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

Thesis Co-Advisor: Prof. Gilead Tadmor

Thesis Reader: Prof. Miriam Leeser

Thesis Reader: Prof. Brad Lehman

Department Chair: Prof. Fabrizio Lombardi

Graduate School Notified of Acceptance:

Director of the Graduate School: Prof. Yaman Yener
“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

Abstract i

Acknowledgments ii

1 Introduction 1

2 The position estimation algorithm for PMSM 4

2.1 Permanent Magnet Synchronous Motors (PMSM) ............... 4

2.1.1 PMSM description ........................................ 4

2.1.2 PMSM model in the αβ axes ............................. 6

2.1.3 PMSM model in the dq axes ............................... 7

2.2 Position estimation algorithm .................................. 8

2.2.1 PWM pattern .............................................. 9

2.2.2 Model discretization ....................................... 12

2.2.3 Parameter estimation ....................................... 14

2.2.4 Mechanical states observer ................................. 16

2.3 Control algorithm ............................................. 17

2.4 Summary ..................................................... 18
3 Hardware design

3.1 General description ........................................ 21
3.2 Signal acquisition ........................................... 23
3.3 The FPGA ..................................................... 25
3.4 The DSP ....................................................... 27
3.5 Construction of the prototype ............................... 28
3.6 Summary ...................................................... 30

4 Word-length effects ........................................... 31

4.1 Overview of the algorithm and implementation issues .... 31
4.2 Scaling ......................................................... 33
4.2.1 Scaling the inputs ....................................... 34
4.2.2 Constructing vector x .................................. 36
4.2.3 Computing estimated parameters ....................... 39
4.2.4 Observer .................................................. 41
4.2.5 Controller ............................................... 43
4.2.6 Counter values ......................................... 47
4.3 Simulations ................................................... 49
4.4 DSP code ...................................................... 49
4.5 Summary ...................................................... 50

5 PWM implementation using FPGA ............................. 52

5.1 General description ......................................... 52
5.2 PWM counter block ........................................ 54
5.3 Registers block ............................................. 55
5.3.1 Interface with the DSP .............................. 56
5.3.2 PWM signals generation ............................. 58
5.3.3 Signal acquisition .................................. 59
5.4 Practical considerations ................................. 60
5.5 Summary ............................................. 62

6 Experimental results ................................. 63
6.1 Experimental setup .................................. 63
6.2 Results ............................................. 66
6.3 Summary ............................................. 67

7 Conclusions ............................................ 71

Bibliography ............................................. 73

A Hardware schematics and PCB ......................... 78

B Matlab files for simulations .............................. 85
B.1 Motor model ........................................ 85
B.1.1 stpmsm.m ....................................... 85
B.2 Floating-point version ............................... 86
B.2.1 simsless.m ....................................... 86
B.2.2 constsl.m ....................................... 88
B.2.3 sless.m .......................................... 90
B.3 Fixed-point version .................................. 93
B.3.1 simslq.m ....................................... 93
B.3.2 constslq.m ........................................ 95
B.3.3 slq.m ........................................ 97
B.3.4 frac16.m .................................... 102
B.3.5 alu.m ........................................ 102
B.3.6 mac.m ........................................ 103

C Assembler code for the DSP .................................. 104
C.1 final.asm ........................................ 104
C.2 sine.asm ...................................... 116
C.3 test.ldf ..................................... 116

D VHDL code for the FPGA ................................... 118
D.1 pwm.vhd ........................................ 118
D.2 pwmcntr.vhd ................................... 120
D.3 regs.vhd ...................................... 121
D.4 pwmpkg.vhd ................................... 128
D.5 Compilation report (edited version) .................. 129

E Inverter design ............................................... 134
E.1 Introduction ...................................... 134
E.2 The power module .................................. 136
E.3 The digital interface .............................. 138
E.4 The current probes ............................... 140
E.5 Design of the PCB ................................. 140
E.6 Mounting and testing ............................. 141
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.7</td>
<td>Schematics</td>
<td>143</td>
</tr>
<tr>
<td>E.8</td>
<td>Printed Circuit Board</td>
<td>145</td>
</tr>
<tr>
<td>E.9</td>
<td>Connectors</td>
<td>148</td>
</tr>
</tbody>
</table>
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
CHAPTER 1. INTRODUCTION

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 pro-
posed 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].
CHAPTER 2. THE POSITION ESTIMATION ALGORITHM FOR PMSM

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.
CHAPTER 2. THE POSITION ESTIMATION ALGORITHM FOR PMSM

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]

\[
\begin{align*}
\mathbf{v} &= \mathbf{R} \mathbf{i} + \mathbf{L} \frac{d\mathbf{i}}{dt} + \omega \frac{d\lambda_r}{d\theta} \\
J \frac{d\omega}{dt} &= \tau_m - B \omega - \tau_i \\
\frac{d\theta}{dt} &= \omega
\end{align*}
\]

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 \( \lambda_r = [\lambda_{r\alpha}, \lambda_{r\beta}]^T \) is the vector of stator phase fluxes due to
the rotor field. The resistance and inductance matrices can be written as

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

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

The load torque is $\tau_l$ and the torque produced by the motor is given by

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

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 $\alpha$ and the $d$ axis equals to $\theta$, the rotation angle of the rotor measured in electrical degrees. In this frame, the PMSM model is [21]

$$
L_d \frac{di_d}{dt} = \omega L_q i_q - R i_d + v_d
$$

$$
L_q \frac{di_q}{dt} = -\omega L_d i_d - R i_q - \omega \Phi + v_q
$$

$$
J \frac{d\omega}{dt} = (\Phi + \Delta L i_d) i_q - B \omega - \tau_l
$$

$$
\frac{d\theta}{dt} = \omega
$$

(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 $\Phi$ 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 ($V_{DC}$) 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].

![PWM schematic and voltage vectors](image)

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 $V_{DC}$, 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.

![Symmetric PWM diagram](image)

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
(V₁...V₆) are generated sequentially (Fig. 2.3). The duty ratio of the subintervals are computed as

\[
\begin{align*}
\zeta_1 &= \frac{1}{6} + \frac{5\sqrt{3}}{12} \frac{v_\alpha}{v_{max}} - \frac{7}{12} \frac{v_\beta}{v_{max}} \\
\zeta_2 &= \frac{1}{6} - \frac{\sqrt{3}}{12} \frac{v_\alpha}{v_{max}} + \frac{11}{12} \frac{v_\beta}{v_{max}} \\
\zeta_{3,4,5,6} &= \frac{1}{6} - \frac{\sqrt{3}}{12} \frac{v_\alpha}{v_{max}} - \frac{1}{12} \frac{v_\beta}{v_{max}}
\end{align*}
\]

(2.3)

where the duty ratios are defined as \(\zeta_i = \frac{\xi_i}{T_{PWM}}\), \(\zeta_1\) is the duty ratio of the subinterval corresponding to the lag vector, \(\zeta_2\) 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

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

(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

$$\mathbf{v}_n = \mathbf{R}\frac{i_{n+1} + i_n}{2} + L\frac{i_{n+1} - i_n}{t_{n+1} - t_n} + \omega\frac{d\lambda_r}{d\theta}$$

(2.5)

for $n = 1, 2, \ldots N$. 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 $i_n$ 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

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

(2.6)
Figure 2.4: Current and voltage waveforms in one PWM period.

for \( n = 1, 2, \ldots N - 1 \).

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 \( \omega \)) 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

\[
\mathbf{w}_n = \mathbf{Lx}_n
\]
where
\[
\begin{align*}
\mathbf{w}_n &= \mathbf{v}_{n+1} - \mathbf{v}_n \\
\mathbf{x}_n &= \left( \frac{i_{n+2} - i_{n+1}}{t_{n+2} - t_{n+1}} - \frac{i_{n+1} - i_n}{t_{n+1} - t_n} \right)
\end{align*}
\]

### 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 \emph{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 \( \theta \) 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
\[
\mathbf{q} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \end{bmatrix} = \frac{1}{L_0^2 - L_1^2} \begin{bmatrix} L_0 \\ -L_1 \cos(2\theta) \\ -L_1 \sin(2\theta) \end{bmatrix} \tag{2.9}
\]
CHAPTER 2. THE POSITION ESTIMATION ALGORITHM FOR PMSM

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 \( \alpha \) and a \( \beta \) component, so we can write

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

for \( n = 1, 2, \ldots, N - 1 \).

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

\[
\mathbf{x} = \mathbf{W} \mathbf{q}
\]  

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

\[
\hat{\mathbf{q}} = \arg \min \| \mathbf{W} \mathbf{q} - \mathbf{x} \|^2
\]

is

\[
\hat{\mathbf{q}} = \left( \mathbf{W}^T \mathbf{W} \right)^{-1} \mathbf{W}^T \mathbf{x} = \mathbf{W}_{ps} \mathbf{x}
\]

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 $W_{pq}$ 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 $x$.

2.2.4 Mechanical states observer

Once the parameter vector $q$ 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

\[
\dot{\Theta} = \gamma_3 \epsilon \tag{2.14}
\]

\[
\dot{\omega} = \Theta + \gamma_2 \epsilon \tag{2.15}
\]

\[
\dot{\alpha} = \omega + \gamma_1 \epsilon \tag{2.16}
\]

where $\dot{\Theta}$, $\dot{\omega}$ and $\dot{\alpha}$ are the estimates of the rotor position, speed and acceleration,
respectively, $\gamma_i$ are design parameters, and $\epsilon$ 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

$$\epsilon = q_2 \cos(2\tilde{\theta}) - q_1 \sin(2\tilde{\theta}) \approx L_1 \sin(2\tilde{\theta})$$  \hspace{1cm} (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 $\gamma_i$. 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.
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 $i_a$ and $i_b$. 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 $c$. 
CHAPTER 2. THE POSITION ESTIMATION ALGORITHM FOR PMSM

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 100 mA, therefore we looked for a resolution of about 1 mA. Since the dynamic range of the phase currents is ±12 A, 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 200 Ω 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 ±12 A, the voltage $v_{cp}$ is limited by $\pm 12 A \cdot \frac{2}{1000} 200 \Omega = \pm 4.8 V$.

The AD9240 works with a single +5 V power supply, therefore we need a signal
CHAPTER 3. HARDWARE DESIGN

conditioning stage to adapt the input \( v_{cp} \) to a suitable range for the ADC. We set the voltage reference \( V\,REF \) to +2.5V, the middle-point of the power supply, to have a symmetric span. The conversion range is therefore ±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 \( V\,REF \), while the other is fed by the desired signal plus a DC component equal to \( V\,REF \). 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.

![Diagram of the ADC configuration](image)

Figure 3.2: Schematics of the ADC configuration.
CHAPTER 3. HARDWARE DESIGN

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].
CHAPTER 3. HARDWARE DESIGN

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 .1μF 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.
CHAPTER 3. HARDWARE DESIGN

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).
CHAPTER 3. HARDWARE DESIGN

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 ±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.

![Prototype board](image)

Figure 3.3: Prototype board (size=6 × 3.5").
CHAPTER 3. HARDWARE DESIGN

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.
CHAPTER 4. WORD-LENGTH EFFECTS

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 $200\Omega$. 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

\[
\begin{align*}
    i_\alpha &= \sqrt{\frac{3}{2}} i_a \\
    i_\beta &= \frac{1}{\sqrt{2}} i_a + \sqrt{2} i_b
\end{align*}
\] (4.1)

In a worst-case scenario, the maximum value of $i_\alpha$ is $\sqrt{\frac{3}{2}} \approx 1.22$, and the maximum value of $i_\beta$ 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

\[
\begin{align*}
  i_\alpha &= \left\lceil \frac{\sqrt{\frac{3}{2}}}{2} \right\rceil [2i_a] \\
  i_\beta &= \left\lceil \frac{1}{2\sqrt{2}} \right\rceil [2i_a] + \left\lceil \frac{\sqrt{2}}{2} \right\rceil [2i_b]
\end{align*}
\]

(4.2)

To convert to fractional notation, we divide by \(i_{a,\beta,\text{max}}\)

\[
\begin{align*}
  \tilde{i}_\alpha &= \left\lceil \frac{\sqrt{\frac{3}{2}}}{2} \right\rceil \frac{i_a}{i_{a,\beta,\text{max}}} \\
  \tilde{i}_\beta &= \left\lceil \frac{1}{2\sqrt{2}} \right\rceil \frac{i_a}{i_{a,\beta,\text{max}}} + \left\lceil \frac{\sqrt{2}}{2} \right\rceil \frac{i_b}{i_{a,\beta,\text{max}}}
\end{align*}
\]

(4.3)

Finally, by defining \(i_{a,\beta,\text{max}} = \frac{i_{a,\beta,\text{max}}}{2} = 25\), we can write the transformation as

\[
\begin{align*}
  \tilde{i}_\alpha &= \left\lceil \frac{\sqrt{\frac{3}{2}}}{2} \right\rceil \tilde{i}_a \\
  \tilde{i}_\beta &= \left\lceil \frac{1}{2\sqrt{2}} \right\rceil \tilde{i}_a + \left\lceil \frac{\sqrt{2}}{2} \right\rceil \tilde{i}_b
\end{align*}
\]

(4.4)

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

We can obtain \(\tilde{i}_a\) and \(\tilde{i}_b\) 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 \(\pm 12.5\,A\). 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 \(\tilde{i}_a\) 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 \( \mathbf{x} \)

We recall that

\[
x_n = \frac{i_{n+2} - i_{n+1}}{t_{n+2} - t_{n+1}} - \frac{i_{n+1} - i_n}{t_{n+1} - t_n}
\]

(4.5)

where the \( \alpha \) and \( \beta \) 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 \( \tilde{d}_n \) the result and notice that the scaling factor is the same as the \( \alpha \beta \) currents, i.e., \( i_{\alpha/\beta,\text{max}} \).

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
CHAPTER 4. WORD-LENGTH EFFECTS

expressed in seconds is

\[ dt_n = T_{CK} \tilde{d}t_n \]  \hspace{1cm} (4.6)

where \( T_{CK} \) is the counter clock period (i.e., 100ns), and \( \tilde{d}t_n \) 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 \( \frac{di}{dt} \approx \frac{1}{T} V \). Since \( V_{\text{max}} \approx 1.2V_{\text{BUS}} \approx 240V \) and \( L_{\text{min}} \approx 3mH \), then we conclude that \( \left( \frac{di}{dt} \right)_{\text{max}} \approx 8 \times 10^4 A/s \).

On the other hand, the slope value, which we will call \( \text{didt}_n \), can be computed as

\[ \text{didt}_n = \frac{di}{dt} = \frac{i_{\alpha\beta,\text{max}} \tilde{d}i_n}{T_{CK} dt_n} = 5 \times 10^8 A/s \frac{\tilde{d}i_n}{dt_n} \]  \hspace{1cm} (4.7)

and since we know the dynamic range of \( \text{didt}_n \), we can conclude that the division \( \frac{\tilde{d}i_n}{dt_n} \) will have a dynamic range of \( \pm \frac{8 \times 10^4}{5 \times 10^8} = \pm 1.6 \times 10^{-4} \).

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 pre-scale the value \( \tilde{d}i_n \) so that the result has a dynamic range close to \((-1,1)\). For this reason, we compute \( \text{didt}_n = \frac{2^{12} \tilde{d}i_n}{dt_n} \); now, the result will have a dynamic range of \( \pm 0.66 \), which is very convenient for a representation in 1.15 format.

The scaling of the output can be found as follows:

\[ \text{didt}_n = \frac{2^{12} \tilde{d}i_n}{dt_n} = \frac{2^{12} T_{CK} di_n}{i_{\alpha\beta,\text{max}} dt_n} = 8.19 \times 10^{-6} A \frac{di_n}{dt_n} \]  \hspace{1cm} (4.8)

therefore, \( \text{didt}_{\text{max}} = \frac{1}{8.19 \times 10^{-6} A/s} = 1.22 \times 10^5 A/s \).

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 \( \tilde{d}_n \) in 1.15 format to the value \( 2^{12} \tilde{d}_n \) 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 \( \text{didt}_n \). We know that the dynamic range of the scaled slope values is \( \pm 0.66 \) 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

\[
\bar{x}_n = \frac{1}{2} \text{didt}_{n+1} - \frac{1}{2} \text{didt}_n
\]

(4.9)

which implies that the scaling factor of the elements of \( \mathbf{x} \) is given by \( \bar{x}_{\text{max}} = 2 \text{didt}_{\text{max}} = 2.44 \times 10^5 \text{A/s} \).

