Polar Field-Oriented Control

with

3rd Harmonic Injection

A Thesis

Presented to

The Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Martin Todd Hess

January 2012

COMMITTEE MEMBERS

TITLE:	POLAR FIELD-ORIENTED CONTROL with 3 RD HARMONIC INJECTION
AUTHOR:	Martin Todd Hess
DATE SUBMITTED:	January 2012
COMMITTEE CHAIR:	Dr. Taufik, Professor of Electrical Engineering
COMMITTEE MEMBER:	Dr. Xiao-Hua (Helen) Yu, Professor of Electrical Engineering
COMMITTEE MEMBER:	Dr. Ahmad Nafisi, Professor of Electrical Engineering

Abstract

POLAR FIELD-ORIENTED CONTROL

with

3RD HARMONIC INJECTION

Martin Todd Hess

Field Oriented Control (FOC), also known as vector control, is a widely used and well documented method for controlling Permanent-Magnet Synchronous Motors (PMSM) and induction motors. Almost invariably the orientation of the stator and rotor (field) fluxes are described in rectangular coordinates. In this thesis we explore the practicality of using polar coordinates.

Third harmonic injection is also a well-known technique that allows full utilization of the bus (DC-link), thus allowing the motor to run to full base speed without the use of field weakening. This technique potentially allows a 15.4% improvement in the available bus. It has fallen out of use since it requires direct knowledge of the terminal voltage vector angle. The use of polar FOC permits the use of third-harmonic injection.

We believe the combination of FOC and third-harmonic injection to be unique, and we present this paper as a novel contribution to the literature on the subject of motor control.

Acknowledgments:

I wish to commend my advisor, Dr. Taufik, Professor of Electrical Engineering, who best exhibits the learn-by-doing philosophy of Cal Poly, and whose demeanor is never intimidating.

The hardest part of writing this thesis was knowing who the audience was to be. After all, I didn't need to teach a panel of professors Ohm's law. Dr. Taufik sagely advised writing to professionals who don't necessarily know anything about my specialty. It will be up to the reader to pass judgment.

I thank Therese, my wife; my *raison d'être*.

Most of all, I thank God and Christ, our only hope, for blessing me with the "talents" and opportunities that made this possible.

TABLE of CONTENTS

List of Figures	ix
Chapter 1: Introduction	1
1.1: Motors & Generators	1
1.1.1: Brush-Type DC Motors	3
1.1.2: PMSM and BLDC Motors	6
1.1.3: Induction Motors	8
1.2: Drives (Inverters)	9
1.2.1: Voltage-Source Drives (Inverters)	10
1.2.2: Current-Source Drives (Inverters)	10
1.2.3: Voltage-Source Drives with Phase Current Feedback	11
Chapter 2: Drive Strategies	14
2.1: A Simple Motor Drive	14
2.1.1: 3-Phase Bridge Rectifier	16
2.1.2: Control Scheme	16
2.2: Third Harmonic Injection	20
2.3: Variable-Frequency, Variable-Voltage Drives	26
2.4: Self-Commutation	27
2.5: Field-Oriented Control of PMSM	30
2.5.1: The Clarke and Park Transforms	31
2.6: Phase Voltage Coupling	34
2.7: Space-Vector PWM	35

Chapter 3: Proposed Polar FOC w/ 3 rd Harmonic Injection	39
3.1: Polar Inverse Clarke and Park Transforms	39
3.2: Polar Park and Clarke Transforms	42
Chapter 4: Implementation	46
4.1: DSP	46
4.2: Language	47
4.3: Numeric Formatting	48
4.4: Special Routines	48
4.4.1: Square Root	49
4.4.2: Arctangent	50
4.5: PWM Block	52
4.6: Analog-to-Digital Converter	55
4.7: Position Sensing	57
4.7.1: Initializing the EIU	59
4.8: The Target Drive	60
4.9: Target Motor	61
4.9.1: Target Encoder	62
4.10: Support Equipment	62
Chapter 5: Evaluation	64
Chapter 6: Proposal II	66
Chapter 7: Evaluation II	70
7.1: Experimental Proof of 3 rd Harmonic Injection	70

7.2: Experimental Proof of Closed-Loop Magnitude Control	74
7.3: Experimental Proof of Closed-Loop Phase Control	75
Chapter 8: Conclusions	77
Epilog: Is there a future for polar FOC?	78
References	79
Appendix A: Source Listing	81
Appendix B: Target Drive Technical Data	194

List of Figures

Fig 1.1: Mechanical v. Electrical Torque.....	2
Fig 1-2: Brush-type DC Motor Construction.....	3
Fig. 1-3: DC Motor / Generator Operation	4
Fig. 1-4: All Windings Used.....	5
Fig. 1-5: Synchronous Motor Construction	6
Fig. 1-6: PMSM Construction.....	7
Fig. 1-7: Squirrel-Cage Induction Motor	8
Fig. 1-8: Voltage-Source Drive (Inverter)	10
Fig. 1-9: Current Source Inverter.....	11
Fig. 1-10: Voltage-Source Inverter with Current Feedback	12
Fig. 2-1: 3-Phase Source and Motor	14
Fig. 2-2: Linear Voltage-Mode Motor Drive.....	14
Fig. 2-3: Phase-Neutral Motor Voltages without 3 rd -Harmonic Injection.....	17
Fig. 2-4: SPICE Neutral-Point Evaluation Circuit.....	18
Fig. 2-5: Normalized Source Neutral-Point Voltage from $90^\circ > \phi > 30^\circ$ with Respect to Ground.	19
Fig. 2-6: Actual Neutral-Point Voltage verses 3 rd Harmonic Voltage.....	20
Fig. 2-7: Motor Controller with Neutral-Point Compensation	22
Fig. 2-8: Control Algorithm with 3rd-Harmonic Injection.....	23
Fig. 2-9: Phase Voltages with respect to the Virtual Ground after 3rd-Harmonic Injection	24
Fig. 2-10: 3 rd -Harmonic Injection as Depicted by Martin	25
Fig. 2-11 Phase-Neutral Voltages after 3rd-Harmonic Injection.....	26

