u3 = iqref-iq;
int3 = int3 + u3*TPWM;
if (abs(int3)>vlim),
   int3 = sign(int3)*vlim;
end
vqc = k3*(u3+a3*int3);

% INSERT VOLTAGES HERE FOR OPEN-LOOP
% vdc = -.0137;
% vqc = 2.2142;

% convert back to alfa-beta frame
sinth = sin(thhat+2*omhat*TPWM);
costh = cos(thhat+2*omhat*TPWM);
valfa = vdc*costh - vqc*sinth;
vbeta = vdc*sinth + vqc*costh;

% find sector and rotate vector to sector 1
if (vbeta>0),
   if (valfa>0),
      if (CN1OSR3*vbeta<valfa),
         sector = 1;
         valr = valfa;
```

```
         vbtr = vbeta;
      else
         sector = 2;
         valr = valfa*.5 + vbeta*SR3O2;
         vbtr = -valfa*SR3O2 + vbeta*.5;
      end
   else
      if (CN1OSR3*vbeta<-valfa),
         sector = 3;
         valr = -valfa*.5 + vbeta*SR3O2;
         vbtr = -valfa*SR3O2 - vbeta*.5;
      else
         sector = 2;
         valr = valfa*.5 + vbeta*SR3O2;
         vbtr = -valfa*SR3O2 + vbeta*.5;
      end
   end
else
   if (valfa>0),
      if (CN1OSR3*vbeta>-valfa),
         sector = 6;
         valr = valfa*.5 - vbeta*SR3O2;
         vbtr = valfa*SR3O2 + vbeta*.5;
      else
         sector = 5;
         valr = -valfa*.5 - vbeta*SR3O2;
         vbtr = valfa*SR3O2 - vbeta*.5;
      end
   else
      if (CN1OSR3*vbeta>valfa),
         sector = 4;
         valr = -valfa;
         vbtr = -vbeta;
      else
         sector = 5;
         valr = -valfa*.5 - vbeta*SR3O2;
         vbtr = valfa*SR3O2 - vbeta*.5;
      end
   end
end

% no limit of voltage magnitude

% find counter values
i = sector;
cnt(i) = (CN1O6 + 5*SR3O12E*valr - 7*CN1O12E*vbtr)*CPTPWM;
if (i==6),
   i = 1;
else
   i = i+1;
end
cnt(i) = (CN1O6 - SR3O12E*valr + 11*CN1O12E*vbtr)*CPTPWM;
if (i==6),
   i = 1;
else
   i = i+1;
end
cnt(i) = (CN1O6 - SR3O12E*valr - CN1O12E*vbtr)*CPTPWM;
if (i==6),
   i = 1;
else
   i = i+1;
end
cnt(i) = (CN1O6 - SR3O12E*valr - CN1O12E*vbtr)*CPTPWM;
if (i==6),
   i = 1;
else
   i = i+1;
end
cnt(i) = (CN1O6 - SR3O12E*valr - CN1O12E*vbtr)*CPTPWM;
if (i==6),
   i = 1;
else
   i = i+1;
end
cnt(i) = (CN1O6 - SR3O12E*valr - CN1O12E*vbtr)*CPTPWM;
```

```
% ------------------------------------------------
% Memory
% ------------------------------------------------

% save last current measurement
ialfa0 = ialfa(6);
ibeta0 = ibeta(6);

% save time intervals for next period
for i=1:6,
   t(i) = tn(i);
end
% load tn
for i=1:6,
   tn(i) = cnt(i)*TCNT;
end
```

B.3 Fixed-point version

The fixed-point version of the algorithm uses the same structure of the floating-point, only that all constants are quantized using the function `frac16.m`, and the operations are simulated using the functions `alu.m` and `mac.m`. The script is called `simslq.m`, the constants are defined in `constslq.m` and the sensorless algorithm is coded in `slq.m`.

B.3.1 `simslq.m`

```
% simslq.m
%
% Simulation of the quantized sensorless algorithm.

% load constants
clear
constslq;

% initialize memory values
ialfa0 = 0;
ibeta0 = 0;
ia = zeros(1,6);
ib = zeros(1,6);
t = ones(1,6)*CPTPWM/6;
tn = t;
pcnt=t;
alhat = 0;
omhat = 0;
thhat = 0;
obserr = 0;
int1 = 0;
int2 = 0;
int3 = 0;

% initial conditions
ial0 = ialfa0;
```

```
ibt0 = ibeta0;
om0 = 0;
th0 = 0;

% repeat NSIM times:
%               run sless algorithm
%               run each subinterval of the PWM cycle on the PMSM model
%                       collecting currents at each switching

t0 = 0;
y0 = [ial0;ibt0;om0;th0];
tout = t0;
yout = y0';
thhatout = th0;
omhatout = om0;

for npwmint = 1:NSIM,

   % recall counter values computed on last PWM period
   cnt = pcnt;

   % run sensorless algorithm
   [pcnt,thtemp,omtemp,valfa,vbeta,ialav,ibtav] = slq(ia,ib,idref,wref);

   % first PWM subinterval
   tf = t0 + cnt(1)*TCNT;
   [ttemp,ytemp] = ode23('stpmsm',[t0 tf],y0,[],vals1,vbts1,Tl);
   nt = length(ttemp);
   tout = [tout; ttemp(2:nt)];
   yout = [yout; ytemp(2:nt,:)];
   thhatout = [thhatout; ones(nt-1,1)*thtemp];
   ia(1) = sqrt(2/3)*ytemp(nt,1);
   ib(1) = ytemp(nt,2)/sqrt(2)-ytemp(nt,1)/sqrt(6);
   t0 = tf;
   y0 = ytemp(nt,:)';

   % second PWM subinterval
   tf = t0 + cnt(2)*TCNT;
   [ttemp,ytemp] = ode23('stpmsm',[t0 tf],y0,[],vals2,vbts2,Tl);
   nt = length(ttemp);
   tout = [tout; ttemp(2:nt)];
   yout = [yout; ytemp(2:nt,:)];
   thhatout = [thhatout; ones(nt-1,1)*thtemp];
   ia(2) = sqrt(2/3)*ytemp(nt,1);
   ib(2) = ytemp(nt,2)/sqrt(2)-ytemp(nt,1)/sqrt(6);
   t0 = tf;
   y0 = ytemp(nt,:)';

   % third PWM subinterval
   tf = t0 + cnt(3)*TCNT;
   [ttemp,ytemp] = ode23('stpmsm',[t0 tf],y0,[],vals3,vbts3,Tl);
   nt = length(ttemp);
   tout = [tout; ttemp(2:nt)];
   yout = [yout; ytemp(2:nt,:)];
   thhatout = [thhatout; ones(nt-1,1)*thtemp];
   ia(3) = sqrt(2/3)*ytemp(nt,1);
   ib(3) = ytemp(nt,2)/sqrt(2)-ytemp(nt,1)/sqrt(6);
   t0 = tf;
   y0 = ytemp(nt,:)';

   % fourth PWM subinterval
   tf = t0 + cnt(4)*TCNT;
   [ttemp,ytemp] = ode23('stpmsm',[t0 tf],y0,[],vals4,vbts4,Tl);
   nt = length(ttemp);
   tout = [tout; ttemp(2:nt)];
   yout = [yout; ytemp(2:nt,:)];
   thhatout = [thhatout; ones(nt-1,1)*thtemp];
   ia(4) = sqrt(2/3)*ytemp(nt,1);
   ib(4) = ytemp(nt,2)/sqrt(2)-ytemp(nt,1)/sqrt(6);
   t0 = tf;
   y0 = ytemp(nt,:)';

   % fifth PWM subinterval
   tf = t0 + cnt(5)*TCNT;
   [ttemp,ytemp] = ode23('stpmsm',[t0 tf],y0,[],vals5,vbts5,Tl);
   nt = length(ttemp);
```

```
   tout = [tout; ttemp(2:nt)];
   yout = [yout; ytemp(2:nt,:)];
   thhatout = [thhatout; ones(nt-1,1)*thtemp];
   ia(5) = sqrt(2/3)*ytemp(nt,1);
   ib(5) = ytemp(nt,2)/sqrt(2)-ytemp(nt,1)/sqrt(6);
   t0 = tf;
   y0 = ytemp(nt,:)';

   % sixth PWM subinterval
   tf = t0 + cnt(6)*TCNT;
   [ttemp,ytemp] = ode23('stpmsm',[t0 tf],y0,[],vals6,vbts6,Tl);
   nt = length(ttemp);
   tout = [tout; ttemp(2:nt)];
   yout = [yout; ytemp(2:nt,:)];
   thhatout = [thhatout; ones(nt-1,1)*thtemp];
   ia(6) = sqrt(2/3)*ytemp(nt,1);
   ib(6) = ytemp(nt,2)/sqrt(2)-ytemp(nt,1)/sqrt(6);
   t0 = tf;
   y0 = ytemp(nt,:)';

end
```

B.3.2 constslq.m

```
% constslq.m
%
% Numeric constants for the quantized sensorless algorithm.

global Wpit TPWM TCNT CPTPWM IABMAX
global ialfa0 ibeta0 t tn alhat omhat thhat obserr int1 int2 int3
global SR3OSR2O2 SR2O2 CN1O2OSR2 CN1OSR3 SR3O2 CN2SR2O4 CN1O6 CN1OSR6 SR2O6

% simulation parameters

NSIM = 2000;    % number of PWM cycles to simulate
TPWM = 2e-4;
TCNT = 1e-7;
CPTPWM = TPWM/TCNT;

wref = frac16(500/1800);
idref = frac16(0/2/TCNT/12.5*TPWM*2^-15);
Tl = 1;

% inverter parameters

VBUS = 200;
EMAX = VBUS/sqrt(2);
TDEAD = 2e-6;
CPTDEAD = TDEAD/TCNT;
vals1 = 2/sqrt(6)*VBUS;
vbts1 = 0;
vals2 = 1/sqrt(6)*VBUS;
vbts2 = 1/sqrt(2)*VBUS;
vals3 = -1/sqrt(6)*VBUS;
vbts3 = 1/sqrt(2)*VBUS;
vals4 = -2/sqrt(6)*VBUS;
vbts4 = 0;
vals5 = -1/sqrt(6)*VBUS;
vbts5 = -1/sqrt(2)*VBUS;
vals6 = 1/sqrt(6)*VBUS;
vbts6 = -1/sqrt(2)*VBUS;

% scale factors

IABMAX = 25;

% general numeric constants (quantized)

SR3OSR2O2 = frac16(sqrt(3/2)/2);
SR2O2 = frac16(sqrt(2)/2);
CN1O2OSR2 = frac16(1/2/sqrt(2));
CN1OSR3 = frac16(1/sqrt(3));
```

```
SR3O2 = frac16(sqrt(3)/2);
CN2SR2O4 = frac16(2*sqrt(2)/4);
CN1O6 = frac16(1/6);
CN1OSR6 = frac16(1/sqrt(6));
SR2O6 = frac16(sqrt(2)/6);

% general numeric constants

SR3OSR2 = sqrt(3/2);
CN1OSR2 = 1/sqrt(2);
CN1OSR6 = 1/sqrt(6);
SR3O12E = sqrt(3)/12/EMAX;
CN1O12E = 1/12/EMAX;

% motor parameters

global pd6m pd12m pq0m pq6m pq12m
global Rsm L0m L1m
global Jm Bm P

P=4;

% Mechanical ( !! normalised with P !! ) constants:
Jm=0.5e-3;
Bm=0.5e-3;

% Motor electrical parameters:
Rsm=0.97;
L0m=7.2e-3;
L1m=-1.8e-3;
Ldm=L0m+L1m;
Lqm=L0m-L1m;

% Constant in q2 and q3 parameters (used in observer error generation)
iL1m=-L1m/(L0m^2-L1m^2);

pd6m=0.001;
pd12m=0.0008;
pq0m=0.1;
pq6m=0.005;
pq12m=0.0008;

% matrix of precomputed voltages, scaled and quantized

Wpit = [
   -sqrt(6)/32/VBUS     -15*sqrt(6)/320/VBUS     45*sqrt(2)/320/VBUS;
    5*sqrt(2)/32/VBUS   -53*sqrt(2)/320/VBUS    -11*sqrt(6)/320/VBUS;
   -3*sqrt(6)/32/VBUS   -29*sqrt(6)/320/VBUS    -9*sqrt(2)/320/VBUS;
   -sqrt(2)/32/VBUS      sqrt(2)/320/VBUS       -33*sqrt(6)/320/VBUS;
   -2*sqrt(6)/32/VBUS   -14*sqrt(6)/320/VBUS    -54*sqrt(2)/320/VBUS;
   -6*sqrt(2)/32/VBUS    54*sqrt(2)/320/VBUS    -22*sqrt(6)/320/VBUS;
    sqrt(6)/32/VBUS      15*sqrt(6)/320/VBUS    -45*sqrt(2)/320/VBUS;
   -5*sqrt(2)/32/VBUS    53*sqrt(2)/320/VBUS     11*sqrt(6)/320/VBUS;
    3*sqrt(6)/32/VBUS    29*sqrt(6)/320/VBUS     9*sqrt(2)/320/VBUS;
    sqrt(2)/32/VBUS      -sqrt(2)/320/VBUS       33*sqrt(6)/320/VBUS];

Wpit = 2*1.22e5/324 * Wpit;

for j=1:3,
   for i=1:10,
      Wpit(i,j)=frac16(Wpit(i,j));
   end
end

clear i j

% observer coefficients

global GAMMA1SQ GAMMA2SQ GAMMA3SQ ACCOEFF SPCOEFF

%%% Observer constants
L0=7.2e-3;
L1=-1.8e-3;

% Constant in q2 and q3 parameters
iL1=-L1/(L0^2-L1^2);
```

```
% Desired observer poles
r1=200; r2=300; w1=100;
p1=-r1+i*w1; p2=-r2; p3=-r1-i*w1;

% Observer constants
GAMMA1 = -real(p1+p2+p3)/2/iL1;
GAMMA2 = real(p1*p2+p1*p3+p2*p3)/2/iL1;
GAMMA3 = -real(p1*p2*p3)/2/iL1;

clear L0 L1 iL1 r1 r2 w1 p1 p2 p3

GAMMA1SQ = frac16(TPWM*GAMMA1*324/2/pi);
GAMMA2SQ = frac16(TPWM*GAMMA2*324/2/1800);
GAMMA3SQ = frac16(TPWM*GAMMA3*324/2/10000);

ACCOEFF = frac16(TPWM*10000/1800);
SPCOEFF = frac16(TPWM*1800/pi);

%%% Controller constants

% Current and voltage limits
ilim=12;
vlim=sqrt(3)/2*VBUS/sqrt(2);

% PI controller constants
r1=15*(1+0.5*i); r2=15*(1-0.5*i);
k2=(Jm*(r1+r2)-Bm)/P/pq0m;
a2=r1*r2/(r1+r2-Bm/Jm);
r1=500; r2=500;
k1=Ldm*(r1+r2)-Rsm;
a1=r1*r2/(r1+r2-Rsm/Ldm);
r1=200; r2=200;
k3=Lqm*(r1+r2)-Rsm;
a3=r1*r2/(r1+r2-Rsm/Lqm);

clear r1 r2

global ERR1COEFFINT ERR1COEFFOUT INT1COEFF
global ERR2COEFFINT ERR2COEFFOUT INT2COEFF
global ERR3COEFFINT ERR3COEFFOUT INT3COEFF

ERR1COEFFINT = frac16(.13824);
ERR1COEFFOUT = frac16(.2682);
INT1COEFF = 1;

ERR2COEFFINT = frac16(2.468e-3);
ERR2COEFFOUT = frac16(.3182);
INT2COEFF = 0.25;

ERR3COEFFINT = frac16(.0369);
ERR3COEFFOUT = frac16(.3466);
INT3COEFF = 1;

global ILIM DLYCOEFF
ILIM=frac16(12/204.8);
DLYCOEFF=frac16(.2292);
```

B.3.3 slq.m

```
function [cnt,thhat,omhat,valfa,vbeta,ialav,ibtav] = slq(ia,ib,idref,wref)

% function [cnt,thhat] = slq(ia,ib,idref,wref)
%
% Performs one round of the sensorless observer and control algorithm
% with quantized coefficients.
%
% inputs:
%                       ia[1..6]:       current measurements
%                       ib[1..6]:       idem
%                       idref, wref:    reference values for the controller
%
```

```
% memory:
%                       ialfa0: last current measurement from previous period
%                       ibeta0: idem
%                       t[1..6]:        time intervals computed two periods before (and used now)
%                       tn[1..6]:       time intervals computed last period (to be used next period)
%                       alhat, omhat, thhat, obserr:    observer variables
%                       int1, int2, int3:       integrators for the PI blocks
%
% outputs:
%                       cnt[1..6]:      count values for the PWM generation
%                       thhat:  estimated position
%
% ge, March 2001

% ------------------------------------------------
% Numeric constants loaded previously from constslq.m
% ------------------------------------------------
global Wpit TPWM TCNT CPTPWM IABMAX
global CN1OSR3 SR3O2 CN106 SR3O12E CN1O12E
global ialfa0 ibeta0 t tn alhat omhat thhat obserr int1 int2 int3
global SR3OSR2O2 SR2O2 CN1O2OSR2 CN1OSR3 SR3O2 CN2SR2O4 CN106 CN1OSR6 SR2O6
global GAMMA1SQ GAMMA2SQ GAMMA3SQ ACCOEFF SPCOEFF
global ERR1COEFFINT ERR1COEFFOUT INT1COEFF
global ERR2COEFFINT ERR2COEFFOUT INT2COEFF
global ERR3COEFFINT ERR3COEFFOUT INT3COEFF
global ILIM DLYCOEFF

% ------------------------------------------------
% Observer
% ------------------------------------------------

% scale current as will be the input to the DSP
ia = ia/IABMAX;
ib = ib/IABMAX;