One last comment about vector \( \mathbf{x} \): in this section we have omitted the subindices \( \alpha \) and \( \beta \), however we must specify the order in which the elements of \( \mathbf{x} \) are stacked.
The vector is constructed as

$$\mathbf{x} = \begin{bmatrix} x_{\alpha,1} \\ x_{\beta,1} \\ x_{\alpha,2} \\ x_{\beta,2} \\ \vdots \\ x_{\alpha,5} \\ x_{\beta,5} \end{bmatrix}$$  \hspace{1cm} (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

$$\hat{\mathbf{q}} = \mathbf{W}_{pi} \mathbf{x}$$  \hspace{1cm} (4.11)

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

<table>
<thead>
<tr>
<th>subinterval</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_\alpha )</td>
<td>( \frac{2}{\sqrt{6}} )</td>
<td>( \frac{1}{\sqrt{6}} )</td>
<td>(-\frac{1}{\sqrt{6}} )</td>
<td>(-\frac{2}{\sqrt{6}} )</td>
<td>(-\frac{1}{\sqrt{6}} )</td>
<td>( \frac{1}{\sqrt{6}} )</td>
</tr>
<tr>
<td>( v_\beta )</td>
<td>0</td>
<td>( \frac{1}{\sqrt{2}} )</td>
<td>( \frac{1}{\sqrt{2}} )</td>
<td>0</td>
<td>(-\frac{1}{\sqrt{2}} )</td>
<td>(-\frac{1}{\sqrt{2}} )</td>
</tr>
</tbody>
</table>

Table 4.1: Inverter \( \alpha/\beta \) voltages in each subinterval.
CHAPTER 4. WORD-LENGTH EFFECTS

We recall that

\[
W = \begin{bmatrix}
    w_{\alpha,1} & w_{\alpha,1} & w_{\beta,1} \\
    w_{\beta,1} & -w_{\beta,1} & w_{\alpha,1} \\
    w_{\alpha,5} & w_{\alpha,5} & w_{\beta,5} \\
    w_{\beta,5} & -w_{\beta,5} & w_{\alpha,5}
\end{bmatrix}
\]  

(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,

\[
W = V \cdot \begin{bmatrix}
    -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\
    \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} \\
    -\frac{2}{\sqrt{6}} & -\frac{2}{\sqrt{6}} & 0 \\
    0 & 0 & -\frac{2}{\sqrt{6}}
\end{bmatrix}
\]  

(4.13)

and

\[
W_{pi} = \frac{1}{V} \cdot \begin{bmatrix}
    -\frac{\sqrt{2}}{32} & \frac{5\sqrt{2}}{32} & -\frac{2\sqrt{2}}{32} \\
    -\frac{15\sqrt{6}}{330} & -\frac{29\sqrt{2}}{330} & \frac{\sqrt{2}}{32} \\
    -\frac{\sqrt{2}}{32} & -\frac{2\sqrt{2}}{32} & -\frac{6\sqrt{2}}{32} \\
    -\frac{5\sqrt{2}}{32} & \frac{3\sqrt{2}}{32} & \frac{\sqrt{2}}{32} \\
    -\frac{45\sqrt{2}}{320} & -\frac{11\sqrt{6}}{320} & -\frac{9\sqrt{2}}{320} \\
    -\frac{33\sqrt{6}}{320} & -\frac{54\sqrt{2}}{320} & -\frac{22\sqrt{6}}{320} \\
    -\frac{11\sqrt{6}}{320} & -\frac{15\sqrt{6}}{330} & -\frac{45\sqrt{2}}{320} \\
    -\frac{9\sqrt{2}}{320} & \frac{33\sqrt{6}}{320}
\end{bmatrix}
\]  

(4.14)

where \( V = V_{BUS} = 200V \).

We observe that \( \max \{|W_{pi}|_{i,j}\} = 1.33 \times 10^{-3} \), so we define \( w_{\max} \) equal to this
value and $\tilde{W}_{pi} = \frac{1}{w_{\text{max}}} W_{pi}$. Therefore, we can write

$$q = W_{pi} \mathbf{x} = w_{\text{max}} x_{\text{max}} \tilde{W}_{pi} \tilde{\mathbf{x}} \approx 324 \tilde{W}_{pi} \tilde{\mathbf{x}}$$  \hspace{1cm} (4.15)

This equation implies 10 multiply-accumulate operations for each element of $q$, 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 $q_{\text{max}} = 324$, then $\tilde{q} = \frac{q}{q_{\text{max}}} = \tilde{W}_{pi} \tilde{\mathbf{x}}$. The computation of the elements of $\tilde{q}$ 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 ($\dot{\alpha}$, $\dot{\omega}$ and $\dot{\theta}$ respectively), based on the observer error $\epsilon$, and the computation of an updated observer error, based on the parameter and position estimates ($q$ and $\theta$ respectively).

For convenience, we will start by studying the error update

$$\epsilon = q_3 \cos 2\dot{\theta} - q_2 \sin 2\dot{\theta}$$  \hspace{1cm} (4.16)

In order to improve the precision, we will define $\epsilon_{\text{max}} = \frac{q_{\text{max}}}{2} = 162$, so we can use the values $2\tilde{q}_2$ and $2\tilde{q}_3$, which have a better resolution, extracting them from the
MAC result by shifting the result to the left 1 bit. Therefore

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

(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

$$\dot{\alpha} = \gamma_3 \epsilon$$

(4.18)

which can be discretized as

$$\hat{\alpha} := \dot{\alpha} + T_{\text{PWM}} \gamma_3 \epsilon$$

(4.19)

We estimate the maximum value of the acceleration from the peak torque as

$$\alpha_{\text{max}} \approx \frac{T_{\text{L, peak}}}{H} \approx 10,000 \text{rad/s}. \text{ Therefore}$$

$$\bar{\alpha} = \frac{\dot{\alpha}}{\alpha_{\text{max}}} := \bar{\alpha} + \frac{T_{\text{PWM}} \gamma_3 \epsilon_{\text{max}}}{\alpha_{\text{max}}} \epsilon$$

(4.20)

We define the new coefficient $$\tilde{\gamma}_3 = \frac{T_{\text{PWM}} \gamma_3 \epsilon_{\text{max}}}{\alpha_{\text{max}}}$$.

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

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

(4.21)

where $$\omega_{\text{max}} = 4,500 \text{rpm} = 1,800 \text{rad/s}$$ and we define $$a_{\text{coeff}} = \frac{T_{\text{PWM}} \epsilon_{\text{max}}}{\omega_{\text{max}}}$$, and

$$\tilde{\gamma}_2 = \frac{T_{\text{PWM}} \gamma_2 \epsilon_{\text{max}}}{\omega_{\text{max}}}.$$  

Finally, a similar expression is obtained for the position

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

(4.22)
where \( \theta_{\max} = \pi \) and we define \( s_{\text{coeff}} = \frac{T_{PWM}}{\theta_{\max}} \omega_{\max} \), and \( \gamma_1 = \frac{T_{PWM}}{\theta_{\max}} \frac{\gamma_1}{\epsilon_{\max}} \). 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 \( \gamma_1 \), \( \gamma_2 \) and \( \gamma_3 \) 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

\[
i_{av} = \sum_{k=1}^{6} \frac{i_{k-1} + i_k}{2} \frac{dt_k}{T_{PWM}} = \frac{i_{\alpha/\beta,\max}}{T_{PWM}} \frac{T_{CK}}{2^{15}} \sum_{k=1}^{6} \left( \frac{i_{k-1}}{2} + \frac{i_k}{2} \right) \frac{d\tilde{t}_k}{2^{15}}
\]

where the sum \( \frac{i_{k-1}}{2} + \frac{i_k}{2} \) will not overflow and \( \frac{d\tilde{t}_k}{2^{15}} \) is the counter value of each subinterval represented in fractional format.

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

\[
i_{\tilde{av}} = \frac{i_{av}}{i_{av,\max}} = 4 \sum_{k=1}^{6} \left( \frac{i_{k-1}}{2} + \frac{i_k}{2} \right) \frac{d\tilde{t}_k}{2^{15}}
\]
CHAPTER 4. WORD-LENGTH EFFECTS

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

\[
\begin{align*}
\tilde{i}_d &= i_{a,av} \cos \hat{\theta} + i_{\beta,av} \sin \hat{\theta} \\
\tilde{i}_q &= -i_{a,av} \sin \hat{\theta} + i_{\beta,av} \cos \hat{\theta}
\end{align*}
\]  

(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 \( k \frac{x+a}{s} \). A discrete version of these blocks uses one variable as the memory of the integrator, and can be expressed as

\[
\begin{align*}
u &= x_{ref} - x \\
\text{int} &= \text{int} + T_{PW} \cdot u \\
y &= k \cdot u + k \cdot a \cdot \text{int}
\end{align*}
\]  

(4.26)

where \( x \) is the input, \( u \) is the intermediate error, \( \text{int} \) 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 \( \text{int}_{\text{max}} \) such that the scaling of \( k \cdot a \cdot \text{int} \) 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

\[
\bar{u}_1 = i_{d,\text{ref}} - \bar{i}_d \\
\text{int}_1 := \int \text{int}_1 + \frac{i_{dq,\text{max}}}{\text{int}_{1,\text{max}}} T_{PWM} \bar{u}_1 \\
v_{dc} = \frac{k_1}{v_{dq,\text{max}}} \bar{u}_1 + \frac{k_1 a_1}{v_{dq,\text{max}}} \text{int}_1
\]

(4.27)

where \( v_{dq,\text{max}} = 2V_{\text{BUS}} = 400V \) and \( \text{int}_{1,\text{max}} = \frac{v_{dq,\text{max}}}{k_1 a_1} = 0.2963 \). The coefficient of \( \text{int}_1 \) in the output equation equals to 1 because of the selection of its scaling.

The block PI2 can be written as

\[
\bar{u}_2 = \omega_{\text{ref}} - \bar{\omega} \\
\text{int}_2 := \int \text{int}_2 + \frac{\omega_{\text{max}}}{\text{int}_{2,\text{max}}} \bar{u}_2 \\
i_{q,\text{ref}} = \frac{k_2}{i_{dq,\text{max}}} \omega_{\text{max}} \bar{u}_2 + \frac{k_2 a_2}{i_{dq,\text{max}}} \text{int}_2
\]

(4.28)
In this case, if we select $\text{int}_{2,\text{max}}$ as before, we would have a very small coefficient $$\frac{\omega_{\text{max}} T_{\text{PWM}}}{\text{int}_{\text{3, max}}} \approx 6 \times 10^{-4}$$ and this would generate steady-state errors due to the low precision of the coefficient. Therefore, we select a smaller value $\text{int}_{2,\text{max}} = \frac{i_{q,\text{max}}}{4 k_2 a_2} = 145.84$ so that the coefficient becomes $2.468 \times 10^{-3}$ 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

$$\begin{align*}
\tilde{u}_3 &= \tilde{i}_{q,\text{ref}} - \tilde{i}_q \\
\text{int}_3 &= \tilde{i}_3 + \frac{i_{d,\text{max}}}{\text{int}_{3,\text{max}}} \tilde{u}_3 \\
\tilde{v}_{qc} &= \frac{k_3 i_{d,\text{max}}}{v_{dq,\text{max}}} \tilde{u}_3 + \frac{k_3 a_3 i_{\text{int}_{3,\text{max}}}}{v_{dq,\text{max}}} \tilde{i}_3
\end{align*}$$

(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

$$\begin{align*}
\tilde{v}_\alpha &= \tilde{v}_{dc} \cos \hat{\theta}_c - \tilde{v}_{qc} \sin \hat{\theta}_c \\
\tilde{v}_\beta &= \tilde{v}_{dc} \sin \hat{\theta}_c + \tilde{v}_{qc} \cos \hat{\theta}_c
\end{align*}$$

(4.30)

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

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

$$\hat{\theta}_c = \hat{\theta} + 2 T_{PWM} \omega \hat{\omega}$$

$$\Rightarrow \tilde{\theta}_c = \tilde{\theta} + \frac{2 T_{PWM} \omega_{max}}{\theta_{max}} \tilde{\omega}$$  \hspace{1cm} (4.31)

\section{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.
CHAPTER 4. WORD-LENGTH EFFECTS

To identify the sector, we first identify the quadrant and then we find out the sector by comparing \( \frac{v_a}{v_o} \) 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 \( v_\beta = \sqrt{3} \left( \frac{V_{BUS}}{\sqrt{2}} - v_a \right) \). If \( v_\beta \) is greater than this value, then the vector is scaled down by a factor \( \frac{1}{v_a + v_\beta} \). 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 \( \sqrt{3} \frac{V_{BUS}}{8} \) (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

\[
\frac{\sqrt{3}}{12 \ v_{\text{max}}} v_a = \frac{1}{\sqrt{6}} \ \bar{v}_a \\
\frac{1}{12 \ v_{\text{max}}} v_\beta = \frac{\sqrt{2}}{6} \ \bar{v}_\beta
\]

(4.32)

given that \( v_{\text{max}} = \frac{V_{BUS}}{\sqrt{2}} \).

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 overshoot 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 20\(MHZ\) 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.
CHAPTER 5. PWM IMPLEMENTATION USING FPGA

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.

![Block diagram of the FPGA design.](image)

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
CHAPTER 5. PWM IMPLEMENTATION USING FPGA

synchronize the system.

The clock signal generated by the DSP is divided by two to obtain a 10 MHz
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 pwmcnt.r vh 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 in-
verter 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.

![Diagram](image)

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

![Diagram](image)

**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 \((t1, \ldots, t6, tdb)\), and one bank for the time values of the next period \((t1, \ldots, t6, tdb)\). There is one bank of registers to store the PWM vectors to be used in each subinterval \((v1, \ldots, v6)\). Register `sta` stores the status of the system.
(stop or running), register \( p \) stores the estimated position of the shaft, and register \( \text{pwm}_{\text{out}} \) 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 \( \text{tadca}1, \text{tadcb}1, \ldots, \text{tadca}6, \text{tadcb}6 \) and at the end of the PWM period copied to another bank \( \text{adca}1, \text{adcb}1, \ldots, \text{adca}6, \text{adcb}6 \).

![Diagram of internal registers](image)

**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 \texttt{ioms.n} signal is asserted by the DSP
when an I/O access is being done; the signals \( \text{wr}_n \) and \( \text{rd}_n \) distinguish a write from a read operation.

The write operation is implemented in a process whose sensitive list includes \( \text{wrs} = \text{wr}_n \text{ OR } \text{ioms}_n \). 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 \( \text{srden} = \text{rd}_n \text{ OR } \text{ioms}_n \). When this signal is 0, the value \( \text{srdata} \) is connected to data bus; otherwise a high impedance output is forced. The values that can be read on the \( \text{srdata} \) 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 \( \text{irq}_n \) is the inverse of the \( \text{new\_period} \) signal. In this way, at the end of a PWM period \( \text{irq}_n \) will go to zero for one clock cycle; this is enough because the interruption that we are using in the DSP is edge sensitive,
CHAPTER 5. PWM IMPLEMENTATION USING FPGA

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

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
CHAPTER 5. PWM IMPLEMENTATION USING FPGA

<table>
<thead>
<tr>
<th>Name</th>
<th>8.3us</th>
<th>8.4us</th>
<th>8.5us</th>
<th>8.6us</th>
<th>8.7us</th>
<th>8.8us</th>
<th>8.9us</th>
<th>9.0us</th>
<th>9.1us</th>
</tr>
</thead>
<tbody>
<tr>
<td>[I] clk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[O] clkout</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[I] rst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[I] wr_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[I] rd_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[I] ioms_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[I] dspa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[I] dspd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[O] dspd</td>
<td>ZZZZ</td>
<td>0450</td>
<td>E450</td>
<td>D450</td>
<td>Z450</td>
<td>D450</td>
<td>E450</td>
<td>0430</td>
<td>E450</td>
</tr>
<tr>
<td>[I] adca</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[I] adcb</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[O] pwm_sig</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>[O] position</td>
<td>5555</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[O] irq_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Read ADC values stored from previous period (one wait state assumed)

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.
CHAPTER 5. PWM IMPLEMENTATION USING FPGA

<table>
<thead>
<tr>
<th>step</th>
<th>subinterval</th>
<th>PWM vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DB</td>
<td>v6 AND v1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>v1</td>
</tr>
<tr>
<td>2</td>
<td>DB</td>
<td>v1 AND v2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>v2</td>
</tr>
<tr>
<td>4</td>
<td>DB</td>
<td>v2 AND v3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>v3</td>
</tr>
<tr>
<td>6</td>
<td>DB</td>
<td>v3 AND v4</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>v4</td>
</tr>
<tr>
<td>8</td>
<td>DB</td>
<td>v4 AND v5</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>v5</td>
</tr>
<tr>
<td>10</td>
<td>DB</td>
<td>v5 AND v6</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>v6</td>
</tr>
</tbody>
</table>

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., 4µs).