Fig. 2-12: Motor Torque	26
Fig. 2-13: Stator Phase-Neutral Circuit	27
Fig. 2-14: Voltage and Current Phasors.....	28
Fig. 2-15: Stator Frame Current Control.....	29
Fig. 2-16: Basic FOC Drive Scheme with Speed Loop	30
Fig. 2-17: The Forward Clarke Transform	32
Fig. 2-18: The Forward Park Transform.....	34
Fig. 2-19: The Inverse Park Transform.....	34
Fig. 2-20: IM FOC with Voltage Decoupling.....	35
Fig. 2-21: Cross-Coupling	36
Fig. 2-22: Voltage-Mode Output Stage	36
Table 2-1: Phase States	37
Fig. 2-23: SVPWM	38
Fig. 3-1: Forward Path of Rectangular FOC.....	39
Fig. 3-2: Polar Reference Frame Transformation	41
Fig. 3-3 Final Forward Path of Polar FOC	41
Fig. 3-4: Polar FOC Feedback Path	42
Fig. 3-5: Polar FOC Feedback with Polar Park	42
Fig. 3-6: Complete Polar FOC with 3rd-Harmonic Injection	43
Fig. 3-7: Polar FOC Proposed by Bose.....	44
Fig. 4-1: Square Root Algorithm	51
Fig. 4-2: Computing the Arctangent	52

Fig. 4-3: PWM Block Diagram.....	53
Fig. 4-4: Center-Based Double-Update PWM.....	54
Fig. 4-5: ADC Transfer Function.....	55
Fig. 4-6: ADC Block Diagram.....	56
Fig. 4-7: PWM Sync and Convert Start.....	56
Fig. 4-8: Encoder Signals.....	57
Fig. 4-9: EIU Block Diagram.....	58
Fig. 4-10: Target Drive	60
Fig. 4-11: Model VSD-1 Motor Drive.....	61
Fig. 4-12: Target Motor	61
Fig. 4.13: Full Test Bench.....	63
Fig. 6.1: Principle of LEM Current Sensing Module.....	68
Fig. 6.2: Proposal II	69
Fig. 7.1: Phase and Phase-to-Phase Voltages without 3 rd Harmonic Injection.....	70
Fig. 7.2: Phase and Phase-to-Phase Voltages with 3 rd Harmonic Injection.....	71
Fig. 7.3: Direct Measurement of R phase w/ and w/o 3 rd Harmonic Injection	73
Fig. 7.4: Magnitude Error	74
Fig. 7.5: Open-Loop Phase Error.....	75
Fig. 7.6: Closed-Loop Phase Error	76

Chapter 1: Introduction

This project arose largely out of curiosity: The driving force behind all discovery. We want to find why FOC is invariably taught and implemented in rectangular form when polar form is an equivalent means of conveying the same information. After all, most of AC electronics is taught using magnitude and phase Bode plots.

3rd harmonic injection, once so popular that I.C. manufacturers designed it into their motor control chips, has fallen out of vogue. The reason for this is obvious: It can't be implemented in rectangular coordinates. But what if we had polar FOC? And so the exploration begins...

1.1: Motors & Generators

We begin by familiarizing the reader with the types of motors and motor controllers that interest us.

The only significant difference between a motor and generator is the direction of power flow: A motor converts electrical energy into mechanical energy, while a generator converts mechanical energy to electrical energy. As a matter of fact, all motors are generators. Motors generate what is known as Back Electro-Motive Force (BEMF) by simple virtue of the fact that there is a conductor moving in a magnetic field. This voltage is given by,

$$V = K_e \cdot \omega \tag{1-1}$$

where,

K_e = BEMF constant in volt-seconds per radian, but note that motor nameplates usually give it in volts per thousand RPM (KRPM) and RPM is designated by n .

ω = Angular velocity in radians per second, but must agree with the units of K_e .

It is necessary to overcome this BEMF voltage, plus $I \cdot R$ and $L \cdot \frac{di}{dt}$ voltages, in order to drive current into the motor and produce torque. Torque in turn is given by,

$$T = K_\tau \cdot I \quad 1-2$$

where,

K_τ = Torque constant, usually in lb-ft per amp, more correctly in Newton-meters per amp.

When K_e and K_τ are both given in SI units, they are numerically identical.

By definition, torque is the cross-product of two vectors. The two vectors in turn define a plane, and the resulting vector is normal to that plane. The magnitude of the resulting vector depends on the magnitudes of vector arguments and the angle between them thus:

$$|\vec{T}| = |\vec{V}_1| \cdot |\vec{V}_2| \cdot \sin \theta \quad 1-3$$

Clearly torque is at a maximum when the vectors are orthogonal. Figure 1-1 shows examples of mechanical and magnetic (motor) torque. The terms will be made clear as we proceed.

(a) Mechanical Torque

(b) Magnetic Torque

Fig 1.1: Mechanical v. Electrical Torque

Electric motors used in industrial applications generally fall into three categories: Brushed (or Brush-type) DC motors, Synchronous AC motors, also known as Brushless-DC motors (BLDC), and induction motors. Each type is briefly reviewed below.