% convert currents to alfa-beta frame
for i=1:6,
        ialfa(i) = SR3OSR2O2*ia(i);
        ibeta(i) = SR2O2*ib(i) + CN1O2OSR2*ia(i);
end

% difference of currents
dialfa(1) = alu('-',ialfa(1),ialfa0);
dibeta(1) = alu('-',ibeta(1),ibeta0);
dialfa(2) = alu('-',ialfa(2),ialfa(1));
dibeta(2) = alu('-',ibeta(2),ibeta(1));
dialfa(3) = alu('-',ialfa(3),ialfa(2));
dibeta(3) = alu('-',ibeta(3),ibeta(2));
dialfa(4) = alu('-',ialfa(4),ialfa(3));
dibeta(4) = alu('-',ibeta(4),ibeta(3));
dialfa(5) = alu('-',ialfa(5),ialfa(4));
dibeta(5) = alu('-',ibeta(5),ibeta(4));
dialfa(6) = alu('-',ialfa(6),ialfa(5));
dibeta(6) = alu('-',ibeta(6),ibeta(5));

% division by time
dialdt(1) = div(dialfa(1)*2^12,t(1),16,16);
dibtdt(1) = div(dibeta(1)*2^12,t(1),16,16);
dialdt(2) = div(dialfa(2)*2^12,t(2),16,16);
dibtdt(2) = div(dibeta(2)*2^12,t(2),16,16);
dialdt(3) = div(dialfa(3)*2^12,t(3),16,16);
dibtdt(3) = div(dibeta(3)*2^12,t(3),16,16);
dialdt(4) = div(dialfa(4)*2^12,t(4),16,16);
dibtdt(4) = div(dibeta(4)*2^12,t(4),16,16);
dialdt(5) = div(dialfa(5)*2^12,t(5),16,16);
dibtdt(5) = div(dibeta(5)*2^12,t(5),16,16);
dialdt(6) = div(dialfa(6)*2^12,t(6),16,16);
dibtdt(6) = div(dibeta(6)*2^12,t(6),16,16);

% construct vector x
x(1) = alu('-',frac16(dialdt(2)/2),frac16(dialdt(1)/2));
x(2) = alu('-',frac16(dibtdt(2)/2),frac16(dibtdt(1)/2));
x(3) = alu('-',frac16(dialdt(3)/2),frac16(dialdt(2)/2));
x(4) = alu('-',frac16(dibtdt(3)/2),frac16(dibtdt(2)/2));
x(5) = alu('-',frac16(dialdt(4)/2),frac16(dialdt(3)/2));
x(6) = alu('-',frac16(dibtdt(4)/2),frac16(dibtdt(3)/2));
```

```
x(7) = alu('-',frac16(dialdt(5)/2),frac16(dialdt(4)/2));
x(8) = alu('-',frac16(dibtdt(5)/2),frac16(dibtdt(4)/2));
x(9) = alu('-',frac16(dialdt(6)/2),frac16(dialdt(5)/2));
x(10) = alu('-',frac16(dibtdt(6)/2),frac16(dibtdt(5)/2));

% find q = Wpi*x = Wpit'*x
for j=2:3, % was 1:3 but q(1) is never used
   q(j) = 0;
   for i=1:10,
      q(j) = mac(q(j),Wpit(i,j),x(i));
   end
end

% observer update
alhat = frac16(mac(alhat,GAMMA3SQ,obserr));
omhat = mac(omhat,ACCOEFF,alhat);
omhat = frac16(mac(omhat,GAMMA2SQ,obserr));
thhat = mac(thhat,SPCOEFF,omhat);
thhat = mac(thhat,GAMMA1SQ,obserr);
% theta hat wrap around
if thhat>1
   thhat = thhat-2;
elseif thhat<-1
   thhat = thhat+2;
end

% error update
cos2thhat = frac16(cos(2*thhat*pi));
sin2thhat = frac16(sin(2*thhat*pi));

obserr = mac(0,frac16(2*q(3)),cos2thhat);
obserr = mac(obserr,-frac16(2*q(2)),sin2thhat);

% ------------------------------------------------
% Controller
% ------------------------------------------------

% average current measurements
ialadd(1) = alu('+',frac16(ialfa0/2),frac16(ialfa(1)/2));
ibtadd(1) = alu('+',frac16(ibeta0/2),frac16(ibeta(1)/2));
ialadd(2) = alu('+',frac16(ialfa(1)/2),frac16(ialfa(2)/2));
ibtadd(2) = alu('+',frac16(ibeta(1)/2),frac16(ibeta(2)/2));
ialadd(3) = alu('+',frac16(ialfa(2)/2),frac16(ialfa(3)/2));
ibtadd(3) = alu('+',frac16(ibeta(2)/2),frac16(ibeta(3)/2));
ialadd(4) = alu('+',frac16(ialfa(3)/2),frac16(ialfa(4)/2));
ibtadd(4) = alu('+',frac16(ibeta(3)/2),frac16(ibeta(4)/2));
ialadd(5) = alu('+',frac16(ialfa(4)/2),frac16(ialfa(5)/2));
ibtadd(5) = alu('+',frac16(ibeta(4)/2),frac16(ibeta(5)/2));
ialadd(6) = alu('+',frac16(ialfa(5)/2),frac16(ialfa(6)/2));
ibtadd(6) = alu('+',frac16(ibeta(5)/2),frac16(ibeta(6)/2));

ialav = mac(0,ialadd(1),t(1)/2^15);
ialav = mac(ialav,ialadd(2),t(2)/2^15);
ialav = mac(ialav,ialadd(3),t(3)/2^15);
ialav = mac(ialav,ialadd(4),t(4)/2^15);
ialav = mac(ialav,ialadd(5),t(5)/2^15);
ialav = mac(ialav,ialadd(6),t(6)/2^15);

ibtav = mac(0,ibtadd(1),t(1)/2^15);
ibtav = mac(ibtav,ibtadd(2),t(2)/2^15);
ibtav = mac(ibtav,ibtadd(3),t(3)/2^15);
ibtav = mac(ibtav,ibtadd(4),t(4)/2^15);
ibtav = mac(ibtav,ibtadd(5),t(5)/2^15);
ibtav = mac(ibtav,ibtadd(6),t(6)/2^15);

% multiply by 4
ialav = 4*ialav;
ibtav = 4*ibtav;

% convert to d-q frame
sinth = frac16(sin(thhat*pi));
costh = frac16(cos(thhat*pi));

id = mac(0,frac16(ialav),costh);
id = frac16(mac(id,frac16(ibtav),sinth));
iq = mac(0,-frac16(ialav),sinth);
```

```
iq = frac16(mac(iq,frac16(ibtav),costh));

% id control block (PI1)
u1 = alu('-',idref,id);
int1 = frac16(mac(int1,u1,ERR1COEFFINT));
if (abs(int1)>1),
   int1 = sign(int1);
end
vdc = mac(0,u1,ERR1COEFFOUT);
vdc = vdc+2*u1+int1;

% w control block (PI2)
u2 = alu('-',wref,omhat);
int2 = frac16(mac(int2,u2,ERR2COEFFINT));
if (abs(int2)>1),
   int2 = sign(int2);
end
iqref = mac(0,u2,ERR2COEFFOUT);
iqref = mac(iqref,int2,INT2COEFF);
if (abs(iqref)>ILIM),
   iqref = sign(iqref)*ILIM;
end

% iq control block (PI3)
u3 = alu('-',iqref,iq);
int3 = frac16(mac(int3,u3,ERR3COEFFINT));
if (abs(int3)>1),
   int1 = sign(int3);
end
vqc = mac(u3,u3,ERR3COEFFOUT);
vqc = alu('+',vqc,int3);

% convert back to alfa-beta frame
thhat2 = thhat + DLYCOEFF*omhat;
sinth = frac16(sin(thhat2*pi));
costh = frac16(cos(thhat2*pi));

valfa = mac(0,vdc,costh);
valfa = frac16(mac(valfa,-vqc,sinth));
vbeta = mac(0,vdc,sinth);
vbeta = frac16(mac(vbeta,vqc,costh));

% find sector and rotate vector to sector 1
if (vbeta>0),
   if (valfa>0),
      if (CN1OSR3*vbeta<valfa),
         sector = 1;
         valr = valfa;
         vbtr = vbeta;
      else
         sector = 2;
         valr = frac16(valfa/2);
         valr = frac16(mac(valr,vbeta,SR3O2));
         vbtr = frac16(vbeta/2);
         vbtr = frac16(mac(vbtr,-valfa,SR3O2));
      end
   else
      if (CN1OSR3*vbeta<-valfa),
         sector = 3;
         valr = -frac16(valfa/2);
         valr = frac16(mac(valr,vbeta,SR3O2));
         vbtr = -frac16(vbeta/2);
         vbtr = frac16(mac(vbtr,-valfa,SR3O2));
      else
         sector = 2;
         valr = frac16(valfa/2);
         valr = frac16(mac(valr,vbeta,SR3O2));
         vbtr = frac16(vbeta/2);
         vbtr = frac16(mac(vbtr,-valfa,SR3O2));
      end
   end
else
   if (valfa>0),
      if (CN1OSR3*vbeta>-valfa),
         sector = 6;
         valr = frac16(valfa/2);
```

```
         valr = frac16(mac(valr,-vbeta,SR302));
         vbtr = frac16(vbeta/2);
         vbtr = frac16(mac(vbtr,valfa,SR302));
      else
         sector = 5;
         valr = frac16(-valfa/2);
         valr = frac16(mac(valr,-vbeta,SR302));
         vbtr = frac16(-vbeta/2);
         vbtr = frac16(mac(vbtr,valfa,SR302));
      end
   else
      if (CN10SR3*vbeta>valfa),
         sector = 4;
         valr = -valfa;
         vbtr = -vbeta;
      else
         sector = 5;
         valr = frac16(-valfa/2);
         valr = frac16(mac(valr,-vbeta,SR302));
         vbtr = frac16(-vbeta/2);
         vbtr = frac16(mac(vbtr,valfa,SR302));
      end
   end
end

% limit voltage magnitude
magvr = frac16(mac(valr,CN10SR3,vbtr));
if (magvr>CN1020SR2)
   magvrs = frac16(mac(0,magvr,CN2SR204));
   valr = div(frac16(valr/4),magvrs,1,1);
   vbtr = div(frac16(vbtr/4),magvrs,1,1);
end

% find counter values
valrs = frac16(mac(0,valr,CN10SR6));
vbtrs = frac16(mac(0,vbtr,SR206));
i=sector;
cnt(i) = alu('+',CN106,valrs);
cnt(i) = alu('+',cnt(i),valrs);
cnt(i) = alu('+',cnt(i),valrs);
cnt(i) = alu('+',cnt(i),valrs);
cnt(i) = alu('+',cnt(i),valrs);
cnt(i) = alu('-',cnt(i),vbtrs);
cnt(i) = alu('-',cnt(i),vbtrs);
cnt(i) = alu('-',cnt(i),vbtrs);
cnt(i) = alu('-',cnt(i),vbtrs);
cnt(i) = alu('-',cnt(i),vbtrs);
cnt(i) = alu('-',cnt(i),vbtrs);
cnt(i) = alu('-',cnt(i),vbtrs);
if (i==6),
   i = 1;
else
   i = i+1;
end
cnt(i) = alu('-',CN106,valrs);
cnt(i) = alu('+',cnt(i),vbtrs);
cnt(i) = alu('+',cnt(i),vbtrs);
cnt(i) = alu('+',cnt(i),vbtrs);
cnt(i) = alu('+',cnt(i),vbtrs);
cnt(i) = alu('+',cnt(i),vbtrs);
cnt(i) = alu('+',cnt(i),vbtrs);
cnt(i) = alu('+',cnt(i),vbtrs);
cnt(i) = alu('+',cnt(i),vbtrs);
cnt(i) = alu('+',cnt(i),vbtrs);
cnt(i) = alu('+',cnt(i),vbtrs);
cnt(i) = alu('+',cnt(i),vbtrs);
if (i==6),
   i = 1;
else
   i = i+1;
end
cnt(i) = alu('-',CN106,valrs);
cnt(i) = alu('-',cnt(i),vbtrs);
if (i==6),
   i = 1;
else
```

```
   i = i+1;
end
cnt(i) = alu('-',CN106,valrs);
cnt(i) = alu('-',cnt(i),vbtrs);
if (i==6),
   i = 1;
else
   i = i+1;
end
cnt(i) = alu('-',CN106,valrs);
cnt(i) = alu('-',cnt(i),vbtrs);
if (i==6),
   i = 1;
else
   i = i+1;
end
cnt(i) = alu('-',CN106,valrs);
cnt(i) = alu('-',cnt(i),vbtrs);

for i=1:6
   cnt(i) = frac16(mac(0,CPTPWM*2^-15,cnt(i)))*2^15;
end

% ------------------------------------------------
% Memory
% ------------------------------------------------

% save last current measurement
ialfa0 = ialfa(6);
ibeta0 = ibeta(6);

% save time intervals for next period
for i=1:6,
   t(i) = tn(i);
end
% load tn
for i=1:6,
   tn(i) = cnt(i);
end
```

B.3.4 frac16.m

```
function q = frac16(x)

% function q = frac16(x)
%
% Performs a quantization of x in the 1.15 format.
% Warning and saturation if x>1-LSB or x<-1.

LSB = 2^-15;

if (x>1 | x<-1)
   warning('The value to quantize is out of range, saturation will happen.')
   if x>1
      q=1-LSB;
   else
      q=-1;
   end
else
        q = quant(x,LSB);
        if q==1
                q=1-LSB;
   end
end
```

B.3.5 alu.m

```
function r = alu(op,x,y)
```

```
% function r = alu(op,x,y)
%
% Performs an ALU operation:
%
%   op = '+' => r = x + y
%   op = '-' => r = x - y
%
% Inputs:  op = operation
%          x = operand in 1.15 format
%          y = operand in 1.15 format
%
% The result will also be in 1.15 format.
% Warning if overflow happens.

switch op
case '+'
   r = x + y;
case '-'
   r = x - y;
otherwise
   error('Unkown operation')
end

if (r>1-2^-15 | r<-1)
   warning('Overflow in the ALU.')
end
```

B.3.6 mac.m

```
function m = mac(mr,mx,my)

% function m = mac(mr,mx,my)
%
% Performs a multiply-and-accumulate operation:
%
%          m = mr + mx*my
%
% Inputs:  mr = previous value of the accumulator,
%               in n.31 format
%          mx = operand in 1.15 format
%          my = operand in 1.15 format
%
% The result will naturally be in (n+1).31 format.
% Error if n+1>8 bits (overflow).

m = mr + mx*my;

if (m>256-2^-31 | m<-256)
   error('Overflow in the MAC.')
end
```

Appendix C

Assembler code for the DSP

C.1 final.asm

```
////////////////////////////////////////////////////
// final.asm
// Final version of the sensorless control program
// for ADSP-2181.
// Gabriel Eirea, June 2001
////////////////////////////////////////////////////

////////////////////////////////////////////////////
// constants
////////////////////////////////////////////////////

// experiment constants
#define omegaref1 0x038E        // initial speed reference (0x38E=50rad/s)
#define omegaref2 0x238E        // final speed reference (0x238E=500rad/s)
//#define omegaref2 0x0E39      // final speed reference (0x0E39=200rad/s)

#define cptpwmfr 0x07D0         // counts per TPWM as a fraction (*2^-15),
                                //   depends on TPWM and TCNT
#define cptdead 0x0014          // counts per TDEAD, depends on TDEAD and TCNT
#define cpt1o6 0x14D            // 1/6*cptpwmfr

#define thhatini 0x6A3E         // initial thhat

// numeric constants
#define sr3osr2o2 0x4E62        // sqrt(3/2)/2
#define cn1o2osr2 0x2D41        // 1/2/sqrt(2)
#define sr2o2 0x5A82            // sqrt(2)/2
#define cn1osr3 0x49E7          // 1/sqrt(3)
#define cn1o2 0x4000            // 1/2
#define sr3o2 0x6EDA            // sqrt(3)/2
#define cn1osr6 0x3441          // 1/sqrt(6)
#define sr2o6 0x1E2B            // sqrt(2)/6
#define cn1o6 0x1555            // 1/6

// observer coefficients
#define gamma1 0x0C7B           // depends on TPWM
#define gamma2 0x0549           // depends on TPWM
#define gamma3 0x53FB           // depends on TPWM
#define accoeff 0x0024          // depends on TPWM
#define spcoeff 0x0EAB          // depends on TPWM
```

```
// controller coefficients
#define err1coeffint 0x11B2     // depends on TPWM, TCNT and VBUS
#define err1coeffout 0x2254     // depends on TPWM, TCNT and VBUS
//#define int1coeff 0x6D71      // depends on TPWM, TCNT and VBUS
#define err2coeffint 0x0051     // depends on TPWM, TCNT and VBUS
#define err2coeffout 0x28BB     // depends on TPWM, TCNT and VBUS
#define int2coeff 0x2000        // depends on TPWM, TCNT and VBUS
#define err3coeffint 0x04B9     // depends on TPWM, TCNT and VBUS
#define err3coeffout 0x2C5D     // depends on TPWM, TCNT and VBUS
//#define int3coeff 0x6D71      // depends on TPWM, TCNT and VBUS

// other controller constants
#define ilimit 0x0780           // current limit
#define delaycoeff 0x1D56       // coefficient for delay correction

// PWM vectors
#define vec1 0x31
#define vec2 0x23
#define vec3 0x2A
#define vec4 0x0E
#define vec5 0x1C
#define vec6 0x15

