Development of a Real Time Test Platform for Motor Drive Algorithms

Viktor Larsson

Luleå University of Technology
MSc Programmes in Engineering
Space Engineering
Department of Computer Science and Electrical Engineering
Division of EISLAB
ABSTRACT
In this thesis a real time test platform for a permanent magnet synchronous motor is developed. The implemented algorithm is Field Oriented Control (FOC) and it is implemented on a Texas Instruments TMS320F2808 Digital Signal Processor (DSP). The platform is developed in a rapid prototyping approach using Matlab/Simulink and the Real Time Workshop (RTW) packages.

With this software the control algorithm and its interface to different DSP modules, such as A/D converter and PWM module, is constructed as a Simulink block scheme. The blocks used come from ordinary Simulink libraries and libraries provided by the RTW packages. From the Simulink block scheme Matlab can auto generate embedded C code adapted for different embedded targets, in this case the 2808 DSP.

The developed real time test platform is also a Simulink model, though different from the algorithm model. When the start simulation command is given in the platform model a Graphical User Interface is loaded which lets the user specify motor parameters and certain algorithm parameters. Once the parameters are chosen RTW generates code from the algorithm model, loads it into the DSP and runs the generated program. From the platform model it is possible to set the reference speed of the motor in real time and monitor/log motor parameters such as actual speed and stator currents.
Preface

The thesis work was carried out at the Department of Power Technologies at ABB Corporate Research in Västerås from March to August 2008.

I would like to thank my supervisor at ABB, Dr. Hector Zelaya de la Parra and my supervisor at the Luleå University of Technology, Professor Kalevi Hyypää.

Finally I would also like to thank Reijo Palola, Student coordinator at ABB, and Dr. Heinz Lendenmann, Manager at the Department of Power Technologies, for giving me the opportunity to carry out my thesis at Corporate Research.

Viktor Larsson
August 2008
# TABLE OF CONTENTS

1 INTRODUCTION.........................................................................................................................1

1.1 PURPOSE .................................................................................................................................1

1.2 THESIS OUTLINE ..................................................................................................................2

2 PREVIOUS WORK .......................................................................................................................3

3 THE CACSD ENVIRONMENT .....................................................................................................4

3.1 BACKGROUND .........................................................................................................................4

3.2 THE CODE GENERATION PROCESS IN THE MATHWORKSTM CACSD ......................5

3.3 USING THE CACSD ENVIRONMENT PROVIDED BY THE MATHWORKSTM ........7

4 THEORETICAL BACKGROUND .................................................................................................10

4.1 THE PERMANENT MAGNET SYNCHRONOUS MOTOR ......................................................10

4.2 FIELD ORIENTED CONTROL ...............................................................................................12

4.2.1 A brief introduction to FOC ..............................................................................................12

4.2.2 Space Vector Modulation PWM .......................................................................................13

5 IMPLEMENTATION ....................................................................................................................16

5.1 THE USED HARDWARE .........................................................................................................17

5.1.1 DSP ...................................................................................................................................17

5.1.2 CPLD .................................................................................................................................17

5.1.3 IPM ..................................................................................................................................17

5.1.4 Motor and Resolver ...........................................................................................................18

5.2 IMPLEMENTATION IN MATLAB .......................................................................................20

5.2.1 The IQmath and DMC libraries .......................................................................................20

5.2.2 Top level ............................................................................................................................21

5.2.3 Function Call Subsystem – FOC ......................................................................................22

5.2.4 Subsystem – Measurements and Scaling .........................................................................23

5.2.4.1 Subsystem – Resolver ...................................................................................................24

5.2.5 Subsystem – Motor Control ..............................................................................................25

5.2.5.1 Subsystem – Current Control PIs ................................................................................26

5.2.5.2 Subsystem – IPM Interface .........................................................................................27

5.2.6 Function Call Subsystem – Speed Calculation ...............................................................30

5.2.7 The Interrupt Service Routines .......................................................................................31

5.2.8 Development of a User Interface ......................................................................................34

6 LABORATORY MEASUREMENTS ...........................................................................................37

7 EVALUATION OF THE MATHWORKSTM CACSD ENVIRONMENT ....................................41

7.1 THE QUALITY OF THE GENERATED CODE ......................................................................41

7.1.1 Readability ........................................................................................................................41

7.1.2 Traceability ........................................................................................................................41

7.1.3 Efficiency ..........................................................................................................................42

7.2 DSP - MATLAB COMMUNICATION PROTOCOLS ..........................................................44

7.2.1 The SCI protocol .............................................................................................................44

7.2.2 The RTDX protocol .........................................................................................................45

7.3 MATLAB USER INTERFACE ...............................................................................................46

7.4 ISSUES ENCOUNTERED DURING THE IMPLEMENTATION ..............................................47

7.4.1 Software issues .................................................................................................................47

7.4.2 Hardware issues ................................................................................................................49

7.5 DEBUGGING ..........................................................................................................................50

8 CONCLUSIONS ........................................................................................................................51

9 FURTHER WORK .......................................................................................................................53

REFERENCES .............................................................................................................................54
<table>
<thead>
<tr>
<th>Appendix</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>List of Abbreviations</td>
<td>56</td>
</tr>
<tr>
<td>B</td>
<td>The Dq Coordinate System</td>
<td>57</td>
</tr>
<tr>
<td>C</td>
<td>Coordinate Transformations</td>
<td>58</td>
</tr>
<tr>
<td>E</td>
<td>Subsystem – IPM Board Interface</td>
<td>61</td>
</tr>
<tr>
<td>F</td>
<td>Space Vector Modulation</td>
<td>62</td>
</tr>
<tr>
<td>G</td>
<td>PID – Controller</td>
<td>64</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

In the present day time efficiency and flexibility is of great importance in research and development processes throughout all industry fields. This also applies to the development, implementation and testing of control algorithms for electrical motors. The rapid development in computer technology during the past decades has revolutionized the field of motor control and enabled much more complex and efficient control algorithms, which nowadays usually are implemented on microprocessors. This development has lead to a more energy efficient use of electrical motors. Something that can generate vast energy savings in industrialized countries where about 40 to 50% of the electrical energy consumption is related to the use of electrical motors.

The process of programming a microprocessor for real time applications is a tedious and time consuming task usually done in Assembly or C. However during the last decade(s) a new approach for microprocessor programming has been developed. It is usually denoted Computer Automated (Aided) Control System Design (CACSD) and it has made the programming process more graphical and thus intuitive. The program is implemented as some form of block scheme and out of this scheme code is auto generated by the CACSD program environment. This approach is in general a lot faster than writing the code by hand, something that allows the developer to spend more time on improving functionality and performance of the program.

1.1 Purpose

The purpose of this thesis is to examine a rapid prototyping approach using CACSD to develop and implement a motor control algorithm on a Digital Signal Processor (DSP). Instead of programming the DSP in Assembly or C the control algorithm will be implemented in the Matlab/Simulink environment developed by the MathWorks\textsuperscript{TM}. The control algorithm is implemented as a block scheme in Simulink and out of this Simulink scheme Matlab can auto generate C-code for an embedded target, for example a DSP, this is done with the packages Real-Time Workshop (RTW) and RTW Embedded Coder.

Worth to point out is that the main goal is not to develop a very efficient implementation in terms of motor step response and speed reference tracking, but rather to develop a real time test platform that is easy to understand/use and to evaluate the MathWorks\textsuperscript{TM} CACSD environment from a user’s perspective. A secondary goal with the thesis is that the report should serve as a future reference document in the rapid prototyping development at Corporate Research.

Throughout the report all the different MathWorks software used; i.e. Matlab, Simulink, Real Time Workshop and the other packages, are frequently referred to as Matlab since exactly which part of the MathWorks software that is involved in the different steps is often irrelevant. Furthermore there will be quite a few abbreviations throughout this report, the first time a concept is abbreviated all of its words are written out followed by its abbreviation in parenthesis. A complete list of all the abbreviations used throughout the report is presented in alphabetical order in appendix A.
1.2 Thesis outline

The first part of the report mentions some earlier work in this field and continues with a general background to the CACSD environment that is provided by the MathWorks™. The second part continues with a brief theoretical background about the used motor type, namely the Permanent Magnet Synchronous Motor (PMSM) and the implemented control algorithm, Field Oriented Control (FOC). In part three a thorough explanation of how the control algorithm was implemented in the MathWorks™ environment is presented. In the fourth part some experimental results are presented and in the fifth and final part of the thesis an evaluation of the used environment is made and conclusions are drawn. The report concludes with some suggestions regarding further work.
2 PREVIOUS WORK

The approach to develop motor control algorithms through rapid prototyping was tested a few years back at Corporate Research in Västerås. The software environment used then was Matlab/Simulink and dSPACE. However the results from that test proved that the dSPACE software was not as flexible as desired, this in combination with rather expensive licenses made it unsuitable for further use.

When reviewing papers published in this field, i.e. digital motor control using automated code generation, it is evident that there has not been so many published. This is most likely due to that automated code generation using CACSD environments is a relatively new development. Another plausible explanation is that this type of software is mainly used in industrial applications and not so much in scientific research.

The implementation of a tuning method for a DC motor controller using a TI DSP and the MathWorks CACSD is found in [1]. A more relevant algorithm is implemented in [2], where Direct Torque Control is implemented on an induction motor using a TI DSP and the MathWorks environment. For a brief but well written introduction to the whole concept of automated code generation using Matlab see [3].

A relevant paper is [4], where the focus is directed more towards the MathWorks CACSD and how it can be configured to support a custom made DSP board. The implementation described is fairly similar to what is presented in this thesis.

It is hard to draw any major conclusions regarding the efficiency of the code generated by CACSD environments, at least in the area of motor control. Most of the published papers come from the companies that develop CACSD environments which makes them questionable. The rest come from the automotive industry, where automated code generation is widely used. In [5] and [6] comparisons are made between hand written and auto generated code produced from Simulink models using Target Link, the code generation tool of dSPACE. In both these papers the auto generated code is more efficient than the hand written code, however the algorithms implemented in these papers are not motor control algorithms. In [7] a rather old but however qualitative analysis is made on a few different CACSD tools used in the automotive industry is made.

Unfortunately it seems that there have not been any recent, independent and qualitative studies made regarding the efficiency of auto generated code from Simulink used in digital motor control applications.
3 THE CACSD ENVIRONMENT

In order to achieve good results using a CACSD environment one must first understand how it works. In this section a general background to CACSD environments will be given followed by an explanation of how the MathWorks™ environment works, it is based on the information provided by the MathWorks™ in [8] to [12]. Finally a brief introduction to the whole concept will be given, from block scheme to generated C code.

3.1 Background

In the present day the time to market is becoming an ever more important issue when developing new products and systems. Developing the most efficient product or system does not always guarantee success today since product life cycles has decreased dramatically during the last years. An efficient way to decrease development time for an embedded application is to use software that utilizes automated code generation.

In traditional development the control algorithm is first developed and simulated in software like Matlab, Maple or Mathematica. When the functionality of the algorithm is proven the algorithm was manually implemented in Assembly or C to target a specific processor. However since product development is an iterative process the algorithm usually has to be modified several times before the product is released. For each modification the code has to be updated and debugged which is very time consuming.