1.1.1: Brush-Type DC Motors

The construction of a Brush-type motor is shown in Figure 1-2 [1].

Fig 1-2: Brush-type DC Motor Construction

Brush-type DC motors have been the mainstay of the servo industry for decades. For example, they were widely used in numerically-controlled (NC) machine tools. This is due to the ease with which these motors can be controlled by simple analog circuitry.

Brush-type motors are constructed with permanent magnets in the stationary frame of the motor (stator). These magnets produce a stationary magnetic flux known as the “field”. They often use an electro-magnet in place of the permanent magnets. The shaft (rotor) contains multiple electromagnets, called the armature (meaning main current-conducting) windings. The current reaches these windings via fixed carbon “brushes” that ride on a segmented copper-alloy ring on the rotor called a “commutator”. The commutator switches current from one armature

winding to the next as the rotor turns. Figure 1-3 [2] shows a classic simplification of a generator. In a motor, the lamp would be a voltage source.

Fig. 1-3: DC Motor / Generator Operation

In this simple case, the BEMF induced would be a rectified sinusoid as given by Faraday's law of induction:

$$\oint \vec{E} \cdot d\vec{l} = -\frac{d}{dt} \iint \vec{B} \cdot d\vec{s} \quad 1-4$$

Where,

B = The magnetic (in this case, the field) flux density measured in Teslas.

ds = The surface vector normal to the plane of the wire loop.

dl = Line vector following the electric field.

E = The electric field intensity in volts per meter.

The **ds** vector is turning sinusoidally, given by $e^{j\omega t}$. Therefore the BEMF would be sinusoidal.

It is rectified because just as the **ds** vector reverses direction relative to **B**, so does the polarity of the brushes on the commutator.

In practice, there are many loops and many commutator pads so that the **ds** vector is always approximately parallel to the **B** vector. The dot product is maximum when the vectors are parallel, but the derivative (a sine becomes a cosine) causes the BEMF to be maximum when

the ds vector is orthogonal to the B vector. The multiple loops cause the BEMF to always be near maximum and so the BEMF is DC.

In fact, all windings are used at once, and numerous ds vectors exist. Figure 1-4 illustrates this. The vertical components of the vectors cancel, and the horizontal components sum to a net vector to the right. The relevance of these multiple vectors will become clearer when we look at brushless motors. The actual flux lines are far more complex [1][3], but not important to our understanding.

Fig. 1-4: All Windings Used

The chief advantage of brush-type motors is that they are easy to control, while there are many disadvantages: The brushes wear out and need periodic replacement; the rotor exhibits a high inertia due to the mass of the stator windings and there is high friction from the brushes; and arcing may occur between the brushes and commutator, making them unsuitable in explosive environments.

1.1.2: PMSM and BLDC Motors

The technical difference between Synchronous AC (PMSM) and Brushless DC (BLDC) motors is that the first is intended for sinusoidal input voltages while the later is optimized for square-wave (also known as trapezoidal or 6-step) inputs. We will revisit PMSM v. BLDC when we look at drives. We will only consider PMSM for our work. The basic construction is the same for both. They are essentially inside-out brush-type motors. A cross-section view is shown in Figure 1-5 [3]. This particular construction is known as a “salient pole” motor, owing to the shape of the rotor. In our motor, the rotor is round.

Fig. 1-5: Synchronous Motor Construction

The permanent magnets are now mounted on the rotor, and the flux produced is still known as the field flux (more commonly, the rotor flux). Rarely, the rotor magnets may be electro-magnets, in which case there are still brushes riding on “slip rings”. In either case, the magnets come in North-South sets known as pole-pairs and denoted N . There are windings mounted in the frame that is referred to as the stator. These windings are always in sets of three, denoted a , b , and c in the Figure, with the number of sets equal to the pole pairs. They are oriented so that the flux vectors produced are $120^\circ/N$ apart electrically. Finally, the windings

may be connected delta or wye, but every delta has an equivalent wye, so we will only consider the wye connection. Multiple sets are connected in parallel. The electrical frequency of the incoming voltages in Hz is N times the mechanical speed of the rotor in RPS. In our theoretical work, we will only consider 2-pole (1 pole-pair) motors, and revisit multiple pole motors when needed.

Figure 1-6a [4] illustrates the 120° orientation of the stator windings for the 2-pole motor. The rotor is again of a salient-pole type. This illustration also introduces the concept of d and q axes. Note that the direct (d) axis is aligned to the rotor's principle flux direction, while the quadrature (q) axis is orthogonal to it.

Fig. 1-6: PMSM Construction

Figure 1-6b [4] shows a 4-pole, permanent-magnet rotor. There are in fact two d and two q axes in this case. For 4 poles ($N=2$) d & q are 45° apart mechanically, but magnetically they

are still 90° apart. The windings are 60° apart mechanically, but electrically they are still 120° apart. The schematic illustrates the electrical equivalent of a permanent magnet.

PMSM and BLDC motors have gained wide popularity in the servo industry. The advantages of brushless motors are essentially the opposite of brushed motors: There is no brush maintenance; there is no arcing; and the rotor exhibits a low inertia and low friction. The principle disadvantages are that the permanent magnets generally use expensive alloys and can become demagnetized in overload conditions; and that they are harder to control.

Commutation, that is the switching of current from winding-to-winding, is now done electronically. This makes them more difficult to control, but they can still be controlled in a rudimentary way by relatively simple electronics. More sophisticated controls (of the type we will be using) require Digital Signal Processing (DSP).

1.1.3: Induction Motors

The construction of an induction is shown in Figure 1-7 [1].