// i/o port addresses
#define pia1 0x0                // ADC value of ia for subinterval 1 (in)
#define pia2 0x1                // ADC value of ia for subinterval 2 (in)
#define pia3 0x2                // ADC value of ia for subinterval 3 (in)
#define pia4 0x3                // ADC value of ia for subinterval 4 (in)
#define pia5 0x4                // ADC value of ia for subinterval 5 (in)
#define pia6 0x5                // ADC value of ia for subinterval 6 (in)
#define pib1 0x8                // ADC value of ib for subinterval 1 (in)
#define pib2 0x9                // ADC value of ib for subinterval 2 (in)
#define pib3 0xA                // ADC value of ib for subinterval 3 (in)
#define pib4 0xB                // ADC value of ib for subinterval 4 (in)
#define pib5 0xC                // ADC value of ib for subinterval 5 (in)
#define pib6 0xD                // ADC value of ib for subinterval 6 (in)
#define pv1 0x0                 // vector for subinterval 1 (out)
#define pv2 0x1                 // vector for subinterval 2 (out)
#define pv3 0x2                 // vector for subinterval 3 (out)
#define pv4 0x3                 // vector for subinterval 4 (out)
#define pv5 0x4                 // vector for subinterval 5 (out)
#define pv6 0x5                 // vector for subinterval 6 (out)
#define pcom 0x6                // command (out)
#define ppos 0x7                // position (out)
#define pt1 0x8                 // time for subinterval 1 (out)
#define pt2 0x9                 // time for subinterval 1 (out)
#define pt3 0xA                 // time for subinterval 1 (out)
#define pt4 0xB                 // time for subinterval 1 (out)
#define pt5 0xC                 // time for subinterval 1 (out)
#define pt6 0xD                 // time for subinterval 1 (out)
#define ptdb 0xE                // time for deadband (out)

// memory mapped registers and associated values
#define iowaitreg 0x3FFE        // address of the IO wait state register
#define onewaitst 0x7FF9        // value for one wait state
#define pfdatareg 0x3FE5        // address of the PFDATA register
#define maskpf1 0x02            // mask to read the PF1 bit
#define maskpf2 0x04            // mask to read the PF2 bit
#define enairqe 0x10            // value to enable IRQE
#define enasat 0x08             // value to enable saturation in AR register
#define dissat 0x00             // value to disable saturation in AR register

// declare sine subroutine
.EXTERN sin;

////////////////////////////////////////////////////
// variables
////////////////////////////////////////////////////
.SECTION/DATA pwmdata;

// currents
.VAR ia[6];                     // measured currents in phase a
.VAR ib[6];                     // measured currents in phase b
.VAR ialfa[7];                  // currents in alpha axis
.VAR ibeta[7];                  // currents in beta axis
.VAR dialfa[6];                 // difference of currents in alpha axis
```

```
.VAR dibeta[6];                 // difference of currents in beta axis
.VAR t[6];                      // subintervals time
.VAR tn[6];                     // subintervals time from previous period
.VAR dialdt[6];                 // dialfa over subinterval time
.VAR dibtdt[6];                 // dibeta over subinterval time
.VAR x[10];                     // vector x
.VAR Wpi[20] =                  // matrix Wpi, only rows 2 and 3
       0xC8A9,0x8F1B,0x9502,0x0221,0xCC59,0x7306,0x3757,0x70E5,0x6AFE,0xFDDF,
       0x5FDA,0xD76B,0xECD4,0x8640,0x8CFA,0xAED6,0xA026,0x2895,0x132C,0x79C0;
.VAR q[2];                      // vector q (estimated parameters)
.VAR alhat;                     // estimated acceleration
.VAR omhat;                     // estimated speed
.VAR thhat;                     // estimated position
.VAR obserr;                    // observer error
.VAR sinth;                     // sine of theta
.VAR costh;                     // cosine of theta
.VAR idref = 0;                 // reference id
.VAR wref = omegaref1;          // reference speed
.VAR int1;                      // integrator for id control block
.VAR int2;                      // integrator for w control block
.VAR int3;                      // integrator for iq control block
.VAR valfa;                     // valfa control output
.VAR vbeta;                     // vbeta control output
.VAR sector;                    // sector of the control output
.VAR/CIRC cnt[6];               // counter values for the PWM

////////////////////////////////////////////////////
// interrupt table
////////////////////////////////////////////////////
.SECTION/CODE itab;

        JUMP start;             // reset
                RTI; RTI; RTI;
        RTI; RTI; RTI; RTI;
        RTI; RTI; RTI; RTI;
        RTI; RTI; RTI; RTI;
        RTI; RTI; RTI; RTI;
        RTI; RTI; RTI; RTI;
        JUMP pwmirq;            // IRQE interrupt
                RTI; RTI; RTI;
        RTI; RTI; RTI; RTI;
        RTI; RTI; RTI; RTI;
        RTI; RTI; RTI; RTI;
        RTI; RTI; RTI; RTI;
        RTI; RTI; RTI; RTI;

////////////////////////////////////////////////////
// initialization code
////////////////////////////////////////////////////
.SECTION/CODE init;

start:

        // set wait states
        AX0 = onewaitst;        // one wait state
        DM(iowaitreg) = AX0;

        // stop PWM generation in FPGA (just in case it is already running)
        AX0 = 0;
        IO(pcom) = AX0;

        // enable interrupt IRQE
        IMASK = enairqe;

        // at the beginning, enable saturation in AR register
        MSTAT = enasat;

        // indirect adressing registers initialization
        L0 = 0;
        L1 = 0;
        L2 = 0;
        L3 = 0;
        L4 = 0;
        M0 = 1;
        M1 = 0;
        M3 = 1;
```

```
        M4 = 1;

        // state variables initialization
        AX0 = 0;
        DM(ialfa) = AX0;        // ialfa0 = 0
        DM(ibeta) = AX0;        // ibeta0 = 0
        I0 = tn;
        I1 = t;
        AX1 = cpt1o6-cptdead;   // load 1/6*cptpwm-cptdead
        CNTR = 6;
DO init_times UNTIL CE;
        DM(I0,M0) = AX1;        // tn(i) = 1/6*cptpwm-cptdead
init_times:
        DM(I1,M0) = AX1;        // t(i) = 1/6*cptpwm-cptdead
        DM(alhat) = AX0;        // alhat = 0
        DM(omhat) = AX0;        // omhat = 0
        AX1 = thhatini;
        DM(thhat) = AX1;        // thhat = initial thhat
        IO(ppos) = AX1;
        DM(obserr) = AX0;       // obserr = 0
        DM(int1) = AX0;         // int1 = 0
        DM(int2) = AX0;         // int2 = 0
        DM(int3) = AX0;         // int3 = 0

        // load vectors on FPGA
        AX0 = vec1;
        IO(pv1) = AX0;
        AX0 = vec2;
        IO(pv2) = AX0;
        AX0 = vec3;
        IO(pv3) = AX0;
        AX0 = vec4;
        IO(pv4) = AX0;
        AX0 = vec5;
        IO(pv5) = AX0;
        AX0 = vec6;
        IO(pv6) = AX0;

        // load times=1/6*TPWM (resulting vector = 0) on FPGA
        AX0 = cpt1o6-cptdead;   // load 1/6*cptpwm-cptdead
        AX1 = cptdead;          // load tdead
        IO(pt1) = AX0;
        IO(pt2) = AX0;
        IO(pt3) = AX0;
        IO(pt4) = AX0;
        IO(pt5) = AX0;
        IO(pt6) = AX0;
        IO(ptdb) = AX1;

        // wait until switch is changed of position
        AX0 = DM(pfdatareg);    // read flags
        AY0 = maskpf2;          // mask for PF2
        AR = AX0 AND AY0;       // find initial flag value
        AY1 = AR;               // move initial value
wait_switch:
        AX0 = DM(pfdatareg);    // read flags
        AR = AX0 AND AY0;       // find PF2 value
        AR = AR XOR AY1;        // compare with initial value
        IF EQ JUMP wait_switch; // loop until change in the flag value

        // start PWM generation in FPGA
        AX0 = 1;
        IO(pcom) = AX0;

        // infinite loop
iloop:
        IDLE;
        JUMP iloop;

/////////////////////////////////////////////////////
// interrupt service rutine
/////////////////////////////////////////////////////
.SECTION/CODE irq;

pwmirq:
```

```
        // if pushbutton is pressed, change speed reference
        AX0 = DM(pfdatareg);    // read flags
        AY0 = maskpf1;          // mask for PF1
        AR = AX0 AND AY0;       // find flag value
        IF EQ JUMP start_alg;   // if it is zero, don't do nothing

        // change speed reference
        AX0 = omegaref2;
        DM(wref) = AX0;

start_alg:

        // read currents from i/o ports
        I0 = ia;
        AX0 = IO(pia1);
        DM(I0,M0) = AX0;
        AX0 = IO(pia2);
        DM(I0,M0) = AX0;
        AX0 = IO(pia3);
        DM(I0,M0) = AX0;
        AX0 = IO(pia4);
        DM(I0,M0) = AX0;
        AX0 = IO(pia5);
        DM(I0,M0) = AX0;
        AX0 = IO(pia6);
        DM(I0,M0) = AX0;
        I2 = ib;
        AX0 = IO(pib1);
        DM(I2,M0) = AX0;
        AX0 = IO(pib2);
        DM(I2,M0) = AX0;
        AX0 = IO(pib3);
        DM(I2,M0) = AX0;
        AX0 = IO(pib4);
        DM(I2,M0) = AX0;
        AX0 = IO(pib5);
        DM(I2,M0) = AX0;
        AX0 = IO(pib6);
        DM(I2,M0) = AX0;

        // convert to alfa-beta
        I0 = ia;
        I1 = ialfa;
        MODIFY(I1,M0);          // skip 1st element (last value of last period)
        MY0 = sr3osr2o2;
        CNTR = 6;
DO conv_alfa UNTIL CE;
        MX0 = DM(I0,M0);        // load ia(i)
        MR = MX0*MY0 (RND);     // ia(i)*sr3osr2o2
conv_alfa:
        DM(I1,M0) = MR1;        // save ialfa(i)
        I0 = ia;
        I2 = ib;
        I3 = ibeta;
        MODIFY(I3,M0);          // skip 1st element (last value of last period)
        MY0 = cn1o2osr2;
        MY1 = sr2o2;
        CNTR = 6;
DO conv_beta UNTIL CE;
        MX0 = DM(I0,M0);        // load ia(i)
        MX1 = DM(I2,M0);        // load ib(i)
        MR = MX0*MY0 (RND);     // ia(i)*cn1o2osr2
        MR = MR+MX1*MY1 (RND);  // ia(i)*cn1o2osr2 + ib(i)*sr2o2
conv_beta:
        DM(I3,M0) = MR1;        // save ibeta(i)

        // difference of currents
        I0 = dialfa;
        I1 = ialfa;
        I2 = dibeta;
        I3 = ibeta;
        AX0 = DM(I1,M0);        // load ialfa(0)
        AX1 = DM(I3,M0);        // load ibeta(0)
        CNTR = 6;
DO subs_curr UNTIL CE;
        AY0 = AX0;              // move ialfa(i-1)
```

```
        AY1 = AX1;              // move ibeta(i-1)
        AX0 = DM(I1,M0);        // load ialfa(i)
        AX1 = DM(I3,M0);        // load ibeta(i)
        AR = AX0-AY0;           // ialfa(i)-ialfa(i-1)
        DM(I0,M0) = AR;         // save dialfa(i)
        AR = AX1-AY1;           // ibeta(i)-ibeta(i-1)
subs_curr:
        DM(I2,M0) = AR;         // save dibeta(i)

        // division by time
        I0 = dialfa;
        I1 = dialdt;
        I2 = dibeta;
        I3 = dibtdt;
        I4 = t;
        CNTR = 6;
DO div_time UNTIL CE;
        AX0 = DM(I4,M4);        // load t(i)
        SR0 = DM(I0,M0);        // load dialfa(i)
        SR = ASHIFT SR0 BY -3 (HI);     // shift dialfa(i)
        AY1 = SR1;              // move shifted dialfa(i)
        AY0 = SR0;
        DIVS AY1,AX0;           // dialfa(i)/t(i)
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DM(I1,M0) = AY0;        // save dialdt(i)
        SR0 = DM(I2,M0);        // load dibeta(i)
        SR = ASHIFT SR0 BY -3 (HI);     // shift dibeta(i)
        AY1 = SR1;              // move shifted dibeta(i)
        AY0 = SR0;
        DIVS AY1,AX0;           // dibeta(i)/t(i)
                DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
div_time:
        DM(I3,M0) = AY0;        // save dibtdt(i)

        // construct vector x
        I0 = x;
        I1 = dialdt;
        I3 = dibtdt;
        SR0 = DM(I1,M0);        // load dialdt(1)
        SR = ASHIFT SR0 BY -1 (LO);     // shift dialdt (divide by 2)
        AX0 = SR0;              // move dialdt(1)/2
        SR0 = DM(I3,M0);        // load dibtdt(1)
        SR = ASHIFT SR0 BY -1 (LO);     // shift dibtdt (divide by 2)
        AX1 = SR0;              // move dibtdt(1)/2
        CNTR = 5;
DO constr_x UNTIL CE;
        AY0 = AX0;              // move dialdt(i-1)/2
        AY1 = AX1;              // move dibtdt(i-1)/2
        SR0 = DM(I1,M0);        // load dialdt(i)
        SR = ASHIFT SR0 BY -1 (LO);     // shift dialdt (divide by 2)
        AX0 = SR0;              // move dialdt(i)/2
        SR0 = DM(I3,M0);        // load dibtdt(i)
        SR = ASHIFT SR0 BY -1 (LO);     // shift dibtdt (divide by 2)
        AX1 = SR0;              // move dibtdt(i)/2
        AR = AX0-AY0;           // dialdt(i)/2-dialdt(i-1)/2
        DM(I0,M0) = AR;         // save x(2*i-1)
        AR = AX1-AY1;           // dibtdt(i)/2-dibtdt(i-1)/2
constr_x:
        DM(I0,M0) = AR;         // save x(2*i)

        // find q=Wpi*x
        I0 = x;
        I1 = Wpi;
        I2 = q;
        MR = 0,                 // clear accumulator
        MX0 = DM(I0,M0);        // load x(1)
        MY0 = DM(I1,M0);        // load Wpi(1,1)
        CNTR = 10;
DO find_q2 UNTIL CE;
        MR = MR+MX0*MY0 (RND),  // MR+x(i)*Wpi(1,i)
```

```
        MX0 = DM(I0,M0);        // load x(i+1)
find_q2:
        MY0 = DM(I1,M0);        // load Wpi(1,i+1)
        SR = ASHIFT MR1 BY 1 (LO);      // multiply result by 2
        SR = SR OR LSHIFT MR0 BY -15 (LO);      // add LSB to increase precision
        DM(I2,M0) = SR0;        // save 2*q(2)

        I0 = x;
        MR = 0,                 // clear accumulator
        MX0 = DM(I0,M0);        // load x(1), Wpi(2,1) already loaded in MY0
        CNTR = 10;
DO find_q3 UNTIL CE;
        MR = MR+MX0*MY0 (RND),  // MR+x(i)*Wpi(2,i)
        MX0 = DM(I0,M0);        // load x(i+1)
find_q3:
        MY0 = DM(I1,M0);        // load Wpi(2,i+1)
        SR = ASHIFT MR1 BY 1 (LO);      // multiply result by 2
        SR = SR OR LSHIFT MR0 BY -15 (LO);      // add LSB to increase precision
        DM(I2,M0) = SR0;        // save 2*q(3)

        // update observer
        MX0 = gamma3;           // load gamma3
        MY0 = DM(obserr);       // load obserr
        MR0 = 0;
        MR1 = DM(alhat);        // load alhat
        MR = MR+MX0*MY0 (RND);  // alhat+gamma3*obserr
        DM(alhat) = MR1;        // save new alhat
        MX0 = gamma2;           // load gamma2
        MX1 = accoeff;          // load accoeff
        MY1 = MR1;              // move alhat
        MR0 = 0;
        MR1 = DM(omhat);        // load omhat
        MR = MR+MX0*MY0 (RND);  // omhat+gamma2*obserr
        MR = MR+MX1*MY1 (RND);  // omhat+gamma2*obserr+accoeff*alhat
        DM(omhat) = MR1;        // save new omhat
        MX0 = gamma1;           // load gamma1
        MX1 = spcoeff;          // load spcoeff
        MY1 = MR1;              // move omhat
        MR0 = 0;
        MR1 = DM(thhat);        // load thhat
        MR = MR+MX0*MY0 (SS);   // thhat+gamma1*obserr
        MR = MR+MX1*MY1 (RND);  // thhat+gamma1*obserr+spcoeff*omhat
        DM(thhat) = MR1;        // save new thhat

        // update observer error
        MSTAT = dissat;         // disable saturation to allow wraparound
        AY0 = MR1;              // move thhat
        AR = MR1 + AY0;         // 2*thhat, if overflow => auto wraparound
        AX0 = AR;
        CALL sin;               // find sin(2*thhat), result in AR
        MX0 = AR;               // move sin(2*thhat)
        AY0 = AX0;              // move 2*thhat
        AX0 = 0x4000;           // this is .5 (corresponds to pi/2)
        AR = AX0 - AY0;         // pi/2 - 2*thhat
        AX0 = AR;
        CALL sin;               // find sin(pi/2-2*thhat) = cos(2*thhat)
        MX1 = AR;               // move cos(2*thhat)
        I2 = q;
        MY0 = DM(I2,M0);        // load 2*q(2)
        MY1 = DM(I2,M0);        // load 2*q(3)
        MR = MX1*MY1 (SS);      // 2*q(3)*cos(2*thhat)
        MR = MR-MX0*MY0 (RND);  // 2*q(3)*cos(2*thhat)-2*q(2)*sin(2*thhat)
        DM(obserr) = MR1;       // save obserr
        MSTAT = enasat;         // enable saturation again

        // average the currents
        I0 = ialfa;
        I1 = ibeta;
        I2 = t;
        AR = DM(I0,M0);         // load ialfa(0)
        SR = ASHIFT AR BY -1 (LO);      // ialfa(0)/2
        AX0 = SR0;              // move ialfa(0)/2
        MR = 0;
        CNTR = 6;
DO alfa_av UNTIL CE;
        AR = DM(I0,M0);         // load ialfa(i)
```