The general idea when using a CACSD software environment is to simplify the programming process by making it more graphical and thus more intuitive. Instead of first developing the control algorithm in a separate program and then implement it in code the designer implements the algorithm in a graphical way, usually as a block scheme or some kind of flow chart. The CACSD environment then translates this scheme into a mathematical model from which it generates processor specific code. This eliminates the need to use several programs during the development process and enables engineers without extensive experience of low level programming to design real time programs for embedded applications.

Today there are a few different CACSD software environments featuring graphical programming and automated code generation for embedded targets available on the market. The most well known are; LabVIEW developed by National Instruments, dSPACE developed by dSPACE GmbH and Real Time Workshop developed by the MathWorks™. This type of software is today predominantly used in the automotive and aerospace industries.
3.2 The code generation process in the MathWorks™ CACSD

To create a program using Matlab it is first implemented as a block scheme in Simulink using time-discrete blocks. In order to go from a Simulink model to a functional C program for an embedded target some additional Matlab packages and an Integrated Development Environment (IDE) are needed. The packages needed are Real Time Workshop, Real Time Workshop Embedded Coder, a Target Support Package and a link to an IDE. Together with an external C cross compiler used as interface to the targeted DSP.

Once the algorithm is implemented in Simulink the model is stored in the standard Simulink format as an .mdl file. The generation of processor specific C code from this Simulink .mdl file is a complex and sophisticated process. The general workflow of how the different Matlab packages interact with one another is depicted in Figure 3.1 below.

The first step in the code generation process is that RTW translates the Simulink .mdl file to a .rtw file, which in principle is a compiled version of the .mdl file. It describes the blocks, inputs, outputs, parameters and other properties of the original model and is implemented in ASCII format. Together with the .rtw file RTW creates a makefile, .mk, from the provided template makefile, .tmf. The template makefile specifies information about the development system such as the compiler to be used and how the generated code should be compiled. To create the makefile RTW copies the content in the template makefile and expands lexical tokens (symbolic names) that describes the models configuration.

In the next step of the process the Target Language Compiler (TLC) translates the .rtw file into target specific C-code using a system target file, a .tlc file, which specifies the rules for the overall code generation, for example to generate code optimized for fixed-point or floating point operations. On block level the code generation is based on libraries of target files, where each Simulink block has a corresponding target file that specifies the rules for that specific block.

To create executable code that can be run on the embedded target a so called make program is needed together with a Matlab link to that program. In this thesis a TI processor is used and the chosen make program is therefore Code Composer Studio (CCS), which is an IDE developed by TI to interface their DSPs and hence the IDE link is Link for Code Composer Studio.

When the C-code is generated Matlab calls the make program, in this case CCS, through the IDE Link and the generated source code together with the makefile, created by RTW earlier in the process, are passed on to make program. CCS uses the makefile to compile the generated source code and link objects files as well as library files into the project. In the next step an .out file is built which finally is loaded into the DSP.
The code generation process described above explains how the code is generated for a specific processor. However it does not really address how code is generated for the specific modules and peripherals in the chosen processor.

This can be done in two different ways. The first and easiest way is by acquiring the proper support package, for example the Target for TI C2000 package which supports the processors in the TI C2000 family and the TMS320F2808 processor used in this thesis. In such a package new blocksets are added to Simulink together with corresponding libraries for the TLC. These new blocks can be used to initialize I/O functions such as PWM generation and AD conversions. Other blocks can be used to enable different communication protocols, such as SCI and CAN, between the host computer and targeted DSP.

The second way to generate code for the peripherals requires a lot more effort since it requires the designer to make the Simulink blocks and corresponding TLC library files himself. The blocks are created by writing s-functions in C/C++/ADA/ FORTRAN or M-code and the TLC library files are written in target language compiler code.

The concept with TLC libraries to set the rules for the code generation for the different Simulink blocks gives the designer a great deal of flexibility, since it is possible to modify these libraries and thus alter the code generation rules to fit any specific demands.
### 3.3 Using the CACSD environment provided by the MathWorks™

This section is aimed to give a brief step-by-step introduction on how the design, code generation and downloading process of a very simple program, called example, to a DSP is performed. In this case the DSP is a TMS320F2808. The program to be implemented makes an ADC readout and depending on the value acquired it sets the duty cycle of a PWM signal to either 25% or 75%. If the readout from the 12-bit ADC is between 0-2047 the duty cycle is set to 25% and it is set to 75% if the readout is between 2048-4095.

The first step is to design the program as a block scheme in Simulink using standard library blocks and processor specific blocks provided by RTW and its sub packages, as depicted in the upper left corner of Figure 3.2.

In this example the switch and the two constant blocks are standard Simulink blocks while the block representing the ADC and the ePWM modules are processor specific blocks provided by the Target for TI C2000 package. In the properties of these blocks the designer can easily specify things like sample time, PWM period/duty cycle and designate the PWM module to be used. The block in the upper left corner of the Simulink model is called the target preferences block and it has some special attributes. It is not connected to any other block and its main purpose is to specify the processor to be used. With this block it is also possible for the user to set features as memory mapping, pin assignments and other peripherals such as SPI, SCI and eCAN.

Several important settings in the model are specified in the configuration parameters pane under the Simulation menu in Simulink. Here the designer specifies important
settings regarding the code generation, optimization, diagnostics and so forth. Under the Real Time Workshop branch in the options tree depicted in Figure 3.3 below, the designer specifies the system target file and the corresponding template makefile that shall be used during the code generation. Examples of other settings that can be specified here are different optimization options for the code generation, code commenting options or whether to support floating-point-/complex numbers.

When the program design in Simulink is finished the user simply pushes the incremental build button in the Simulink window, see ring in figure 3.2, in order to generate, compile, build and load the code into the DSP, all of these in one automatic process. The process can be followed in the Matlab command window, as seen in the lower left corner in figure 3.2, and it ends with CCS building and loading the generated program into the DSP. However before the code generation begins Simulink examines the model and checks for inconsistencies such as data type errors and illegal rate transitions. If no errors are found the model is compiled into an .rtw file.

When the code is successfully generated it can easily be examined directly in the CCS window that is opened during the code generation process. It is worth to point out that the names given to the different blocks in the Simulink scheme are included in the generated code as comments to the code generated for that block. Therefore it is important for the designer to name the blocks according to their functions in order to simplify any eventual debugging in CCS.
In Figure 3.4 the code generated to implement the function in the Simulink example model is depicted. Noteworthy is that the code generated here is understandable, this due to the simplicity of the Simulink model. A more extensive and complicated model will not generate code that is particularly readable.

```c
/* Model step function */
void example_step(void)
{
  /* local block i/o variables */
  real_T rth_Switch;

  /* S-Function Block: <Root>/ADC (cJ80xadc) */
  {
    AdcRegDs.ADCCTRL2.bit.RST_SEQ1 = 1; // Reset SEQ1 module
    AdcRegDs.ADCST.bit.INT_SEQ1_CLR = 1; // Clear INT sequencer
    AdcRegDs.ADCCTRL2.bit.SOC_SEQ1 = 1; // Software Trigger
    while (AdcRegDs.ADCST.bit.INT_SEQ1 == 0) {
      // Wait for sequencer INT bit to clear
      asm(" RST #11 || NOP");
      example_B.ADC = (AdcRegDs.ADCRESULT0) >> 4;
    }

    /* Switch: <Root>/Switch incorporates:
    * Constant: <Root>-25% Duty cycle
    * Constant: <Root>-75% Duty cycle
    */
    if (example_B.ADC >= example_P.Switch_Threshold) {
      rth_Switch = example_P.DutyCycle_Value_1;
    } else {
      rth_Switch = example_P.DutyCycle_Value;
    }

    /* Update CMRA value for efPWM */
    { EFwm1Regs.CMRA.half.CMPA = (uint16_T)(((uint32_T)EFwm1Regs.TEPRD * rth_Switch)/100));
    }
}
```

Figure 3.4 The section of the generated C code that implements the Simulink blocks in the example program.
4 THEORETICAL BACKGROUND

To implement/develop a motor controller one must first have a basic understanding of the motor type and control algorithm used. This section of the report will give a brief introduction to the Permanent Magnet Synchronous Motor and the control algorithm Field Oriented Control.

4.1 The Permanent Magnet Synchronous Motor

The principle behind an electrical motor is to utilize electrical energy to create interacting magnetic fields and thus electromagnetic forces, which are exploited to achieve a rotational movement.

As in all electrical motors a PMSM has two principal parts. A moving part and a stationary part, where the former is called the rotor and the latter is called the stator. In a PMSM the rotor is as the name suggests made out of surface mounted permanent magnets. The phase windings are found in the stator and can be configured in different ways, for example as sinusoidal-, trapezoidal- or the more common concentrated winding. There are usually three phase windings since most PMSM are three phase motors, thus each winding is excited by a different phase. When a current runs through the windings a force is exerted on the current due to the magnetic field of the rotor. This phenomenon is called the Lorentz force and is described by equation (4.1)

$$\vec{F} = l \cdot (I \times \vec{B}),$$  \hspace{0.5cm} (4.1)

where F is the force exerted, l is the length of the conductor, I is the current flowing in the conductor and B is the external magnetic field present. The counter force to equation (4.1) is what causes the rotation of the rotor and transforms electrical energy to mechanical energy.

In Figure 4.1 the cross section of a simplified PMSM with concentrated windings is depicted. The three phase windings are called u, v and w respectively and the rotor is modelled as a simple bar magnet, thus this motor has two poles.

Figure 4.1 A PMSM with concentrated windings. The three phases are represented by u, v and w respectively. The rotor is here represented by a simple bar magnet.
As can be seen in equation (4.1) the force, and thus the torque created, reaches its maximum value when the current and the magnetic field are orthogonal to one another. Hence the goal of the motor control algorithm should be to keep the current flowing in the windings orthogonal to the magnetic field of the rotor. The torque equation for a PMSM can be derived by calculating the active power consumption for the three phases and divide the sum of those with the mechanical angular speed of the rotor. This leads to the following torque equation

\[
T = \frac{3}{2} p [\psi_m \times I_q - (L_q - L_d) I_q \times I_d],
\]

(4.2)

where \( p \) is the number of pole pairs, \( L_d \) is the direct axis stator inductance, \( L_q \) is the quadrature axis stator inductance \( \psi_m \) is the magnetic flux linked with the stator winding (i.e. the permanent magnet magnetic flux), \( I_q \) is the quadrature axis stator current and \( I_d \) is the direct axis stator current. For an explanation of the dq-coordinate system see appendix B. For a PMSM with surface mounted magnets, \( L_q \) is equal to \( L_d \). Hence equation 4.2 can be simplified to equation 4.3

\[
T = \frac{3}{2} p \psi_m \times I_q.
\]

(4.3)

For a more thorough derivation of the torque equation please refer to Chapter 12 in [13] or one of the other standard textbooks in the subject like [14] or [15].