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
**CHAPTER 5. PWM IMPLEMENTATION USING FPGA**

<table>
<thead>
<tr>
<th>Name</th>
<th>1.75us</th>
<th>1.8us</th>
<th>1.85us</th>
<th>1.9us</th>
<th>1.95us</th>
<th>2.0us</th>
<th>2.05us</th>
<th>2.1us</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td><img src="image" alt="clk" /></td>
<td><img src="image" alt="clk" /></td>
<td><img src="image" alt="clk" /></td>
<td><img src="image" alt="clk" /></td>
<td><img src="image" alt="clk" /></td>
<td><img src="image" alt="clk" /></td>
<td><img src="image" alt="clk" /></td>
<td><img src="image" alt="clk" /></td>
</tr>
<tr>
<td>clkout</td>
<td><img src="image" alt="clkout" /></td>
<td><img src="image" alt="clkout" /></td>
<td><img src="image" alt="clkout" /></td>
<td><img src="image" alt="clkout" /></td>
<td><img src="image" alt="clkout" /></td>
<td><img src="image" alt="clkout" /></td>
<td><img src="image" alt="clkout" /></td>
<td><img src="image" alt="clkout" /></td>
</tr>
<tr>
<td>rst</td>
<td><img src="image" alt="rst" /></td>
<td><img src="image" alt="rst" /></td>
<td><img src="image" alt="rst" /></td>
<td><img src="image" alt="rst" /></td>
<td><img src="image" alt="rst" /></td>
<td><img src="image" alt="rst" /></td>
<td><img src="image" alt="rst" /></td>
<td><img src="image" alt="rst" /></td>
</tr>
<tr>
<td>wr_n</td>
<td><img src="image" alt="wr_n" /></td>
<td><img src="image" alt="wr_n" /></td>
<td><img src="image" alt="wr_n" /></td>
<td><img src="image" alt="wr_n" /></td>
<td><img src="image" alt="wr_n" /></td>
<td><img src="image" alt="wr_n" /></td>
<td><img src="image" alt="wr_n" /></td>
<td><img src="image" alt="wr_n" /></td>
</tr>
<tr>
<td>rd_n</td>
<td><img src="image" alt="rd_n" /></td>
<td><img src="image" alt="rd_n" /></td>
<td><img src="image" alt="rd_n" /></td>
<td><img src="image" alt="rd_n" /></td>
<td><img src="image" alt="rd_n" /></td>
<td><img src="image" alt="rd_n" /></td>
<td><img src="image" alt="rd_n" /></td>
<td><img src="image" alt="rd_n" /></td>
</tr>
<tr>
<td>ioms_n</td>
<td><img src="image" alt="ioms_n" /></td>
<td><img src="image" alt="ioms_n" /></td>
<td><img src="image" alt="ioms_n" /></td>
<td><img src="image" alt="ioms_n" /></td>
<td><img src="image" alt="ioms_n" /></td>
<td><img src="image" alt="ioms_n" /></td>
<td><img src="image" alt="ioms_n" /></td>
<td><img src="image" alt="ioms_n" /></td>
</tr>
<tr>
<td>dspa</td>
<td><img src="image" alt="dspa" /></td>
<td><img src="image" alt="dspa" /></td>
<td><img src="image" alt="dspa" /></td>
<td><img src="image" alt="dspa" /></td>
<td><img src="image" alt="dspa" /></td>
<td><img src="image" alt="dspa" /></td>
<td><img src="image" alt="dspa" /></td>
<td><img src="image" alt="dspa" /></td>
</tr>
<tr>
<td>dspd</td>
<td><img src="image" alt="dspd" /></td>
<td><img src="image" alt="dspd" /></td>
<td><img src="image" alt="dspd" /></td>
<td><img src="image" alt="dspd" /></td>
<td><img src="image" alt="dspd" /></td>
<td><img src="image" alt="dspd" /></td>
<td><img src="image" alt="dspd" /></td>
<td><img src="image" alt="dspd" /></td>
</tr>
<tr>
<td>adca</td>
<td><img src="image" alt="adca" /></td>
<td><img src="image" alt="adca" /></td>
<td><img src="image" alt="adca" /></td>
<td><img src="image" alt="adca" /></td>
<td><img src="image" alt="adca" /></td>
<td><img src="image" alt="adca" /></td>
<td><img src="image" alt="adca" /></td>
<td><img src="image" alt="adca" /></td>
</tr>
<tr>
<td>adcb</td>
<td><img src="image" alt="adcb" /></td>
<td><img src="image" alt="adcb" /></td>
<td><img src="image" alt="adcb" /></td>
<td><img src="image" alt="adcb" /></td>
<td><img src="image" alt="adcb" /></td>
<td><img src="image" alt="adcb" /></td>
<td><img src="image" alt="adcb" /></td>
<td><img src="image" alt="adcb" /></td>
</tr>
<tr>
<td>pwm_sig</td>
<td><img src="image" alt="pwm_sig" /></td>
<td><img src="image" alt="pwm_sig" /></td>
<td><img src="image" alt="pwm_sig" /></td>
<td><img src="image" alt="pwm_sig" /></td>
<td><img src="image" alt="pwm_sig" /></td>
<td><img src="image" alt="pwm_sig" /></td>
<td><img src="image" alt="pwm_sig" /></td>
<td><img src="image" alt="pwm_sig" /></td>
</tr>
<tr>
<td>position</td>
<td><img src="image" alt="position" /></td>
<td><img src="image" alt="position" /></td>
<td><img src="image" alt="position" /></td>
<td><img src="image" alt="position" /></td>
<td><img src="image" alt="position" /></td>
<td><img src="image" alt="position" /></td>
<td><img src="image" alt="position" /></td>
<td><img src="image" alt="position" /></td>
</tr>
<tr>
<td>irq_n</td>
<td><img src="image" alt="irq_n" /></td>
<td><img src="image" alt="irq_n" /></td>
<td><img src="image" alt="irq_n" /></td>
<td><img src="image" alt="irq_n" /></td>
<td><img src="image" alt="irq_n" /></td>
<td><img src="image" alt="irq_n" /></td>
<td><img src="image" alt="irq_n" /></td>
<td><img src="image" alt="irq_n" /></td>
</tr>
</tbody>
</table>

**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 adcb0 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.
CHAPTER 5. PWM IMPLEMENTATION USING FPGA

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.
CHAPTER 6. EXPERIMENTAL RESULTS

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.

![Experimental setup](image)

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:
CHAPTER 6. EXPERIMENTAL RESULTS

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 $1\text{Nm}$, ($\approx 141\text{inoz}$) while the speed reference changed abruptly from 50 to $500\text{rad/s}$ (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 $500\text{rad/s}$ (electrical) with a constant torque of $1\text{Nm}$. 
Figure 6.3: Speed step response, position estimation error in rad (electrical).

A better performance can be achieved if we use a less aggressive 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 1 Nm and the speed remained constant at 200 rad/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
state, the position estimation error is about $.25\text{rad}$ (electrical), while the error reported for the floating-point version is $.15\text{rad}$ [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.
CHAPTER 6. EXPERIMENTAL RESULTS

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

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


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 PMM, suited for an ODE solver.
    %
    % state vector:
    %  [i alpha; 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 Rs Lq L1m

global Jm Bm P

ial = x(1);
ibt = x(2);
em = x(3);
APPENDIX B. MATLAB FILES FOR SIMULATIONS

th = x(4);
phid = pdm*sin(6*th) + pd12m*sin(12*th);
phiq = pqm + pqm*cos(6*th) + pq12m*cos(12*th);
phial = phid*cos(th) - phiq*sin(th);
phibt = phid*sin(th) + phiq*cos(th);

lia1dot = val - Rm*ial - om*(2*Lm*(-ial*sin(2*th)+ibt*cos(2*th))+phiial);
libtdot = vbt - Rm*ibt - om*(2*Lm*(ial*cos(2*th)+ibt*sin(2*th))+phiibt);
nia1dot = (lia1dot*(Lqm-Lim*cos(2*th))-Lm*Libtdot*sin(2*th)) / (Lqm-Lim^2);
hicdot = (libtdot*(Lqm-Lim*cos(2*th))-Lm*lia1dot*sin(2*th)) / (Lqm-Lim^2);
tau1 = P*(Lm*((1-ial^2)*sin(2*th)+2*ial+ibt*cos(2*th)+ial*phial+ibt*phibt));
comdot = 1/Lm*(tau1 - Rm*om - T1);
thdot = om;
xdot = [lia1dot;libtdot;comdot;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;
ial = zeros(1,6);
ib = zeros(1,6);
t = ones(1,6)*TPWM/6;
tn = t;
pctn = t/TCNT;
alhat = 0;
omhat = 0;
thaln = 0;
oth = 0;
ialsn = 0;
t1 = 0;
t2 = 0;
t3 = 0;

% initial conditions
ial0 = ialfa0;
ib0 = ibeta0;
com0 = 0;
th0 = 0;
% repeat NSIM times
% run sensorless algorithm
% run each subinterval of the PWM cycle on the PMEM model
% (including deadtime), collecting currents at each switching

t0 = 0;
y0 = [ia0;ibt0;om0;th0];
tout = t0;
yout = y0';
thatout = th0;
for npmi = 1:NSIM,
    % recall counter values computed on last PWM period
    cnt = cntt;
    % run sensorless algorithm
    [cntt,ttemp,va1f,va1beta] = sless(ia,ib,idref,wref);
    % first PWM subinterval
    tf = t0 + cnt(1)*TCKT;
    [ttemp,ytemp] = ode23('stpmem',[t0 tf],y0,[],wals1,vbts1,T1);
    nt = length(ttemp);
    tout = [t0; ttemp(2:nt)];
    yout = [y0; ytemp(2:nt,:)];
    thatout = [thatout; ones(nt-1,1)*ttemp];
    ia(1) = ytemp(nt,1);
    ib(1) = ytemp(nt,2);
    t0 = tf;
y0 = ytemp(nt,:);'';
    % second PWM subinterval
    tf = t0 + cnt(2)*TCKT;
    [ttemp,ytemp] = ode23('stpmem',[t0 tf],y0,[],wals2,vbts2,T1);
    nt = length(ttemp);
    tout = [t0; ttemp(2:nt)];
    yout = [y0; ytemp(2:nt,:)];
    thatout = [thatout; ones(nt-1,1)*ttemp];
    ia(2) = ytemp(nt,1);
    ib(2) = ytemp(nt,2);
    t0 = tf;
y0 = ytemp(nt,:);'';
    % third PWM subinterval
    tf = t0 + cnt(3)*TCKT;
    [ttemp,ytemp] = ode23('stpmem',[t0 tf],y0,[],wals3,vbts3,T1);
    nt = length(ttemp);
    tout = [t0; ttemp(2:nt)];
    yout = [y0; ytemp(2:nt,:)];
    thatout = [thatout; ones(nt-1,1)*ttemp];
    ia(3) = ytemp(nt,1);
    ib(3) = ytemp(nt,2);
    t0 = tf;
y0 = ytemp(nt,:);'';
    % fourth PWM subinterval
    tf = t0 + cnt(4)*TCKT;
    [ttemp,ytemp] = ode23('stpmem',[t0 tf],y0,[],wals4,vbts4,T1);
    nt = length(ttemp);
    tout = [t0; ttemp(2:nt)];
    yout = [y0; ytemp(2:nt,:)];
    thatout = [thatout; ones(nt-1,1)*ttemp];
    ia(4) = ytemp(nt,1);
    ib(4) = ytemp(nt,2);
    t0 = tf;
y0 = ytemp(nt,:);'';
    % fifth PWM subinterval
    tf = t0 + cnt(5)*TCKT;
    [ttemp,ytemp] = ode23('stpmem',[t0 tf],y0,[],wals5,vbts5,T1);
    nt = length(ttemp);
    tout = [t0; ttemp(2:nt)];
    yout = [y0; ytemp(2:nt,:)];
    thatout = [thatout; ones(nt-1,1)*ttemp];
    ia(5) = ytemp(nt,1);
    ib(5) = ytemp(nt,2);
APPENDIX B. MATLAB FILES FOR SIMULATIONS

88

c0 = tf;
y0 = ytemp(nt,:);

% sixth PWM subinterval
tf = t0 + cnt(6)*TCNT;
[tempytemp] = ode23('stpmem', [t0 tf], y0, [], vals6, vbs6, T1);
nt = length(temp);
tout = [tout; temp(2:nt)];
yst = [yst; ytemp(2:nt,:)];
thatout = [thatout; ones(nt-1,1)*temp];
M1 = ytemp(nt,1);
M2 = ytemp(nt,2);
t0 = tf;
y0 = ytemp(nt,:);'

eend

B.2.2 constsl.m

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

global Wp1 TFw1 TCNT CFTPM1 GAMMA1 GAMMA2

global k1 a1 k2 a2 k3 a3 ilim vlim

global CM105 SR301 SR3012C CM1012E

global iafa0 ibeta0 t tn alhat omhat thatz obserr inti int2 int3

% simulation parameters

NSIM = 4000;  % number of PWM cycles to simulate
TFw1 = 2e-4;
Wref = 500;
Idref = 0;
T1 = 1;

% PWM counter parameters

TCNT = 1e-7;
CFTPM1 = TFw1/TCNT;

% inverter parameters

VBUS = 200;
EMAX = VBUS/sqrt(2);
NDEAD = 2e-6;
CDEAD = TCNT/TCNT;
vals1 = 2/sqrt(6)*VBUS;
vbs1 = 0;
vals2 = 1/sqrt(6)*VBUS;
vbs2 = 1/sqrt(2)*VBUS;
vals3 = -1/sqrt(6)*VBUS;
vbs3 = 1/sqrt(2)*VBUS;
vals4 = -2/sqrt(6)*VBUS;
vbs4 = 0;
vals5 = -1/sqrt(6)*VBUS;
vbs5 = -1/sqrt(2)*VBUS;
vals6 = 1/sqrt(6)*VBUS;
vbs6 = -1/sqrt(2)*VBUS;

% general numeric constants

SR3012R2 = sqrt(3)/2;
SR3012D2 = sqrt(3)/2;
CM105 = 1/6;
CM105R2 = 1/sqrt(2);
CM105R3 = 1/sqrt(3);
CM105R6 = 1/sqrt(6);
SR3012E = sqrt(3)/12/EMAX;
CM1012E = 1/12/EMAX;

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

% Motor electrical parameters:
Rm=0.97;  
Lm=7.2e-3;  
Lms=1.8e-3;  
Ldm=Lm+1*Lm;  
Lqm=Lm*Lm;

% Constant in q2 and q3 parameters (used in observer error generation)
ilm=-Lm/(L0e-2-Lim^2);
pd6m=0.001;  
pd12m=0.0008;  
pq0m=0.1;  
pq2m=0.006;  
pq12m=0.0008;

% matrix of precomputed voltages

Wpi = [  
    -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  -64*sqrt(2)/320/VBUS;  
    -5*sqrt(2)/32/VBUS  53*sqrt(2)/320/VBUS  11*sqrt(6)/320/VBUS;  
    sqrt(6)/32/VBUS  15*sqrt(6)/320/VBUS  -45*sqrt(2)/320/VBUS;  
    -sqrt(2)/32/VBUS  sqrt(2)/320/VBUS  -33*sqrt(6)/320/VBUS;  
    sqrt(2)/32/VBUS  29*sqrt(2)/320/VBUS  9*sqrt(6)/320/VBUS;  
    sqrt(2)/32/VBUS  -sqrt(2)/320/VBUS  33*sqrt(6)/320/VBUS;  
    ];

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

% Constant in q2 and q3 parameters
ilm=L1/(L0e-2-L1^2);

% Desired observer poles
ri=200;  
ri=300;  
ri=100;  
pi1=-ri+i*wi;  
pi2=-ri;  
p3=-ri-i*wi;

% Observer constants
GAMMA1 = -real(pi+pi2+pi3)/2/il;  
GAMMA2 = real(pi+pi2+pi3)/2/il;  
GAMMA3 = -real(pi+pi2+pi3)/2/il;

clear L0 L1 il1 ri r2 w1 p1 p2 p3

% Controller constants

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

% PI controller constants
ri=15*(1+0.5*i);  
ri=15*(1-0.5*i);  
k2=(Jm*(ri+2)-Rm)/P/pq0m;  
a2=r1*r2/(v1+2-Rm/Jm);  
r1=500;  
r2=500;  
ki=Ldm*(ri+2)-Rm;  
a1=r1*r2/(v1+2-Rm/Ldm);  
r1=200;  
r2=200;  
k3=Lqm*(ri+2)-Rm;  
a3=r1*r2/(v1+2-Rm/Lqm);
B.2.3  sless.m

function [cnt,thhat,alalfa,wbeta] = 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:
%  alalfa0: 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 const1.m
%---------------------------------------------------------------
global Wpit TPWM TCONT CTPWM GAMMA1 GAMMA2 GAMMA3
global k1 a1 k2 a2 k3 a3 ilim vlim
global CN1003 SR322 CN106 SR3012E CN1012E
global alalfa0 ibeta0 t tn alhat omhat thhat obserr int1 int2 int3
%
% Observer
%---------------------------------------------------------------
% scale current?
%-----------------------------------------------
% for i=1:6,
%  % modified for simulation only!!!
%  alalfa(i) = SR300R2*ia(i);    % alalfa(i) = CM10RR2*ib(i)+ia(i);
%  ibeta(i) = CM10RR2*ib(i);
%  end
% construct vector x
%-----------------------------------------------
x(1) = (alalfa(2)-alalfa(1))/t(2) - (alalfa(1)-alalfa0)/t(1);
x(2) = (ibeta(2)-ibeta(1))/t(2) - (ibeta(1)-ibeta0)/t(1);
x(3) = (alalfa3-alalfa2)/t(3) - (alalfa2-alalfa1)/t(2);
x(4) = (ibeta3-ibeta2)/t(3) - (ibeta2-ibeta1)/t(2);
x(5) = (alalfa4-alalfa3)/t(4) - (alalfa3-alalfa2)/t(3);
x(6) = (ibeta4-ibeta3)/t(4) - (ibeta3-ibeta2)/t(3);
x(7) = (alalfa5-alalfa4)/t(5) - (alalfa4-alalfa3)/t(4);
x(8) = (ibeta5-ibeta4)/t(5) - (ibeta4-ibeta3)/t(4);
x(9) = (alalfa6-alalfa5)/t(6) - (alalfa5-alalfa4)/t(5);
x(10) = (ibeta6-ibeta5)/t(6) - (ibeta5-ibeta4)/t(5);
% find q = Wpit*x = Wpit'*x
% for j=1:3,
%  q(j) = 0;
%  for i=1:10,
%     q(j) = q(j) + Wpit(i,j)*x(i);
%  end