```
        SR = ASHIFT AR BY -1 (LO);      // ialfa(i)/2
        AY0 = SR0;
        AR = AX0 + AY0;         // ialfa(i-1)/2 + ialfa(i)/2
        MY0 = DM(I2,M0);        // load t(i)
        MR = MR+AR*MY0 (RND);   // (ialfa(i-1)+ialfa(i))/2*t(i)
alfa_av:
        AX0 = AY0;              // move ialfa(i)/2
        SR = ASHIFT MR1 BY 2 (LO);      // multiply by 4
        MX0 = SR0;              // move result (ialfa average) to MX0

        I2 = t;
        AR = DM(I1,M0);         // load ibeta(0)
        SR = ASHIFT AR BY -1 (LO);      // ibeta(0)/2
        AX0 = SR0;              // move ibeta(0)/2
        MR = 0;
        CNTR = 6;
DO beta_av UNTIL CE;
        AR = DM(I1,M0);         // load ibeta(i)
        SR = ASHIFT AR BY -1 (LO);      // ibeta(i)/2
        AY0 = SR0;
        AR = AX0 + AY0;         // ibeta(i-1)/2 + ibeta(i)/2
        MY0 = DM(I2,M0);        // load t(i)
        MR = MR+AR*MY0 (RND);   // (ibeta(i-1)+ibeta(i))/2*t(i)
beta_av:
        AX0 = AY0;              // move ibeta(i)/2
        SR = ASHIFT MR1 BY 2 (LO);      // multiply by 4
        MY0 = SR0;              // move result (ibeta average) to MY0

        // convert to d-q frame
        MSTAT = dissat;         // disable saturation to allow wraparound
        AX0 = DM(thhat);        // load thhat
        CALL sin;               // find sin(thhat)
        DM(sinth) = AR;         // save sin(thhat)
        AY0 = AX0;              // move thhat
        AX0 = 0x4000;           // this is .5 (corresponds to pi/2)
        AR = AX0 - AY0;         // pi/2 - thhat
        AX0 = AR;
        CALL sin;               // find sin(pi/2-thhat) = cos(thhat)
        DM(costh) = AR;         // save costh
        MSTAT = enasat;         // enable saturation again
        MX1 = MY0;              // move ibeta
        MY0 = DM(sinth);        // load sin(thhat)
        MY1 = AR;               // move cos(thhat)
        MR = MX0*MY1 (RND);     // ialfa*cos(thhat)
        MR = MR+MX1*MY0 (RND);  // ialfa*cos(thhat)+ibeta*sin(thhat)
        AY0 = MR1;              // move id
        MR = MX1*MY1 (RND);     // ibeta*cos(thhat)
        MR = MR-MX0*MY0 (RND);  // ibeta*cos(thhat)-ialfa*sin(thhat)
        AY1 = MR1;              // move iq

        // id control block
        AX0 = DM(idref);        // load idref
        AR = AX0-AY0;           // idref-id
        MR1 = DM(int1);         // load int1
        MR0 = 0;
        MY0 = err1coeffint;     // load err1coeffint
        MR = MR+AR*MY0 (RND);   // int1+(idref-id)*err1coeffint
        IF MV SAT MR;           // saturate int1
        DM(int1) = MR1;         // save int1
        MY1 = err1coeffout;     // load err1coeffout
        MR = MR+AR*MY1 (RND);   // int1+(idref-id)*err1coeffout
        IF MV SAT MR;
        AY0 = MR1;              // move
        AF = AR+AY0;            // int1+(idref-id)*(err1coeffout+1)
        AR = AR+AF;             // int1+(idref-id)*(err1coeffout+2)
        MX0 = AR;               // move vdc

        // w control block
        AX0 = DM(wref);         // load wref
        AY0 = DM(omhat);        // load omhat
        AR = AX0-AY0;           // wref-what
        MR1 = DM(int2);         // load int2
        MR0 = 0;
        MY0 = err2coeffint;     // load err2coeffint
        MR = MR+AR*MY0 (RND);   // int2+(wref-what)*err2coeffint
        IF MV SAT MR;           // saturation of int2
```

```
        DM(int2) = MR1;         // save int2
        MY0 = int2coeff;        // load int2coeff
        MY1 = err2coeffout;     // load err2coeffout
        MR = MR1*MY0 (SS);      // int2*int2coeff
        MR = MR+AR*MY1 (RND);   // int2*int2coeff+(wref-what)*err2coeffout
        AR = ABS MR1;           // find absolute value of iqref
        AY0 = ilimit;           // load ilimit
        NONE = AR-AY0;          // compare
        IF GT JUMP sat_iqref;   // if abs(iqref)>ilimit, saturate
        AX0 = MR1;              // move iqref, no saturation
        JUMP iq_control;        // continue with controller
sat_iqref:
        AX0 = ilimit;           // load absolute limit
        NONE = PASS MR1;        // check sign of original iqref
        IF GT JUMP iq_control;  // if positive, leave ilimit at AX0
        AR = -AX0;              // if negative, negate
        AX0 = AR;               // move -ilimit to AX0

iq_control:
        // iq control block
        AR = AX0-AY1;           // iqref-iq
        MR1 = DM(int3);         // load int3
        MR0 = 0;
        MY0 = err3coeffint;     // load err3coeffint
        MR = MR+AR*MY0 (RND);   // int3+(iqref-iq)*err3coeffint
        IF MV SAT MR;           // saturation of int3
        DM(int3) = MR1;         // save int3
        MY1 = err3coeffout;     // load err3coeffout
        MR = MR+AR*MY1 (RND);   // int3+(iqref-iq)*err3coeffout
        IF MV SAT MR;
        AY0 = MR1;              // move
        AR = AR+AY0;            // int3+(iqref-iq)*(err3coeffout+1)
        AX1 = AR;               // move vqc

        // convert back to alfa-beta frame
        MSTAT = dissat;         // disable saturation to allow wraparound
        AX0 = DM(omhat);        // load omhat
        AR = PASS AX0;          // move omhat to AR
        MY0 = delaycoeff;       // load delaycoeff
        MR = AR*MY0 (RND);      // delaycoeff*omhat
        AY0 = DM(thhat);        // load thhat
        AR = MR1+AY0;           // thhat+delaycoeff*omhat, allow wraparound
        AX0 = AR;               // move corrected thhat
        CALL sin;               // find sin(corrthhat)
        MY0 = AR;               // move sin(corrthhat)
        AY0 = AX0;              // move corrthhat
        AX0 = 0x4000;           // this is .5 (corresponds to pi/2)
        AR = AX0 - AY0;         // pi/2-corrthhat, allow wraparound
        AX0 = AR;               // move pi/2-corrthhat
        CALL sin;               // find sin(pi/2-corrthhat) = cos(corrthhat)
        MY1 = AR;               // move cos(corrthhat)
        MSTAT = enasat;         // enable saturation again
        MR = MX0*MY1 (SS);      // vdc*costh
        MX1 = AX1;              // move vqc
        MR = MR-MX1*MY0 (RND);  // vdc*costh-vqc*sinth
        DM(valfa) = MR1;        // save valfa
        MR = MX0*MY0 (SS);      // vdc*sinth
        MR = MR+MX1*MY1 (RND);  // vdc*sinth+vqc*costh
        DM(vbeta) = MR1;        // save vbeta

        // find sector and rotate vector
        AR = PASS MR1;          // check sign of vbeta
        IF LE JUMP vbn;
vbp:
        AX0 = DM(valfa);
        AR = PASS AX0;          // check sign of valfa
        IF LE JUMP vbpvan;
vbpvap:
        MX1 = MR1;              // move vbeta
        MR1 = AX0;              // move valfa
        MR0 = 0;
        MY0 = cn1osr3;          // load 1/sqrt(3)
        MR = MR-MX1*MY0 (RND);  // valfa-vbeta*cn1osr3
        AR = PASS MR1;
        IF LE JUMP sector2;
sector1:
```

```
        AX1 = 1;
        DM(sector) = AX1;       // sector = 1
                                // valfa and vbeta remain the same
        JUMP endsec;
sector2:
        AX1 = 2;
        DM(sector) = AX1;       // sector = 2
        MX0 = AX0;              // move valfa
        MY0 = cn1o2;            // load 1/2
        MY1 = sr3o2;            // load sqrt(3)/2
        MR = MX0*MY0 (SS);      // valfa/2
        MR = MR+MX1*MY1 (RND);  // valfa/2+vbeta*sqrt(3)/2
        DM(valfa) = MR1;        // save rotated valfa
        MR = MX1*MY0 (SS);      // vbeta/2
        MR = MR-MX0*MY1 (RND);  // vbeta/2-valfa*sqrt(3)/2
        DM(vbeta) = MR1;        // save rotated vbeta
        JUMP endsec;
vbpvan:
        MX1 = MR1;              // move vbeta
        MR1 = AX0;              // move valfa
        MR0 = 0;
        MY0 = cn1osr3;          // load 1/sqrt(3)
        MR = MR+MX1*MY0 (RND);  // valfa+vbeta*cn1osr3
        AR = PASS MR1;
        IF GE JUMP sector2;
sector3:
        AX1 = 3;
        DM(sector) = AX1;       // sector = 3
        MX0 = AX0;              // move valfa
        MY0 = cn1o2;            // load 1/2
        MY1 = sr3o2;            // load sqrt(3)/2
        MR = MX1*MY1 (SS);      // vbeta*sqrt(3)/2
        MR = MR-MX0*MY0 (RND);  // vbeta*sqrt(3)/2-valfa/2
        DM(valfa) = MR1;        // save rotated valfa
        MR = 0;
        MR = MR-MX1*MY0 (SS);   // -vbeta/2
        MR = MR-MX0*MY1 (RND);  // -vbeta/2-valfa*sqrt(3)/2
        DM(vbeta) = MR1;        // save rotated vbeta
        JUMP endsec;
vbn:
        AX0 = DM(valfa);
        AR = PASS AX0;          // check sign of valfa
        IF LE JUMP vbnvan;
vbnvap:
        MX1 = MR1;              // move vbeta
        MR1 = AX0;              // move valfa
        MR0 = 0;
        MY0 = cn1osr3;          // load 1/sqrt(3)
        MR = MR+MX1*MY0 (RND);  // valfa+vbeta*cn1osr3
        AR = PASS MR1;
        IF LE JUMP sector5;
sector6:
        AX1 = 6;
        DM(sector) = AX1;       // sector = 6
        MX0 = AX0;              // move valfa
        MY0 = cn1o2;            // load 1/2
        MY1 = sr3o2;            // load sqrt(3)/2
        MR = MX0*MY0 (SS);      // valfa/2
        MR = MR-MX1*MY1 (RND);  // valfa/2-vbeta*sqrt(3)/2
        DM(valfa) = MR1;        // save rotated valfa
        MR = MX1*MY0 (SS);      // vbeta/2
        MR = MR+MX0*MY1 (RND);  // vbeta/2+valfa*sqrt(3)/2
        DM(vbeta) = MR1;        // save rotated vbeta
        JUMP endsec;
sector5:
        AX1 = 5;
        DM(sector) = AX1;       // sector = 5
        MX0 = AX0;              // move valfa
        MY0 = cn1o2;            // load 1/2
        MY1 = sr3o2;            // load sqrt(3)/2
        MR = 0;
        MR = MR-MX0*MY0 (SS);   // -valfa/2
        MR = MR-MX1*MY1 (RND);  // -valfa/2-vbeta*sqrt(3)/2
        DM(valfa) = MR1;        // save rotated valfa
        MR = MX0*MY1 (SS);      // valfa*sqrt(3)/2
        MR = MR-MX1*MY0 (RND);  // valfa*sqrt(3)/2-vbeta/2
```

```
        DM(vbeta) = MR1;        // save rotated vbeta
        JUMP endsec;
vbnvan:
        MX1 = MR1;              // move vbeta
        MR1 = AX0;              // move valfa
        MR0 = 0;
        MY0 = cn1osr3;          // load 1/sqrt(3)
        MR = MR-MX1*MY0 (RND);  // valfa-vbeta*cn1osr3
        AR = PASS MR1;
        IF GE JUMP sector5;
sector4:
        AX1 = 4;
        DM(sector) = AX1;       // sector = 4
        AR = -AX0;              // -valfa
        DM(valfa) = AR;         // save rotated valfa
        AX0 = MX1;              // move vbeta
        AR = -AX0;              // -vbeta
        DM(vbeta) = AR;         // save rotated vbeta
endsec:

        // limit voltage magnitude
        MR1 = DM(valfa);        // load valfa
        MX0 = DM(vbeta);        // load vbeta
        MY0 = cn1osr3;          // load 1/sqrt(3)
        MR = MR+MX0*MY0 (RND);  // valfa+vbeta/sqrt(3)
        AY0 = cn1o2osr2;        // load 1/2/sqrt(2)
        AR = MR1-AY0;           // valfa+vbeta/sqrt(3) - 1/2/sqrt(2)
        IF LE JUMP nolimit;
limit:
        MY0 = sr2o2;            // load sqrt(2)/2
        MR = MR1*MY0 (RND);     // (valfa+vbeta/sqrt(3))*sqrt(2)/2
        AX0 = MR1;              // move result
        SI = DM(valfa);         // load valfa
        SR = ASHIFT SI BY -2 (LO);      // valfa/4
        AY1 = SR0;              // move valfa/4
        DIVS AY1,AX0;           // (valfa/4)/((valfa+vbeta/sqrt(3))*sqrt(2)/2)
                DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DM(valfa) = AY0;        // save new value of valfa
        SI = DM(vbeta);         // load vbeta
        SR = ASHIFT SI BY -2 (LO);      // vbeta/4
        AY1 = SR0;              // move vbeta/4
        DIVS AY1,AX0;           // (vbeta/4)/((valfa+vbeta/sqrt(3))*sqrt(2)/2)
                DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DM(vbeta) = AY0;        // save new value of vbeta
nolimit:

        // compute counter values
        I0 = cnt;               // pointer to cnt
        L0 = 6;                 // circular buffer length 6
        AX0 = DM(sector);       // load sector
        AX1 = cn1o6;            // load 1/6
        MX0 = DM(valfa);        // load valfa
        MY0 = cn1osr6;          // load 1/sqrt(6)
        MR = MX0*MY0 (RND);     // valfa*1/sqrt(6)
        AY0 = MR1;              // move new valfa
        MX0 = DM(vbeta);        // load vbeta
        MY0 = sr2o6;            // load sqrt(2)/6
        MR = MX0*MY0 (RND);     // vbeta*sqrt(2)/6
        AY1 = MR1;              // move new vbeta
        AR = AX0-1;             // sector-1
        M2 = AR;
        MODIFY(I0,M2);          // point to cnt(sector)
        AR = AX1+AY0;
        AR = AR+AY0;
        AR = AR+AY0;
        AR = AR+AY0;
        AR = AR+AY0;
        AR = AR-AY1;
        AR = AR-AY1;
        AR = AR-AY1;
```

```
        AR = AR-AY1;
        AR = AR-AY1;
        AR = AR-AY1;
        AR = AR-AY1;            // 1/6+5*valfa-7*vbeta
        DM(I0,M0) = AR;         // save cnt(sector)
        AR = AX1-AY0;
        AR = AR+AY1;
        AR = AR+AY1;
        AR = AR+AY1;
        AR = AR+AY1;
        AR = AR+AY1;
        AR = AR+AY1;
        AR = AR+AY1;
        AR = AR+AY1;
        AR = AR+AY1;
        AR = AR+AY1;
        AR = AR+AY1;            // 1/6-valfa+11*vbeta
        DM(I0,M0) = AR;         // save cnt(sector+1)
        AR = AX1-AY0;
        AR = AR-AY1;            // 1/6-valfa-vbeta
        CNTR = 4;
DO load_cntr UNTIL CE;
load_cntr:
        DM(I0,M0) = AR;         // save cnt(i)
        L0 = 0;                 // disable circular buffer

        // convert counter values from fraction to integer
        I0 = cnt;
        MY0 = cptpwmfr;         // load cptpwmfr
        AY0 = cptdead;          // load cptdead
        CNTR = 6;
DO scale_cnt UNTIL CE;
        MX0 = DM(I0,M1);        // load cnt(i)
        MR = MX0*MY0 (SS);      // cnt(i)*cptpwmfr
        AR = MR1-AY0;           // substract deadband time
scale_cnt:
        DM(I0,M0) = AR;         // save integer cnt(i)

        // output counter values and position
        I0 = cnt;
        AX0 = DM(I0,M0);
        I0(pt1) = AX0;
        AX0 = DM(I0,M0);
        I0(pt2) = AX0;
        AX0 = DM(I0,M0);
        I0(pt3) = AX0;
        AX0 = DM(I0,M0);
        I0(pt4) = AX0;
        AX0 = DM(I0,M0);
        I0(pt5) = AX0;
        AX0 = DM(I0,M0);
        I0(pt6) = AX0;
        AX0 = DM(thhat);
        I0(ppos) = AX0;

        // save last current measurement
        I0 = ialfa;             // pointer to ialfa
        M2 = 6;
        MODIFY(I0,M2);          // point to last value
        AX0 = DM(I0,M0);
        I0 = ialfa;
        DM(I0,M0) = AX0;        // copy last value to first position
        I0 = ibeta;             // pointer to ibeta
        MODIFY(I0,M2);          // point to last value
        AX0 = DM(I0,M0);
        I0 = ibeta;
        DM(I0,M0) = AX0;        // copy last value to first position

        // save time intervals for next period
        I0 = tn;
        I1 = t;
        I2 = cnt;
        CNTR = 6;
DO copy_times UNTIL CE;
        AX0 = DM(I0,M1);        // load tn(i), don't increment I0
        AX1 = DM(I2,M0);        // load cnt(i), increment I2
```