There are a number of advantages with a PMSM. It lacks rotor windings which reduce the copper losses. There is no mechanical commutator which implies a lower moment of inertia and that less maintenance is needed. These advantages makes PMSMs well suited for demanding applications like high-performance servo drives. However since the only control variables are the stator currents, the motor control algorithms for PMSMs requires heavy real-time computations in comparison with other motor types. But over the last decades the performance of advanced microprocessors such as DSPs has increased dramatically while at the same time the cost for them has decreased. This has lead to an increasing use of PMSMs during the last two decades.
4.2 Field Oriented Control

One of the most common control algorithms used with PMSMs and induction motors is FOC. The idea behind FOC is to control the stator currents in the time invariant dq-reference frame. The torque equation for a surface mounted PMSM was stated in equation 4.3 and from that equation it is clear that the stator q current, $I_q$, directly controls the torque of the motor since the rotor flux, $\psi_m$, is constant due to the permanent magnets. The basic control scheme for FOC of a three phase PMSM is illustrated in Figure 4.2 below. In section 4.2.1 below the FOC algorithm is described briefly, for a more extensive coverage please refer to [16] and [17].

4.2.1 A brief introduction to FOC

As can be seen in figure 4.2 FOC really consists of two nested loops. The inner loop controls the stator currents in the dq reference frame and thus the torque; it is often referred to as the current loop. The outer loop controls the speed of the motor and is mostly referred to as the speed loop.

The measured input signals in FOC are two of the stator phase currents, $i_u$ and $i_v$ in Figure 4.2, and position sensor information that enable calculation of the rotor angular position and speed. Only two of the stator phase currents need to be measured, since in a symmetric three phase system the third phase current can be deduced from the two others. Typical examples of position sensors that can be used are resolvers, quadrature encoders or absolute encoders.

![Figure 4.2 The control scheme for FOC on a PMSM [16].](image)

---

Master Thesis - Development of a Real Time Test Platform for Motor Drive Algorithms
The two measured phase currents are transformed into the dq-reference frame using the Clarke transformation (uvw → αβ) followed by the Park transformation (αβ → dq). For more information regarding these transformations see Appendix C.

As stated in the previous section the idea in FOC is to control the torque of the motor by controlling its currents in the dq reference frame. Hence the d and q currents are the regulated variables in the inner current loop. From equation 3.3 it is evident that the torque produced depends only on the q current, hence the reference value for the d current is set to zero.

The d and q currents obtained from the transformations are compared to the reference values and the difference is regulated by two PI controllers, one for each current. The resulting control signals are subjected to the inverse Park transformation to give the voltage control signals in the stationary αβ reference frame.

The αβ signals constitute the components of the desired stator voltage vector, i.e. the stator voltage required to produce the desired speed/torque. With this vector the required switching pattern for the 3-phase inverter driving the motor can be determined. This is done using a technique called Space Vector Modulation PWM (SVM PWM) explained in section 4.2.2 below.

The outer speed loop overlays the inner current loop and gives the q current reference, the controller in the outer loop is also performed by a PI controller as seen in Figure 4.2. In practice the control frequency of the speed loop is often made a factor five to ten lower than that of the current loop. This is done since the mechanical time constants are typically much larger than the electrical time constants. Typical values for the frequency of the current control loop are 4–8 kHz.

### 4.2.2 Space Vector Modulation PWM

SVM PWM is a technique, used in digital motor control, to determine the appropriate duty cycles for the switching devices in a three phase inverter. The technique has been proven to generate less harmonic distortion than other comparable techniques such as the subharmonic method, see [18], and is therefore the most commonly used technique with FOC.

With the duty cycles calculated by SVM PWM the inverter can synthesize the desired voltage vector generated by the two current PI controllers. Figure 4.3 below depicts a simplified three phase inverter, the three phases are as in the previous section called u, v and w. The switching devices, for example Isolated Gate Bipolar Transistors (IGBT), are switched so that three are always on and three are always off. To prevent vertical conduction and a resulting short circuit over the supply, the two switching devices belonging to one phase may never by on at the same time. For this reason the switching devices for a phase are driven by complementary signals with dead bands. By dead band is here meant a time period where both the complementary signals are low simultaneously, thus separating the active regions of the complementary signals from each other in time.

---

1 Here uvw represents the three different phases.
The conditions above generate $2^3=8$ possible switching states for the inverter. These switching states are usually referred to as voltage space vectors (VSV).

To simplify the notation of the space vectors one usually denotes the switching state of a phase with a 1 or a 0. In this report a 1 means that the upper switching device is on and the lower switching device is off and vice versa for the 0 state. The nomenclature for a VSV is here (uvw), in terms of the phase order.

Two of the space vectors, (111) and (000), are known as zero vectors, since the phase to phase voltages for these two vectors are zero. The six nonzero, or active, space vectors are shown in the $\alpha\beta$ reference frame depicted in Figure 4.4.

An example of a desired stator voltage vector $V_s$ can also be seen in Figure 4.4. The desired stator voltage vector can approximately be synthesized by applying an appropriate switching pattern of VSV. The switching pattern to produce the voltage vector consists of the two active adjacent VSV, for $V_s$ in Figure 4.4 these are (100) and (110), together with the zero vectors, (111) and (000).
This is described by equation 4.4

\[ \bar{V}_s = \frac{T_1\bar{V}_1 + T_2\bar{V}_2 + T_6\bar{V}_6 + T_7\bar{V}_7}{T}, \]  

(4.4)

where \( \bar{V}_1 \) corresponds to vector (100), \( \bar{V}_2 \) to vector (110), \( \bar{V}_6 \) to zero vector (000), \( \bar{V}_7 \) to zero vectors (111), \( T \) is the switching period and \( T_{i,i=0,1,2,7} \) is the period of time that vector \( i \) is applied during the switching period. Figure 4.5 shows the resulting PWM signals for the three phases.

Figure 4.5 The PWM signals required to synthesize the desired stator voltage vector \( \bar{V}_s \).

As can be seen in Figure 4.5 the generated PWM signals are symmetric around the middle of the PWM period. This configuration implies that only the switching devices belonging to one phase has to be switched at anyone time in order to change the VSV, thus minimizing the switching frequency and the harmonic contents in the system.

In appendix E the derivation of the different phase duty cycles required to synthesize the desired stator voltage vector, \( \bar{V}_s \), can be found. Additional information about SVM PWM can be found in [19] and [20].
5 IMPLEMENTATION

The CACSD environment provided by the MathWorks™ and TI was used in a rapid prototyping approach to implement FOC on a small PMSM. The used motor type is normally used in a six axis industrial robots. An existing control board previously developed at Corporate Research was used in the implementation.

The principal components of this control board is the TMS320F2808 DSP implementing the control algorithm, an Intelligent Power Module (IPM), i.e. a three phase inverter, driving the motor and a Complex Programmable Logic Device (CPLD) used to generate the required dead bands for the switching devices in the IPM. The board also includes DC/DC converters, hardware for filtering / voltage level shifting for the ADC and other basic circuity. The board is programmed using the Joint Test Action Group (JTAG) interface.

The control board was originally designed to be able to control three motors simultaneously and hence it was equipped with three IPM’s. However to drive three IPM’s with complementary signals would require eighteen PWM signals, something that the 2808 DSP does not have enough channels for. Hence a CPLD was added to generate complementary PWM signals with dead bands. The position sensor used to determine the rotors mechanical position was a resolver. In Figure 5.1 below a functional diagram of the control board and its interface to the motor is depicted.

![Simplified functional diagram of the used hardware.](image-url)
5.1 The used hardware

5.1.1 DSP

The DSP on the control board is as mentioned before a TMS320F2808 from Texas Instruments. This is a 32-bit fixed-point processor using the Harvard architecture. This model is mainly used in controller applications, like digital motor control, digital power supplies and advanced sensing applications in automotive and medical industries. Some of its key features are summarized in Table 5.1. For more information about the DSP see [21].

Table 5.1 The key features of the TI 2808 DSP.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock frequency</td>
<td>100 MHz</td>
</tr>
<tr>
<td>RAM</td>
<td>36 kB</td>
</tr>
<tr>
<td>PWM</td>
<td>16-CH</td>
</tr>
<tr>
<td>Flash</td>
<td>18 kB</td>
</tr>
<tr>
<td>ADC</td>
<td>1 16-Ch 12-Bit</td>
</tr>
<tr>
<td>ADC Conversion Time</td>
<td>160 ns</td>
</tr>
<tr>
<td>CAP/QEP/CAN</td>
<td>4/2/2</td>
</tr>
<tr>
<td>I2C</td>
<td>1 Ch</td>
</tr>
<tr>
<td>UART</td>
<td>2 SCI Ch</td>
</tr>
<tr>
<td>SPI</td>
<td>2 Ch</td>
</tr>
<tr>
<td>Nr of GPIO</td>
<td>35</td>
</tr>
<tr>
<td>TIMERS</td>
<td>3 32-Bit, 1 WD</td>
</tr>
<tr>
<td>Core Supply</td>
<td>1.8 V</td>
</tr>
<tr>
<td>IO Supply</td>
<td>3.3V</td>
</tr>
</tbody>
</table>

5.1.2 CPLD

The CPLD used is a XILINX XC9572XL which contains 72 macrocells with 1600 usable gates. It has 44 I/O pins, supports frequencies up to 178 MHz and uses a 3.3 V supply. It was already programmed in a previous project to generate dead bands of 5 µs. This was found to be an appropriate time interval for the application in question and hence the available program was used unaltered. For more information regarding the CPLD see [22].

5.1.3 IPM

The IPM used to drive the motor is a PM50CLA120 manufactured by Mitsubishi Electric. Table 5.2 depicts some of its properties. More information about the IPM is found in [23].

Table 5.2 The properties of the Mitsubishi IPM.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>V Supply</td>
<td>&lt; 800V</td>
</tr>
<tr>
<td>t dead band</td>
<td>&gt; 2.5 µs</td>
</tr>
<tr>
<td>f PWM</td>
<td>&lt; 20kHz</td>
</tr>
<tr>
<td>P Rated</td>
<td>5.5kW/7.5kW</td>
</tr>
<tr>
<td>V Control Supply</td>
<td>15 V</td>
</tr>
<tr>
<td>V Forward voltage</td>
<td>2.5V</td>
</tr>
</tbody>
</table>
5.1.4 Motor and Resolver

The used PMSM was a small servo motor named 3HAC 17346-1 manufactured by Tamagawa and mainly used by ABB in robot applications. Some of the main properties of the motor are found in Table 5.3.

Table 5.3 The properties of the Tamagawa motor.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of poles</td>
<td>6</td>
</tr>
<tr>
<td>Rated speed</td>
<td>2200 rpm</td>
</tr>
<tr>
<td>$V_{RMS}$</td>
<td>190 V</td>
</tr>
<tr>
<td>$T_{RMS}$ (at 1100 rpm)</td>
<td>0.57 Nm</td>
</tr>
<tr>
<td>$J_{Rotor}$</td>
<td>$6.08 \cdot 10^{-5}$ kgm$^2$</td>
</tr>
<tr>
<td>$R_{20}$</td>
<td>11.9 $\Omega$/phase</td>
</tr>
</tbody>
</table>

Mounted in the motor is a Tamagawa Smartsyn TS2640N871E172 resolver used for position sensing. A resolver is in principle a rotating transformer and consists of three inductors. One is mounted on the rotor and the two other are mounted orthogonal to each other but on the stator, as illustrated in Figure 5.2.