APPENDIX B. MATLAB FILES FOR SIMULATIONS
APPENDIX B. MATLAB FILES FOR SIMULATIONS

end

% observer update
ahat = ah + TFW*GAMMA3*oberr;\nomhat = oh + TFW*(ahat+GAMMA2*oberr);
that = th + TFW*(omhat+GAMMA1*oberr);

% error update
oberr = q(3)*cos(2*thath) - q(2)*sin(2*thath);

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

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

% average current measurements
ialav = (ialav(i-1)+ialav(i))*t(i);
ibtav = (ibtav(i-1)+ibtav(i))*t(i);
for i=2:6,
    ialav = ialav + (ialav(i-1)+ialav(i))*t(i);
    ibtav = ibtav + (ibtav(i-1)+ibtav(i))*t(i);
end
ialav = ialav/2/TFW;
ibtav = ibtav/2/TFW;

% convert to d-q frame
sint = sin(thath);
cost = cos(thath);
id = ialav*cost + ibtav*sint;
ig = -ialav*sint + ibtav*cost;

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

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

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

% INSERT VOLTAGES HERE FOR OPEN-LOOP
% vdc = -.0437;
% vqc = 2.2142;

% convert back to alfa-beta frame
sint = sin(thath*2*omhat*TFW);
cost = cos(thath*2*omhat*TFW);
valfa = vdc*cost - vqc*sint;
vbeta = vdc*sint + vqc*cost;

% find sector and rotate vector to sector 1
if (vb>0),
    if (valfa>0),
        sector = 1;
        valr = valfa;
    end
end
vbtr = vbeta;
else
    sector = 2;
    valr = valfa*.5 + vbeta*SR302;
    vbtr = -valfa*SR302 + vbeta*.5;
end
else
    if (CN10SR3*beta<valfa),
        sector = 3;
        valr = -valfa*.5 + vbeta*SR302;
    else
        sector = 2;
        valr = valfa*.5 + vbeta*SR302;
        vbtr = -valfa*SR302 + vbeta*.5;
    end
else
    if (valfa>0),
        if (CN10SR3*beta<valfa),
            sector = 6;
            valr = valfa*.5 - vbeta*SR302;
        else
            sector = 5;
            valr = -valfa*.5 - vbeta*SR302;
            vbtr = valfa*SR302 - vbeta*.5;
        end
    else
        if (CN10SR3*beta>valfa),
            sector = 4;
            valr = -valfa;
            vbtr = -vbeta;
        else
            sector = 5;
            valr = -valfa*.5 - vbeta*SR302;
            vbtr = valfa*SR302 - vbeta*.5;
        end
    end
end

% no limit of voltage magnitude
% find counter values
i = sector;
cnt(i) = (CN106 + 5*SR3012E+valr - 7*CN1012E+vbtr)*CPTPWM;
if (i==0),
    i = 1;
else
    i = i+1;
end
cnt(i) = (CN106 - SR3012E+valr + 11*CN1012E+vbtr)*CPTPWM;
if (i==0),
    i = 1;
else
    i = i+1;
end
cnt(i) = (CN106 - SR3012E+valr - CN1012E+vbtr)*CPTPWM;
if (i==0),
    i = 1;
else
    i = i+1;
end
cnt(i) = (CN106 - SR3012E+valr - CN1012E+vbtr)*CPTPWM;
if (i==0),
    i = 1;
else
    i = i+1;
end
cnt(i) = (CN106 - SR3012E+valr - CN1012E+vbtr)*CPTPWM;
if (i==0),
    i = 1;
else
    i = i+1;
end
cnt(i) = (CN106 - SR3012E+valr - CN1012E+vbtr)*CPTPWM;
if (i==0),
    i = 1;
else
    i = i+1;
end
cnt(i) = (CN106 - SR3012E+valr - CN1012E+vbtr)*CPTPWM;
APPENDIX B. MATLAB FILES FOR SIMULATIONS

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
columnslq;

% initialize memory values
ialfa0 = 0;
beta0 = 0;
i0 = zeros(1,6);
i1 = zeros(1,6);
t = ones(1,6)*CPTPW/6;
yn = t;
pnswt;
alpha = 0;
emhat = 0;
that = 0;
eberr = 0;
int1 = 0;
int2 = 0;
int3 = 0;

% initial conditions
ial0 = ialfa0;
APPENDIX B. MATLAB FILES FOR SIMULATIONS

ibt0 = ibeta0;
omb0 = 0;
th0 = 0;

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

t0 = 0;
y0 = [ia0; ibt0; omb0; th0];
tout = t0;
yout = y0';
thatout = th0;
omhatout = omb0;
for npwmint = 1:NSIM,

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

% run sensorless algorithm
[ctn, t, temp,oref, wref, ib0, ombi, ibti, y0] = slq(ia, ib, idref, wref);

% first PWM subinterval
if t0 + ctn(1)<TCTM;
[tttemp, ytemp] = ode23('stpmem', [t0 tf], y0, [], wals1, vbeta1, T1);
nt = length(ttemp);
tout = [t0; ttemp(2:nt)];
yout = [y0; ytemp(2:nt)];
thatout = [thatout; ones(nt-1,1)*ttemp];
ia1(1) = sqrt(2/3)*ytemp(nt,1);
ib1(1) = ytemp(nt,3)/sqrt(2)-ytemp(nt,1)/sqrt(6);
t0 = tf;
y0 = ytemp(nt,1)';
end

% second PWM subinterval
if t0 + ctn(2)<TCTM;
[tttemp, ytemp] = ode23('stpmem', [t0 tf], y0, [], wals2, vbeta2, T1);
nt = length(ttemp);
tout = [t0; ttemp(2:nt)];
yout = [y0; ytemp(2:nt)];
thatout = [thatout; ones(nt-1,1)*ttemp];
ia2(1) = sqrt(2/3)*ytemp(nt,1);
ib2(1) = ytemp(nt,3)/sqrt(2)-ytemp(nt,1)/sqrt(6);
t0 = tf;
y0 = ytemp(nt,1)';
end

% third PWM subinterval
if t0 + ctn(3)<TCTM;
[tttemp, ytemp] = ode23('stpmem', [t0 tf], y0, [], wals3, vbeta3, T1);
nt = length(ttemp);
tout = [t0; ttemp(2:nt)];
yout = [y0; ytemp(2:nt)];
thatout = [thatout; ones(nt-1,1)*ttemp];
ia3(1) = sqrt(2/3)*ytemp(nt,1);
ib3(1) = ytemp(nt,3)/sqrt(2)-ytemp(nt,1)/sqrt(6);
t0 = tf;
y0 = ytemp(nt,1)';
end

% fourth PWM subinterval
if t0 + ctn(4)<TCTM;
[tttemp, ytemp] = ode23('stpmem', [t0 tf], y0, [], wals4, vbeta4, T1);
nt = length(ttemp);
tout = [t0; ttemp(2:nt)];
yout = [y0; ytemp(2:nt)];
thatout = [thatout; ones(nt-1,1)*ttemp];
ia4(1) = sqrt(2/3)*ytemp(nt,1);
ib4(1) = ytemp(nt,3)/sqrt(2)-ytemp(nt,1)/sqrt(6);
t0 = tf;
y0 = ytemp(nt,1)';
end

% fifth PWM subinterval
if t0 + ctn(5)<TCTM;
[tttemp, ytemp] = ode23('stpmem', [t0 tf], y0, [], wals5, vbeta5, T1);
nt = length(ttemp);
APPENDIX B. MATLAB FILES FOR SIMULATIONS

95

tout = [tout; ttemp(2:nt)];
yout = [yout; ytemp(2:nt,:)];
thatout = [thatout; ones(nt-1,1)*ttemp];
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
if tf = t0 + cnt(6)*TCNT;
[ttemp,ytemp] = ode23('spmms',[t0 tf],y0,[]);vals6,vbeta6,T1);
nt = length(ttemp);
tout = [tout; ttemp(2:nt)];
yout = [yout; ytemp(2:nt,:)];
thatout = [thatout; ones(nt-1,1)*ttemp];
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 tref TFWM TCTN CPTFWM IABMAX
global iabs0 ibeta0 ttn alhah ombat ththa obserr ini int2 int3
global SR30SR202 SR202 CNI02SR2 CNI03SR3 SR302 CN2SR204 CN106 CN10SR6 SR206
% simulation parameters

NSIM = 2000; % number of PWM cycles to simulate
TFWM = 2e-4;
TCTN = 1e-7;
CPTFWM = TFWM/TCTN;

wref = frac16(500/1800);
idref = frac16((0/2)/TCTN/12.5*TFWM/2^-15);
T1 = 1;

% inverter parameters

VBUS = 200;
EMAX = VBUS/sqrt(2);
TEDAD = 2e-6;
CPTDEAD = TEDAD/TCTN;
vals1 = 2/sqrt(6)*VBUS;
vbs1 = 0;
vals2 = 1/sqrt(6)*VBUS;
vbs2 = 1/sqrt(2)*VBUS;
vals3 = -1/sqrt(6)*VBUS;
vbs3 = 1/sqrt(2)*VBUS;
vals4 = -2/sqrt(6)*VBUS;
vbs4 = 0;
vals5 = -1/sqrt(6)*VBUS;
vbs5 = -1/sqrt(2)*VBUS;
vals6 = 1/sqrt(6)*VBUS;
vbs6 = -1/sqrt(2)*VBUS;

% scale factors

IABMAX = 26;
% general numeric constants (quantized)

SR30SR202 = frac16(sqrt(3)/2)/2;
SR202 = frac16(sqrt(3)/2);
CNI02SR2 = frac16(i/2/sqrt(2));
CNI03SR3 = frac16(i/sqrt(3));
SR2 = frac6(sqrt(3)/2);
CM2SR4 = frac6(2*sqrt(2)/4);
CM16 = frac6(1/6);
CM1SR6 = frac6(1/sqrt(6));
SR26 = frac6(sqrt(2)/6);

% general numeric constants
SR3SR2 = sqrt(3/2);
CM1SR2 = 1/sqrt(2);
CM1SR6 = 1/sqrt(6);
SR312E = sqrt(3)/12/EMAX;
CM112E = 1/12/EMAX;

% motor parameters
global pd6m pd12m pq0m pq6m pq12m
global Rsm Lsm Lm
global jm Jm Ps

Pm =;

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

% Motor electrical parameters:
Rsm=0.97;
Lsm=7.2e-3;
Lm=1.8e-3;
Lqsm=Lsm+Lm;
Lq=Lsm-Lm;

% Constant in q2 and q3 parameters (used in observer error generation)
ilm=-Lm/(L0e^2-Lm^2);

pd6m=0.001;
pd12m=0.0008;
pq0m=0.1;
pq6m=0.006;
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 -32*sqrt(6)/320/VBUS;
    -sqrt(6)/32/VBUS 15*sqrt(6)/320/VBUS -46*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 = 21.2265/32A * Wpit;

for j=1:3,
    for i=1:10,
        Wpit(i,j)=frac8(Wpit(i,j));
    end
end

clear i j

% observer coefficients
global GAMMA1SG GAMMA2SQ GAMMA3SQ ACCSPEFF SPOEFF

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

% Constant in q2 and q3 parameters
ilim=L1/(L0^2-L1^2);
APPENDIX B. MATLAB FILES FOR SIMULATIONS

% 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*i11;
GAMMA2 = real(p1*p2*p3+p2*p3+p3*p1)/2/i11;
GAMMA3 = -real([p1*p2*p3]/2*i11;
clear L0 L1 r1 r2 w1 p1 p2 p3
GAMMA1SQ = frac16(TPMW+GAMMA1+324/2/pi);
GAMMA2SQ = frac16(TPMW+GAMMA2+324/2*1800);
GAMMA3SQ = frac16(TPMW+GAMMA3+324/2*10000);
ACDQEFF = frac16(TPMW=10000/18000);
SPCQEFF = frac16(TPMW=18000/pi);

% Controller constants
% Current and voltage limits
ilim=12;
vlim=sqrt(3)/2*VBUS/sqrt(2);
% PI controller constants
r1=15*(1.0-0.1); r2=15*(1.0-0.1);
k2=(k2*(ri+1.2-1800))

%3r1=500; r2=500;
k1=rdm*(ri+1.2-1800);
alpha1=r1=0.15*(ri+1.2-1800);
r1=300; r2=300;
k3=rdm*(ri+1.2-1800);
alpha1=r1=0.15*(ri+1.2-1800);
clear r1 r2
global ERR1QEFFINT ERR1QEFFOUT INT1QEFF
global ERR2QEFFINT ERR2QEFFOUT INT2QEFF
global ERR3QEFFINT ERR3QEFFOUT INT3QEFF
ERR1QEFFINT = frac16(.13884);
ERR1QEFFOUT = frac16(.26502);
INT1QEFF = 1;
ERR2QEFFINT = frac16(2.468e-3);
ERR2QEFFOUT = frac16(.31882);
INT2QEFF = 0.25;
ERR3QEFFINT = frac16(.0369);
ERR3QEFFOUT = frac16(.34866);
INT3QEFF = 1;
global ILIM DLYQEFF
ILIM=f16(.32408);
DLYQEFF=f16(.25);

B.3.3 slq.m

function [cnt,thhat,omhat,wa,va,thb,thb,ib,ibk,ikav,ikbav] = 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:
iafa0: 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, that, obserr: observer variables
int1, int2, int3: integrators for the PI blocks

% outputs:
cnt[1...6]: count values for the PWM generation
that: estimated position

% ge, March 2001

% Numeric constants loaded previously from constsg.m

% Observe

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

% convert currents to alpha-beta frame
for i=1:6,
    ialfa(i) = SRSR2Q2*fta(i);
    ibeta(i) = SRSR2Q2*fb(i) + CN16*SRSR2Q2*ia(i);
end

% difference of currents
dalfa(1) = alu('-',ialfa(1),ialfa0);
dbeta(1) = alu('-',ibeta(1),ibeta0);
dalfa(2) = alu('-',ialfa(2),ialfa1);
dbeta(2) = alu('-',ibeta(2),ibeta1);
dalfa(3) = alu('-',ialfa(3),ialfa2);
dbeta(3) = alu('-',ibeta(3),ibeta2);
dalfa(4) = alu('-',ialfa(4),ialfa3);
dbeta(4) = alu('-',ibeta(4),ibeta3);
dalfa(5) = alu('-',ialfa(5),ialfa4);
dbeta(5) = alu('-',ibeta(5),ibeta4);
dalfa(6) = alu('-',ialfa(6),ialfa5);
dbeta(6) = alu('-',ibeta(6),ibeta5);

% division by time
dialt(1) = div(dalfa(1)*2^12,t(1),16,16);
dibdt(1) = div(dbeta(1)*2^12,t(1),16,16);
dialt(2) = div(dalfa(2)*2^12,t(2),16,16);
dibdt(2) = div(dbeta(2)*2^12,t(2),16,16);
dialt(3) = div(dalfa(3)*2^12,t(3),16,16);
dibdt(3) = div(dbeta(3)*2^12,t(3),16,16);
dialt(4) = div(dalfa(4)*2^12,t(4),16,16);
dibdt(4) = div(dbeta(4)*2^12,t(4),16,16);
dialt(5) = div(dalfa(5)*2^12,t(5),16,16);
dibdt(5) = div(dbeta(5)*2^12,t(5),16,16);
dialt(6) = div(dalfa(6)*2^12,t(6),16,16);
dibdt(6) = div(dbeta(6)*2^12,t(6),16,16);