```
        DM(I0,M0) = AX1;        // save new tn(i), increment I0
copy_times:
        DM(I1,M0) = AX0;        // save new t(i), increment I1

        // end of the interrupt routine
        RTI;

/////////////////////////////////////////////////////
// end of program
/////////////////////////////////////////////////////
```

C.2 sine.asm

```
/*
        Sine Approximation
                y = sin(x)
        Calling Parameters
                AX0 = x in scaled 1.15 format
                M3 = 1
                L3 = 0
        Return Values
                AR = y in 1.15 format
        Altered Registers
                AY0,AF,AR,MY1,MX1,MF,MR,SR,I3
        Computation Time
                25 cycles
        ( remember cos(x) = sin(pi/2-x) )
*/

.GLOBAL sin;

.SECTION/DATA sindata;

.VAR sin_coeff[5] = 0x3240, 0x0053, 0xAACC, 0x08B7, 0x1CCE;

.SECTION/CODE sin;

sin:
        I3=sin_coeff;                   // Pointer to coeff. buffer
        AY0=0x4000;
        AR=AX0, AF=AX0 AND AY0;         // Check 2nd or 4th quad.
        IF NE AR=-AX0;                  // If yes, negate input
        AY0=0x7FFF;
        AR=AR AND AY0;                  // Remove sign bit
        MY1=AR;
        MF=AR*MY1 (RND), MX1=DM(I3,M3); // MF = x^2
        MR=MX1*MY1 (SS), MX1=DM(I3,M3); // MR = C_1 x
        CNTR=3;
        DO approx UNTIL CE;
        MR=MR+MX1*MF (SS);
approx:
        MF=AR*MF (RND), MX1=DM(I3,M3);
        MR=MR+MX1*MF (SS);
        SR=ASHIFT MR1 BY 3 (HI);
        SR=SR OR LSHIFT MR0 BY 3 (LO);  // Convert to 1.15 format
        AR=PASS SR1;
        IF LT AR=PASS AY0;              // Saturate if needed
        AF=PASS AX0;
        IF LT AR=-AR;                   // Negate output if needed
        RTS;
```

C.3 test.ldf

```
ARCHITECTURE(ADSP-2181)
SEARCH_DIR( $ADI_DSP\218x\lib )
$OBJECTS=$COMMAND_LINE_OBJECTS;

MEMORY
{
    seg_itab    { TYPE(PM RAM) START(0x00000) END(0x0002f) WIDTH(24) }
    seg_init    { TYPE(PM RAM) START(0x00030) END(0x00fff) WIDTH(24) }
    seg_irq     { TYPE(PM RAM) START(0x01000) END(0x02fff) WIDTH(24) }
```

```
    seg_sin     { TYPE(PM RAM) START(0x03000) END(0x03fff) WIDTH(24) }

    seg_pwmdata { TYPE(DM RAM) START(0x00000) END(0x00fff) WIDTH(16) }
    seg_sindata { TYPE(DM RAM) START(0x01000) END(0x01fff) WIDTH(16) }
}

PROCESSOR p0
{
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST )
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

    SECTIONS
    {
        sec_init
        {
            INPUT_SECTIONS( $OBJECTS(init) )
        } >seg_init

                sec_itab
        {
            INPUT_SECTIONS( $OBJECTS(itab) )
        } >seg_itab

                sec_irq
                {
                        INPUT_SECTIONS( $OBJECTS(irq) )
                } >seg_irq

                sec_sin
                {
                        INPUT_SECTIONS( $OBJECTS(sin) )
                } >seg_sin

                sec_pwmdata
        {
            INPUT_SECTIONS( $OBJECTS(pwmdata) )
        } >seg_pwmdata

                sec_sindata
        {
            INPUT_SECTIONS( $OBJECTS(sindata) )
        } >seg_sindata

        }
}
```

Appendix D

VHDL code for the FPGA

D.1 pwm.vhd

```
-- pwm.vhd
-- Digital PWM generation in an FPGA.
-- Top-level design file.
-- ge, Feb 2001

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY altera;
USE altera.maxplus2.ALL;
LIBRARY work;
USE work.pwmpkg.ALL;

ENTITY pwm IS
        PORT (
                clk:            IN              STD_LOGIC;
                rst:            IN              STD_LOGIC;
                rd_n:           IN              STD_LOGIC;
                wr_n:           IN              STD_LOGIC;
                ioms_n:         IN              STD_LOGIC;
                dspa:           IN              STD_LOGIC_VECTOR (3 DOWNTO 0);
                dspd:           INOUT           STD_LOGIC_VECTOR (15 DOWNTO 0);
                adca:           IN              adctype;
                adcb:           IN              adctype;
                clkout:         OUT             STD_LOGIC;
                irq_n:          OUT             STD_LOGIC;
                pwm_sig:        OUT             pwmvec;
                position:       OUT             postype;
                tp:             OUT             STD_LOGIC_VECTOR(3 DOWNTO 0);
                inp75:          IN              STD_LOGIC;
                inp79:          IN              STD_LOGIC;
                inp80:          IN              STD_LOGIC;
                inp81:          IN              STD_LOGIC
        );
END pwm;

ARCHITECTURE flex6k OF pwm IS

        SIGNAL new_time:        timetype;
        SIGNAL status:          STD_LOGIC;
        SIGNAL step:            steptype;
```

```
        SIGNAL new_step:        STD_LOGIC;
        SIGNAL new_period:      STD_LOGIC;
        SIGNAL clko:            STD_LOGIC;
        SIGNAL adcar:           adctype;
        SIGNAL adcbr:           adctype;
        -- added for debug
        SIGNAL interr:          STD_LOGIC;

BEGIN

        -- bit reversing of ADC input because of mistake in PCB design
        adcar(13) <= adca(0);
        adcar(12) <= adca(1);
        adcar(11) <= adca(2);
        adcar(10) <= adca(3);
        adcar(9) <= adca(4);
        adcar(8) <= adca(5);
        adcar(7) <= adca(6);
        adcar(6) <= adca(7);
        adcar(5) <= adca(8);
        adcar(4) <= adca(9);
        adcar(3) <= adca(10);
        adcar(2) <= adca(11);
        adcar(1) <= adca(12);
        adcar(0) <= adca(13);
        adcbr(13) <= adcb(0);
        adcbr(12) <= adcb(1);
        adcbr(11) <= adcb(2);
        adcbr(10) <= adcb(3);
        adcbr(9) <= adcb(4);
        adcbr(8) <= adcb(5);
        adcbr(7) <= adcb(6);
        adcbr(6) <= adcb(7);
        adcbr(5) <= adcb(8);
        adcbr(4) <= adcb(9);
        adcbr(3) <= adcb(10);
        adcbr(2) <= adcb(11);
        adcbr(1) <= adcb(12);
        adcbr(0) <= adcb(13);

        -- process to divide the clock frequency by 2
        PROCESS (clk)
                VARIABLE tflipflop:     STD_LOGIC;
        BEGIN
                IF (clk'EVENT AND clk = '1') THEN
                        tflipflop := NOT tflipflop;
                END IF;
                clko <= tflipflop;
        END PROCESS;

        -- instantiate the 'regs' module
        reg:    regs PORT MAP (
                                clk => clko,
                                rst => rst,
                                ioms_n => ioms_n,
                                rd_n => rd_n,
                                wr_n => wr_n,
                                step => step,
                                new_step => new_step,
                                new_period => new_period,
                                address => dspa,
                                adca => adcar,
                                adcb => adcbr,
                                data => dspd,
                                curr_vector => pwm_sig,
                                new_time => new_time,
                                position => position,
                                status => status,
                                irq_n => interr
                                );

        -- instantiate the PWM counter
        cntr:   pwmcntr PORT MAP (
                                clk => clko,
                                rst => status,
                                new_time => new_time,
```

```
                                step => step,
                                new_period => new_period,
                                new_step => new_step
                                );

        -- output the divided clock to control the ADCs
        clkout <= clko;

        -- output the interrupt request signal
        irq_n <= interr;

        -- testpoints
        tp(0) <= new_step;
        tp(1) <= clko;
        tp(2) <= new_period;
        tp(3) <= interr;

END flex6k;
```

D.2 pwmcntr.vhd

```
-- pwmcntr.vhd
-- PWM counter, generates variable length times for each step of the PWM cycle.
-- ge, Feb 2001
--
-- This module loads a count value and counts down until it reaches zero, in
-- that case it increases the step number, generates a 'new_step' signal, loads
-- a new value and starts again. If the step number is 11, then it goes back to 0
-- and generates a 'new_period' signal.
--

LIBRARY altera;
USE altera.maxplus2.ALL;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY work;
USE work.pwmpkg.ALL;

ENTITY pwmcntr IS
        PORT (
                clk:            IN  STD_LOGIC;
                rst:            IN  STD_LOGIC;
                new_time:       IN  timetype;
                step:           OUT steptype;
                new_period:     OUT STD_LOGIC;
                new_step:       OUT STD_LOGIC
        );
END pwmcntr;

ARCHITECTURE flex6k OF pwmcntr IS

BEGIN

        -- there is only one synchronous process
        PROCESS (clk)

                -- variable for the counter
                VARIABLE cnt:   timetype;
                -- variable for the step
                VARIABLE stp:   steptype;
                -- variable to indicate the begining of a new PWM period
                VARIABLE np:    STD_LOGIC;
                -- variable to indicate the begining of a new step
                VARIABLE ns:    STD_LOGIC;

        BEGIN

                IF (clk'EVENT AND clk = '1') THEN

                        IF (rst = '0') THEN

                                -- reset all variables
                                cnt := 0;
                                stp := 0;
```

```
                        np := '0';
                        ns := '0';

                ELSIF (cnt = 0) THEN

                        -- load counter with new value
                        cnt := new_time - 1;

                        -- indicate new step
                        ns := '1';

                        -- increase step number and check boundary condition
                        IF (stp = 11) THEN
                                stp := 0;
                                np := '1';
                        ELSE
                                stp := stp + 1;
                        END IF;

                ELSE

                        -- decrease counter and erase np and ns variables
                        cnt := cnt - 1;
                        np := '0';
                        ns := '0';

                END IF;

        END IF;

        -- connect variables to their respective signals
        step <= stp;
        new_period <= np;
        new_step <= ns;

END PROCESS;

END flex6k;
```

D.3 regs.vhd

```
-- regs.vhd
-- Internal registers to store PWM vectors and duty-cycles for each of them.
-- ge, Feb 2001
--
-- This module provides the following functions:
--
--              1) Registers to store the PWM vectors, the duty-cycles, the dead-band
--              time, the status of the PWM generator and the position of the shaft,
--              all of them values written by the DSP on output ports.
--
--              2) Registers to store the ADC values corresponding to the current
--              measurements of two phases at the end of each one of the six PWM
--              subintervals; this values can be red by the DSP from input ports.
--
--              3) Generation of the PWM output according to the step (subinterval)
--              number.
--
--              4) Generation of the interrupt signal at the end of the PWM period.
--
--              5) Indication of the duration of the next step ('new_time' output),
--              to be used as an input to the 'pwmcntr' module.
--
--      PORT ADDRESSES (as seen from the DSP):
--
--              address         write           read
--              ---------------------------------------
--              0               vector 1        ADC a 1
--              1               vector 2        ADC a 2
--              2               vector 3        ADC a 3
--              3               vector 4        ADC a 4
--              4               vector 5        ADC a 5
--              5               vector 6        ADC a 6
--              6               status          status
```

```
--              7               position        ---
--              8               time 1          ADC b 1
--              9               time 2          ADC b 2
--              A               time 3          ADC b 3
--              B               time 4          ADC b 4
--              C               time 5          ADC b 5
--              D               time 6          ADC b 6
--              E               time db         ---
--              F               ---             ---
--
--      STATUS:
--
--              0       stop
--              1       run
--
--      STEP ORDER:
--
--              step    vector
--              ----------------------
--              0       db = v1 AND v6
--              1       v1
--              2       db = v2 AND v1
--              3       v2
--              4       db = v3 AND v2
--              5       v3
--              6       db = v4 AND v3
--              7       v4
--              8       db = v5 AND v4
--              9       v5
--              10      db = v6 AND v5
--              11      v6
--
--              NOTE: the PWM output is active HIGH
--
--      TIME VALUES (DUTY-CYCLE): the actual time corresponding to each step is equal
--      to N*T where N is the value written in the register and T is the PWM
--  generator clock period (2 times the DSP clock period).
--

LIBRARY altera;
USE altera.maxplus2.ALL;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
LIBRARY work;
USE work.pwmpkg.ALL;

ENTITY regs IS
        PORT (
                clk:                    IN STD_LOGIC;
                rst:                    IN STD_LOGIC;
                ioms_n:                 IN STD_LOGIC;
                rd_n:                   IN STD_LOGIC;
                wr_n:                   IN STD_LOGIC;
                step:                   IN steptype;
                new_step:               IN STD_LOGIC;
                new_period:             IN STD_LOGIC;
                address:                IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                adca:                   IN adctype;
                adcb:                   IN adctype;
                data:                   INOUT STD_LOGIC_VECTOR (15 DOWNTO 0);
                curr_vector:            OUT pwmvec;
                new_time:               OUT timetype;
                position:               OUT postype;
                status:                 OUT STD_LOGIC;
                irq_n:                  OUT STD_LOGIC
        );
END regs;

ARCHITECTURE flex6k OF regs IS

        -- generic write signal
        SIGNAL wrs:             STD_LOGIC;

        -- signals for the value of each vector
        SIGNAL sv1:             pwmvec;
```

```
        SIGNAL sv2:             pwmvec;
        SIGNAL sv3:             pwmvec;
        SIGNAL sv4:             pwmvec;
        SIGNAL sv5:             pwmvec;
        SIGNAL sv6:             pwmvec;

        -- signal to select the appropriate vector at each step
        SIGNAL sel_vec:         pwmvec;

        -- signals for the value of each standby time
        SIGNAL st1:             timevec;
        SIGNAL st2:             timevec;
        SIGNAL st3:             timevec;
        SIGNAL st4:             timevec;
        SIGNAL st5:             timevec;
        SIGNAL st6:             timevec;
        SIGNAL stdb:            timevec;

        -- signals for the value of each active time
        SIGNAL stt1:            timevec;
        SIGNAL stt2:            timevec;
        SIGNAL stt3:            timevec;
        SIGNAL stt4:            timevec;
        SIGNAL stt5:            timevec;
        SIGNAL stt6:            timevec;
        SIGNAL sttdb:           timevec;

        -- status signal and its extended version
        SIGNAL stat:            STD_LOGIC;
        SIGNAL srstat:          STD_LOGIC_VECTOR (15 DOWNTO 0);

        -- signals for the ADC values
        SIGNAL sadca1:          STD_LOGIC_VECTOR (15 DOWNTO 0);
        SIGNAL sadcb1:          STD_LOGIC_VECTOR (15 DOWNTO 0);
        SIGNAL sadca2:          STD_LOGIC_VECTOR (15 DOWNTO 0);
        SIGNAL sadcb2:          STD_LOGIC_VECTOR (15 DOWNTO 0);
        SIGNAL sadca3:          STD_LOGIC_VECTOR (15 DOWNTO 0);
        SIGNAL sadcb3:          STD_LOGIC_VECTOR (15 DOWNTO 0);
        SIGNAL sadca4:          STD_LOGIC_VECTOR (15 DOWNTO 0);
        SIGNAL sadcb4:          STD_LOGIC_VECTOR (15 DOWNTO 0);
        SIGNAL sadca5:          STD_LOGIC_VECTOR (15 DOWNTO 0);
        SIGNAL sadcb5:          STD_LOGIC_VECTOR (15 DOWNTO 0);
        SIGNAL sadca6:          STD_LOGIC_VECTOR (15 DOWNTO 0);
        SIGNAL sadcb6:          STD_LOGIC_VECTOR (15 DOWNTO 0);

        -- enable signal for a memory read
        SIGNAL srden:           STD_LOGIC;

        -- signal for the data to be red
        SIGNAL srdata:          STD_LOGIC_VECTOR (15 DOWNTO 0);

        -- signal for the position
        SIGNAL pos:             postype;

BEGIN

        -- generic write signal (goes to zero when the DSP writes to any port)
        wrs <= wr_n OR ioms_n;

        -- write process
        PROCESS (wrs, rst)

                -- variables to store the vector values
                VARIABLE v1:    pwmvec;
                VARIABLE v2:    pwmvec;
                VARIABLE v3:    pwmvec;
                VARIABLE v4:    pwmvec;
                VARIABLE v5:    pwmvec;
                VARIABLE v6:    pwmvec;

                -- variables to store the standby time values
                VARIABLE t1:    timevec;
                VARIABLE t2:    timevec;
                VARIABLE t3:    timevec;
                VARIABLE t4:    timevec;
                VARIABLE t5:    timevec;
```

```
        VARIABLE t6:    timevec;
        VARIABLE tdb:   timevec;

        -- variable to store the status bit
        VARIABLE sta:   STD_LOGIC;

        -- variable to store the position
        VARIABLE p:     postype;

BEGIN

        IF (rst = '0') THEN

                -- reset the status bit in case of a hardware reset
                sta := '0';

        ELSIF (wrs'EVENT AND wrs = '1') THEN

                -- load the vectors only if the system is not running
                IF (sta = '0' AND address = "0000") THEN
                        v1 := data (5 DOWNTO 0);
                END IF;
                IF (sta = '0' AND address = "0001") THEN
                        v2 := data (5 DOWNTO 0);
                END IF;
                IF (sta = '0' AND address = "0010") THEN
                        v3 := data (5 DOWNTO 0);
                END IF;
                IF (sta = '0' AND address = "0011") THEN
                        v4 := data (5 DOWNTO 0);
                END IF;
                IF (sta = '0' AND address = "0100") THEN
                        v5 := data (5 DOWNTO 0);
                END IF;
                IF (sta = '0' AND address = "0101") THEN
                        v6 := data (5 DOWNTO 0);
                END IF;