![Figure 5.2 A simplified resolver.](image)

A sinusoidal excitation voltage, $U_0$, is applied to the inductor on the rotor. This creates a time varying magnetic field which induces the two voltages $U_1$ and $U_2$ in the two inductors mounted on the stator. The amplitude of the induced voltages depends on the rotor angle $\theta$. This is described mathematically through equations 5.1 – 5.3,

$$U_0(t) = \hat{u}_0 \cdot \sin(2\pi f_{exc} t),$$

$$U_1(t) = \hat{u}_0 \cdot \sin(2\pi f_{exc} t) \cdot \kappa \cdot \cos(\theta),$$

$$U_2(t) = \hat{u}_0 \cdot \sin(2\pi f_{exc} t) \cdot \kappa \cdot \sin(\theta).$$

Where $\hat{u}_0$ is the amplitude of the voltage applied to the inductor on the rotor, $f_{exc}$ is the frequency of the excitation signal, $\kappa$ is the transformation ratio between the inductors and $\theta$ is the mechanical angle of the rotor relative to the reference position.
The rotor angle $\theta$ can easily be determined by measuring $U_1$ and $U_2$ and taking the four quadrant arctangent of equation 5.3 divided by equation 5.2, see equation 5.4a-b.

$$\theta = \arctan\left(\frac{U_2(t)}{U_1(t)}\right), \text{ if } U_2(t) \geq 0 \tag{5.4a}$$

$$\theta = \pi + \arctan\left(\frac{U_2(t)}{U_1(t)}\right), \text{ if } U_2(t) < 0 \tag{5.4b}$$

The Smartsyn resolver uses a sinusoidal excitation signal of 10 kHz generated from a low pass filtered PWM signal from the DSP.
5.2 Implementation in Matlab

The FOC algorithm was implemented on a system running Matlab R2007b and the accompanying Simulink version 7.0. The additional packages required for target specific code generation for the TI C2000 DSP family are; Real Time Workshop® v7.0, Real Time Workshop® Embedded Coder v5.0, Link to Code Composer Studio™ v3.1 and Target for TI C2000™. In addition to the Matlab environment Code Composer Studio v3.3 is needed in order to build the generated program and program the DSP.

An implementation of FOC in Matlab/Simulink adapted for code generation with RTW can easily result in a model which is hard to follow. Therefore the model developed in the following sections is divided into several subsystems, where each one is described separately in order to make the model more intuitive and easy to follow.

Since the F2808 DSP is a 32-bit fixed point processor the data type mainly used in the Simulink model below is fixdt(1,32,17) in Matlab nomenclature, which corresponds to a fixed point data type. The first number determines whether the data is signed or unsigned, in this case it is signed. The middle number, in this case thirtytwo, indicates the length of the word, and the last number indicates the length of the fractional part of the word. The range of the data type specified above is -16383 to 16384 and the precision is \(2^{-17}=0.0000076294\). This range should be more than enough for this implementation where the highest possible number is 4095 and comes from the 12-bit analog to digital converter (ADC). The precision should also be more than enough since it is also limited by the 12-bit ADC.

5.2.1 The IQmath and DMC libraries

Included in the Target for TI C2000 package are two quite special libraries namely the IQmath and the DMC (Digital Motor Control) library. The blocks in these libraries correspond to functions in the TI C28x IQmath and DMC assembly code library which are written to be optimized for the processors in the C2000 family. This implies that the code generated from blocks in these libraries is very efficient.

The blocks in these libraries can also be used in Simulink simulations together with some of the basic Simulink blocks. The functions that are implemented by the blocks in these libraries are for example Clarke-, Park transformations, a PID controller, a Space Vector Generator, trigonometric functions and fixed point multiplications. In general all input and output signals for the IQmath and DMC blocks are fixed point numbers.
5.2.2 Top level

The top level of the developed program model, which is called FOC PMSM, is shown in Figure 5.3 below. The target preferences block specifies the settings of the processor used, in this case the F2808 DSP. The coloured block loads a Graphical User Interface (GUI) when double clicked. The GUI lets the user set certain algorithm parameters and it is explained more in section 5.2.8.

There are two Function Call Subsystems (FCS's) in the scheme, the upper one implements the major parts of the FOC algorithm and is called the FOC FCS. However as was mentioned in section 4.2 above, the outer speed control loop in FOC is usually run at a lower frequency than the inner current control loop. Hence the rotor shaft speed calculation is implemented in a separate FCS which is executed separately from the FOC FCS. This subsystem is called the Speed calculation FCS and is explained in section 5.2.6.

In the Hardware Interrupt block, see upper left corner of Figure 5.3, the designer can set the Interrupt Service Routines (ISR's) that are to be used in the program and how these shall be prioritized. The ISR's generated by the Hardware Interrupt block are connected to the FCS's which are triggered to be executed asynchronously once its interrupt is thrown. Since FCS’s are executed asynchronously all blocks placed inside a FCS must have their sample time set to -1, which implies that the sample time is inherited.

The right part of figure 5.3 depicts the properties pane of the Hardware Interrupt block. To generate an ISR the designer chooses the four parameters, specified on row one to four, that are used by Matlab to describe an interrupt. The first two parameters are the CPU and PIE interrupt numbers, found on row one and two respectively. These numbers corresponds to a position in the F2808 interrupt table, see [24], and specifies the actual peripheral interrupt.

In Figure 5.3 two ISR's are initialized, here presented in coordinate form (CPU number, PIE number). The first interrupt has coordinates (1, 1), which corresponds to an event coming from the ADC module, this interrupt triggers the execution of the FOC FCS. Interrupt number two has coordinates (3, 4) which corresponds to an event coming from the fourth ePWM module and it triggers the execution of the Speed calculation FCS.

On the third row the Simulink task priority is set, a low value corresponds to a high priority. Here the ISR connected to the FOC subsystem has a higher priority since it is the main task and thus forms the base rate of the model. The fourth row specifies the preemption conditions, which means that a task with preemption flag set to 1 can have its execution preempted by a higher priority task. Hence in this model the ISR generated by the ADC can preempt the ISR generated by the ePWM. In order to initialize the ePWM ISR an additional subsystem is needed, se left part of Figure 5.3, this subsystem and the timing of the ISR's are described in section 5.2.7 below.

In figure 5.3 it can be seen that there are rate transition blocks connected between the output and input ports of the FCS's. This is a requirement for all signal paths between blocks running on different sample rates. The rate transition block improves the data integrity in the system and to some extent also lets the designer determine the level of data integrity in the system.
5.2.3 Function Call Subsystem – FOC

This subsystem just splits the FOC function call subsystem into two different parts to achieve a better overview of the model. The left part deals with measurements and corresponding scaling while the right part implements the actual FOC algorithm.

Figure 5.3 The top level in the implementation, left window, and the properties of the Hardware Interrupt block, right window.

Figure 5.4 The FOC subsystem is split into two parts to improve overview.
5.2.4 Subsystem – Measurements and Scaling

The blocks in this subsystem, the left one in Figure 5.4, addresses the input signals of the FOC algorithm or more precisely two of the phase currents, the speed reference and the two resolver signals needed for the estimation of the rotor position. The subsystem is depicted in Figure 5.5 below.

In the left part of Figure 5.5 the properties pane of the ADC block is shown. This pane lets the designer specify the settings for the ADC module. Here the ADC start of conversion is set to be triggered by ePWM events. This is important since it enables the designer to decide where in the PWM cycle samples are to be taken. Thus it is possible to avoid sampling during the transients coming from the IPM, which is controlled by the PWM signals. Where in the PWM cycle the trigger signal is to be generated is set in the block properties of the PWM module.

The option to post an interrupt at the end of conversion is here enabled, this generates the event that triggers the first interrupt, i.e. (1, 1), in the Hardware Interrupt block as was described in section 5.2.2.

The speed reference for the motor is here read from a location in the memory. Worth to point out is that the speed reference could also be given using the CAN, RTDX, SPI or SCI interfaces.

There are two additional subsystems present in this subsystem. The upper is the Current Scaling subsystem which shifts the sampled phase currents back to zero reference, rescales them to desired levels and changes the data type from integer to fixed point. The lower Resolver subsystem handles the resolver signals and is explained in the next section. For a more extensive coverage of the Current Scaling subsystems refer to Appendix D.
5.2.4.1 Subsystem – Resolver

The resolver outputs two signals, commonly denoted as the cos and sin signals which corresponds to equations 5.2 and 5.3 respectively. With these two signals the rotor position can be determined using equation 5.4. The scheme implementing this can be seen in Figure 5.6.

![Figure 5.6 The resolver subsystem.](image)

In the left part of the scheme the data type of the signals from the ADC is changed from integer to fixed point and the value is shifted back to zero level. Then the angular position of the rotor is calculated in per unit, i.e. [0, 1] which corresponds to [0, 2\pi] in radians. This calculation is performed by calculating the four quadrant arctangent of the cos and sin signals coming from the resolver, see equation 5.4. The four quadrant arctangent function is implemented by the Arctangent IQN block.

The angle calculated from the resolver signals is the mechanical angle of the rotor, which is not the relevant angle for the Park and Clark transformations performed further on. For these transformations the electrical angle of the rotor is relevant. Hence the mechanical angle is multiplied with the number of rotor pole pairs and a resolver offset angle is deducted, the latter due to the physical orientation of the resolver inductors. These operations can cause the angle to become greater than one, however the range of the angle is [0, 1]. To implement the angular wrap around a fractional part IQN block is used, i.e. only the decimal part of the number is kept. In this way a modulus operation around one is performed.
5.2.5 Subsystem – Motor Control

The Motor Control subsystem is the right subsystem in Figure 5.4 and it implements the main parts of the FOC algorithm, described in section 4.2. The input signals to this system are the measured and scaled motor currents, the electrical angle of the rotor and the external speed reference. The implementation is depicted in Figure 5.7 and has been made quite similar to the block diagram of the FOC algorithm found in figure 4.2 for coherency reasons. Though here the two PI current controllers in the inner current loop have been lumped together to form a separate subsystem.

As can be seen in the figure the coordinate transformations, the speed PI controller and the space vector generation is performed by DMC (Digital Motor Control) blocks that are included in the Target Support Package, thus enabling a more efficient code generation. The transformation blocks simply implements the equations in Appendix C and the PID block implements a PID controller with antiwindup correction that has the derivative gain set to zero, for additional information refer to Appendix G. The Space Vector Generator block implements the inverse Clarke transformation and then calculates the duty cycle ratios needed for the PWM signals to generate the appropriate space vectors, as was outlined in section 4.2.1 and appendix E. For a more detailed description of how the blocks in the DMC library functions see [25].

In the figure there are three additional subsystems dealing with: the current PI controllers in the inner loop, the interface to the IPM and the interface to the IPM board. The two former subsystems are described in sections 5.2.5.1 and 5.2.5.2, while the latter subsystem is presented in appendix E.

Figure 5.7 The implementation of the principal parts of the FOC algorithm.
5.2.5.1 Subsystem – Current Control PIs

This subsystem implements the PI controllers used for the d and q current regulation as described in section 4.2. The controllers are implemented with PID controllers from the DMC library with the derivative gain set to zero. The control signals generated by the PI controllers constitutes the desired voltage space vector in dq coordinates.

In the right part of Figure 5.9 the resulting magnitude of the desired voltage space vector is calculated. If the magnitude is greater than unity the previous value of the desired voltage space vector is feed through the switch instead of the present value. The reason for this operation is to ensure that the duty cycle ratios calculated by the Space Vector Generator block further on will be correctly scaled.