% construct vector x
x(1) = alu('-',frac16(dialt(2)/2),frac16(dialt(1)/2));
x(2) = alu('-',frac16(dibdt(2)/2),frac16(dibdt(1)/2));
x(3) = alu('-',frac16(dialt(3)/2),frac16(dialt(2)/2));
x(4) = alu('-',frac16(dibdt(3)/2),frac16(dibdt(2)/2));
x(5) = alu('-',frac16(dialt(4)/2),frac16(dialt(3)/2));
x(6) = alu('-',frac16(dibdt(4)/2),frac16(dibdt(3)/2));
x(7) = au('r',frac16(dialdt(5)/2),frac16(dialdt(4)/2));
x(8) = au('r',frac16(dibdct(5)/2),frac16(dibdct(4)/2));
x(9) = au('r',frac16(dialdt(6)/2),frac16(dialdt(5)/2));
x(10) = au('r',frac16(dibdct(6)/2),frac16(dibdct(5)/2));

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

% observer update
ahlhat = frac16(mac(alhat,GAMMA36Q,obserr));
ombhat = mac(ombhat,ACQUFF,ahlhat);
ombhat = frac16(mac(ombhat,GAMMA32Q,obserr));
thalm = mac(thalm,SPQQFF,ombhat);
thalm = mac(thalm,GAMMAG1Q,obserr);

% theta hat wrap around
if that>1
 that = that-2;
else if that<-1
 that = that+2;
end

% error update
cos2that = frac16(cos(2*that*pi));
sin2that = frac16(sin(2*that*pi));

obserr = mac(0,frac16(2*q(3)),cos2that);
obserr = mac(obserr,-frac16(2*q(2)),sin2that);

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

% average current measurements
ialadd(1) = au('r',frac16(ialfa0(2)/2),frac16(ialfa(t(1)/2))); ibadd(1) = au('r',frac16(ialfa0(2)/2),frac16(ialfa(t(1)/2)));
ialadd(2) = au('r',frac16(ialfa0(2)/2),frac16(ialfa(t(2)/2))); ibadd(2) = au('r',frac16(ialfa0(2)/2),frac16(ialfa(t(2)/2)));
ialadd(3) = au('r',frac16(ialfa0(2)/2),frac16(ialfa(t(3)/2))); ibadd(3) = au('r',frac16(ialfa0(2)/2),frac16(ialfa(t(3)/2)));
ialadd(4) = au('r',frac16(ialfa0(2)/2),frac16(ialfa(t(4)/2))); ibadd(4) = au('r',frac16(ialfa0(2)/2),frac16(ialfa(t(4)/2)));
ialadd(5) = au('r',frac16(ialfa0(2)/2),frac16(ialfa(t(5)/2))); ibadd(5) = au('r',frac16(ialfa0(2)/2),frac16(ialfa(t(5)/2)));
ialadd(6) = au('r',frac16(ialfa0(2)/2),frac16(ialfa(t(6)/2))); ibadd(6) = au('r',frac16(ialfa0(2)/2),frac16(ialfa(t(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);
ibav = mac(0,ibadd(1),t(1)/2*15);
ibav = mac(ibav,ibadd(2),t(2)/2*15);
ibav = mac(ibav,ibadd(3),t(3)/2*15);
ibav = mac(ibav,ibadd(4),t(4)/2*15);
ibav = mac(ibav,ibadd(5),t(5)/2*15);
ibav = mac(ibav,ibadd(6),t(6)/2*15);

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

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

id = mac(0,frac16(ialav),costh);
id = frac16(mac(id,frac16(ibav),sinth));
ig = mac(0,-frac16(ialav),sinth);
\texttt{iq = frac16(mac(iq,frac16(ibtav),costh));}
\texttt{\% id control block (PI1)}
\texttt{u1 = alu('-',idref,id);}
\texttt{int1 = frac16(mac(int1,u1,ERR1CQEFFINT));}
\texttt{if (abs(int1)>1),}
\texttt{int1 = sign(int1);}
\texttt{end}
\texttt{vdc = mac(Q,u1,ERR1CQEFFOUT);}
\texttt{vdc = vdc+u1+int1;}
\texttt{\% w control block (PI2)}
\texttt{u2 = alu('-',wref,omhat);}
\texttt{int2 = frac16(mac(int2,u2,ERR2CQEFFINT));}
\texttt{if (abs(int2)>1),}
\texttt{int2 = sign(int2);}
\texttt{end}
\texttt{iqref = mac(0,u2,ERR2CQEFFOUT);}
\texttt{iqref = mac(iqref,int2,INT3QEFF);}
\texttt{if (abs(iqref)>ILIM),}
\texttt{iqref = sign(iqref)*ILIM;}
\texttt{end}
\texttt{\% iq control block (PI3)}
\texttt{u3 = alu('-',iqref,iq);}
\texttt{int3 = frac16(mac(int3,u3,ERR3CQEFFINT));}
\texttt{if (abs(int3)>1),}
\texttt{int3 = sign(int3);}
\texttt{end}
\texttt{vqc = mac(u3,u3,ERR3CQEFFOUT);}
\texttt{vqc = alu('+',vqc,int3);}
\texttt{\% convert back to alfa-beta frame}
\texttt{that2 = that + DLYQ*omhat;}
\texttt{sinth = frac16(sin(that2*pi));}
\texttt{costh = frac16(cos(that2*pi));}
\texttt{valfa = mac(0,vdc,costh);}
\texttt{valfa = frac16(mac(valfa,-vqc,sinth));}
\texttt{vbeta = mac(0,vdc,sinth);}
\texttt{vbeta = frac16(mac(vbeta,vqc,costh));}
\texttt{\% find sector and rotate vector to sector 1}
\texttt{\% (vbeta>0),}
\texttt{if (valfa>0),}
\texttt{\% (CN1SR3<vbeta-valfa),}
\texttt{sector = 1;}
\texttt{valr = valfa;}
\texttt{vbtr = vbeta;}
\texttt{else}
\texttt{sector = 2;}
\texttt{valr = frac16(valfa/2);}
\texttt{valr = frac16(mac(valr,vbeta,SR302));}
\texttt{vbtr = frac16(vbeta/2);}
\texttt{vbtr = frac16(mac(vbtr,-valfa,SR302));}
\texttt{end}
\texttt{else}
\texttt{if (CN1SR3<vbeta-valfa),}
\texttt{sector = 3;}
\texttt{valr = -frac16(valfa/2);}
\texttt{valr = frac16(mac(valr,vbeta,SR302));}
\texttt{vbtr = -frac16(vbeta/2);}
\texttt{vbtr = frac16(mac(vbtr,-valfa,SR302));}
\texttt{else}
\texttt{sector = 2;}
\texttt{valr = frac16(valfa/2);}
\texttt{valr = frac16(mac(valr,vbeta,SR302));}
\texttt{vbtr = frac16(vbeta/2);}
\texttt{vbtr = frac16(mac(vbtr,-valfa,SR302));}
\texttt{end}
\texttt{else}
\texttt{if (valfa>0),}
\texttt{if (CN1SR3<vbeta-valfa),}
\texttt{sector = 6;}
\texttt{valr = frac16(valfa/2);
APPENDIX B. MATLAB FILES FOR SIMULATIONS

valr = frac16(mac(valr,-vbeta,SR302));
vhrt = frac16(vbeta/2);
vbtr = frac16(mac(vhtr,valfa,SR302));
else
    sector = 5;
    valr = frac16(-valfa/2);
    valr = frac16(mac(valr,-vbeta,SR302));
    vhtr = frac16(-vbeta/2);
    vhtr = frac16(mac(vhtr,valfa,SR302));
end
else
    if (CN1SR3>vbeta>valfa),
        sector = 4;
        valr = -valfa;
        vhtr = -vbeta;
        else
            sector = 5;
            valr = frac16(-valfa/2);
            valr = frac16(mac(valr,-vbeta,SR302));
            vhtr = frac16(-vbeta/2);
            vhtr = frac16(mac(vhtr,valfa,SR302));
        end
    end
end

% limit voltage magnitude
magv = frac16(mac(valr,CN1SR3,vhtr));
if (magv>CN10SR2)
    magv = frac16(mac(0,magv,CN2SR2));
    valr = div(frac16(valr/4),magvrs,i,i,1);
    vhtr = div(frac16(vhtr/4),magvrs,i,i,1);
end

% find counter values
valrs = frac16(mac(0,valr,CN10SR6));
vbrts = frac16(mac(0,vbtr,SR206));
i=sector;
cnt(1) = au('+',CN1DS,valrs);
cnt(1) = au('+',cnt(1),valrs);
cnt(1) = au('-+',cnt(1),valrs);
cnt(1) = au('+',cnt(1),valrs);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
if (i==0),
i = 1;
else
i = i+1;
end

cnt(1) = au('-+',CN1DS,vbtr);
cnt(1) = au('+',cnt(1),vbtr);
cnt(1) = au('+',cnt(1),vbtr);
cnt(1) = au('+',cnt(1),vbtr);
cnt(1) = au('+',cnt(1),vbtr);
cnt(1) = au('+',cnt(1),vbtr);
cnt(1) = au('+',cnt(1),vbtr);
cnt(1) = au('+',cnt(1),vbtr);
cnt(1) = au('+',cnt(1),vbtr);
cnt(1) = au('+',cnt(1),vbtr);
cnt(1) = au('+',cnt(1),vbtr);
cnt(1) = au('+',cnt(1),vbtr);
if (i==0),
i = 1;
else
i = i+1;
end

cnt(1) = au('-+',CN1DS,vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
cnt(1) = au('-',cnt(1),vbtr);
if (i==0),
i = 1;
else
i = i+1;
APPENDIX B. MATLAB FILES FOR SIMULATIONS

```matlab
i = i+1;
end
cnt(i) = alu('-',CN106,walrs);
cnt(i) = alu('-',cnt(i),vbrtm);
if (i==0),
i = 1;
else
i = i+1;
end
cnt(i) = alu('-',CN106,walrs);
cnt(i) = alu('-',cnt(i),vbrtm);
i = 1;
else
i = i+1;
end
cnt(i) = alu('-',CN106,walrs);
cnt(i) = alu('-',cnt(i),vbrtm);
for i=1:6
cnt(i) = frac16(mac(Q,CPTFMN-15,cnt(i)))*2^15;
end

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

% save last current measurement
ialfa0 = ialpha(i);
ibeta0 = ibeta(i);

% save time intervals for next period
for i=1:6,
r(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('Unknown operation')
end
if (r>1-2^-15 | r<-1)
  warning('Overflow in the ALU.')
end

B.3.6 mac.m

function m = mac(mx,my)
% function m = mac(mx,my)
% Performs a multiply-and-accumulate operation:
% m = mx + my
% Inputs:  mx = previous value of the accumulator,
%          my = operand in 1.15 format
% The result will naturally be in (n+1).31 format.
% Error if n+1>8 bits (overflow).
% m = mx + my;
if (m>255-2^-31 | m<-256)
  error('Overflow in the MAC.')
end
Appendix C

Assembler code for the DSP

C.1 final.asm

.getAppC-1.0

)findViewById();

setContentView(R.layout.activity_main);

getWindow().setSoftInputMode(WindowManager.LayoutParams.SOFT_INPUT_ADJUST_RESIZE);

initWidgets();

getWindow().setWindowAnimations(R.style.AppTheme);
APPENDIX C. ASSEMBLER CODE FOR THE DSP

// controller coefficients
#define errlcoefff 0x1B2 // depends on TFM, TCHT and VBUS
#define errlcoefout 0x2254 // depends on TFM, TCHT and VBUS
//Define intlcoeff 0x6D71 // depends on TFM, TCHT and VBUS
#define err2coefff 0x09051 // depends on TFM, TCHT and VBUS
#define err2coefout 0x2B8B // depends on TFM, TCHT and VBUS
#define int2coeff 0x2000 // depends on TFM, TCHT and VBUS
#define err3coefff 0x4B9 // depends on TFM, TCHT and VBUS
#define err3coefout 0x2C5D // depends on TFM, TCHT and VBUS
//Define int3coeff 0x6D71 // depends on TFM, TCHT and VBUS

// other controller constants
#define ilimit 0x0780 // current limit
#define delaycoeff 0x1DB6 // coefficient for delay correction

// FWM vectors
#define vec1 0x31
#define vec2 0x23
#define vec3 0x52
#define vec4 0x4E
#define vec5 0x1C
#define vec6 0x1E

// i/o port addresses
#define pin1 0x0 // ADC value of ia for subinterval 1 (in)
#define pin2 0x1 // ADC value of ia for subinterval 2 (in)
#define pin3 0x2 // ADC value of ia for subinterval 3 (in)
#define pin4 0x3 // ADC value of ia for subinterval 4 (in)
#define pin5 0x4 // ADC value of ia for subinterval 5 (in)
#define pin6 0x5 // ADC value of ia for subinterval 6 (in)
#define pin7 0x6 // ADC value of ib for subinterval 1 (in)
#define pin8 0x7 // ADC value of ib for subinterval 2 (in)
#define pin9 0x8 // ADC value of ib for subinterval 3 (in)
#define pin10 0x9 // ADC value of ib for subinterval 4 (in)
#define pin11 0xA // ADC value of ib for subinterval 5 (in)
#define pin12 0xB // 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 pcm 0x5 // command (out)
#define ppo 0x6 // position (out)
#define pt1 0x8 // time for subinterval 1 (out)
#define pt2 0x9 // time for subinterval 2 (out)
#define pt3 0xA // time for subinterval 3 (out)
#define pt4 0xB // time for subinterval 4 (out)
#define pt5 0xC // time for subinterval 5 (out)
#define pt6 0xD // time for subinterval 6 (out)
#define ptdb 0xE // time for deadband (out)

// memory mapped registers and associated values
#define lowaitreg 0x3FFE // address of the I/Q wait state register
#define onewait 0x7FF9 // value for one wait state
#define fdatareg 0x3FFB // address of the PF DATA register
#define maskpf1 0x002 // mask to read the PF1 bit
#define maskpf2 0x004 // mask to read the PF2 bit
#define enaflag 0x80 // value to enable INRQ
#define disnaf 0x00 // value to enable saturation in AR register
#define disnaf 0x00 // value to disable saturation in AR register

// declare sine subroutine

.EXTRN sin;

//variables

.SECTION/DATA pendata;

// currents
.VAR ila[5]; // measured currents in phase a
.VAR iba[5]; // measured currents in phase b
.VAR iapk[7]; // currents in alpha axis
.VAR ibapk[7]; // currents in beta axis
.VAR dia[6]; // difference of currents in alpha axis

APPENDIX C. ASSEMBLER CODE FOR THE DSP

```assembly
.VAR dbeta[6]; // difference of currents in beta axis
.VAR t[i];   // subintervals time
.VAR tnn[6];  // subintervals time from previous period
.VAR dialfa[6]; // dialfa over subinterval time
.VAR dbdt[4];  // dbeta over subinterval time
.VAR x[i][0]; // vector x
.VAR Wp[20] = // matrix Wpi, only rows 2 and 3
0x3A4B, 0x2F1B, 0x9502, 0x3D21, 0x0C69, 0x7306, 0x3757, 0x70E5, 0x6AFE, 0xFD0F,
0x9FDF, 0x2976H, 0x6C04, 0x6490, 0x2C04, 0x4606, 0x6296, 0x123C, 0x7900;
.VAR q[2];   // vector q (estimated parameters)
.VAR alhat; // estimated acceleration
.VAR omhat;  // estimated speed
.VAR thhat;  // estimated position
.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 va;    // vaf control output
.VAR vb;    // vbta control output
.VAR vr;    // sector of the control output
.VAR/CRC 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;
    JUMP powirq; // IRGE interrupt
    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
    EM(iowaitreg) = AX0;

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

    // enable interrupt IRGE
    ENMAK = enairq;

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

    // indirect addressing 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(iaiga) = AX0;      // iaiga0 = 0
DM(ibeta) = AX0;      // ibeta0 = 0
IO = tn;
II = t;
A11 = cplt06-cptdead; // load 1/6*cfptpm-cptdead
CNTM = 6;
DO init_times UNTIL CE;
   DM(II,II) = AX1;  // tn(i) = 1/6*cfptpm-cptdead
   DM(II) = AX1;
   DM(I1,II) = AX1;  // t(i) = 1/6*cfptpm-cptdead
   DM(alhat) = AX0;  // alhat = 0
   DM(ombat) = AX0;  // ombat = 0
   AX1 = tihatini;
   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
   DM(10,10) = AX0;

   // 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 = cplt06-cptdead; // load 1/6*cfptpm-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(pfdatar); // read flags
   AY0 = maskkp2;    // mask for PP2
   AR = AX0 AND AY0; // find initial flag value
   AY1 = AR;         // move initial value

   wait_switch:
      AX0 = DM(pfdatar); // read flags
      AR = AX0 AND AY0; // find PP2 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
   loop:
      IDLE;
      JUMP loop;

   // interrupt service routine
   .SECTION CODE irq:
pwmirq:
APPENDIX C. ASSEMBLER CODE FOR THE DSP

// if pushbutton is pressed, change speed reference
AXO = DM(p 않습니다0); // read flags
AYO = maskpni; // mask for PF1
AR = AXO AND AYO; // find flag value
IF EQ JUMP start alg; // if it is zero, don't do nothing