Fig. 1-7: Squirrel-Cage Induction Motor

Induction motors have stator windings like those of the synchronous motor, but there are no permanent magnets on the rotor. Instead there are conducting bars mounted on the rotor and parallel to the rotor axis. These bars are shorted at the ends. The assembly is known as a squirrel cage. As the rotating stator flux cuts through the bars, a voltage is induced. Since they are shorted at the ends, a current flows and a rotor flux is produced. In order to maintain the rotor flux generation, the rotor must be turning at a speed other than the stator flux. This difference is known as the “slip frequency”.

Like the PMSM and BLDC motors, it is possible to use windings in place of the squirrel cage, along with brushes and slip rings. In this case the windings are shorted or loaded by some external resistance. Induction motors are inexpensive and very rugged. They are immensely popular in general-purpose applications such as pumps and blowers. They have not become popular in the servo industry due to the difficulty of controlling them precisely. Recent developments in vector control promise to increase their popularity in servo applications. The advantages of induction motors are that they are inexpensive and rugged, while the principle disadvantage is that they are difficult to control precisely.

The techniques to be developed in this thesis are equally applicable to synchronous and induction machines. However, due to the additional complexities of controlling the induction motor, we will confine ourselves entirely to permanent-magnet, synchronous motors.

1.2: Drives (Inverters)

We will next consider some motor drive (inverter) topologies.

1.2.1: Voltage-Source Drives (Inverters)

A basic voltage-mode drive is shown in Figure 1-8 [5]. It consists of four parts:

1. Uncontrolled rectifier bridge. (Converts AC to DC.)
2. DC-link. (Filters the DC to a constant voltage.)
3. Inverter (Converts DC to AC).
4. Control electronics. (Not shown).

Fig. 1-8: Voltage-Source Drive (Inverter)

The inductance “L” in Figure 1-8 is often omitted for cost and weight reasons. This type of topology is used to implement “Variable-Frequency Variable-Voltage” control. As the name implies, the voltage to the motor is what is controlled. These drives are used when the speed is what is what is to be controlled.

1.2.2: Current-Source Drives (Inverters)

A current source drive is shown in Figure 1-9 [5]. Note that the capacitor in the DC-link section is gone and current sensing is added. The inductor is now a necessity, since it acts as a constant current source. This type of control is used when torque is what is to be controlled, but

it is rarely used for servo applications due to slow response. It consists of the same four elements, but the rectifier is now controlled (typically SCR's).

Fig. 1-9: Current Source Inverter

1.2.3: Voltage-Source Drives with Phase Current Feedback

By far the most common controller topology for servo use is a voltage-source inverter that includes current feedback from the individual motor phases.

The control algorithm compares the actual motor current with a desired set point (desired torque) and uses the error to control the motor voltage. In this way, the voltage-source inverter is effectively turned into a fast-responding current source inverter. Figure 1-10 shows the control strategy of one such implementation.

This is the topology we will use and is discussed at length in the coming chapters. In particular, we will examine and use Field-Oriented Control (FOC). In FOC, the AC feedback currents are transformed onto a rotating reference frame so that they essentially become DC

values.

Fig. 1-10: Voltage-Source Inverter with Current Feedback

The “textbook” approach to 3-phase, variable-speed motor drives does not fully utilize the available bus (DC-link) voltage, and hence the motor cannot reach “base” speed. This is true for both “variable-frequency, variable-voltage” and “field-oriented” (vector) control. A well-documented technique known as “3rd-harmonic injection” exists to solve the problem in variable-voltage, variable frequency drives. As will become clear, 3rd-harmonic injection requires knowledge of the stator flux angle. Instead, other techniques have been developed and are lumped into a technique called Space-Vector Pulse-Width Modulation (SVPWM). In addition, “field weakening” techniques exist to extend the speed even beyond base speed. We mentioned earlier that the rotor flux is sometimes called field flux from its brush-type cousin. If a component of the stator flux is allowed to oppose the field flux (instead of at right-angles), then field flux is effectively weakened. This in turn allows the motor to develop less BEMF and therefore run at higher speed. In field weakening the motor switches from “constant torque” to

“constant power” operation. In constant power operation, torque is lost. Therefore using this method to achieve base speed prevents full torque.

3rd-harmonic injection addresses the fundamental reason why a drive can’t fully utilize the available bus. However, the conventional method of implementing field oriented control in rectangular components (D and Q) means the necessary angular information to compute the 3rd harmonic is not available. This difficulty will be made clear in Chapter 2. In the technique presented here, we implement vector control in polar coordinates, so as to preserve the necessary angular information.

In Chapter 2 we begin with a thorough investigation of the fundamental problem, and introduce 3rd-harmonic injection. We then review torque control using a voltage-mode inverter, field-oriented control, and SVPWM. In Chapter 3 we present in detail our proposal for implementing 3rd-harmonic injection in a field-oriented, current-controlled, voltage mode motor controller (drive).

Chapter 2: Drive Strategies

2.1: A Simple Motor Drive

Consider a 3-phase motor driven by a 3-phase source as shown in Figure 2-1.

Fig. 2-1: 3-Phase Source and Motor

Obviously the phase-to-phase motor voltage is equal to the phase-to-phase source voltage. Ideally, the neutral-point voltages will also be the same, even though the connection doesn't physically exist.

Now consider the voltage-mode inverter drive shown in Figure 2-2.