Figure 5.9 The current control PIs followed by a magnitude limiter.
5.2.5.2 Subsystem – IPM Interface

Figure 5.10 shows the scaling of the PWM duty cycle ratios. The input is a vector containing the three duty cycle ratios generated by the Space Vector Generator block, see Figure 5.7. These duty cycle ratios ranges from -1 to 1 and are centred round zero, the range is due to the magnitude limiting operation described in the previous section. The scaling is performed by adding a one to the duty cycle ratios and to multiply with 50 and thus get the duty cycles in percent.

Following the scaling is a saturation block and a switch. These blocks are required for the start-up phase of the control board. The upper switching devices in the IPM are fed through bootstrapping and therefore the PWM signals must be active for 10ms at start-up to charge the bootstrapped capacitors. This is implemented by the switch and the counter block, as long as the counter has a value below the reference value the duty cycles are set to 100% and once the counter reaches the reference value the correct duty cycles are sent through the switch. The saturation blocks before the switch ensures that the duty cycles are limited to be in the 5% to 95% span, in order to ensure that the bootstrapped capacitors remain charged during normal operation.

The correctly scaled PWM signals are then sent to the input ports of the PWM blocks in the very right part of Figure 5.10. The properties of the PWM block are shown in Figure 5.11. Here it is possible to set practically all features belonging to the PWM module of the F2808 in a simple graphical way. The controllable features ranges from setting of timer period, duty cycle, dead band generation, symmetrical/unsymmetrical waveforms, to the control of when in the PWM cycle ADC conversions are to occur.

When using FOC symmetrical PWM signals are required, hence the counting mode for the PWM module is set to Up-Down; see the left part of Figure 5.11. This means that time counter, CTR, in the PWM register first counts up from zero to the set period value, PRD, and then starts to count down to zero. The settings governing the active period of the PWM are found in the pane depicted in the right part of Figure 5.11 and are set by choosing the set-, clear- or do nothing action at certain counter values. For a graphical representation of the PWM counter, its different counter events and the PWM waveform generated with the settings chosen in Figure 5.11 refer to the right part Figure 5.12.
Figure 5.11 The properties pane of the PWM block.

Figure 5.12 Left part, synchronisation PWM-ADC. Right part, the PWM counter and the generated signals.
In the left part of Figure 5.12 the pane enabling synchronisation between PWM- and ADC modules is depicted. In digital motor control this synchronisation is very important, due to the switching of the IPM which causes transients in the system. The ADC should not start conversion close in time to the on/off switching of the IPM, i.e. near a low/high or high/low transition in the PWM signal. As can be seen in the figure the ADC conversion is set to start every time that CTR=PRD, which is exactly in the middle of the active period of the PWM signal, se right half of Figure 5.12. Thus the effect of the switching transients will be minimized.

Worth to note here is that the generated PWM period will be twice as long as the set Timer initial period, see Figure 5.11, since the counter first counts up during one timer period and then counts down during one timer period. Hence the PWM period will be two timer periods.
5.2.6 Function Call Subsystem – Speed Calculation

This scheme implements the second FCS in the model; it is here the speed calculation is made. In Figure 5.13 the scheme is depicted. The speed calculation itself is just a simple numerical derivation, implemented by a unit delay block and sum block taking the difference between the present and the previous sample of the electric angle which is the input signal to the system. The difference operation is followed by a scaling with the sample time to calculate the actual speed in revolutions per second. However the calculated speed is the electrical speed which is usually not the relevant one, hence the electrical speed has to be divided by the pole pair number to get the mechanical speed.

It is here worth to point out that sample time in this subsystem is different from the rest of the model, since this subsystem implements the outer speed control loop. Although the speed PI controller is placed inside the FOC subsystem, this does not change the behaviour of the speed loop since the inputs to the PI is updated with the sample time of the speed control loop.

There is a problem with the speed calculation; it suffers from the angular wraparound which occurs at the start of every new revolution. Assume that the angle changes one tenth in per unit between two consecutive samples, for example from 0.4 to 0.5 which gives an angular change of 0.5 - 0.4 = 0.1. However if the angle wraps around zero in between the two samples, for example from 0.95 to 0.05, this would indicate an angular change of 0.05 - 0.95 = - 0.90 which is not the case.

To get around this issue the magnitude of the speed is calculated and if it is above a certain threshold value, i.e. a value that indicates a very high speed, the previous value of the calculated speed is used instead. This is implemented by an absolute value block, a unit delay block and switch block in the right part of Figure 5.13.

Figure 5.13 The calculation of the rotor shaft speed.
5.2.7 The Interrupt Service Routines

In the program generated out of the developed model described in the previous sections, there will be two ISR’s, one implementing the main FOC algorithm, i.e. the current loop, and one implementing the speed calculation. When working with multiple ISR’s in real time applications timing is of uttermost importance. If the timing between the ISR’s in the system is not done properly the system behaviour will at best be poor or more likely unpredictable. It is important that these two interrupts are synchronized since they share information amongst each other.

Figure 5.14 shows the timing diagram for the ISR’s in the system, the upper half depicts the faster current control ISR and the lower part depicts the speed control ISR. The vertical arrows indicate when the interrupts occur and the gray shaded area following an arrow illustrates the execution time required within the interrupt time period.

When deciding the sample times in the system the most obvious part is to choose times that exceeds the execution times of the ISR’s. Additionally, in the present system the time period for the speed control ISR should be longer than the current control ISR. Though only longer is not enough, it should preferably be an integer amount times longer. If this is not the case, the calculation of the shaft speed will sometimes be misleading since there will not always be a constant number of current control interrupts between two consecutive speed control interrupts. Furthermore the interrupts should also be synchronized with each other, if not they will drift form one another in time, something that can cause problems and unpredictable behaviour, for example if both interrupts were to occur at the same time.

The generation of the base rate interrupt, i.e. the current control ISR, was outlined in section 5.2.2 and 5.2.5.2. Though it can be worth to recapitulate, it was initialized using the properties of the ADC and ePWM modules respectively. The first ePWM module synchronizes the ADC conversion to be made in the middle of the PWM period as it sends a trigger signal to the ADC module which starts the conversion. When the conversion is made the interrupt is thrown and the FOC subsystem is executed.

To initialize the speed control ISR is a bit more complicated. The initialization would ideally be done using one of the event timers in the 2808, however unfortunately there is no event timer block for the 2808 in target for TI C2000. Even though there is such a block available for the 2812, which is the other DSP in the C2000 family. To get around this issue one can use the timer in one of the unused PWM modules. Here the timer in PWM module 4 is used as an alternative event timer. To initialize the event that triggers
the interrupt is a bit tricky since it for some reason is not possible to trigger this event directly in the properties of the ePWM block, something that was possible for the ADC module see Figure 5.3. To trigger the event write to memory blocks have to be added to the model. These blocks should be configured to write directly to the relevant registers in the DSP memory.

Figure 4.15 shows the Initialize ePWM Interrupt subsystem, located in the top level of the model, together with the properties set in this subsystem. A write to memory block 2 initializes the triggering event to be sent on every event and a write to memory block 3 specifies that event to be CTR = PRD. For information regarding the ePWM module and its registers see [26] and [27].

![Figure 5.15 The Initialize ePWM subsystem, to the left, the ePWM settings, in the middle and the write to memory settings to the right.](image)

In the middle part of Figure 5.15 the properties of ePWM module 4 is shown. It can be seen that it is synchronized with the EPWMxSYNCI signal. This is a signal that can be set in ePWM module 1 to synchronize this module with the other ePWM modules. From Figure 5.11 it is evident that the synchronisation occurs when CTR=Zero in ePWM module 1.

The ePWM counters together with the resulting interrupt sequence can be viewed in Figure 5.16 below where the speed control ISR has twice the period of the current control ISR.
Figure 5.16 The ePWM counters and the resulting interrupt cycles when the speed control ISR has twice the period of the current control ISR.
5.2.8 Development of a User Interface

As in all other real time test platforms it is important to have a user friendly interface in order to simplify user to computer communication. Usually a graphical interface with for example buttons and sliders is perceived as better than a command based interface such as a prompt.

When using large Simulink models containing many blocks and subsystems it is a cumbersome task to find all instances of a parameter when it is changed/updated. The most practical way of solving this issue is by using parameters. These parameters can be set in a couple of different ways, for example directly in the Matlab prompt or by running scripts from the prompt. However for users other than the designer it can be difficult to know which parameters that need to be set and which values these parameters are allowed to take.

One way of solving these problems is by designing a Graphical User Interface (GUI) using the GUI Builder included in Matlab. With this package it is possible to custom design a GUI in a fairly easy way. The GUI builder package itself lets the designer pick buttons, sliders, graph displays, etcetera etcetera and put them together in a way much like building a Simulink model. When the GUI is saved Matlab auto generates an M-file and a figure file, the M-file already contains the basic code belonging to the figure. The only thing remaining for the designer is to write the code to be executed when a button is pressed or a slider is changed. The next step is to link this GUI and possibly some other scripts to some button or block in the Simulink model. In this way the designer can achieve a user interface that is fully understandable and intuitive to use, even for users with only basic Matlab skills. The GUI designed can be viewed in Figure 5.17.

![Figure 5.17 The designed GUI.](image)

However to have a real time test platform the user must be able to communicate with the DSP in some way and thus control/monitor the motor. Hence a user interface Simulink model was developed for the program model.
The user interface model is depicted in Figure 5.18 below. To use the real time test platform the user simply presses the play button in the user interface model. When this is done a script is run and the GUI from Figure 5.17 is loaded and lets the user set the required parameters, then the user interface model proceeds by generating the code for the FOC PMSM model, loads it into the DSP through the CCS link and runs the program. Once the generated program is running on the DSP the user can in real time set the reference speed by changing the value in the speed reference block. The rotor speed calculated on the DSP, or any other desired variables, can then be monitored in real time on the scope. This data can also be processed later in Matlab since it is saved to the workspace using the simout block.

![User Interface Model](image)

Figure 5.18 The User interface model.

The reason that an extra model has to be created is that the host blocks used for the different communication protocols and the scope are not a part of the generated program that will run on the embedded target.

For the program model, described in sections 5.2.2-5.2.7, to handle the communication requested by the user interface target side SCI Receive and Transmit blocks have to be added to the program model. The updated top level in the program model is found in Figure 5.19. The added SCI blocks are placed inside a new FCS which is triggered by a FIFO (First In First Out) register interrupt thrown from the SCI Receive block when the receive FIFO is full. When the SCI Communication FCS is executed the received speed reference is written into the memory location used for the speed reference in the FOC subsystem, see figure 5.5, and the calculated speed is sent to the host computer.

The concept described above can easily be extended to apply for more parameters and/or settings if desired.
Figure 5.19 The corresponding top level in the program model.
6 LABORATORY MEASUREMENTS

The FOC algorithm implemented in the previous section was tested in the laboratory on the hardware described in Section 5.1. The experimental setup is shown in Figure 6.1. During this test the goal was not to make extensive and very precise measurements but rather to verify the functionality of the generated program and the possibility to monitor motor variables using the developed user interface.

Thus the software user interface used was the same as the one described in section 5.2.8. This user interface however will most likely degrade the performance of the FOC algorithm implemented since the communication protocol used is SCI. This protocol has, as mentioned before, a rather low data rate which means that parameter data can not be transmitted during each base rate interrupt cycle, unless the base rate time period is increased substantially. Instead parameter data is sent when the FIFO buffer interrupt is executed as mentioned in the previous section. With the configuration used this occurs about every 50 ms.