// change speed reference
AXO = omegaref2;
DM(ωref) = AXO;

start alg:

// read currents from i/o ports
IO = ia;
AXO = IO(pία1);
DM(ΙO,ΜΟ) = AXO;
AXO = IO(pία2);
DM(ΙO,ΜΟ) = AXO;
AXO = IO(pία3);
DM(ΙO,ΜΟ) = AXO;
AXO = IO(pία4);
DM(ΙO,ΜΟ) = AXO;
AXO = IO(pία5);
DM(ΙO,ΜΟ) = AXO;
AXO = IO(pία6);
DM(ΙO,ΜΟ) = AXO;
I2 = Ib;
AXO = IO(pib1);
DM(I2,ΜΟ) = AXO;
AXO = IO(pib2);
DM(I2,ΜΟ) = AXO;
AXO = IO(pib3);
DM(I2,ΜΟ) = AXO;
AXO = IO(pib4);
DM(I2,ΜΟ) = AXO;
AXO = IO(pib5);
DM(I2,ΜΟ) = AXO;
AXO = IO(pib6);
DM(I2,ΜΟ) = AXO;

// convert to alfa-beta
IO = ia;
ii = ialsa;
MODIFY(ΙΙ,ΜΟ); // skip ist element (last value of last period)
MYO = sr30sr2o2;
CNTN = 6;
D0 conv.alfa UNTIL CE;
MXO = DM(ΙO,ΜΟ); // load ia(i)
MR = MXO*MYO (NND); // ia(i)*sr30sr2o2
conv.alfa:
DM(Ι1,ΜΟ) = MR1; // save ialsa(i)
I0 = ia;
I2 = Ib;
I3 = ibeta;
MODIFY(Ι3,ΜΟ); // skip ist element (last value of last period)
MYO = cn102b2r2;
MY1 = sr2o2;
CNTN = 6;
D0 conv.beta UNTIL CE;
MXI = DM(I2,ΜΟ); // load ib(i)
MR = MXI*MYO (NND); // ia(i)*cn102b2r2
MR = MR+MXI*MY1 (NND); // ia(i)*cn102b2r2 + ib(i)*sr2o2
conv.beta:
DM(Ι3,ΜΟ) = MR1; // save ibeta(i)

// difference of currents
IO = dialfa;
ii = ialsa;
I2 = dibeta;
I3 = ibeta;
AXO = DM(Ι1,ΜΟ); // load ialsa(i)
AI1 = DM(Ι3,ΜΟ); // load ibeta(i)
CNTN = 6;
D0 subc. curr UNTIL CE;
AYO = AXO; // move ialsa(i-1)
APPENDIX C. ASSEMBLER CODE FOR THE DSP

AYi = AX;  // move ibeta(i-1)
AX = DM(I1,MO);  // load ila(i)
AX1 = DM(I3,MO);  // load ibeta(i)
AR = AXO-AYO;  // ila(i)-ila(i-1)
DM(IQ,MO) = AR;  // save dialfa(i)
AN = AX1-AY1;  // ibeta(i)-ibeta(i-1)

subs_curv:
DM(I2,MO) = AR;  // save dibeta(i)

// division by time
IQ = dialfa;
i1 = dialdt;
i2 = dibeta;
i3 = db td;
T4 = t;
CNTR = 6;
D0 div_time UNTIL CE;
AXO = DM(I4,MA);  // load t(i)
SRQ = DM(IQ,MO);  // load dialfa(i)
SR = ASHIFT SOE BY -3 (HI);  // shift dialfa(i)
AY1 = SR1;  // move shifted dj alf a(i)
AYO = SR0;
DIV5 AY1,AXO;  // dialfa(i)/t(i)
DIVQ AXO; DIVQ AXO; DIVQ AXO;
DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO;
DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO;
DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO;
DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO;

div_time:
DM(I3,MO) = AYO;  // save dibeta(i)

// con struct vector x
IQ = x;
i1 = dialdt;
i3 = dibtdt;
SRQ = DM(I1,MO);  // load dialdt(i)
SR = ASHIFT SOE BY -1 (LQ);  // shift dialdt (divide by 2)
AXO = SR0;  // move dialdt(i)/2
SRQ = DM(I3,MO);  // load dibtdt(i)
SR = ASHIFT SOE BY -1 (LQ);  // shift dibtdt (divide by 2)
AX1 = SR0;  // move dibtdt(i)/2
CNTR = 5;
D0 constr_x UNTIL CE;
AYO = AXO;  // move dial dt(i-1)/2
AY1 = AX1;  // move dibtdt(i-1)/2
SRQ = DM(I1,MO);  // load dialdt(i)
SR = ASHIFT SOE BY -1 (LQ);  // shift dialdt (divide by 2)
AXO = SR0;  // move dialdt(i)/2
SRQ = DM(I3,MO);  // load dibtdt(i)
SR = ASHIFT SOE BY -1 (LQ);  // shift dibtdt (divide by 2)
AX1 = SR0;  // move dibtdt(i)/2
AR = AXO-AYO;  // dialdt(i)/2-dialdt(i-1)/2
DM(IQ,MO) = AR;  // save x(2i+1)
AN = AX1-AY1;  // dibtdt(i)/2-dibtdt(i-1)/2

const x:
DM(IQ,MO) = AR;  // save x(2i)

// find q=Wpi*x
IQ = x;
i1 = Wpi;
i2 = q;
MR = 0;  // clear accumulator
AXO = DM(IQ,MO);  // load x(i)
MYO = DM(I1,MO);  // load Wp i(i,1)
CNTR = 10;
D0 find_q2 UNTIL CE;
MR = MR+AXO*MYO (RND),  // MR*x(i)=Wpi(i,1)
APPENDIX C. ASSEMBLER CODE FOR THE DSP 110

find_q2:
MXX = DM(IO,MO); // load x(i+1)
MYO = DM(II,MO); // load Wpi(i+1)
SR = ASHIFT MR1 BY 1 (LD); // multiply result by 2
SR = SR OR LSHIFT MR0 BY -15 (LS); // add LSB to increase precision
DM(I2,MO) = SRO; // save 2^q(2)

IQ = x;
MR = 0; // clear accumulator
MXX = DM(IO,MO); // load x(i), Wpi(2,1) already loaded in MYO
CNTN = 10;
DO find_q3 UNTIL CE;
MR = MR+MXO+MYO (RND); // MR+x(i)*Wpi(2,1)
MYO = DM(IO,MO); // load x(i+1)

find_q3:
MYO = DM(IO,MO); // load Wpi(2,i+1)
SR = ASHIFT MR1 BY 1 (LD); // multiply result by 2
SR = SR OR LSHIFT MR0 BY -15 (LS); // add LSB to increase precision
DM(I2,MO) = SRO; // save 2^q(3)

// update observer
MXX = gamma3; // load gamma3
MYO = DM(observ); // load obserr
MRQ = 0;
MR1 = DM(ahat); // load ahat
MR = MR+MXO+MYO (RND); // ahat+gamma3*obserr
DM(ahat) = MR1; // save new ahat
MXX = gamma2; // load gamma2
MX1 = accoeff; // load accoeff
MY1 = MR1; // move ahat
MRQ = 0;
MR1 = DM(omhat); // load omhat
MR = MR+MXO+MYO (RND); // omhat+gamma2*obserr
MR = MR+MX1+MY1 (RND); // omhat+gamma2*obserr+acoeff+ahat
DM(omhat) = MR1; // save new omhat
MXX = gamma1; // load gamma1
MX1 = spcoeff; // load spcoeff
MY1 = MR1; // move omhat
MRQ = 0;
MR1 = DM(thhat); // load thhat
MR = MR+MXO+MYO (SS); // thhat+gamma1*obserr
MR = MR+MX1+MY1 (RND); // thhat+gamma1*obserr+spcoeff+omhat
DM(thhat) = MR1; // save new thhat

// update observer error
MsTAT = disat; // disable saturation to allow wraparound
AYO = MR1; // move thhat
AR = MR1 + AYO; // 2*thhat, if overflow => auto wraparound
AQO = AR;
CALL sin; // find sin(2*thhat), result in AR
MXX = AR; // move sin(2*thhat)
AYO = AQO; // move 2*thhat
AQO = 0; // this is .5 (corresponds to pi/2)
AR = AQO - AYO; // pi/2 - 2*thhat
AQO = AR;
CALL sin; // find sin(pi/2-2*thhat) = cos(2*thhat)
MX1 = AR; // move cos(2*thhat)
I2 = q;
MYO = DM(I2,MO); // load 2^q(2)
MY1 = DM(I2,MO); // load 2^q(3)
MR = MX1+MY1 (SS); // 2^q(3)*cos(2*thhat)
MR = MR-MXO+MYO (RND); // 2^q(3)*cos(2*thhat)-2^q(2)*sin(2*thhat)
DM(observ) = MR1; // save observ
MsTAT = ensat; // enable saturation again

// average the currents
IQ = ilalfa; I1 = ibeta;
I2 = t;
AR = DM(IO,MO); // load ilalfa(q)
SR = ASHIFT AR BY -1 (LD); // ilalfa(q)/2
AQO = SRO; // move ilalfa(q)/2
MR = 0; //
CNTN = 6;
DO alfa_av UNTIL CE;
AR = DM(IO,MO); // load ilalfa(i)
SR = ASHFAR BY -1 (IQ);  // ialfa(i)/2
AYQ = SR0;
AR = AXO + AYO;  // ialfa(i)/2 + ialfa(i)/2
MYO = DM(I2,MO);  // load t(i)
MR = MR+AR+MYO (NND);  // (ialfa(i-1)+ialfa(i))/2+t(i)

alpha_av:
AXO = AYO;  // move ialfa(i)/2
SR = ASHFAR MR1 BY 2 (IQ);  // multiply by 4
MXO = SR0;  // move ialfa(i)/2

JZ = t;
AR = DM(I1,MO);  // load ialfa(i)
SR = ASHFAR AR BY -1 (IQ);  // ialfa(i)/2
AYQ = SR0;
AR = AXO + AYO;  // ialfa(i-1)/2 + ialfa(i)/2
MYO = DM(I2,MO);  // load t(i)
MR = MR+AR+MYO (NND);  // (ialfa(i-1)+ialfa(i))/2+t(i)

beta_av:
AXO = AYO;  // move ialfa(i)/2
SR = ASHFAR MR1 BY 2 (IQ);  // multiply by 4
MYO = SR0;  // move ialfa(i)/2

// convert to d-q frame

MSTAT = dissat;  // disable saturation to allow wraparound
AXO = DM(thhat);  // load thhat
CALL sin;  // find sin(thhat)
DM(sinth) = AR;  // save sin(thhat)
AYQ = AXO;  // move thhat
AXO = 0x40QQ;  // this is .5 (corresponds to pi/2)
AR = AXO - AYO;  // pi/2 - thhat
AXO = AR;

CALL sin;  // find sin(pi/2-thhat) = cos(thhat)
DM(coth) = AR;  // save coth
MSTAT = ensat;  // enable saturation again
MX1 = MYO;  // move i beta
MYO = DM(sinth);  // load sin(thhat)
MY1 = AR;  // move cos(thhat)
MR = MR+MYO (NND);  // ialfa*cos(thhat)
MX = MR+MYO (NND);  // ialfa*cos(thhat)+ibeta*sin(thhat)
AYO = MR1;  // move id
MR = MR+MYO (NND);  // ialfa*cos(thhat)
AY1 = MR1;  // move iq

// id control block
AXO = DM(idref);  // load idref
AR = AXO-AYO;  // idref-id
MR1 = DM(int1);  // load int1
MRQ = 0;
MYO = erricoeffint;  // load erricoeffint
MR = MR+AR+MYO (NND);  // int1(idref-id)*erricoeffint
IF MY SAT MR;  // saturate int1
DM(int1) = MR1;  // save int1
MR1 = erricoeffout;  // load erricoeffout
MR = MR+AR+MYO (NND);  // int1(idref-id)*erricoeffout
IF MY SAT MR;  // saturate int2
AYO = MR1;  // move
AF = AR-AYO;  // int1(idref-id)*(erricoeffout+1)
AR = AR+AF;  // int1(idref-id)*(erricoeffout+2)
NAO = AR;  // move vdc

// w control block
AXO = DM(wref);  // load wref
AYO = DM(othhat);  // load othhat
AR = AXO-AYO;  // wref-what
MR1 = DM(int2);  // load int2
MRQ = 0;
MYO = erricoeffint;  // load erricoeffint
MR = MR+AR+MYO (NND);  // int2(wref-what)*erricoeffint
IF MY SAT MR;  // saturation of int2
APPENDIX C. ASSEMBLER CODE FOR THE DSP

112

DM(int3) = M1; // save int3
MYO = int2coeff; // load int2coeff
MY1 = err2coeffout; // load err2coeffout
MR = M1+MYO (SS); // int2+int2coeff
MR = M1+MY+MY1 (NND); // int2+int2coeff+(valref-what)*err2coeffout
AN = ABS M11; // find absolute value of iqref
AYO = ilimit; // load ilimit
NONE = AR-AYO; // compare
IF GT JUMP sat_iqref; // if abs(iqref)>ilimit, saturate
AXO = M11; // move iqref, no saturation
JUMP iq_control; // continue with controller

sat_iqref:
AXO = ilimit; // load absolute limit
NONE = PASS M11; // check sign of original iqref
IF GT JUMP iq_control; // if positive, leave ilimit at AXO
AR = -AXO; // if negative, negate
AXO = AR; // move -ilimit to AXO

iq_control:
// iq_control block
AR = AXO-A1; // iqref-iq
M11 = DM(int3); // load int3
MR = 0; // MYO = err2coeffint
MR = M1+MY+MYO (NND); // int3+(iqref-iq)*err2coeffint
IF MY SAT M1; // saturation of int3
DM(int3) = M11; // save int3
MY1 = err2coeffout; // load err2coeffout
MR = M1+MY+MY1 (NND); // int3+(iqref-iq)*err2coeffout
IF MY SAT MR;
AYO = M11; // move
AR = M1+AYO; // int3+(iqref-iq)*(err2coeffout+1)
AX1 = AR; // move vq

// convert back to alpha-beta frame
MSTAT = dissat; // disable saturation to allow wraparound
AXO = DM(that); // load that
AR = PASS AXO; // move that to AR
MYO = delaycoeff; // load delaycoeff
MR = AN+MYO (NND); // delaycoeff+omhat
AYO = DM(that); // load that
AR = M1+AYO; // that+delaycoeff+omhat, allow wraparound
AXO = AR; // move corrected that
CALL sin; // find sin(pi/2-corrthat)
MYO = AR; // move sin(corrthat)
AYO = AXO; // move corrthat
AXO = 0x4000; // this is 6.5 (corresponds to pi/2)
AR = AXO - AYO; // pi/2-corrthat, allow wraparound
AXO = AR; // move pi/2-corrthat
CALL sin; // find sin(pi/2-corrthat) = cos(corrthat)
MY1 = AR; // move cos(corrthat)
MSTAT = enssat; // enable saturation again
MY = MYO+MY1 (SS); // vdc+costh
MX1 = AX1; // move vq
MR = MR-MX1+MYO (NND); // vdc+costh-vq=sinthat
DM(valfa) = M11; // save valfa
MR = MYO+MY (SS); // vdc=sinthat
MR = M1+MX1+MY1 (NND); // vdc+sinthat=vq+costh
DM(vbeta) = M11; // save vbeta