Fig. 2-2: Linear Voltage-Mode Motor Drive

The purpose of using a motor controller is to drive the motor at frequencies and voltages other than the source frequency and voltage. Ideally, the controller should be able to drive the motor up to at least its rated frequency and voltage, i.e. equal to a direct connection from the source to the motor, as in Figure 2-1. Note that, at this point, we are not yet considering a PWM scheme. The inverter half-bridges shown in Figure 2-2 are simple class-B, push-pull, emitter-followers. We have a low-power 3-phase control algorithm driving the inverter stage, and the inverter-stage outputs (the motor phases) simply “follow” the control inputs.

In a typical drive, such as the one in Figure 2-2, the bus (often called the DC-link) is unipolar. However, because we will be dealing with sinusoids, and sinusoidal trig functions have bipolar results, it is convenient to think of having a virtual ground at $\frac{1}{2}$ of the bus (DC-link) voltage. R_1 , R_2 , and the ground symbol in Figure 2-2 illustrate the idea. R_1 and R_2 simply constitute a voltage divider to establish this mid-point voltage, which we then ground. The bus thus appears to be symmetrically bipolar, denoted by the node names *Bus+* and *Bus-*. The resistors and the ground connection do not, in general, exist in an actual drive. Pay special attention to the fact that the three outputs of the “control algorithm” are referenced to the virtual ground, as shown by the grounding of the neutral point of the control algorithm. Since the motor phase voltages follow the control voltages, it follows that the motor’s neutral-point is also at ground potential. Assuming that any given transistor in the inverter is ideal and fully turned on, it should be clear that the corresponding peak motor-phase to virtual ground voltage can never exceed the rails, i.e. $\frac{1}{2}$ of the full bus voltage.

2.1.1: 3-Phase Bridge Rectifier

In the 3-phase bridge rectifier, only two diodes are conducting at any instant. The phase with the most positive voltage, with respect to the other two phases, sources the positive side of the filter capacitor. Similarly, the most negative phase sinks the negative side of the filter cap. Therefore the capacitor charges to the peak of the phase-phase source voltage.

By the law of cosines,

$$V_{ph-ph} = \sqrt{V_{ph-n}^2 + V_{ph-n}^2 - 2 \cdot V_{ph-n} \cdot V_{ph-n} \cdot \cos \theta} = \sqrt{3} \cdot V_{ph-n} \quad 2-1$$

where $\theta=120^\circ$.

Thus, with respect to the virtual ground, each half of the bus is $= \pm V_{ph-n} \cdot \sqrt{3}/2$.

2.1.2: Control Scheme

If we now wish to convert the DC bus voltage back to a 3-phase AC voltage, we need to define a control algorithm for the inverter. If we wish to replicate the original sinusoidal input, as when the motor was tied directly to the source, an obvious choice would seem to be:

$$\begin{aligned} A &= M \cdot \sin(\phi) \\ B &= M \cdot \sin(\phi + 120^\circ) \\ C &= M \cdot \sin(\phi - 120^\circ) \end{aligned} \quad 2-2$$

where ϕ is the electrical angle of the phase-to-neutral motor voltage in radians at any given instant of time, and is equal to the motor's desired electrical angular frequency times time, or ωt . M is the desired peak phase-neutral (virtual ground) output voltage but it can never be greater than the bus which is $V_{ph-n(source)} \cdot \sqrt{3}/2$. This is the maximum phase-to-neutral (virtual ground)

voltage the motor can see, and is less than the direct connection to the source. Figure 2-3 illustrates the resulting output with a normalized source phase-to-neutral voltage of 1.

Fig. 2-3: Phase-Neutral Motor Voltages without 3rd-Harmonic Injection

So the obvious choice of a control scheme from equation 2-2 fails. We can't deliver the same phase-to-phase, voltage to the motor that we could by connecting the motor directly to the source as in Figure 2-1.

In order to choose a better control strategy, we need to examine what the source's neutral-point voltage is with respect to our virtual ground. Note that we grounded the neutral-point in the control algorithm, but the source's neutral-point is floating. Figure 2-4 shows the circuit we will use for evaluation and simulation. The source voltages are,

$$V_1 = \sin(\phi)$$

$$V_2 = \sin(\phi + 120^\circ)$$

$$V_3 = \sin(\phi - 120^\circ)$$

$$V_4 = \left[1 - \frac{\sqrt{3}}{2} \right] \sin(3\phi)$$

$$V_5 = \sqrt{3}$$

2-3

The purpose of V_4 will become clear shortly, and V_5 simulates a fully charged, infinitely large capacitor (no ripple).

Fig. 2-4: SPICE Neutral-Point Evaluation Circuit

Since the equivalent circuit changes as the various diodes turn on and off, we will need to derive the equation for the neutral-point voltage (node 2 in Figure 2-4) in a piecewise fashion.

Consider V_1 , V_2 and V_3 over the range $90^\circ > \phi > 60^\circ$. In this segment V_1 is the most positive so D_2 and D_3 are reverse biased. V_2 and V_3 are both negative, but not as negative as $V_1 - V_5$ (V_C), so D_4 , D_5 , and D_6 are also all reverse-biased. The neutral point voltage (starting at (virtual) ground and ending at node 2 in Figure 2-4) is,

$$V_N = V_{R1} - V_1 = \frac{\sqrt{3}}{2} - \sin(\phi) \quad 2-4$$

At the endpoints, $V_N(90^\circ) = -0.134$ and $V_N(60^\circ) = 0$. Similarly, in the region $60^\circ > \phi > 30^\circ$, V_3 is the most negative so D_4 and D_5 are reverse biased, along with D_1 , D_2 , and D_3 . The neutral-point voltage is,

$$V_N = -V_{R2} - V_3 = -\frac{\sqrt{3}}{2} - \sin(\phi - 120^\circ) \quad 2-5$$

At the endpoints, $V_N(60^\circ) = 0$ and $V_N(30^\circ) = 0.134$. The total region $90^\circ > \phi > 30^\circ$, completes $\frac{1}{2}$ -cycle of a waveform. The neutral point with respect to the virtual ground went from a negative peak of -0.134 , through 0, to a positive peak of $+0.134$. This $\frac{1}{2}$ -cycle is illustrated in Figure 2-5.