                -- load the standby time values
                IF (address = "1000") THEN
                        t1 := data (13 DOWNTO 0);
                END IF;
                IF (address = "1001") THEN
                        t2 := data (13 DOWNTO 0);
                END IF;
                IF (address = "1010") THEN
                        t3 := data (13 DOWNTO 0);
                END IF;
                IF (address = "1011") THEN
                        t4 := data (13 DOWNTO 0);
                END IF;
                IF (address = "1100") THEN
                        t5 := data (13 DOWNTO 0);
                END IF;
                IF (address = "1101") THEN
                        t6 := data (13 DOWNTO 0);
                END IF;
                IF (address = "1110") THEN
                        tdb := data (13 DOWNTO 0);
                END IF;

                -- load the status bit
                IF (address = "0110") THEN
                        sta := data(0);
                END IF;

                -- load the position
                IF (address = "0111") THEN
                        p := data;
                END IF;

        END IF;

        -- connect the signals to the registers
        sv1 <= v1;
        sv2 <= v2;
```

```
        sv3 <= v3;
        sv4 <= v4;
        sv5 <= v5;
        sv6 <= v6;
        st1 <= t1;
        st2 <= t2;
        st3 <= t3;
        st4 <= t4;
        st5 <= t5;
        st6 <= t6;
        stdb <= tdb;
        stat <= sta;
        pos <= p;

END PROCESS;

-- synchronous process
PROCESS (clk)

        -- variable to store the latched vector output
        VARIABLE pwmout:        pwmvec;

        -- variables to store the active times
        VARIABLE tt1:           timevec;
        VARIABLE tt2:           timevec;
        VARIABLE tt3:           timevec;
        VARIABLE tt4:           timevec;
        VARIABLE tt5:           timevec;
        VARIABLE tt6:           timevec;
        VARIABLE ttdb:          timevec;

        -- variables to store the temporary ADC values
        VARIABLE tadca1:        adctype;
        VARIABLE tadcb1:        adctype;
        VARIABLE tadca2:        adctype;
        VARIABLE tadcb2:        adctype;
        VARIABLE tadca3:        adctype;
        VARIABLE tadcb3:        adctype;
        VARIABLE tadca4:        adctype;
        VARIABLE tadcb4:        adctype;
        VARIABLE tadca5:        adctype;
        VARIABLE tadcb5:        adctype;

        -- variables to store the firm ADC values
        VARIABLE adca1:         adctype;
        VARIABLE adcb1:         adctype;
        VARIABLE adca2:         adctype;
        VARIABLE adcb2:         adctype;
        VARIABLE adca3:         adctype;
        VARIABLE adcb3:         adctype;
        VARIABLE adca4:         adctype;
        VARIABLE adcb4:         adctype;
        VARIABLE adca5:         adctype;
        VARIABLE adcb5:         adctype;
        VARIABLE adca6:         adctype;
        VARIABLE adcb6:         adctype;

BEGIN

        IF (clk'EVENT AND clk='1') THEN

                -- latch the PWM output to avoid glitches
                IF (stat = '0') THEN
                        -- system stopped, then output must be zero
                        pwmout := pwm_off;
                ELSE
                        -- latch the selected PWM vector
                        pwmout := sel_vec;
                END IF;

                -- update active times when a PWM period is complete or when the
                --    system is stopped (so that when we write time values they
                --    go directly to the active registers)
                IF (new_period = '1' OR stat = '0') THEN
                        tt1 := st1;
                        tt2 := st2;
```

```
                tt3 := st3;
                tt4 := st4;
                tt5 := st5;
                tt6 := st6;
                ttdb := stdb;
        END IF;

        -- latch the ADC data; at the end of the period copy to the firm bank,
        --     from where the DSP will read them during the next period.
        IF (new_step = '1') THEN
                CASE step IS
                        WHEN 2          =>      tadca1 := adca;
                                                tadcb1 := adcb;
                        WHEN 4          =>      tadca2 := adca;
                                                tadcb2 := adcb;
                        WHEN 6          =>      tadca3 := adca;
                                                tadcb3 := adcb;
                        WHEN 8          =>      tadca4 := adca;
                                                tadcb4 := adcb;
                        WHEN 10         =>      tadca5 := adca;
                                                tadcb5 := adcb;
                        WHEN 0          =>      adca1 := tadca1;
                                                adcb1 := tadcb1;
                                                adca2 := tadca2;
                                                adcb2 := tadcb2;
                                                adca3 := tadca3;
                                                adcb3 := tadcb3;
                                                adca4 := tadca4;
                                                adcb4 := tadcb4;
                                                adca5 := tadca5;
                                                adcb5 := tadcb5;
                                                adca6 := adca;
                                                adcb6 := adcb;
                        WHEN OTHERS     =>
                END CASE;
        END IF;

END IF;

-- connect the output of the current vector
curr_vector <= pwmout;

-- connect the signals for the active times
stt1 <= tt1;
stt2 <= tt2;
stt3 <= tt3;
stt4 <= tt4;
stt5 <= tt5;
stt6 <= tt6;
sttdb <= ttdb;

-- connect the signals for the ADC values (the MSB is inverted to convert
-- to 2's complement, the value is shifted 1 bit to the left and
-- extended to 16 bits with sign extension)
sadca1 (0) <= '0';
sadca1 (13 DOWNTO 1) <= adca1 (12 DOWNTO 0);
sadca1 (14) <= NOT adca1(13);
sadca1 (15) <= NOT adca1(13);
sadcb1 (0) <= '0';
sadcb1 (13 DOWNTO 1) <= adcb1 (12 DOWNTO 0);
sadcb1 (14) <= NOT adcb1(13);
sadcb1 (15) <= NOT adcb1(13);
sadca2 (0) <= '0';
sadca2 (13 DOWNTO 1) <= adca2 (12 DOWNTO 0);
sadca2 (14) <= NOT adca2(13);
sadca2 (15) <= NOT adca2(13);
sadcb2 (0) <= '0';
sadcb2 (13 DOWNTO 1) <= adcb2 (12 DOWNTO 0);
sadcb2 (14) <= NOT adcb2(13);
sadcb2 (15) <= NOT adcb2(13);
sadca3 (0) <= '0';
sadca3 (13 DOWNTO 1) <= adca3 (12 DOWNTO 0);
sadca3 (14) <= NOT adca3(13);
sadca3 (15) <= NOT adca3(13);
sadcb3 (0) <= '0';
sadcb3 (13 DOWNTO 1) <= adcb3 (12 DOWNTO 0);
```

```
        sadcb3 (14) <= NOT adcb3(13);
        sadcb3 (15) <= NOT adcb3(13);
        sadca4 (0) <= '0';
        sadca4 (13 DOWNTO 1) <= adca4 (12 DOWNTO 0);
        sadca4 (14) <= NOT adca4(13);
        sadca4 (15) <= NOT adca4(13);
        sadcb4 (0) <= '0';
        sadcb4 (13 DOWNTO 1) <= adcb4 (12 DOWNTO 0);
        sadcb4 (14) <= NOT adcb4(13);
        sadcb4 (15) <= NOT adcb4(13);
        sadca5 (0) <= '0';
        sadca5 (13 DOWNTO 1) <= adca5 (12 DOWNTO 0);
        sadca5 (14) <= NOT adca5(13);
        sadca5 (15) <= NOT adca5(13);
        sadcb5 (0) <= '0';
        sadcb5 (13 DOWNTO 1) <= adcb5 (12 DOWNTO 0);
        sadcb5 (14) <= NOT adcb5(13);
        sadcb5 (15) <= NOT adcb5(13);
        sadca6 (0) <= '0';
        sadca6 (13 DOWNTO 1) <= adca6 (12 DOWNTO 0);
        sadca6 (14) <= NOT adca6(13);
        sadca6 (15) <= NOT adca6(13);
        sadcb6 (0) <= '0';
        sadcb6 (13 DOWNTO 1) <= adcb6 (12 DOWNTO 0);
        sadcb6 (14) <= NOT adcb6(13);
        sadcb6 (15) <= NOT adcb6(13);

END PROCESS;

-- select PWM vector (before latch) according to step number
WITH step SELECT
        sel_vec <=      (sv1 AND sv6)   WHEN 0,
                        sv1             WHEN 1,
                        (sv2 AND sv1)   WHEN 2,
                        sv2             WHEN 3,
                        (sv3 AND sv2)   WHEN 4,
                        sv3             WHEN 5,
                        (sv4 AND sv3)   WHEN 6,
                        sv4             WHEN 7,
                        (sv5 AND sv4)   WHEN 8,
                        sv5             WHEN 9,
                        (sv6 AND sv5)   WHEN 10,
                        sv6             WHEN 11,
                        pwm_off         WHEN OTHERS;

-- select new_time (time of next step) according to step number
WITH step SELECT
        new_time <=     CONV_INTEGER(UNSIGNED(stt1))    WHEN 0,
                        CONV_INTEGER(UNSIGNED(sttdb))   WHEN 1,
                        CONV_INTEGER(UNSIGNED(stt2))    WHEN 2,
                        CONV_INTEGER(UNSIGNED(sttdb))   WHEN 3,
                        CONV_INTEGER(UNSIGNED(stt3))    WHEN 4,
                        CONV_INTEGER(UNSIGNED(sttdb))   WHEN 5,
                        CONV_INTEGER(UNSIGNED(stt4))    WHEN 6,
                        CONV_INTEGER(UNSIGNED(sttdb))   WHEN 7,
                        CONV_INTEGER(UNSIGNED(stt5))    WHEN 8,
                        CONV_INTEGER(UNSIGNED(sttdb))   WHEN 9,
                        CONV_INTEGER(UNSIGNED(stt6))    WHEN 10,
                        CONV_INTEGER(UNSIGNED(sttdb))   WHEN 11,
                        0                               WHEN OTHERS;

-- signal to read the status bit
srstat(0)               <= stat;
srstat(15 DOWNTO 1)     <= "000000000000000";

-- read enable signal (goes to zero when the DSP reads any port)
srden <= rd_n OR ioms_n;

-- select the data to be red
WITH address SELECT
        srdata <=       sadca1 WHEN "0000",
                        sadca2 WHEN "0001",
                        sadca3 WHEN "0010",
                        sadca4 WHEN "0011",
                        sadca5 WHEN "0100",
                        sadca6 WHEN "0101",
```

```
                                srstat WHEN "0110",
                                sadcb1 WHEN "1000",
                                sadcb2 WHEN "1001",
                                sadcb3 WHEN "1010",
                                sadcb4 WHEN "1011",
                                sadcb5 WHEN "1100",
                                sadcb6 WHEN "1101",
                                "0000000000000000" WHEN OTHERS;

        -- implement tristate buffer for the data bus
        data <= srdata WHEN srden = '0' ELSE
                "ZZZZZZZZZZZZZZZZ";

        -- interrupt request when a period is complete (must be edge sensitive)
        irq_n <= NOT new_period;

        -- output status signal, used to reset the counter
        status <= stat;

        -- output position, used for evaluation of the performance
        position <= pos;

END flex6k;
```

D.4 pwmpkg.vhd

```
-- pwmpkg.vhd
-- Package for the PWM generation design.
-- ge, Feb 2001

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY altera;
USE altera.maxplus2.ALL;

PACKAGE pwmpkg IS

        SUBTYPE timetype        IS INTEGER RANGE 0 TO 16383; -- 14 bits
        SUBTYPE timevec         IS STD_LOGIC_VECTOR (13 DOWNTO 0); -- 14 bits
        SUBTYPE pwmvec          IS STD_LOGIC_VECTOR (5 DOWNTO 0); -- 6 bits
        SUBTYPE steptype        IS INTEGER RANGE 0 TO 15; -- 4 bits
        SUBTYPE adctype         IS STD_LOGIC_VECTOR (13 DOWNTO 0); -- 14 bits
        SUBTYPE postype         IS STD_LOGIC_VECTOR (15 DOWNTO 0); -- 16 bits

        CONSTANT pwm_off:       pwmvec := "000000";

        COMPONENT pwmcntr
                PORT (
                        clk:            IN  STD_LOGIC;
                        rst:            IN  STD_LOGIC;
                        new_time:       IN  timetype;
                        step:           OUT steptype;
                        new_period:     OUT STD_LOGIC;
                        new_step:       OUT STD_LOGIC
                );
        END COMPONENT;

        COMPONENT regs
                PORT (
                        clk:            IN STD_LOGIC;
                        rst:            IN STD_LOGIC;
                        ioms_n:         IN STD_LOGIC;
                        rd_n:           IN STD_LOGIC;
                        wr_n:           IN STD_LOGIC;
                        step:           IN steptype;
                        new_step:       IN STD_LOGIC;
                        new_period:     IN STD_LOGIC;
                        address:        IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                        adca:           IN adctype;
                        adcb:           IN adctype;
                        data:           INOUT STD_LOGIC_VECTOR (15 DOWNTO 0);
                        curr_vector:    OUT pwmvec;
                        new_time:       OUT timetype;
                        position:       OUT postype;
```

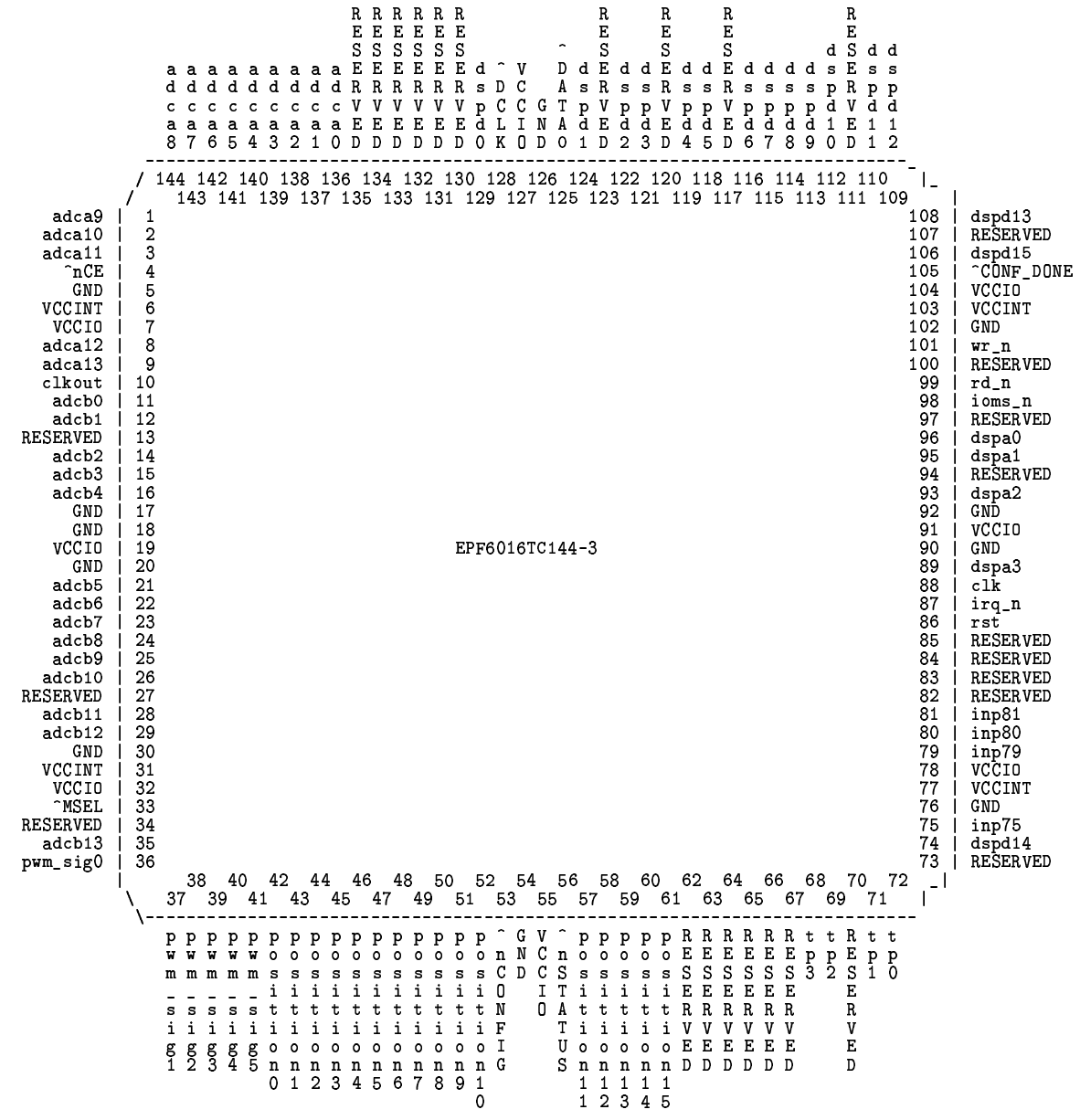
```
                    status:         OUT STD_LOGIC;
                    irq_n:          OUT STD_LOGIC
            );
    END COMPONENT;

END pwmpkg;
```

D.5 Compilation report (edited version)

```
Project Information                                   c:\gabriel\pwmfpga\pwm.rpt

MAX+plus II Compiler Report File
Version 9.6 3/22/2000
Compiled: 06/27/2001 11:02:23

Copyright (C) 1988-2000 Altera Corporation


***** Project compilation was successful

PWM


** DEVICE SUMMARY **

Chip/                    Input   Output   Bidir                          LCs
POF       Device         Pins    Pins     Pins      LCs    % Utilized

pwm       EPF6016TC144-3   41       28       16      992         75 %

User Pins:                 41       28       16


** PIN/LOCATION/CHIP ASSIGNMENTS **

                Actual
    User      Assignments
Assignments   (if different)     Node Name

pwm@136                          adca0
pwm@137                          adca1
pwm@138                          adca2
pwm@139                          adca3
pwm@140                          adca4
pwm@141                          adca5
pwm@142                          adca6
pwm@143                          adca7
pwm@144                          adca8
pwm@1                            adca9
pwm@2                            adca10
pwm@3                            adca11
pwm@8                            adca12
pwm@9                            adca13
pwm@11                           adcb0
pwm@12                           adcb1
pwm@14                           adcb2
pwm@15                           adcb3
pwm@16                           adcb4
pwm@21                           adcb5
pwm@22                           adcb6
pwm@23                           adcb7
pwm@24                           adcb8
pwm@25                           adcb9
pwm@26                           adcb10
pwm@28                           adcb11
pwm@29                           adcb12
pwm@35                           adcb13
pwm@88                           clk
pwm@10                           clkout
pwm@96                           dspa0
pwm@95                           dspa1
pwm@93                           dspa2
pwm@89                           dspa3
pwm@129                          dspd0
```