// find sector and rotate vector
AR = PASS MR1; // check sign of vbeta
IF LE JUMP vbp;

vbp:
AXO = DM(valfa); // check sign of valfa
AR = PASS AXO; // if LE JUMP vbpvan;

vbpvan:
MX1 = M11; // move vbeta
MX1 = AXO; // move valfa
MR = 0;
MYO = cnlosr3; // load 1/sqrt(3)
MR = MR-MX1+MYO (NND); // valfa-vbeta+cnlosr3
AR = PASS MR1;
IF LE JUMP sector2;

sector2:
AX1 = 1;
DM(sector) = AX1;  // sector = 1
        // valfa and vbets remain the same
JUMP endsec;

sector2:
AX1 = 2;
DM(sector) = AX1;  // sector = 2
MXO = AX0;  // move valfa
MYO = cn1o2;  // load 1/2
MY1 = sr3o2;  // load sqrt(3)/2
MR = MXO+MYO (SS);  // valfa/2
MR = MX1+MY1 (RRD);  // valfa/2+vbeta*sqrt(3)/2
DM(valfa) = MR1;  // save rotated valfa
MR = MX1+MYO (SS);  // vbeta/2
MR = MR-MXO+MY1 (RRD);  // vbeta/2-valfa*sqrt(3)/2
DM(vbeta) = MR1;  // save rotated vbeta
JUMP endsec;

vbvan:
MX1 = MR1;  // move vbeta
MX1 = AX0;  // move valfa
MYO = cn0;  // load 1/sqrt(5)
MYO = cn1o3+MR (RRD);  // valfa+vbeta*cn1o3
AR = PASS MR1;
IF LE JUMP sector2;

sector3:
AX1 = 3;
DM(sector) = AX1;  // sector = 3
MXO = AX0;  // move valfa
MYO = cn1o2;  // load 1/2
MY1 = sr3o2;  // load sqrt(3)/2
MR = MX1+MY1 (RRD);  // vbeta*sqrt(3)/2
MR = MXO+MYO (RRD);  // vbeta*sqrt(3)/2-valfa/2
DM(valfa) = MR1;  // save rotated valfa
MR = 0;
MR = MR-MX1+MYO (SS);  // -vbeta/2
MR = MR-MXO+MY1 (RRD);  // -vbeta/2-valfa*sqrt(3)/2
DM(vbeta) = MR1;  // save rotated vbeta
JUMP endsec;

vbn:
AXO = DM(valfa);  // check sign of valfa
AR = PASS AX0;
IF LE JUMP vbvan;

vbvanp:
MX1 = MR1;  // move vbeta
MX1 = AX0;  // move valfa
MYO = cn0;  // load 1/sqrt(5)
MYO = cn1o3+MR (RRD);  // valfa+vbeta*cn1o3
AR = PASS MR1;
IF LE JUMP sector5;

sector6:
AX1 = 6;
DM(sector) = AX1;  // sector = 6
MXO = AX0;  // move valfa
MYO = cn1o2;  // load 1/2
MY1 = sr3o2;  // load sqrt(3)/2
MR = MXO+MYO (SS);  // valfa/2
MR = MR-MX1+MY1 (RRD);  // valfa/2-vbeta*sqrt(3)/2
DM(valfa) = MR1;  // save rotated valfa
MR = MX1+MYO (SS);  // vbeta/2
MR = MR-MXO+MY1 (RRD);  // vbeta/2-valfa*sqrt(3)/2
DM(vbeta) = MR1;  // save rotated vbeta
JUMP endsec;

sector5:
AX1 = 5;
DM(sector) = AX1;  // sector = 5
MXO = AX0;  // move valfa
MYO = cn1o2;  // load 1/2
MY1 = sr3o2;  // load sqrt(3)/2
MR = 0;
MR = MR-MXO+MYO (SS);  // -valfa/2
MR = MR-MX1+MY1 (RRD);  // -valfa/2-vbeta*sqrt(3)/2
DM(valfa) = MR1;  // save rotated valfa
MR = MXO+MY1 (SS);  // valfa*sqrt(3)/2
MR = MR-MX1+MYO (RRD);  // valfa*sqrt(3)/2-valfa/2
APPENDIX C. ASSEMBLER CODE FOR THE DSP

```
DM(vbeta) = MR1;           // save rotated vbeta
JMP endsec;

vbmnan:
   MX1 = MR1;               // move vbeta
   MR1 = AX0;               // move valfa
   MYO = 0;                 // load 1/sqrt(3)
   MYO = cmlsr3;            // load 1/sqrt(3)
   MH = MR1+MYO (NND);      // valfa-vbeta-cmlsr3
   AR = PASS MR1;
   IF GE JMP sector5;

sector4:
   AX1 = 4;
   DM(sector) = AX1;        // sector = 4
   AR = -AX0;               // -valfa
   DM(valfa) = AR;          // save rotated valfa
   AXO = MX1;               // move vbeta
   AR = -AXO;               // -vbeta
   DM(vbeta) = AR;          // save rotated vbeta
   endsec:

   // limit voltage magnitude
   Mq1 = DM(valfa);         // load valfa
   MXO = DM(vbeta);         // load vbeta
   MYO = cmlsr3;            // load 1/sqrt(3)
   MH = Mq1+MXO+MYO (NND);  // valfa+beta/sqrt(3)
   MYO = cmlsr2;            // load 1/2/sqrt(2)
   AY0 = Mq1-AX0;           // valfa+beta/sqrt(3) - 1/2/sqrt(2)
   IF LE JMP nolimit;

limit:
   MYO = yr2o2;             // load sqrt(2)/2
   MH = Mq1+MYO (NND);      // (valfa+beta/sqrt(3))*sqrt(2)/2
   AXO = MR1;               // move result
   SI = DM(valfa);          // load valfa
   SR = ASHIFT SI BY -2 (1Q); // valfa/4
   AY1 = SRO;               // move valfa/4
   DIVS AY1,AXO;            // (valfa/4)/(valfa+beta/sqrt(3))*sqrt(2)/2
   DIVQ AXO;                // DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO;
   DIVQ AXO;                // DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO;
   DIVQ AXO;                // DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO;
   DM(valfa) = AY0;         // save new value of valfa
   SI = DM(vbeta);          // load vbeta
   SR = ASHIFT SI BY -2 (1Q); // vbeta/4
   AY1 = SRO;               // move vbeta/4
   DIVS AY1,AXO;            // (vbeta/4)/(valfa+beta/sqrt(3))*sqrt(2)/2
   DIVQ AXO;                // DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO;
   DIVQ AXO;                // DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO;
   DIVQ AXO;                // DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO; DIVQ AXO;
   DM(vbeta) = AY0;         // save new value of vbeta

nolimit:

   // compute counter values
   IQ = cnt;                // pointer to cnt
   LQ = 6;                  // circular buffer length 6
   AX0 = DM(sector);        // load sector
   AX1 = cmlsr6;            // load 1/6
   MXO = DM(valfa);         // load valfa
   MYO = cmlsr6;            // load 1/sqrt(6)
   MH = MXO+MYO (NND);      // valfa+1/sqrt(6)
   AY0 = MR1;               // move new valfa
   MXO = DM(vbeta);         // load vbeta
   MYO = yr2o6;             // load sqrt(2)/6
   MH = MXO+MYO (NND);      // vbeta*sqrt(2)/6
   AY1 = MR1;               // move new vbeta
   AR = AX0-1;              // sector-1
   M2 = AR;                 // point to cnt(sector)
   AR = AX1+AY0;
   AR = AR+AY0;
   AR = AR+AY0;
   AR = AR+AY0;
   AR = AR-AY1;
   AR = AR-AY1;
```
AN = AN-AY1;
AN = AN-AY1;
AN = AN-AY1;
AN = AN-AY1;  // 1/6-valfa-7*vbta
DM(1Q,MO) = AN;  // save cnt(sector)
AN = AX1-AY0;
AN = AN+AY1;
AN = AN+AY1;
AN = AN-AY1;
AN = AN+AY1;
AN = AN+AY1;
AN = AN+AY1;
AN = AN-AY1;
AN = AN+AY1;  // 1/6-valfa+11*vbta
DM(1Q,MO) = AN;  // save cnt(sector+i)
AN = AX1-AY0;
AN = AN-AY1;  // 1/6-valfa-vbta
CNTR = 4;
DO load_cntn UNTIL CE;
    load_cntn:
        DM(1Q,MO) = AN;  // save cnt(i)
        LO = 0;  // disable circular buffer
        // convert counter values from fraction to integer
        IO = cnt;
        MYO = cptpwmsfr;
        AYO = cptealdead;  // load cptealdead
        CNTR = 6;
        DO scale_cnt UNTIL CE;
            AXO = DM(1Q,MO);  // load cnt(i)
            MX = MXO*MYO (SS);  // cnt(i)*cptealdead
            AN = AX1-AY0;  // subtract deadband time
            scale_cnt:
                DM(1Q,MO) = AN;  // save integer cnt(i)
                // output counter values and position
                IO = cnt;
                AXO = DM(1Q,MO);
                IO(pt1) = AXO;
                AXO = DM(1Q,MO);
                IO(pt2) = AXO;
                AXO = DM(1Q,MO);
                IO(pt3) = AXO;
                AXO = DM(1Q,MO);
                IO(pt4) = AXO;
                AXO = DM(1Q,MO);
                IO(pt5) = AXO;
                AXO = DM(1Q,MO);
                IO(pt6) = AXO;
                AXO = DM(thbar);
                IO(ppos) = AXO;
                // save last current measurement
                IO = ialfa;  // pointer to ialfa
                M2 = 6;
                MODIFY(1O,M2);  // point to last value
                AXO = DM(1Q,MO);
                IO = ialfa;
                DM(1Q,MO) = AXO;  // copy last value to first position
                IO = ibeta;  // pointer to ibeta
                MODIFY(1O,M2);  // point to last value
                AXO = DM(1Q,MO);
                IO = ibeta;
                DM(1Q,MO) = AXO;  // copy last value to first position
                // save time intervals for next period
                IO = tn;
                1 = t;
                2 = cnt;
                CNTR = 6;
        DO copy_times UNTIL CE;
            AXO = DM(1Q,M1);  // load tn(i), don't increment IO
            AX1 = DM(12,MO);  // load cnt(i), increment 22
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
   AR, AF, AR, MY1, MX1, MF, MR, SR, I3
   Computation Time
   26 cycles
   ( remember cos(x) = sin(pi/2-x) )
*/

.GLOBAL sin;

.SECTION/DATA sin.data;

.VAR sin_coeff[5] = 0x3240, 0x0053, 0x08B7, 0x10CE;

.SECTION/CODE sin;

sin:

    I3=min_coeff;   // Pointer to coeff. buffer
    AYO=0x4000;
    AR=AX0, AF=AX0 AND AYO;   // Check 2nd or 4th quad.
    IF WE AR=AX0;   // If yes, negate input
    AYO=0x7FFF;
    AR=AR AND AYO;    // 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
    CWTR=3;
    DO approx UNTIL CE;
    MR=MR+MX1+MF (SS);

    approx:
    MF=AR+MF (SS), MX1=DM(I3,M3);
    MR=MR+MX1+MF (SS);
    SR=ASHIFT MR1 BY 3 (WI);
    SR=SR OR LSHIFT MR0 BY 3 (LQ); // Convert to 1.15 format
    AR=PASS SR1;
    IF LT AR=PASS AYO;   // Saturate if needed
    AF=PASS AYO;
    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_init { TYPE(PM RAM) START(0x00000) END(0x00000) WIDTH(24) }
    seg_tab  { TYPE(PM RAM) START(0x00000) END(0x00000) WIDTH(24) }
    seg_irq  { TYPE(PM RAM) START(0x01000) END(0x02000) WIDTH(24) }
}
APPENDIX C. ASSEMBLER CODE FOR THE DSP

```assembly
seg.sin  { TYPE(PM RAM) START(0x03000) END(0x03fff) WIDTH(24) }
seg.pwndata { TYPE(IN RAM) START(0x02000) END(0x02fff) WIDTH(16) }
seg.sindata { TYPE(IN RAM) START(0x01000) END(0x01fff) WIDTH(16) }
}
PROCESSOR p0
{
    LINK_AGAINST( $COMMAND_LINE_OUTPUT_FILE )
    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.pwndata
        {
            INPUT_SECTIONS( $OBJECTS(pwndata) )
        } >seg.pwndata
        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;
    ions_n: IN STD_LOGIC;
    depa: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
    depd: INOUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    adca: IN adctype;
    adcb: IN adctype;
    clkcout: OUT STD_LOGIC;
    irq_n: OUT STD_LOGIC;
    pwm_sig: OUT pwmvec;
    position: OUT postype;
    tp: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
    imp75: IN STD_LOGIC;
    imp70: IN STD_LOGIC;
    imp80: IN STD_LOGIC;
    imp81: IN STD_LOGIC
  );
END pwm;

ARCHITECTURE flexk OF pwm IS
  SIGNAL new_time: timetype;
  SIGNAL status: STD_LOGIC;
  SIGNAL step: stepype;

118
```vhdl
-- bit reversing of ADC input because of mistake in PCB design
adc13 <= adca(0);
adc12 <= adca(1);
adc11 <= adca(2);
adc10 <= adca(3);
adc9 <= adca(4);
adc8 <= adca(5);
adc7 <= adca(6);
adc6 <= adca(7);
adc5 <= adca(8);
adc4 <= adca(9);
adc3 <= adca(10);
adc2 <= adca(11);
adc1 <= adca(12);
adc0 <= adca(13);
adcbr13 <= aadc(0);
adcbr12 <= aadc(1);
adcbr11 <= aadc(2);
adcbr10 <= aadc(3);
adcbr9 <= aadc(4);
adcbr8 <= aadc(5);
adcbr7 <= aadc(6);
adcbr6 <= aadc(7);
adcbr5 <= aadc(8);
adcbr4 <= aadc(9);
adcbr3 <= aadc(10);
adcbr2 <= aadc(11);
adcbr1 <= aadc(12);
adcbr0 <= aadc(13);
```

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

```vhdl
-- instantiate the 'reg' module
reg: reg PORT MAP (
  clk => cliko,
  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 => dma,
  adca => adcar,
  adcb => adcbh,
  data => dspd,
  curr_vector => pwm_sig,
  new_time => new_time,
  position => position,
  status => status,
  int_n => interr
);
```

```vhdl
-- instantiate the PWM counter
<ctr: pwmctr PORT MAP (
  clk => clk,
  rst => status,
  new_time => new_time,
```
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)
  BEGIN
    -- variable for the counter
    VARIABLE cnt: timetype;
    -- variable for the step
    VARIABLE stp: steptype;
    -- variable to indicate the beginning of a new PWM period
    VARIABLE np: STD_LOGIC;
    -- variable to indicate the beginning of a new step
    VARIABLE ns: STD_LOGIC;

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

    IF (clk'EVENT AND clk = '1') THEN
      IF (rst = '0') THEN
        -- reset all variables
        cnt := 0;
        stp := 0;
      ELSE
        -- increment counter
        cnt := cnt + 1;
        IF (cnt = new_time) THEN
          -- load new value
          np := '1';
          ns := '1';
          stp := 0;
        END IF;
      END IF;
    END IF;
  END PROCESS;

END;
APPENDIX D. VHDL CODE FOR THE FPGA

```
np := 'O';
sn := 'O';
ELSIF (cnt = 0) THEN
  -- load counter with new value
  cnt := new_time - i;
  -- 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;
ELSIF (cnt = 0) THEN
  -- decrease counter and erase np and ns variables
  cnt := cnt - 1;
  np := 'O';
  ns := 'O';
END IF;
END IF;

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

D.3 regs.vhd

```vhdl
-- 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; these values can be read 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 'pwmctrl' 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
```
APPENDIX D. VHDL CODE FOR THE FPGA

```vhdl
-- position
-- time 1  ADC b 1
-- time 2  ADC b 2
-- time 3  ADC b 3
-- time 4  ADC b 4
-- time 5  ADC b 5
-- time 6  ADC b 6
-- time db

-- 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.pwm&.ALL;

ENTITY regs IS
PORT (  clk:  IN STD_LOGIC;
    rst:  IN STD_LOGIC;
    ios_m:  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 flexib OF regs IS

-- generic write signal
SIGNAL wrs:  STD_LOGIC;

-- signals for the value of each vector
SIGNAL svi:  pwmvec;
```
APPENDIX D. VHDL CODE FOR THE FPGA

SIGNAL sw2: pwnvec;
SIGNAL sw3: pwnvec;
SIGNAL sw4: pwnvec;
SIGNAL sw5: pwnvec;
SIGNAL sw6: pwnvec;

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

-- 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 stdb: 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 sadc1: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL sadcb1: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL sadc2: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL sadc3: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL sadca: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL sadc4: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL sadc5: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL sadc6: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL sadcb6: STD_LOGIC_VECTOR (15 DOWNTO 0);

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

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

-- signal for the position
SIGNAL pos: pos_type;

BEGIN

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

-- write process
PROCESS (wr, rst)

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

-- variables to store the standby time values
VARIABLE t1: timevec;
VARIABLE t2: timevec;
VARIABLE t3: timevec;
VARIABLE t4: timevec;
VARIABLE t5: timevec;
APPENDIX D. VHDL CODE FOR THE FPGA

```vhdl
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 (6 DOWNTO 0);
    END IF;
    IF (sta = '0' AND address = "0001") THEN
        v2 := data (6 DOWNTO 0);
    END IF;
    IF (sta = '0' AND address = "0010") THEN
        v3 := data (6 DOWNTO 0);
    END IF;
    IF (sta = '0' AND address = "0011") THEN
        v4 := data (6 DOWNTO 0);
    END IF;
    IF (sta = '0' AND address = "0100") THEN
        v5 := data (6 DOWNTO 0);
    END IF;
    IF (sta = '0' AND address = "0101") THEN
        v6 := data (6 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;
```