Fig. 2-5: Normalized Source Neutral-Point Voltage from $90^\circ > \phi > 30^\circ$ with Respect to Ground

We now repeat the process for the remaining ten 30° segments. Since we completed $\frac{1}{2}$ -cycle from 90° to 30° , we expect to complete 3 cycles of the neutral-point voltage in 360° of the source.

2.2: Third Harmonic Injection

Now consider the function,

$$\left[1 - \frac{\sqrt{3}}{2}\right] \sin(3\phi) \quad 2-6$$

This function matches the neutral-point waveform at every $n \cdot 30^\circ$ point. However, it is not an exact match, as shown in Figure 2-6.

Fig. 2-6: Actual Neutral-Point Voltage versus 3rd Harmonic Voltage

To find the significance of the error, we first find the error as the difference between equations 2-4 and 2-6. We then find the angle of the maximum error by taking the derivative of

the error and setting it equal to zero. Finally, we plug the angle back into the error equation.

These steps are summarized in equations 2-7 – 2-10.

$$\varepsilon = \left[\frac{\sqrt{3}}{2} - \sin \varphi \right] - \left[1 - \frac{\sqrt{3}}{2} \right] \cdot \sin(3 \cdot \varphi) \quad 2-7$$

$$\varphi = \cos^{-1} \sqrt{\frac{16 - 9 \cdot \sqrt{3}}{24 - 12 \cdot \sqrt{3}}} = 69^\circ \quad 2-8$$

$$\varepsilon = \left[\frac{\sqrt{3}}{2} - \sin 69^\circ \right] - \left[1 - \frac{\sqrt{3}}{2} \right] \cdot \sin 207^\circ = 0.0067 \quad 2-9$$

$$\% \varepsilon = \frac{0.0067}{\frac{\sqrt{3}}{2}} \cdot 100\% = 0.78\% \quad 2-10$$

Note that % error reflects the effect of the error on the available bus.

What we have discovered at this point is that referencing the neutral of our control algorithm to the virtual ground (1/2 of the bus) was a bad choice. If we wish the control algorithm to mimic the source, then the control's neutral point with respect to ground must mimic the source's neutral point with respect to ground. Note how, in Figure 2-1, the source and motor neutral points will be the same, even though they are not physically connected. So what we need to do is to insert a function that corrects the neutral point of the control algorithm relative to the virtual ground. Such a scheme is illustrated in Figure 2-7. The 3rd harmonic voltage is arbitrarily close to the desired correction, and easy to calculate, so that what we will use.

Fig. 2-7: Motor Controller with Neutral-Point Compensation

The new control algorithm becomes,

$$\begin{aligned}
 V_A &= V_1 + V_4 = M \cdot \sin(\phi) + \left[1 - \frac{\sqrt{3}}{2} \right] \cdot \sin(3 \cdot \phi) \\
 V_B &= V_2 + V_4 = M \cdot \sin(\phi + 120) + \left[1 - \frac{\sqrt{3}}{2} \right] \cdot \sin(3 \cdot \phi) \\
 V_C &= V_3 + V_4 = M \cdot \sin(\phi - 120) + \left[1 - \frac{\sqrt{3}}{2} \right] \cdot \sin(3 \cdot \phi)
 \end{aligned}
 \tag{2-11}$$

where M can now have a maximum amplitude of 1. Let us consider this carefully by examining Figure 2-8.

Fig. 2-8: Control Algorithm with 3rd-Harmonic Injection

Note carefully the node marked “Motor Neutral Point” in Figure 2-8. From the *motor’s* perspective, its phase-neutral voltages can now have the desired normalized amplitude of 1. At the same time, V_4 deducts from the phase-neutral voltage as each of the three phases passes through their respective peaks. Similarly, V_4 augments the phase-neutral voltage as each of the three phases passes through their respective valleys. In this way, the phase voltages, *with respect to ground*, never exceed the rails, that is, the normalized bus voltage of $\sqrt{3}/2$. The phase-to-ground voltages, along with the 3rd-harmonic (V_4) voltage, are shown in Figure 2-9.

The use of 3rd harmonic injection is well documented in the literature. Martin [6], for example, does a comparative analysis of various PWM techniques including 3rd harmonic injection. Compare the flattened phase-to-neutral in his analysis of 3rd-harmonic injection (shown in Figure 2-10) with our own simulation in Figure 2-9.

Fig. 2-9: Phase Voltages with respect to the Virtual Ground after 3rd-Harmonic Injection

The 3rd-harmonic technique was so widely used in the 1990's that some specialized motor control I.C.'s include a 3rd-harmonic function. One of these, Fairchild Semiconductor, correctly states, "... V_{BUS} is not being fully utilized." [7] However, in much of the literature, 3rd harmonic injection is touted as a method for achieving higher phase-phase voltages as if we are gaining something extra. Martin states, "...line to line voltages remain sinusoidal and are increased up to a 15% respect to pure sinusoidal modulation (sic)."

Fig. 2-10: 3rd-Harmonic Injection as Depicted by Martin

This type of wording is misleading for several reasons. 3rd harmonic injection is a *necessary* part of the control scheme if we are to achieve something as mundane as just driving a motor to the level of the source.