Furthermore the parameters in the PI controllers are not tuned in order to maximize the performance of the algorithm. The parameter values used in the PI controllers are summarized in Table 6.1. The period times used were 400 µs for the current control ISR and 2 ms for the speed control ISR, i.e. the speed control loop runs five times faster than the speed control loop.

Unfortunately time restrictions made it impossible to setup a proper test bench hence the motor had to be tested without a load and using a reduced bus voltage of 90 V rather then the nominal 190 V.

<table>
<thead>
<tr>
<th>Gain</th>
<th>Speed PI</th>
<th>d-Current PI</th>
<th>q-Current PI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proportional</td>
<td>0.1</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>Integral</td>
<td>0.1</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>Integral correction</td>
<td>0.1</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>Derivative</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 6.1: The experimental setup used during the measurements.

Table 6.1 The parameter values used in the PI controllers.
In Figure 6.2 the speed of the motor and the resulting scaled q current is depicted. After about two and a half second the speed reference is subjected to a step change from five to twenty revolutions per second. As expected the magnitude of the q current increases as the speed increases. From the figure it is evident that the speed reference tracking performance degrades with higher reference speeds, thus the resulting q current is not very constant. It can also be seen in the figure that the amplitude of the oscillation round the reference speed and the q current is increasing slightly over time even though the speed reference is held constant. Both these phenomena most likely depend on the untuned PI controllers which seem to make the system slightly unstable. Unfortunately there was no time available to further study this instability since that was not the main purpose in the thesis.

Figure 6.2: The upper graph depicts the motor speed, the resulting scaled q current is shown in the lower graph.

Figure 6.3 depicts a similar step change in the speed reference and the resulting q current PI control signal, i.e. the desired stator q voltage. It is evident that the magnitude of the control signal increases as the speed reference is increased. Noteworthy is that the control signal is relatively high even at this rather low speed, this depends on the fact the motor was run at a reduced bus voltage, 90 V compared to the nominal 190 V.
The next measurement is shown in Figure 6.4 and shows a similar step change as in the two previous figures. The corresponding speed PI control signal clearly shows that the control signal saturates as the speed reference step is applied and then quickly returns to fluctuate round zero as the speed reaches the reference. The high control signal values in the beginning of the plot occur during the motor acceleration phase from zero to five revolutions per second. The high oscillation in the control signal is as mentioned previously due to the untuned and slightly unstable PI controllers.

In figure 6.5 the scaled d and q currents and the motor speed are depicted. In the figure it is evident that the d current is fluctuating round zero which is the d current reference. The q current is as expected larger than zero and it is also here increasing slightly over time due the same reasons as mentioned above.

Figure 6.3: The upper graph shows the motor speed and the lower graph shows the resulting q current PI control signal.
Figure 6.4: The upper graph shows the motor speed and the lower graph shows the resulting speed PI control signal.

Figure 6.5: The upper graph depicts the scaled d current, the right graph depicts the scaled q current and the lower graph shows the motor speed.
7 EVALUATION OF THE MATHWORKS CACSD ENVIRONMENT

7.1 The quality of the generated code

Once the Simulink model is designed and working as intended the next step in the process is the code generation and the evaluation of its quality. What is here meant by code quality and how is it measured?

Concepts that often are mentioned are readability, traceability and efficiency. The two former are subjective and can unfortunately not be directly measured.

7.1.1 Readability

With readability is meant how easy it is for a moderately experienced programmer to understand what the generated code is doing. It can be said to depend on the formatting of the generated code, variable naming, programming constructs used, program hierarchy and commenting.

The generated code is formatted quite well since there are usually not too many characters on each row, blank spaces are placed round mathematical operators and brackets are placed on separate rows. During the code generation the programming constructs used tend to improve readability since for example; if/else statements with indentations are used rather than the conditional operator. For a section of generated code please see Figure 3.4 in Section 3.3.

The names given to subsystems and blocks in the Simulink model are added as comments in the code when they are not optimized away, something that clearly improves readability and traceability. However what is not good in terms of readability are the variable names which can become quite long and sometimes not very intuitive. Though what really makes the code hard to follow is the lack of hierarchy in the generated program. In principle all the generated code is placed in one long C-file, which for obvious reasons is very hard to follow. The additional C-files used in the project are mostly imported TI library files.

7.1.2 Traceability

The term traceability refers to the possibility to correlate generated code back to the blocks in the Simulink scheme or vice versa. This is implemented quite well in Matlab, something that was mentioned above is that the names given to the blocks in the Simulink schemes are included as comments in the generated code. However it is a tedious task to manually search the code for these comments. Matlab also features a more practical way of doing this. In the configuration parameters the designer can choose to generate an html report which is presented when the code is generated, in this report code-to-block highlighting can be included. When this is done hyperlinks which link generated code to its respective Simulink block are included in the report. These hyperlinks also work in the opposite direction, which means that it is possible to mark a Simulink block and let Matlab find the code segment generated from that block.
7.1.3 Efficiency

Code efficiency usually refers to the execution time of the program, its stack usage and RAM/ROM usage. In contrast to readability and tractability these things can be determined objectively.

When implementing real time motor control algorithms perhaps the most interesting issue is the execution time, which sets the upper limit for the frequency of the control loop. Once the code is generated it is possible to determine the execution time and system stack usage of the program. This is done using the profile command.

However for Matlab to determine the execution time of the generated program some settings have to be enabled in the Configuration Parameters pane in the Simulink model and the generated program has to be run on the target processor, i.e. the targeted processor must be physically connected to the computer.

The determination of the execution time is done by running the profile command in the Matlab prompt. This runs the generated program on the target processor during a number of interrupt cycles and a specified number of samples are collected. These are processed by Matlab and the result is presented as a graph and an HTML report. In the report the execution time of the generated program is presented together with task overrun information. It is here worth to point out that the execution time is highly dependant on the blocks and settings used in the Simulink model. In Target for TI C2000 there are two libraries included, described in section 5.2.1, that has blocks that are specially designed to generate efficient code. These two libraries are the IQ-Math library and the DMC library.

To illustrate the above mentioned dependency a simple program implementing the calculation of the electrical rotor position, in per unit, using the resolver input signals was generated in two different versions. The first one using the available IQ-Math blocks, as was done in section 5.2.4.1 above, while the second one is designed using standard Simulink blocks. The two models and the resulting execution times can be seen in Figure 7.1.

One can clearly see that there is a substantial difference in execution time, a factor 7.5, between the two models. This clearly shows that it is important to use the available IQ-Math/DMC blocks when code efficiency is desired. Unfortunately the number of available IQ-Math/DMC blocks is rather limited, something that might lead to long execution times when implementing algorithms that require blocks that are not available in those libraries. One way out of this problem is for the designer to write/use C-functions that implement the desired operations and link these functions to Matlab.

However there is a bug in the profile command in the present version of Target for TI C2000, the command does not work together with asynchronous interrupts. Something which clearly is a problem for the FOC PMSM model developed in this report and hence its execution time can not be measured directly.

Though the command is still useful and can be used for partial implementations of the main program, as was done in the example here.

Empirically the execution time for the current control ISR has been determined to be somewhere between 116 $\mu$s to 123 $\mu$s. This sets the maximum frequency of the current control loop to about 8 kHz.
One important feature regarding code efficiency determination is missing in Matlab. It is not possible to determine RAM and ROM usage from Matlab using prompt commands. There is however an alternate way to acquire this information, when the project is build in Code Composer Studio a memory allocation text file containing this information is created in the build directory.

Figure 7.1: Execution times for two Simulink models implementing the resolver subsystem but using different block sets.
7.2 DSP - Matlab Communication Protocols

The 2808 DSP supports three different communication protocols that can be used in communication between the DSP and Matlab. These are; CAN, SCI and RTDX (Real Time Data Exchange). The control board used in this project only supports SCI and RTDX communication, hence these protocols were the ones studied.

7.2.1 The SCI protocol

SCI is a full duplex protocol using the computers COM port. In Matlab it is implemented by adding SCI Receive- and Transmit blocks to the model. These blocks handles the DSP side of the communication link. To send and receive data via Matlab there are separate host-side SCI blocks, which are placed in a separate Simulink model as was described in Section 5.2.8.

For the communication to work between the host-side Simulink model, including the host-side SCI blocks, and the generated program running on the DSP, i.e. the target-side, the two Simulink models has to be linked together by a script. To receive the data sent from the generated program the host side model has to be run in synchronization with the program on the embedded target, i.e. the host side model and the program on the embedded target has to be started at the same time and in a certain sequence or Matlab will not be able to recognize the incoming data. The data that is received can then be saved to a file or be monitored in the host-side model by for example a standard Simulink scope.

However there are some limitations due to the Matlab data handling. Matlab is a heavy application and loads the computer CPU extensively when the Simulink host-side model is run. Hence if the incoming data rate is too high, i.e. data is transmitted every interrupt cycle, Matlab is unable to process it effectively in real time and has to buffer the data extensively. Hence if long tests with good time resolution are desired it is uncertain whether Matlab can buffer all the incoming data.

It is here worth to point out that there are some limitations in the SCI communication due to the Matlab implementation. For example there are only ten different data rates available to choose from out of the ~65000 possible data rates. The major limitation is that the highest available data rate is 115 200 bits/s compared to the highest possible data rate of DSP which is 6 250 000 bits/s [26]. This is clearly a drawback when implementing FOC. Assume one wants to monitor the rotational speed of the rotor with a reasonable accuracy, say by transmitting a 32-bit number. The length of the data package for a 32-bit number is 6 bytes long including header and terminator. This corresponds to 48 bits and it will take 48/115200 seconds to transmit, i.e. 417 µs. A time period that is not very compatible with an application whose base rate interrupt has a period time in the 125 - 250 µs range, i.e. a current control loop frequency of 4-8 kHz. Thus data can not usually be sent during each interrupt cycle at least not with any precision.

Another limitation is that it does not seem to be possible to send single SCI messages from Matlab to the DSP. This means that when SCI is used to set the speed reference as in section 5.2.8 the reference speed is transmitted continuously even though the reference is not changed.
7.2.2 The RTDX protocol

The Real Time Data Exchange protocol was developed by TI to enable real time debugging; data is sent over the JTAG connector. When using RTDX a bidirectional communication path is opened between the host computer and the target DSP and data can be transmitted. When data is not being transmitted the RTDX communication path is not interfering with the application running on the DSP. Noteworthy is that RTDX is not intended for variable monitoring or data logging purposes since it is not meant to transmit data at regular time intervals but rather upon requests.

RTDX is like SCI implemented by adding receive and transmit blocks to the model, however there are no host blocks for RTDX since data is transmitted upon requests from the Matlab prompt.