APPENDIX D. VHDL CODE FOR THE FPGA

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)
BEGIN
  -- 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 tt2db:      timevec;
  -- variables to store the temporary ADC values
  VARIABLE tadc1:      adctype;
  VARIABLE tadc1b:     adctype;
  VARIABLE tadc2:      adctype;
  VARIABLE tadc2b:     adctype;
  VARIABLE tadc3:      adctype;
  VARIABLE tadc3b:     adctype;
  VARIABLE tadc4:      adctype;
  VARIABLE tadc4b:     adctype;
  VARIABLE tadc5:      adctype;
  VARIABLE tadc5b:     adctype;
  -- variables to store the firm ADC values
  VARIABLE adc1:       adctype;
  VARIABLE adc1b:      adctype;
  VARIABLE adc2:       adctype;
  VARIABLE adc2b:      adctype;
  VARIABLE adc3:       adctype;
  VARIABLE adc3b:      adctype;
  VARIABLE adc4:       adctype;
  VARIABLE adc4b:      adctype;
  VARIABLE adc5:       adctype;
  VARIABLE adc5b:      adctype;
  VARIABLE adc6:       adctype;
  VARIABLE adc6b:      adctype;

  IF (clk'EVENT AND clk='1') THEN
    IF (stat = '0') THEN
      IF (new_period = '1') THEN
        -- system stopped, then output must be zero
        pwmout := pwm_off;
      ELSE
        -- latch the selected PWM vector
        pwmout := sel_vec;
      END IF;
    ELSE
      -- 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;
    END IF;
  END IF;
END PROCESS;
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 =>
      tadc1 := adc1;
      tadb1 := adcb;
    WHEN 4 =>
      tadc2 := adc2;
      tadb2 := adcb;
    WHEN 6 =>
      tadc3 := adc3;
      tadb3 := adcb;
    WHEN 8 =>
      tadc4 := adc4;
      tadb4 := adcb;
    WHEN 10 =>
      tadc5 := adc5;
      tadb5 := adcb;
    WHEN 0 =>
      adc1 := tadc1;
      adcb1 := tadb1;
      adc2 := tadc2;
      adcb2 := tadb2;
      adc3 := tadc3;
      adcb3 := tadb3;
      adc4 := tadc4;
      adcb4 := tadb4;
      adc5 := tadc5;
      adcb5 := tadb5;
      adc6 := adca;
      adcb6 := adcb;
  WHEN OTHERS =>
  END CASE;
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 i bit to the left and
-- extended to 16 bits with sign extension)
sadc1 (0) <= '0';
sadc1 (13 DOWNTO 1) <= adc1 (12 DOWNTO 0);
sadc1 (14) <= NOT adc1(13);
sadc1 (15) <= NOT adc1(13);
sadb1 (0) <= '0';
sadb1 (13 DOWNTO 1) <= adcb1 (12 DOWNTO 0);
sadb1 (14) <= NOT adcb1(13);
sadb1 (15) <= NOT adcb1(13);
sadc2 (0) <= '0';
sadc2 (13 DOWNTO 1) <= adc2 (12 DOWNTO 0);
sadc2 (14) <= NOT adc2(13);
sadc2 (15) <= NOT adc2(13);
sadb2 (0) <= '0';
sadb2 (13 DOWNTO 1) <= adcb2 (12 DOWNTO 0);
sadb2 (14) <= NOT adcb2(13);
sadb2 (15) <= NOT adcb2(13);
sadc3 (0) <= '0';
sadc3 (13 DOWNTO 1) <= adc3 (12 DOWNTO 0);
sadc3 (14) <= NOT adc3(13);
sadc3 (15) <= NOT adc3(13);
sadb3 (0) <= '0';
sadb3 (13 DOWNTO 1) <= adcb3 (12 DOWNTO 0);
APPENDIX D. VHDL CODE FOR THE FPGA

127

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 <=
(s1 AND sv6) WHEN 0,
s1 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,
else 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(stt2)) WHEN 1,
CONV_INTEGER(UNSIGNED(sttda)) WHEN 2,
CONV_INTEGER(UNSIGNED(sttd)) WHEN 3,
CONV_INTEGER(UNSIGNED(sttda)) WHEN 4,
CONV_INTEGER(UNSIGNED(sttda)) WHEN 5,
CONV_INTEGER(UNSIGNED(sttdb)) WHEN 6,
CONV_INTEGER(UNSIGNED(sttdb)) WHEN 7,
CONV_INTEGER(UNSIGNED(sttda)) WHEN 8,
CONV_INTEGER(UNSIGNED(sttda)) WHEN 9,
CONV_INTEGER(UNSIGNED(stt6)) WHEN 10,
CONV_INTEGER(UNSIGNED(sttdb)) WHEN 11,
else WHEN OTHERS;

-- signal to read the status bit
srsta(0) <= stat;
srsta(15 DOWNTO 1) <= "0000000000000000";

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

-- select the data to be read
WITH address SELECT
srdat <=
sdca1 WHEN "00000",
sdca2 WHEN "00010",
sdca3 WHEN "00100",
sdca4 WHEN "00110",
sdca5 WHEN "01000",
sdca6 WHEN "01010","
APPENDIX D. VHDL CODE FOR THE FPGA

---

```vhdl
-- generate PWM
data <= srdata WHEN srden = '0' ELSE '1'
```

---

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

---

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

---

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

END flexB;

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 stepvec IS INTEGER RANGE 0 TO 15; -- 4 bits
SUBTYPE steptype 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: IN steptype;
  new_period: OUT STD_LOGIC;
  new_step: OUT STD_LOGIC;
);
END COMPONENT;

COMPONENT reg3
PORT (
  clk: IN STD_LOGIC;
  rst: IN STD_LOGIC;
  ions_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: IN OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
  curr_vector: OUT pwmvec;
  new_time: OUT timevec;
  position: OUT postype;
);```
## D.5 Compilation report (edited version)

**Project Information**

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 **

<table>
<thead>
<tr>
<th>Chip/Device</th>
<th>Input Pins</th>
<th>Output Pins</th>
<th>Bidir Pins</th>
<th>LFs</th>
<th>% Utilized</th>
</tr>
</thead>
<tbody>
<tr>
<td>pwm</td>
<td>41</td>
<td>28</td>
<td>16</td>
<td>992</td>
<td>75%</td>
</tr>
</tbody>
</table>

** User Pins:**

|            | 41  | 28  | 16  |

** PIN/LOCATION/CHIP ASSIGNMENTS **

<table>
<thead>
<tr>
<th>User Assignments</th>
<th>Actual Assignments (if different)</th>
<th>Node Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pwm136</td>
<td>adc0</td>
<td></td>
</tr>
<tr>
<td>pwm137</td>
<td>adc1</td>
<td></td>
</tr>
<tr>
<td>pwm138</td>
<td>adc2</td>
<td></td>
</tr>
<tr>
<td>pwm139</td>
<td>adc3</td>
<td></td>
</tr>
<tr>
<td>pwm140</td>
<td>adc4</td>
<td></td>
</tr>
<tr>
<td>pwm141</td>
<td>adc5</td>
<td></td>
</tr>
<tr>
<td>pwm142</td>
<td>adc6</td>
<td></td>
</tr>
<tr>
<td>pwm143</td>
<td>adc7</td>
<td></td>
</tr>
<tr>
<td>pwm144</td>
<td>adc8</td>
<td></td>
</tr>
<tr>
<td>pwm145</td>
<td>adc9</td>
<td></td>
</tr>
<tr>
<td>pwm146</td>
<td>adc10</td>
<td></td>
</tr>
<tr>
<td>pwm147</td>
<td>adc11</td>
<td></td>
</tr>
<tr>
<td>pwm148</td>
<td>adc12</td>
<td></td>
</tr>
<tr>
<td>pwm149</td>
<td>adc13</td>
<td></td>
</tr>
<tr>
<td>pwm150</td>
<td>adc14</td>
<td></td>
</tr>
<tr>
<td>pwm151</td>
<td>adc15</td>
<td></td>
</tr>
<tr>
<td>pwm152</td>
<td>adc16</td>
<td></td>
</tr>
<tr>
<td>pwm153</td>
<td>adc17</td>
<td></td>
</tr>
<tr>
<td>pwm154</td>
<td>adc18</td>
<td></td>
</tr>
<tr>
<td>pwm155</td>
<td>adc19</td>
<td></td>
</tr>
<tr>
<td>pwm156</td>
<td>adc20</td>
<td></td>
</tr>
<tr>
<td>pwm157</td>
<td>adc21</td>
<td></td>
</tr>
<tr>
<td>pwm158</td>
<td>adc22</td>
<td></td>
</tr>
<tr>
<td>pwm159</td>
<td>adc23</td>
<td></td>
</tr>
<tr>
<td>pwm160</td>
<td>adc24</td>
<td></td>
</tr>
<tr>
<td>pwm161</td>
<td>adc25</td>
<td></td>
</tr>
<tr>
<td>pwm162</td>
<td>adc26</td>
<td></td>
</tr>
<tr>
<td>pwm163</td>
<td>adc27</td>
<td></td>
</tr>
<tr>
<td>pwm164</td>
<td>adc28</td>
<td></td>
</tr>
<tr>
<td>pwm165</td>
<td>adc29</td>
<td></td>
</tr>
<tr>
<td>pwm166</td>
<td>adc30</td>
<td></td>
</tr>
<tr>
<td>pwm167</td>
<td>adc31</td>
<td></td>
</tr>
<tr>
<td>pwm168</td>
<td>adc32</td>
<td></td>
</tr>
<tr>
<td>pwm169</td>
<td>adc33</td>
<td></td>
</tr>
<tr>
<td>pwm170</td>
<td>adc34</td>
<td></td>
</tr>
<tr>
<td>pwm171</td>
<td>adc35</td>
<td></td>
</tr>
<tr>
<td>pwm172</td>
<td>clk</td>
<td></td>
</tr>
<tr>
<td>pwm173</td>
<td>clkout</td>
<td></td>
</tr>
<tr>
<td>pwm174</td>
<td>depa0</td>
<td></td>
</tr>
<tr>
<td>pwm175</td>
<td>depa1</td>
<td></td>
</tr>
<tr>
<td>pwm176</td>
<td>depa2</td>
<td></td>
</tr>
<tr>
<td>pwm177</td>
<td>depa3</td>
<td></td>
</tr>
<tr>
<td>pwm178</td>
<td>depa4</td>
<td></td>
</tr>
</tbody>
</table>

---

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

---

```vhdl
END pwmkg;
```

---

```vhdl
```
** FILE HIERARCHY **

```
|reg:reg|
|pmctr:pmctr| lpm_add_sub:148| lpm_add_sub:148| addcore:addder|
|pmctr:pmctr| lpm_add_sub:148| altshift:reslut_ext_latency_ffs|
|pmctr:pmctr| lpm_add_sub:148| altshift:flow_ext_latency_ffs|
|pmctr:pmctr| lpm_add_sub:177| addcore:addder|
|pmctr:pmctr| lpm_add_sub:177| altshift:reslut_ext_latency_ffs|
|pmctr:pmctr| lpm_add_sub:177| altshift:carry_ext_latency_ffs|
|pmctr:pmctr| lpm_add_sub:177| altshift:flow_ext_latency_ffs|
|pmctr:pmctr| lpm_add_sub:247| addcore:addder|
|pmctr:pmctr| lpm_add_sub:247| altshift:reslut_ext_latency_ffs|
|pmctr:pmctr| lpm_add_sub:247| altshift:carry_ext_latency_ffs|
|pmctr:pmctr| lpm_add_sub:247| altshift:flow_ext_latency_ffs|
```

***** Logic for device 'pm' compiled without errors.
Device: EPF6Q16TC144-3

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
APPENDIX D. VHDL CODE FOR THE FPGA

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).
VCCO = 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 to
& = JTAG pin used for I/O. When used as user I/O, JTAG pins must be kept stable before and during configuration. JTAG

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total dedicated input pins used:</td>
<td>1/4 (25%)</td>
</tr>
<tr>
<td>Total I/O pins used:</td>
<td>84/113    (74%)</td>
</tr>
<tr>
<td>Total logic cells used:</td>
<td>96/21320  (75%)</td>
</tr>
<tr>
<td>Average fan-in:</td>
<td>3.50/4    (87%)</td>
</tr>
<tr>
<td>Total fan-in:</td>
<td>3476/6580 (65%)</td>
</tr>
<tr>
<td>Total input pins required:</td>
<td>41</td>
</tr>
<tr>
<td>Total output pins required:</td>
<td>28</td>
</tr>
<tr>
<td>Total bidirectional pins required:</td>
<td>16</td>
</tr>
<tr>
<td>Total reserved pins required:</td>
<td>0</td>
</tr>
<tr>
<td>Total logic cells required:</td>
<td>96/2</td>
</tr>
<tr>
<td>Total flipflops required:</td>
<td>349</td>
</tr>
<tr>
<td>Total packed registers required:</td>
<td>0</td>
</tr>
<tr>
<td>Total logic cells in carry chains:</td>
<td>0</td>
</tr>
<tr>
<td>Total number of carry chains:</td>
<td>0</td>
</tr>
<tr>
<td>Total logic cells in cascade chains:</td>
<td>0</td>
</tr>
<tr>
<td>Total number of cascade chains:</td>
<td>0</td>
</tr>
<tr>
<td>Logic cells inserted for fitting:</td>
<td>1</td>
</tr>
<tr>
<td>Synthesized logic cells:</td>
<td>157/1320  (11%)</td>
</tr>
</tbody>
</table>

** 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

<table>
<thead>
<tr>
<th>Option</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARRY_CHAIN</td>
<td>ignore</td>
</tr>
<tr>
<td>CARRY_CHAIN_LENGTH</td>
<td>25</td>
</tr>
<tr>
<td>CASCADE_CHAIN</td>
<td>ignore</td>
</tr>
<tr>
<td>CASCADE_CHAIN_LENGTH</td>
<td>2</td>
</tr>
<tr>
<td>DECOMPOSE_GATES</td>
<td>on</td>
</tr>
<tr>
<td>DUPLICATE_LOGIC_EXTRACTION</td>
<td>on</td>
</tr>
<tr>
<td>MINIMIZATION</td>
<td>full</td>
</tr>
<tr>
<td>MULTI_LEVEL_FACTORING</td>
<td>on</td>
</tr>
<tr>
<td>NOT_GATE_PUSH_BACK</td>
<td>on</td>
</tr>
<tr>
<td>REDUCE_LOGIC</td>
<td>on</td>
</tr>
<tr>
<td>REFACCTORIZATION</td>
<td>on</td>
</tr>
<tr>
<td>REGISTER_OPTIMIZATION</td>
<td>on</td>
</tr>
<tr>
<td>RESYNTHESIZE_NETWORK</td>
<td>on</td>
</tr>
<tr>
<td>SLOW_SLEW_RATE</td>
<td>off</td>
</tr>
<tr>
<td>SUBFACTORIZATION</td>
<td>on</td>
</tr>
<tr>
<td>IGNORE_SOFT_BUFFERS</td>
<td>on</td>
</tr>
<tr>
<td>USE_LPM_FOR_ANDL_OPS</td>
<td>off</td>
</tr>
</tbody>
</table>

Other logic synthesis settings:

Automatic Global Clock = on
Automatic Global Clear = on
APPENDIX D. VHDL CODE FOR THE FPGA

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 TDU 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 = 94,896K
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 February 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 carefully, 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.

![Block Diagram of the Inverter](image-url)

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 precision 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 $0.47 \, \mu F$ (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°C, which is the maximum junction temperature admissible.
After some approximations, we obtain an estimation of the thermal resistance needed. For practical purposes, a heat sink with a thermal resistance of 0.5°C/W was selected, for which the joint temperature is less than 150°C 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-emitter-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.
C:\Program Files\ExpressPCB\inverter.pcb (Top layer)

Figure E.6: Inverter PCB (top layer)
Figure E.7: Inverter PCB (bottom layer)
E.9 Connectors

<table>
<thead>
<tr>
<th>DB15 pin</th>
<th>PCB name</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>PWM1</td>
</tr>
<tr>
<td>14</td>
<td>PWM4</td>
</tr>
<tr>
<td>15</td>
<td>PWM2</td>
</tr>
<tr>
<td>6</td>
<td>PWM5</td>
</tr>
<tr>
<td>7</td>
<td>PWM3</td>
</tr>
<tr>
<td>13</td>
<td>PWM6</td>
</tr>
<tr>
<td>4</td>
<td>FO1</td>
</tr>
<tr>
<td>11</td>
<td>FO2</td>
</tr>
<tr>
<td>3</td>
<td>FO3</td>
</tr>
<tr>
<td>10</td>
<td>FO4</td>
</tr>
<tr>
<td>5</td>
<td>DVCC</td>
</tr>
<tr>
<td>12</td>
<td>DGND</td>
</tr>
</tbody>
</table>

Table E.1: PWM connectors

<table>
<thead>
<tr>
<th>HDR pin</th>
<th>PCB name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPU</td>
</tr>
<tr>
<td>2</td>
<td>AGNDU</td>
</tr>
<tr>
<td>3</td>
<td>CPV</td>
</tr>
<tr>
<td>4</td>
<td>AGNDV</td>
</tr>
<tr>
<td>5</td>
<td>CPW</td>
</tr>
<tr>
<td>6</td>
<td>AGNDW</td>
</tr>
<tr>
<td>7</td>
<td>AGND</td>
</tr>
</tbody>
</table>

Table E.2: Current probes connector