A second deceptive point is the notion that we are applying some strange flattened-out phase-neutral waveform to the motor. But this strange waveform only applies if we reference the new phase voltages to the virtual ground (bus midpoint voltage) as shown in Figure 2-8. Figure 2-11, simulated from Figure 2-8, shows that the phase-to-neutral voltages are purely sinusoidal, and they now have a magnitude of 1.

Fig. 2-11: Phase-Neutral Voltages after 3rd-Harmonic Injection

2.3: Variable-Frequency, Variable-Voltage Drives

What we have at this point is a drive where we can control the frequency and amplitude of the motor voltages. This is known as a Variable-Frequency, Variable-Voltage drive. By definition, synchronous motors turn at a ratio of the electrical frequency. (The ratio of the electrical to mechanical speed being the number of pole-pairs.) If the motor *can* turn it *will* turn at the applied frequency. We must also apply enough voltage to overcome the BEMF and drive current into the motor to produce torque.

Recall Figure 1-1b, reprinted here.

Fig. 2-12: Motor Torque

The stator flux rotates at the commanded frequency. The rotor flux is attracted to the stator flux causing rotor rotation. But as the rotor flux angle approaches the stator flux angle, the torque approaches zero (see equation 1-3). Therefore, the system quiesces at a torque equal to the load, so θ varies with the load.

Now consider the equivalent circuit of one phase as shown in Figure 2-13 [3].

Fig. 2-13: Stator Phase-Neutral Circuit

E_1 is the M (voltage magnitude) of our controller and E_2 is the BEMF. We can vary M in proportion to the velocity (and hence the BEMF), but we must always insure we have enough current so that we can produce torque to drive the load when θ slips to the maximum of 90° .

A disadvantage to this drive then is that it must always deliver an apparent power at least equal to the worst-case real power requirement. This results in efficiency loss from I^2R heating in both the drive and motor at less than full power. Another significant disadvantage is the varying angle between the commanded voltage and the rotor. This makes the drive unsuitable for servo applications. However, the phase-neutral voltage angle is always known; indeed, it is commanded. This makes implementation of 3rd harmonic injection simple.

2.4: Self-Commutation

If we attach a position sensor (optical encoder, resolver, etc.) to the shaft, we can use this to generate the terminal voltage vector angle. In other words, we can cause the angle between E_1

and E_2 to be anything we desire and maintain it (e.g. 90°). This is analogous to the action of the mechanical brushes and commutator ring in a brush-type motor. The motor will now speed up until the terminal - BEMF voltage differential results in a current, and hence torque, that equals the load torque. This eliminates the need to control both frequency and voltage. It also eliminates the possibility of the motor falling out of commutation due to an insufficient terminal voltage. However, we do lose direct control of the speed, and worse, the power angle between the phase current and BEMF is still a function of the motor impedances and load. Consider the terminal and BEMF voltage vectors and the current vector shown in Figure 2-14 [3]:

Fig. 2-14: Voltage and Current Phasors

By self commutation, we are now in control of the terminal voltage angle relative to the BEMF, but not the current. Such a drive was presented in Figure 1-10 and is repeated below.

This system is known to work well and has been used in commercial servo systems for decades. Where precise velocity control is needed, an additional velocity control loop is closed around the current (torque) loop.

Fig. 2-15: Stator Frame Current Control

Even so, this scheme is still flawed. The control loop exists in the stator (or stationary) reference frame. In-other-words, the inputs and outputs to and from the control loops are AC values. If we postulate a frictionless (or even low friction system) system running at a constant high velocity, then the torque command is zero (torque would produce acceleration). Yet the voltage outputs from the loops must be high-amplitude, high-frequency (relatively) signals. If the input is zero, the error must come from the feedback. Yet it can't be real (in phase with the BEMF) or we would again have acceleration. Therefore the motor must again adjust itself so as to cause some imaginary current flow. The efficiency of this scheme is therefore sensitive to the gain and bandwidth that can be achieved by the compensators.

This scheme does still allow us to easily use 3rd-harmonic injection, since the angle of the terminal voltage vector is directly measured.

2.5: Field-Oriented Control of PMSM

In Field Oriented Control, we define a rotating reference frame. We can attach this frame to any of the moving fluxes; rotor, air-gap, or stator. The rotor flux is especially easy to use since it rotates with the shaft. We will again attach an angle sensor to the shaft and align it so as to have its zero angle output aligned to the rotor flux. The measured currents transferred to the rotating frame now become stationary in steady state. The components of the vector in polar form give us the magnitude and phase of the signal with respect to the rotor flux. It is universally common to use the rectangular form, where the components are referred to as D (direct) and Q (quadrature). The process of converting the signals from the stationary (stator) reference frame to the rotating frame is known as the Park Transform.

A typical FOC system is shown in Figure 2-16 [8].

Fig. 2-16: Basic FOC Drive Scheme with Speed Loop

Finally, the 3-phase signals are related to each other by 120° . We need to convert these to orthogonal coordinates before we can use them. This process is known as the Clarke transform and uses simple projection of the vectors onto the orthogonal coordinates.

With the currents on a rotating frame, the error amplifiers no longer have to process AC signals. In fact, if the BEMF was purely sinusoidal, the signals would be pure DC in the steady state. Once the components of the voltage vector have been computed, we simply reverse the Park and Clarke transforms to produce our 3-phase voltage commands.

We will talk about the block marked SVPWM (Space-Vector Pulse-Width Modulation) shown, in Figure 2-16, in the next section. There is nothing inherently necessary about using PWM. FOC would work just as well with the linear stage from Figure 2-2. And, unfortunately, many authors don't explicitly show the inverse Clarke transform. In Figure 2-16, the inverse Clarke transform is buried in the SVPWM block. For now, think of the SVPWM as the inverse Clarke. It is the inverse Clarke that converts the 2-phase orthogonal vectors into the 3-phase vectors.