Since RTDX data is received or transmitted only upon requests from the prompt it is well suited for use together with Matlab GUI’s in debugging purposes. It is easy to program a GUI with buttons that are programmed to respond with RTDX data transmissions. This concept could of course be further extended to allow readouts of a multitude of signals without interfering with the running program since the RTDX channels does not transmit data continuously. If a multitude of variables were to be monitored over an SCI channel which transmits continuously the period of the base rate would have to be increased dramatically, thus altering the behaviour of the FOC algorithm.
7.3 Matlab User Interface

The user interface of the RTW and Target for TI C2000 packages is in principle the same as in Simulink. To design a program the designer, as mentioned before, drags blocks into the Simulink workspace and connects them together with signal lines. This is a much more intuitive way to design a program than the usual way of writing a series of code files and linking those together. It gives a much better overview during the design process since the block scheme of the program is a part of the design process rather than a tool of understanding and explaining it. This should guarantee that there never are any discrepancies between the block scheme and the code of the program.

To specify the settings for a DSP module block one simply double clicks on the block as with standard Simulink blocks. As in all DSP programming there are usually quite a few settings for each module and a question that usually comes up is what all these settings mean. Here the Matlab help is very useful, by right clicking on the block in question and choosing help, the Matlab help window regarding this block is opened automatically. An example of what the help looks like for the ADC module can be seen in Figure 7.2.

![Figure 7.2: The Matlab help for the ADC module.](image)

The help for each module briefly describes its properties and sometimes include hyperlinks to other relevant topics which at times is rather helpful. However the Matlab help is nowhere near as extensive as the help files provided by the manufacturer of the DSP, in this case Texas Instruments. This leads to a situation where the designer sometimes needs to consult the manufacturers help files in order to understand the more advanced settings of certain modules. In this situation there is a discrepancy since the manufacturers help files assume that the reader is designing the program in the standard way by writing C-code when he in fact is designing it using Matlab.

This is not a major problem but it sometimes leads to confusion since not all the available functions described in the manufacturer’s help files are implemented in Matlab.
7.4 Issues encountered during the implementation

When working with a relatively new technology such as automated code generation for embedded targets some problematic issues are to be expected. Here it is important to distinguish the different issues and the problems they cause from one another.

The most important issue is the reliability of the generated code, i.e. that the code actually has the anticipated functionality. This has never been a problem throughout this project; the generated code has always functioned correctly and there have never been any compilation errors. This is most likely due to the Target Language Compiler (TLC) in RTW which generates the code for all the different processor family packages, of which Target for TI C2000 is one. Hence a lot of development effort has been spent on the TLC and it appears to be a very mature technology.

7.4.1 Software issues

What has been an issue in the project are which blocks that are supported by code generation and which are not. The main problem is that all Simulink blocks that support code generation are not gathered in one or several blocksets. Of course all the blocks belonging to RTW and Target for TI C2000 support code generation, but to design the model more blocks are needed such as summing- and delay blocks, these blocks are found in the standard Simulink libraries. One illustrative example is that the unit delay block is supported but the integer delay block is not, even though they are basically identical in functionality. Both blocks are seen in Figure 7.3.

Figure 7.3: To the left the Unit Delay block that is supported for code generation. At the right the Integer Delay block which is not supported.

Another problem is that some of the standard Simulink blocks that support code generation does not support fixed point numbers. When using these blocks the signal must first be converted to integer or floating point then passed on to the block and then after the block be converted back to fixed point, something that of course degrades the efficiency of the generated code.

However the major issues encountered has not been regarding whether or not blocks support code generation or not but rather which DSP settings that are supported in the code generation. An illustrative example of this problem is how the two hardware interrupts were initialized in the model that was developed. The interrupt generated in the ADC module, see section 5.2.2, was initialized by selecting the Post interrupt at end of conversion option in the properties pane of the ADC block.

The interrupt generated by the PWM module was initialized in a different and not very intuitive way, as described in section 5.2.7. This inconsistency in which settings that are implemented in different modules is quite confusing and disturbing for the designer. Especially since the initialization of the PWM interrupt is not described anywhere in the Matlab documentation and only can be found in one of the available demos.
When working in a rapid prototyping approach it is desirable for the designer to have the possibility to alter parameter values in real time or at least to alter them without rebuilding the entire program. This feature is possible in Matlab when using Simulink external mode, see [8], which enables signal logging and parameter tuning. However this feature is only available using the CAN communication protocol.

Since the controller board used in this project did not support CAN communication external mode could not be used and therefore real time parameter tuning becomes more problematic. To illustrate this refer to Figure 7.4. Here it can be seen that in the properties of the PWM block it is possible to specify the timer period source through an input port which makes it possible to alter the PWM period in real time by connecting this port to for example an SCI receive block. This can be compared with the switch block in which it is not possible to specify the threshold through an input port. This means that in order to alter the threshold in the switch block the code for the entire model has to be regenerated when external mode is not available.

Worth to point out is that parameter tuning and external mode is not compatible with IQmath/DMC blocks even when using CAN communication. This means that when the gains in for example a PID controller is to be changed, the whole program has to be regenerated.

![Figure 7.4](image)

Figure 7.4: The possibility to specify parameters through input ports can be used for real time parameter tuning when using protocols other than CAN.

Once the code is generated it is important for the designer to be able to determine how efficient the generated code is. As was mentioned in Section 7.1.3 the profile command used to determine the execution time of a generated program suffers from a bug and can not be used on models with asynchronous tasks. Though according to the MathWorks support this bug is supposed to be fixed in the next version of Target for TI C2000. However there is another desirable command that is missing completely namely a command to determine the RAM/ROM usage of the generated code.
7.4.2 Hardware issues

Other issues that can be cumbersome arise when the DSP is used with hardware other than the standard one. For example in this project the oscillator crystal used had a frequency of 25 MHz and not 20 MHz which is the standard. Unfortunately it is not possible to alter the CPU multiplier directly in Matlab. Since Matlab assumes that a 20 MHz crystal is used the multiplier is per default always five. So when a 25 MHz crystal is used the DSP will in fact be overclocked to 125MHz rather than the default 100 MHz. This leads to timing problems since Matlab assumes that the DSP is running at 100 MHz and thus all time periods specified in for example PWM modules will in reality be 25% shorter than the ones actually set.

To get around this problem the designer has to locate the TI system control file that Matlab links into the generated projects and manually change the multiplier value. This is of course not so hard to do once the relevant file is located. However to get such information one must most likely contact the Matlab support.
7.5 Debugging

The debugging process when working with automated code generation is as anticipated fairly different compared to code written in the usual way. In the standard design process most syntax errors are found during the compilation of the program and are corrected in an iterative process until the code complies with the design rules.

In Matlab the syntax errors are found before the code is compiled, as Matlab executes the build command it checks the Simulink model and controls whether the design rules are obeyed before the code is generated. Instead of an error message from the compiler the designer gets an error message from Simulink and the build process is halted before the code generation starts. Therefore there are seldom compilation errors.

The second part of the debugging process concerns the code functionality i.e. does the code function in the desired way. Here the automated code generation has both advantages and disadvantages. The main advantage is the fact the code is auto generated. Thus registers, counters and memory sections are initialized correctly and without human mistakes. Since these are the main sources of functionality errors this is clearly an advantage and saves a lot of time and effort for the designer. However the main disadvantage is also due to the fact that the code is auto generated. The code generation is not perfect and if the generated code has errors these are very hard to find, although it appears to happen very rarely.

The most common debugging that has to be done when using automated code generation is to locate functionality errors in the algorithm implemented in Simulink. These errors should generally be found during simulations before the code is actually generated. However if they remain after the simulations they can be hard to find since it is quite complicated to use break points in Matlab.

An easier way to locate functionality errors from Matlab is by using RTDX communication to monitor variable values. Though this method requires some effort since RTDX blocks has to placed in the model before the code is generated.
8 CONCLUSIONS

Working with Matlab/Simulink and RTW to develop and implement motor control algorithms has as expected both advantages and disadvantages. One great advantage is that an engineer without extensive knowledge of embedded hardware and low level programming can develop and implement motor control algorithms. However the greatest advantage is the rapid prototyping possibilities that the Matlab environment offers. It is in principle possible to go from model to functional code in a matter of minutes.

Though nothing comes without a price; what is gained in development time has a cost in flexibility, code quality and money. When using Matlab to program a DSP the designer is initially somewhat limited in what is possible to implement. Not all features of the DSP are possible to implement using Matlab. Most often the engineer does not exactly know all of the features of the DSP that will be used in the final application. Hence problems might come up when desired features are not fully supported by Matlab. Most of the time it is possible to get around these problems using alternate solutions. An example is the initialization of the ePWM interrupt in Section 5.2.7, however these alternate solutions can be tricky and require some time to find.

The readability is in principle forfeited since almost all of the generated code is placed in one file. It is possible to understand individual statements and sections of the code, but it is almost impossible to get an overview of what the generated program is doing without also studying the Simulink scheme that it is generated from.

As far as the code efficiency is concerned it is hard to say how efficient the generated code is in comparison to hand written code since no comparison was made. One thing is however clear, the auto generated code is not very efficient when standard Simulink function blocks are used, as was shown in Section 7.1.3.

The main conclusion that can be drawn from this thesis is that Matlab/Simulink and RTW is quite well suited when a rapid prototyping approach is desired. If high performance and flexibility is desired this software is probably not the best choice.

The possibility to design GUIs and run DSP communication through Matlab makes it quite easy to design a user interface platform for an external user. These interfaces are easy to handle for users with Simulink experience. However they are not so well suited for precise measurements, at least when using the SCI protocol.

The code generation technology is mature. However what is not as mature is what is supported in code generation; there are quite a few Simulink blocks that does not support code generation. This is disturbing since it is not always possible to see which blocks that are supported and which are not, at least before the code is to be generated. One thing that is worth to keep in mind is that in this thesis a motor control algorithm was implemented and in Target for TI C2000 there is a digital motor control library included. Hence not so many problems with unsupported Simulink blocks were encountered. It can be suspected that a desired implementation without this type of specific library might be more difficult to implement. These libraries also make the designed model somewhat processor specific since the DMC and IQmath libraries can only be used with C2000 processors. If a truly processor independent algorithm is to be implemented only standard Simulink blocks can be used. Something that will result in generated code that is quite slow.
Furthermore there are quite a few RTW options that are out of the scope for this report and have not been used. It is possible to influence the code generation in several quite advanced ways. Some examples of what can be influenced are: the code commenting, the data type naming conventions and where data definitions/declarations are made. More advanced features like code generation rules can also be controlled by the designer. This makes it possible to design and add new libraries to Matlab and thus create support for new types of implementations and/or processors. However it is questionable whether the latter feature is worth the effort unless the created libraries will be used extensively.
9 FURTHER WORK

Below a few suggestions are given on topics that could be interesting to examine further.

- An interesting feature that has not been evaluated in this thesis is the possibility to perform parameter tuning using Simulink external mode. This feature makes it possible to tune model parameters in real time using the CAN communication protocol. However, this feature has some limitations since it for example is not possible to tune parameters in IQmath and DMC blocks, something that severely limits its usefulness in at least the application presented in this thesis.

- Implement the Direct Torque Control (DTC) algorithm, which is the motor control algorithm mainly used by ABB. This should be done to determine whether or not Matlab can generate code that has a sufficiently low execution time. Something that is not certain since DTC has much stricter timing constraints than FOC.

- Investigate the limitations of the DSP to Matlab communication, i.e. which limitations there are when designing user interfaces.

- Evaluate how difficult it is to integrate previously written C/Assembly functions with RTW.

- Evaluate a different CACSD environment. The LabVIEW environment developed by National Instruments can also be used for real time motor control, at least when using an FPGA.