```
pwm@124                         dspd1
pwm@122                         dspd2
pwm@121                         dspd3
pwm@119                         dspd4
pwm@118                         dspd5
pwm@116                         dspd6
pwm@115                         dspd7
pwm@114                         dspd8
pwm@113                         dspd9
pwm@112                         dspd10
pwm@110                         dspd11
pwm@109                         dspd12
pwm@108                         dspd13
pwm@74                          dspd14
pwm@106                         dspd15
pwm@75                          inp75
pwm@79                          inp79
pwm@80                          inp80
pwm@81                          inp81
pwm@98                          ioms_n
pwm@87                          irq_n
pwm@42                          position0
pwm@43                          position1
pwm@44                          position2
pwm@45                          position3
pwm@46                          position4
pwm@47                          position5
pwm@48                          position6
pwm@49                          position7
pwm@50                          position8
pwm@51                          position9
pwm@52                          position10
pwm@57                          position11
pwm@58                          position12
pwm@59                          position13
pwm@60                          position14
pwm@61                          position15
pwm@36                          pwm_sig0
pwm@37                          pwm_sig1
pwm@38                          pwm_sig2
pwm@39                          pwm_sig3
pwm@40                          pwm_sig4
pwm@41                          pwm_sig5
pwm@99                          rd_n
pwm@86                          rst
pwm@72                          tp0
pwm@71                          tp1
pwm@69                          tp2
pwm@68                          tp3
pwm@101                         wr_n


** FILE HIERARCHY **

|regs:reg|
|pwmcntr:cntr|
|pwmcntr:cntr|lpm_add_sub:148|
|pwmcntr:cntr|lpm_add_sub:148|addcore:adder|
|pwmcntr:cntr|lpm_add_sub:148|altshift:result_ext_latency_ffs|
|pwmcntr:cntr|lpm_add_sub:148|altshift:carry_ext_latency_ffs|
|pwmcntr:cntr|lpm_add_sub:148|altshift:oflow_ext_latency_ffs|
|pwmcntr:cntr|lpm_add_sub:177|
|pwmcntr:cntr|lpm_add_sub:177|addcore:adder|
|pwmcntr:cntr|lpm_add_sub:177|altshift:result_ext_latency_ffs|
|pwmcntr:cntr|lpm_add_sub:177|altshift:carry_ext_latency_ffs|
|pwmcntr:cntr|lpm_add_sub:177|altshift:oflow_ext_latency_ffs|
|pwmcntr:cntr|lpm_add_sub:247|
|pwmcntr:cntr|lpm_add_sub:247|addcore:adder|
|pwmcntr:cntr|lpm_add_sub:247|altshift:result_ext_latency_ffs|
|pwmcntr:cntr|lpm_add_sub:247|altshift:carry_ext_latency_ffs|
|pwmcntr:cntr|lpm_add_sub:247|altshift:oflow_ext_latency_ffs|


***** Logic for device 'pwm' compiled without errors.
```

```
Device: EPF6016TC144-3

FLEX 6000 Configuration Scheme: Passive Serial

Device Options:
    User-Supplied Start-Up Clock               = OFF
    Auto-Restart Configuration on Frame Error  = OFF
    Release Clears Before Tri-States           = OFF
    Enable Chip_Wide Reset                     = OFF
    Enable Chip-Wide Output Enable             = OFF
    Enable INIT_DONE Output                    = OFF
    Enable JTAG Support                        = OFF
    MultiVolt I/O                              = OFF
```



```
N.C. = No Connect. This pin has no internal connection to the device.
VCCINT = Dedicated power pin, which MUST be connected to VCC (5.0 volts).
VCCIO = Dedicated power pin, which MUST be connected to VCC (5.0 volts).
GND = Dedicated ground pin or unused dedicated input, which MUST be connected to GND.
RESERVED = Unused I/O pin, which MUST be left unconnected.

^ = Dedicated configuration pin.
+ = Reserved configuration pin, which is tri-stated during user mode.
* = Reserved configuration pin, which drives out in user mode.
PDn = Power Down pin.
@ = Special-purpose pin.
# = JTAG Boundary-Scan Testing/In-System Programming or Configuration Pin. The JTAG inputs TMS and TDI should be tied t
& = JTAG pin used for I/O. When used as user I/O, JTAG pins must be kept stable before and during configuration.  JTAG


Total dedicated input pins used:                1/4      ( 25%)
Total I/O pins used:                           84/113    ( 74%)
Total logic cells used:                       992/1320   ( 75%)
Average fan-in:                                3.50/4    ( 87%)
Total fan-in:                                 3476/5280   ( 65%)

Total input pins required:                      41
Total output pins required:                     28
Total bidirectional pins required:              16
Total reserved pins required                     0
Total logic cells required:                    992
Total flipflops required:                      349
Total packed registers required:                 0
Total logic cells in carry chains:               0
Total number of carry chains:                    0
Total logic cells in cascade chains:             0
Total number of cascade chains:                  0
Logic cells inserted for fitting:                1

Synthesized logic cells:                       157/1320   ( 11%)


** COMPILATION SETTINGS & TIMES **

Processing Menu Commands
------------------------

Design Doctor                             = off

Logic Synthesis:

   Synthesis Type Used                    = Multi-Level

   Default Synthesis Style                = NORMAL

      Logic option settings in 'NORMAL' style for 'FLEX6000' family

      CARRY_CHAIN                         = ignore
      CARRY_CHAIN_LENGTH                  = 32
      CASCADE_CHAIN                       = ignore
      CASCADE_CHAIN_LENGTH                = 2
      DECOMPOSE_GATES                     = on
      DUPLICATE_LOGIC_EXTRACTION          = on
      MINIMIZATION                        = full
      MULTI_LEVEL_FACTORING               = on
      NOT_GATE_PUSH_BACK                  = on
      REDUCE_LOGIC                        = on
      REFACTORIZATION                     = on
      REGISTER_OPTIMIZATION               = on
      RESYNTHESIZE_NETWORK                = on
      SLOW_SLEW_RATE                      = off
      SUBFACTOR_EXTRACTION                = on
      IGNORE_SOFT_BUFFERS                 = on
      USE_LPM_FOR_AHDL_OPERATORS          = off

   Other logic synthesis settings:

      Automatic Global Clock              = on
      Automatic Global Clear              = on
```

```
      Automatic Global Preset             = on
      Automatic Global Output Enable      = on
      Automatic Fast I/O                  = off
      Automatic Register Packing          = off
      Automatic Open-Drain Pins           = on
      Automatic Implement in EAB          = off
      Optimize                            = 5

Default Timing Specifications: None

Cut All Bidir Feedback Timing Paths       = on
Cut All Clear & Preset Timing Paths       = on

Ignore Timing Assignments                 = off

Functional SNF Extractor                  = off

Linked SNF Extractor                      = off
Timing SNF Extractor                      = on
Optimize Timing SNF                       = off
Generate AHDL TDO File                    = off
Fitter Settings                           = NORMAL
Smart Recompile                           = off
Total Recompile                           = off

Interfaces Menu Commands
------------------------

EDIF Netlist Writer                       = off
Verilog Netlist Writer                    = off
VHDL Netlist Writer                       = off

Compilation Times
-----------------

   Compiler Netlist Extractor             00:00:05
   Database Builder                       00:00:03
   Logic Synthesizer                      00:00:46
   Partitioner                            00:00:02
   Fitter                                 00:00:32
   Timing SNF Extractor                   00:00:03
   Assembler                              00:00:01
   --------------------------             --------
   Total Time                             00:01:32