- It would be interesting to make a comparison between the auto generated code and hand written code in terms of code efficiency when implementing the same functionality in both cases.

- Study how the code generation process can be optimized in the configuration parameters and how much influence it has on the generated program.

- Examine the possibilities to use the same model both for code generation and simulation. There is a form of multiplexer block available which enables the designer to create different branches in a model, one used for simulation and one used for code generation.
REFERENCES


[23] “Mitsubishi <Intelligent Power Modules> PM50CLA120”, Mitsubishi Electric, May 2005


## APPENDIX

### A List of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td>APWM</td>
<td>Asymmetric Pulse Width Modulation</td>
</tr>
<tr>
<td>CACSD</td>
<td>Computer Automated (Aided) Control System Design</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>CAP</td>
<td>Capture Module</td>
</tr>
<tr>
<td>CCS</td>
<td>Code Composer Studio</td>
</tr>
<tr>
<td>DMC</td>
<td>Digital Motor Control</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>DTC</td>
<td>Direct Torque Control</td>
</tr>
<tr>
<td>eCAN</td>
<td>enhanced Control Area Network</td>
</tr>
<tr>
<td>eCAP</td>
<td>enhanced Capture Module</td>
</tr>
<tr>
<td>ePWM</td>
<td>enhanced Pulse Width Modulation</td>
</tr>
<tr>
<td>FCS</td>
<td>Function Call Subsystem</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FOC</td>
<td>Field Oriented Control</td>
</tr>
<tr>
<td>GPIO</td>
<td>General Purpose Input Output</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IGBT</td>
<td>Isolated Gate Bipolar Transistor</td>
</tr>
<tr>
<td>IPM</td>
<td>Intelligent Power Module</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>I2C</td>
<td>Inter Integrated Circuit</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group</td>
</tr>
<tr>
<td>PMSM</td>
<td>Permanent Magnet Synchronous Motor</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse Width Modulation</td>
</tr>
<tr>
<td>QEP</td>
<td>Quadrature Encoder Pulse circuit</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RTDX</td>
<td>Real Time Data Exchange</td>
</tr>
<tr>
<td>SCI</td>
<td>Serial Communication Interface</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>SVM</td>
<td>Space Vector Modulation</td>
</tr>
<tr>
<td>TI</td>
<td>Texas Instruments</td>
</tr>
<tr>
<td>TLC</td>
<td>Target Language Compiler</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver Transmitter</td>
</tr>
<tr>
<td>VSV</td>
<td>Voltage Space Vector</td>
</tr>
<tr>
<td>WD</td>
<td>Watch Dog</td>
</tr>
</tbody>
</table>
B The dq coordinate system

In motor applications the dq coordinate system is often used, in this system the axis are fixed in respect to the rotor. The d-axis or direct axis is oriented along the rotor flux linkage vector, $\Psi_m$. The q-axis or quadrature axis orientation is given by the cross product of the vector of angular velocity, $\omega$, and the d-axis.

Figure B1 The dq coordinate system.
C Coordinate transformations

The Clarke transformation

Transforms a symmetric three phase system, where \( i_u \), \( i_v \), and \( i_w \) represent the three phase currents, into an orthogonal and stationary coordinate system, represented by the \( \alpha \) and \( \beta \) coordinates as seen in figure C1 below.

![Clarke Transformation Diagram](image-url)

Figure C1 The Clarke transformation.

The Clarke transformation is described by equations C1 – C7

\[
\begin{align*}
\beta &= \alpha, \quad (C1) \\
\beta &= -\frac{1}{2} i_u + \frac{\sqrt{3}}{2} i_\beta, \quad (C2) \\
\beta &= -\frac{1}{2} i_u - \frac{\sqrt{3}}{2} i_\beta. \quad (C3)
\end{align*}
\]

Substitute C1 into C2

\[
\beta = -\frac{1}{2} i_u + \frac{\sqrt{3}}{2} i_\beta, \quad (C4)
\]

and rearranging terms gives

\[
\beta = \frac{(2 i_v + i_u)}{\sqrt{3}}. \quad (C5)
\]

Thus the Clarke transformation is given by

\[
\begin{align*}
\beta &= i_u, \quad (C6) \\
\beta &= \frac{(2 i_v + i_u)}{\sqrt{3}}. \quad (C7)
\end{align*}
\]
The Park transformation

Transforms the stationary $\alpha$–$\beta$ coordinate system into the rotating $d$-$q$ coordinate system, where the $d$ axis is oriented aligned with the rotor flux.

The Park transformation is described by equations C4-C5

\[
\begin{align*}
    i_d &= \cos(\theta_e) \cdot i_\alpha + \sin(\theta_e) \cdot i_\beta, \\
    i_q &= -\sin(\theta_e) \cdot i_\alpha + \cos(\theta_e) \cdot i_\beta,
\end{align*}
\]  
(C4)
(C5)

where $\theta_e$ represents the angle between the $\alpha$ axis and the $d$ axis.

The inverse Park transformation is the transformation from $d$-$q$ coordinates back to $\alpha$–$\beta$ coordinates and is described by equations C6-C7

\[
\begin{align*}
    i_\alpha &= \cos(\theta_e) \cdot i_d - \sin(\theta_e) \cdot i_q, \\
    i_\beta &= \sin(\theta_e) \cdot i_d + \cos(\theta_e) \cdot i_q.
\end{align*}
\]  
(C6)
(C7)
D Subsystem – Current Scaling

In the current scaling subsystem the current samples are first shifted back to zero offset level. In the next step the values are shifted arithmetically six digits to the left i.e. divided by $2^6 = 64$. The reason for the shifting is due to the PI regulators controlling the current, see appendix E. The reference value for the q current is the output from the speed PI regulator which output has the range $[-1, 1]$ and the reference value for the d current is zero. Hence the other PI inputs, i.e. the feedback inputs, has to be in a similar range to ensure proper operation of the controllers.

Figure D1 The Current Scaling Subsystem.
E  Subsystem – IPM board interface

This subsystem handles some settings on the IPM board, such as hardware scaling factors for the current measurements, motor brake control, activation of the PWM signals coming from the CPLD and board revision number.

The eCAP block has here been configured to run the eCAP1 module in APWM mode, i.e. as an asymmetric PWM channel. It generates a 10 kHz signal with 50% duty cycle which is filtered to a sinusoidal waveform on the IPM interface board and then used as the resolver excitation signal.
**F  Space Vector Modulation**

Assume that the desired stator voltage vector $V_{S,Ref}$ is located between space vectors $V_1$ and $V_2$. To synthesize $V_{S,Ref}$ vector $V_1$ is applied during time period $T_1$ and $V_2$ is applied during time period $T_2$, during the reminder of the PWM period, $T_0$, the two zero vectors are applied. This can be summarized by the following equations

\[ T = T_1 + T_2 + T_0, \]  
\[ \bar{V}_{S,Ref} = \frac{T_1}{T} \cdot V_1 + \frac{T_2}{T} \cdot V_2, \]  

where $T$ is equal to the PWM period time. Equation F2 is a simplified version of equation 4.4, the zero vectors are omitted since their contribution is zero.

However the given reference vector is given in the $\alpha\beta$ reference system, hence it has to be projected onto $V_1$ and $V_2$

\[ V_{s\beta,Ref} = \frac{T_1}{T} \sin(60^\circ) \cdot |V_2|, \]  
\[ V_{s\alpha,Ref} = \frac{T_1}{T} |V_1| + \frac{T_2}{T} \cos(60^\circ) \cdot |V_2|. \]  

It can be shown [25] that the magnitude of a voltage space vector after normalization by the maximum phase voltage (i.e. line to neutral) is $2\sqrt{3}$. Hence (F3) and (F4) can be rewritten to

\[ T_2 = T \cdot V_{s\beta,Ref}, \]  
\[ T_1 = \frac{\sqrt{3}T \cdot V_{s\alpha,Ref} - T_2}{2} = \frac{T(\sqrt{3} \cdot V_{s\alpha,Ref} - V_{s\beta,Ref})}{2}. \]

The time period that the active voltage space vectors are applied can be written in fractions of the total PWM period.

Figure F1: The projection of the desired stator voltage vector onto the two adjacent space vectors
The duty cycles, in fractions, for the three phases can be deduced from Figure F.2 and are given by:

\[ t_w = \frac{1-t_1-t_2}{2}, \]  
\[ t_v = t_w + t_2, \]  
\[ t_u = t_v + t_1. \]  

In (F9) above the one represents the PWM period in fractions, i.e. \( T/T=1 \). It is here worth to point out that when the duty cycles for the phases are calculated with equations (F9)-(F11) both the zero vectors, (000) and (111), are applied equal amounts of time during a PWM period.

The reasoning above is valid when the desired stator voltage vector is located between voltage space vectors (100) and (110). However the reasoning for desired stator voltage vectors that lie between other voltage space vectors is analogous and straight forward.

Figure F2: The generated PWM waveform.
G  PID – Controller

The PID controller with integral antiwindup correction can be described by equations (G1) to (G12). A controller with antiwindup correction is normally used in digital motor control since all physical systems tend to saturate at some point, i.e. there is a maximum speed for the motor. When this occurs the control loop is not closed anymore since the output does not directly depend on the input. This leads to a drift in the integral term and a long recovery time for the system once it desaturates. The drift can be compensated by adding an antiwindup term that takes the difference between the saturated and unsaturated output signals scales it and adds it to the original integral term.

\[
\begin{align*}
    u_{\text{presat}}(t) &= u_p(t) + u_i(t) + u_d(t), \\
    u_p(t) &= K_p e(t), \\
    u_i(t) &= \frac{K_p}{T_i} \int_0^t e(\tau) d\tau + K_c (u(t) - u_{\text{presat}}(t)), \\
    u_d(t) &= K_p T_d \frac{de(t)}{dt}.
\end{align*}
\]

Where

\[
u = \begin{cases} 
    u_{\text{presat}}, & u_{\text{min}} < u_{\text{presat}} \leq u_{\text{max}} \\
    u_{\text{max}}, & u_{\text{presat}} \geq u_{\text{max}} \\
    u_{\text{min}}, & u_{\text{presat}} < u_{\text{min}}
\end{cases}
\]

\[
e = \text{ref} - \text{fdb},
\]

\[
K_d = \frac{T_d}{T},
\]

\[
K_c = \frac{T}{T_i}.
\]

Since a PI controller is used in the model the derivative term \(K_d\) should be zero, i.e. \(T_d\) is set to zero.
With the conditions above (G1), (G2) and (G3) becomes in discrete form

\[
\begin{align*}
    u_{\text{presat}}(n) &= u_{\rho}(n) + u_i(n), \quad \text{(G9)} \\
    u_{\rho}(n) &= K_{\rho} e(n) = K_{\rho} \left[ \text{ref}(n) - fdb(n) \right], \quad \text{(G10)} \\
    u_i(n+1) &= u_i(n) + \frac{K_{\rho}}{T_i} e(n) + K_c \left[ u(n) - u_{\text{presat}}(n) \right] \\
                 &= u_i(n) + \frac{K_{\rho} K_c}{T} \left[ \text{ref}(n) - fdb(n) \right] + K_c \left[ u(n) - u_{\text{presat}}(n) \right]. \quad \text{(G11)}
\end{align*}
\]