Memory Allocated
----------------

Peak memory allocated during compilation  = 24,896K
```

Appendix E

Inverter design

This appendix describes the design and construction of an inverter for 3-phase AC motors. It consists basically of a power module with six IGBTs, gate drives and protection included. Proper electrical isolation has been provided for the PWM signals. Three current sensors have been added to measure the line currents. The schematics and PCB layout plots of this design are included.

E.1 Introduction

The inverter described in this appendix has been designed and constructed between Febraury and May of 2000, with the purpose of being integrated to the experimental test-bed of the Power Electronics and Motion Control Systems Laboratory at Northeastern University. In particular, this inverter is used in experiments on sensorless control for Permanent Magnet Synchronous Motors.

The power module selected can drive up to 15A and 600V. However, the limitations

of the DC power supply (200V, 11A) impose these limits. Therefore, the maximum power that this inverter can transfer to the load is specified as 2kW. With a different DC power supply, higher values could be achieved, but protection circuits should be studied carefuly, because the simple snubber which is used in this design could not be effective in more stressful conditions.

Figure E.1 shows a block diagram of the inverter. The thick lines show the path followed by the high currents. Three independent power supplies have been used to minimize noise interference.

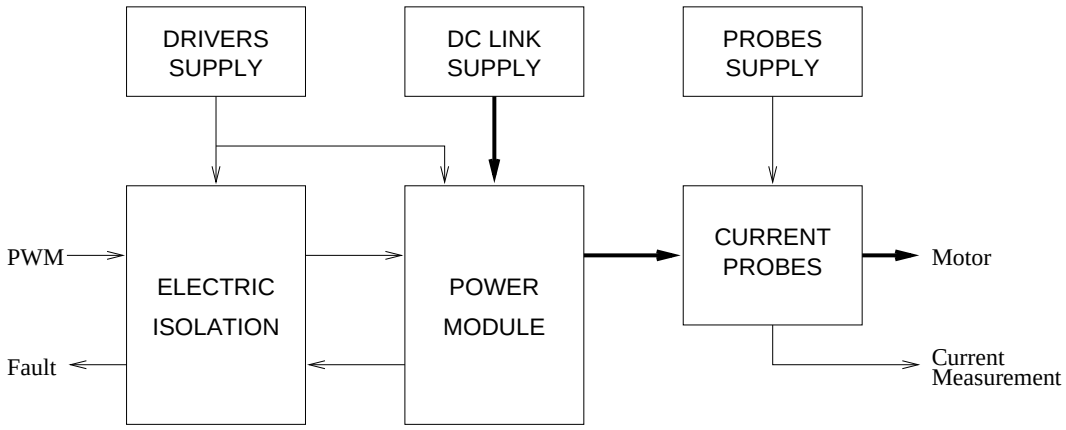


Figure E.1: Block diagram of the inverter

The design has been simplified by the use of a power module, which integrates in the same package the three legs of the inverter, their associated gate drive circuits and additional protection circuitry. This module and the associated components are described in section E.2.

The PWM signals that control the inverter, have been isolated from the digital section by using optocouplers. The same consideration has been taken for the fault

signals generated by the power module, which indicate the activation of internal protection. Section E.3 describes this circuitry and an additional enable/disable switch which has proven to be very useful during the experiments.

For the measurement of the line currents, three current probes with their independent power source were introduced in the design. The output of these probes is converted from current to voltage using precission resistances. This is described in section E.4.

In section E.5 we present the design of the Printed Circuit Board, while section E.6 provides some details about the construction and testing of the whole drive.

More technical information is provided in sections E.7, E.8 and E.9 with the schematics, PCB layout plots and connectors pin list respectively.

E.2 The power module

The power module selected was the PM20CSJ060 (U12) from Powerex (Mitsubishi). It contains a complete three-phase IGBT inverter with the gate driver circuit included, plus a protection circuit which protects against short circuit, over current, over temperature and under voltage.

The power supply of the gate driver circuitry is provided by the M57140-01 (U1), from the same manufacturer. It provides four isolated outputs of 15V: one for each of the upper IGBTS and one for the three lower ones. The outputs are isolated from the 20V input, which is provided by a conventional DC power supply.

The DC link voltage is provided by a programmable voltage source, HP6575A from Hewlett Packard, which can provide as much as 200V/11A.

Due to the relative low power of the design, it was enough to use a single capacitor as a snubber, in order to control the transient voltages. For that reason, a low-inductane capacitor of .47 μF (C6) was placed between the P and N terminals.

The heat sink

The main parameter for the selection of the heat sink is the thermal resistance. In order to find an adequate value for this parameter, an estimation of the power dissipated by the power module and a thermal model are needed.

The power dissipated by the module can be classified in four groups:

- IGBT conduction losses
- IGBT switching losses
- Free-wheel diode conduction losses
- Free-wheel diode recovery losses

For the thermal calculation the model in fig. E.2 is used. The power is equivalent to a current, the temperature to a voltage, and the relation between both is called the thermal resistance. T_j is the temperature in the junction of the semiconductor, T_c is the temperature in the case of the module, T_f is the temperature in the fins of the power sink and T_a is the ambient temperature. The thermal resistances represent the ability of the heat to flow in a material interface; the values of R_{jc} and R_{cf} are given by the manufacturer and the only parameter we can change is R_{fa}, by selecting the heat sink. Therefore, the objective is to find a suitable value for R_{fa} that makes T_j less than 150^oC, which is the maximum junction temperature admissible.

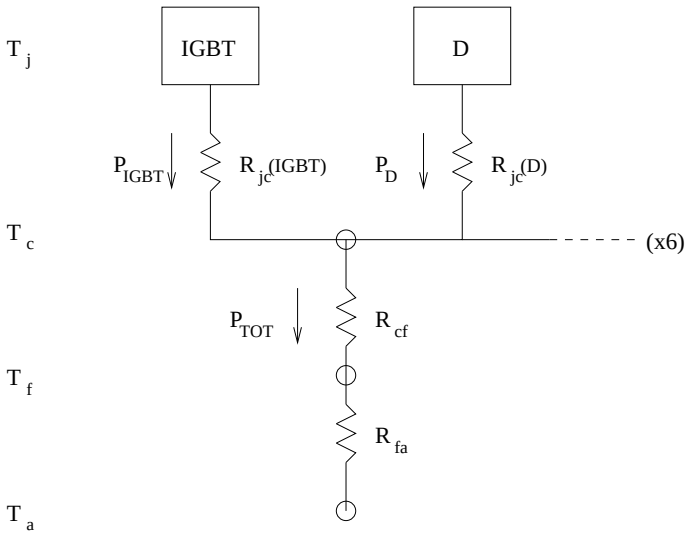


Figure E.2: Thermal calculation model

After some approximations, we obtain an estimation of the thermal resistance needed.For practical purposes, a heat sink with a thermal resistance of $.5^oC/W$ was selected, for which the joint temperature is less than 150^oC even under extremely conservative assumptions.

The heat sink selected with this thermal resistance was the model 392-120AB from Wakefield Engineering.

E.3 The digital interface

The optocouplers HCPL-2211 (U2 to U7) provide electric isolation of the digital PWM signals. The inputs of the power module are active low, but we desired to have active high PWM inputs. Therefore, inversion of the signal is achieved in the optocouplers, since the conduction of the diode is possible when the input is in the low state.

A .1 μF bypass capacitor is placed between VCC and GND on the output of the

optocouplers to filter undesired high frequency noise.

An additional enable/disable switch (SW1) was added in order to provide a quick and convenient way of deactivate all the inputs of the power module at the same time. As we said before, the PWM signals that are fed into the board, are active high, then the enable/disable function is being done by the AND function (U16 and U17). The switch SW1, when in position 2, generates a high value in the input of the AND gates, who are enabled to copy the input to the output. On the other hand, when the switch is in position 3, the resistor R16 forces a low value at the input of the AND gates, and therefore a constant low value at their output. Finally, the LEDs D1 (red) and D2 (green) provide visual information of the switch state.

The fault signals generated by the power module can be used to monitor the functioning of the inverter, in particular to see if a protection has been activated. For that reason it has been considered important to provide a feedback route to this signals in two possible ways: on one hand connecting them to the same connector that brings the digital PWM signals (J5), and on the other hand connecting them to a dedicated connector (J6) which can be used for example to provide visual information. Being the fault outputs of the power module of the open collector type, with a limited sink current, the diodes of the optocouplers U8 to U11 have been connected between the fault output and their correspondent VCC. The output of these optocouplers is a phototransistor, for that reason they have been connected in a common-emmiter-like configuration.

E.4 The current probes

The current probes selected were CLN-25 from F.W.Bell (U13 to U15). Since the maximum current we were proposed to work with was 11A, in order to maximize the resolution the measuring range selected was 12A, which implies a connection with 2 turns of the current. The turn ratio in this configuration is 2/1000 and an output current up to 24 mA.

To convert the current output to a voltage value, precisor resistances of 200Ω (R11 to R13) were used. Additionally, small capacitors of 10nF (C11 to C13) in parallel provide filtering of very high frequencies. The voltage range of the output has a maximum of 24mA x 200Ω = 4.8V.

An independent power supply of ±15V was needed for the current probes. The bypass capacitors C7 to C10 provide stabilization of the supply voltages.

E.5 Design of the PCB

The schematics drawing was made using an evaluation software (MicroSim), given the relative simplicity of the design.

The Printed Circuit Board (PCB) was manufactured by the company ExpressPCB (www.expresspcb.com). The design of the PCB was made using the software provided by this manufacturer, which is very simple and easy to use. However, this software is not suitable for complex designs, because it doesn't provide many important CAD tools like Design Rules Check and connectivity with a schematics file.

In this design, the size of the board was not optimized. Instead, we concentrated in the layout of the components and the path followed by the most critical routes.

The traces between the optocouplers and the power module inputs were kept as short and straight as possible. Decoupling capacitors were put close to all optocouplers and power sources.

The high-current traces have been designed wide enough to avoid overheating, and with smooth curves whenever possible.

All connectors and the switch were placed on the edge of the board. In general the inputs are on the left and the outputs on the right.

E.6 Mounting and testing

First, the traces of the PCB were tested using a multimeter. Special attention was paid to the power traces and all the connections to the power module.

The components were soldered to the PCB. IC sockets were used for the DIP packages (optocouplers, AND gates and line receiver). The heat sink was mounted over the power module using a silicon compound in the surface contact.

The PCB was mounted on a hardwood board. The two independent power sources (+20VDC for the gate driver and ±15VDC for the current probes) were mounted in the same board. The AC part of both power sources were connected together to a cable with a plug suitable for a standard 110VAC socket.

Without inserting the optocouplers in their sockets, a PWM signal was injected and their polarity was tested, in particular it was checked that no two transistors on the same leg were on at the same time. Then the optocouplers were inserted and a small DC link voltage was applied to the power module, without connecting the motor yet; the 3-phase voltage outputs were checked in this operation condition.

The motor was connected and an open loop control was implemented to make it spin at different speeds. The current measurements were checked.

Finally, a closed loop control was implemented. The results were satisfactory and the inverter was ready to be used in the experiments.

E.7 Schematics

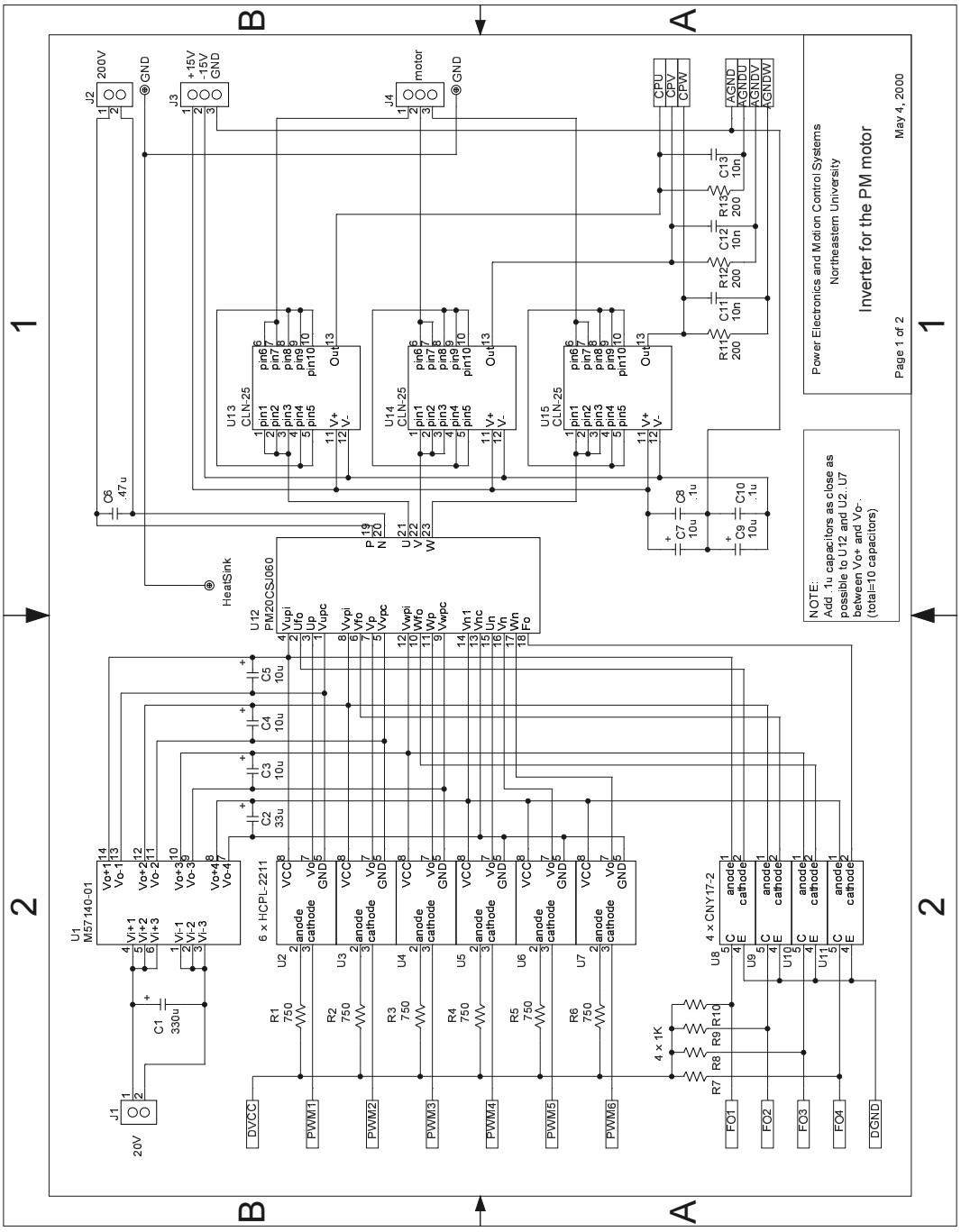


Figure E.3: Inverter schematics (1 of 2)

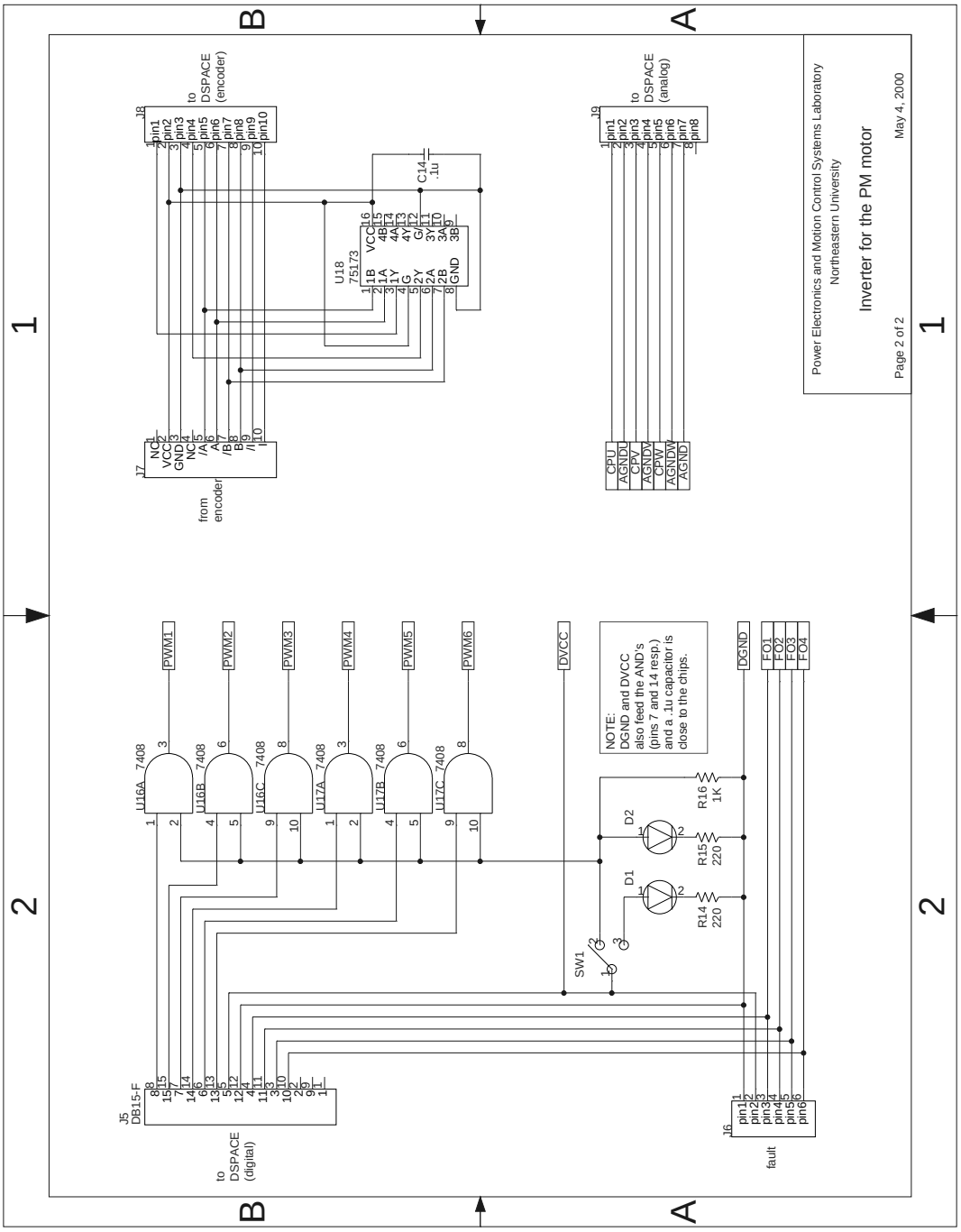


Figure E.4: Inverter schematics (2 of 2)

E.8 Printed Circuit Board

Note: the plots are not to scale.

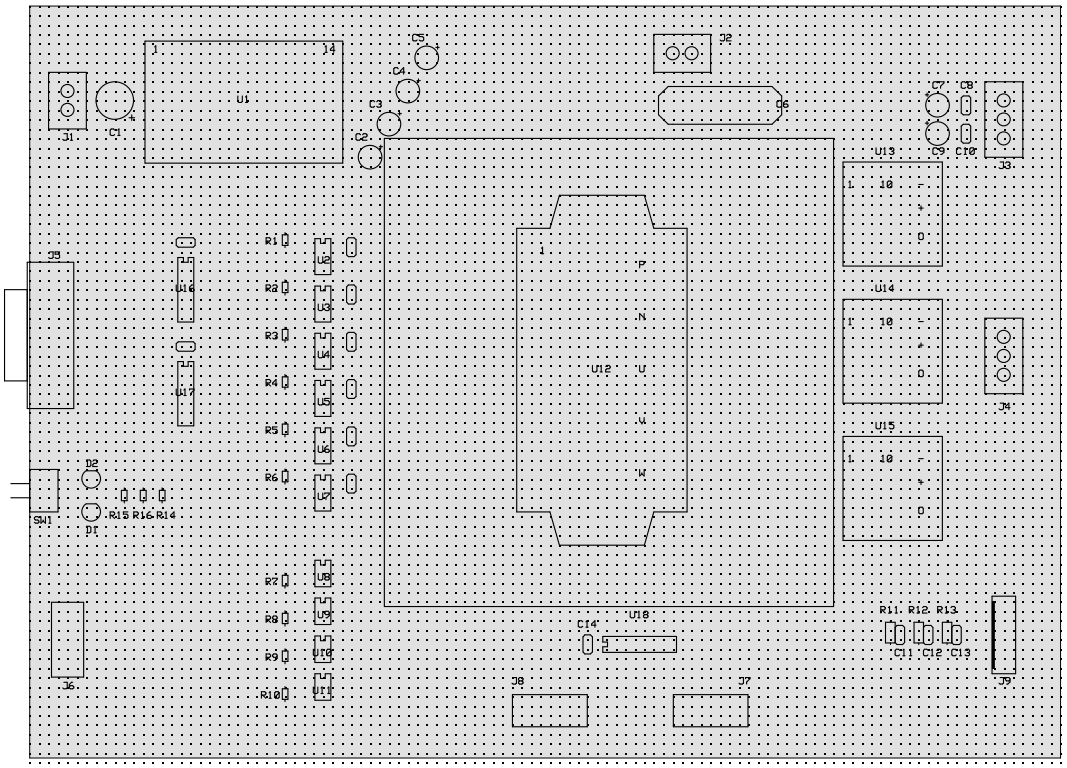
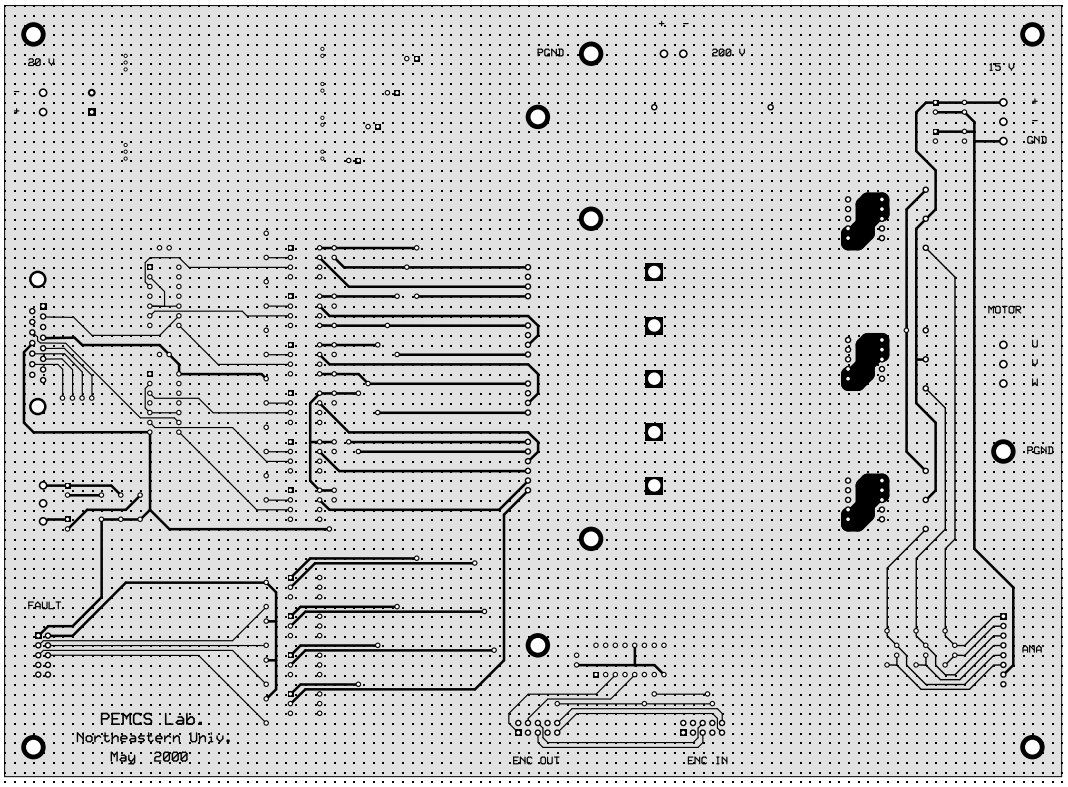
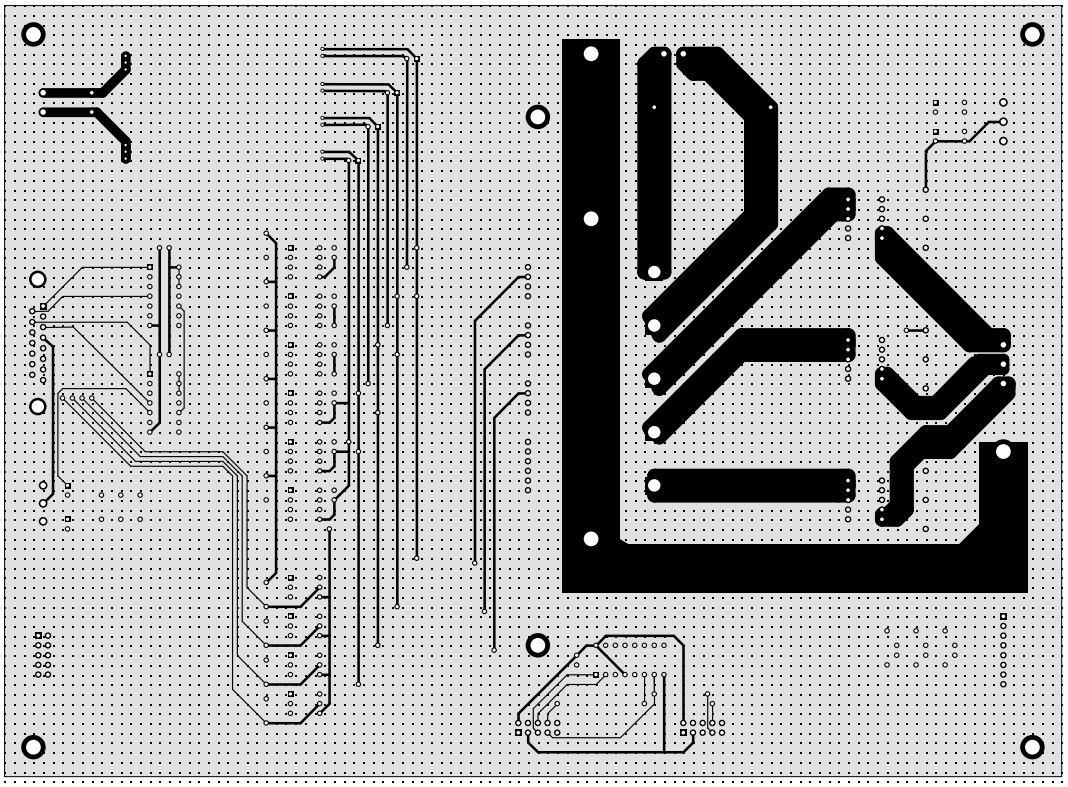


Figure E.5: Inverter PCB (silkscreen)



C:\Program Files\ExpressPCB\inverter.pcb (Top layer)

Figure E.6: Inverter PCB (top layer)



C:\Program Files\ExpressPCB\inverter.pcb (Bottom layer)

Figure E.7: Inverter PCB (bottom layer)

E.9 Connectors

DB15 pin	PCB name
8	PWM1
14	PWM4
15	PWM2
6	PWM5
7	PWM3
13	PWM6
4	FO1
11	FO2
3	FO3
10	FO4
5	DVCC
12	DGND

Table E.1: PWM connectors

HDR pin	PCB name
1	CPU
2	AGNDU
3	CPV
4	AGNDV
5	CPW
6	AGNDW
7	AGND

Table E.2: Current probes connector