We will use the basic structure shown in Figure 2-16 to prove our proposed scheme. Note that we want the Q-axis (torque producing) current to be exactly in phase with the BEMF, so that the i_{sdref} value shown in Figure 2-16 is simply zero. Also, we mentioned previously that we have lost direct control of the velocity. Figure 2-16 includes a velocity control loop to compensate for this.

2.5.1: The Clarke and Park Transforms

In the last section, we introduced the Park and Clarke transforms. We will now define them in a more formal fashion.

The three stator fluxes vary in magnitude, but are fixed in direction. The interaction of the three fluxes has some net magnitude and direction, and it is this vector that we desire to represent in an orthogonal coordinate system. The fluxes are proportional to the phase currents, so we may also say this is a current vector.

The transformation is accomplished by projecting the three vectors onto the orthogonal axis. Figure 2-17 [9] clearly illustrates this along with the final equations:

Fig. 2-17: The Forward Clarke Transform

The development of these equations proceeds as shown in equations 2-12 through 2-15:

$$i_{\alpha} = i_A - \frac{1}{2}i_B - \frac{1}{2}i_C \quad 2-12$$

$$i_{\beta} = \frac{\sqrt{3}}{2}i_B - \frac{\sqrt{3}}{2}i_C$$

But by Kirchoff's current law for the phase currents,

$$i_A + i_B + i_C = 0 \quad 2-13$$

$$\therefore i_C = -i_A - i_B$$

Substituting,

$$\begin{aligned}
i_\alpha &= \frac{3}{2}i_A \\
i_\beta &= \frac{\sqrt{3}}{2}i_A + \sqrt{3}i_B
\end{aligned}
\tag{2-14}$$

Finally we normalize the result (this will make more sense when we talk about number formats in chapter 4).

$$\begin{aligned}
i_\alpha &= i_A \\
i_\beta &= \frac{1}{\sqrt{3}}i_A + \frac{2}{\sqrt{3}}i_B
\end{aligned}
\tag{2-15}$$

The inverse Clarke transform is literally the inverse and is found by letting $A=\alpha$ and solving for B in equation 2-15. Finally, $C = -A-B$. In the inverse Clarke case voltages are used instead of currents, since we are converting the output of the control loop into the phase voltages. Voltages do not, as a rule, follow Kirchoff's current law, but in this case they do, since $I = I/Z$ and Z is the same for each phase.

$$\begin{aligned}
V_A &= V_\alpha \\
V_B &= \frac{\sqrt{3}}{2}V_\beta - \frac{1}{2}V_\alpha \\
V_C &= -\frac{\sqrt{3}}{2}V_\beta - \frac{1}{2}V_\alpha
\end{aligned}
\tag{2-16}$$

The Park transform projects a vector from the orthogonal stationary (stator) reference frame onto another orthogonal rotating (rotor) reference frame. The two frames are separated at any instant by the angle θ . The transformation is carried out by projecting the components from one frame to the other as was done for the Clarke transform. Figure 2-18 [9] illustrates the technique.

Fig. 2-18: The Forward Park Transform

Similarly, the inverse Park transform projects a vector from the rotating reference frame back to the stationary frame. This is illustrated in Figure 2-19 [9]:

Fig. 2-19: The Inverse Park Transform

2.6: Phase Voltage Coupling

Although controls of the form shown in Figure 2-16 are common, they are not mathematically tractable. In both cases, it appears that V_D^* and V_Q^* are computed from the I_D and I_Q errors in a decoupled sense. This is not the case. Consider simply that a sinusoidal current flowing through the stator inductance will induce a sinusoidal voltage in quadrature to the current. Without invoking complicated math, it is obvious that D-axis voltages will produce Q-axis currents and vice-versa. Some control algorithms take this into account. Figure 2-20 [10] shows a control scheme for an induction motor that includes a decoupling algorithm. The

justification for ignoring the decoupling in Figure 2-20 comes from two sources: The first is the basic idea that a Type 1 control will develop any output necessary to force the input error to zero [11].

Fig. 2-20: IM FOC with Voltage Decoupling

in the steady state step response. The second is that even though V_D may affect both I_D and I_Q , I_Q affects V_Q and V_Q in turn affects I_D so that eventually we achieve closed-loop control. The same argument applies to the effect V_Q . This cross-coupling concept is shown in Figure 2-21.

2.7: Space-Vector PWM

Although linear amplifiers as shown in Figure 2-2 exist, they are much less common than PWM amplifiers. We will be using a PWM scheme known as Space-Vector Pulse-Width Modulation (SVPWM), so a few words are needed.

Fig. 2-21: Cross-Coupling

The output stage will consist of IGBT's configured as shown in Figure 1-8 and is repeated here.

Fig. 2-22: Voltage-Mode Output Stage

Clearly both the upper and lower stage cannot be on at the same time in any half-bridge or the DC-link would be short-circuited. At the same time, turning both off leaves the phase

voltage undefined. We require then that for each phase, either the upper or lower device is on (sourcing or sinking). With at least one upper transistor on and one lower transistor on (the third phase will also have either the upper or lower on), the voltage vector, can only point in one of six directions. If all uppers or all lowers are on, the phase-phase voltages are all zero. We call these the “zero vectors”.

To obtain intermediate angles, we simply alternate between two adjacent states with whatever duty cycle is needed to create an average that gives the desired direction. Moreover, we can alternate with the zero vectors to create any desired magnitude. Table 2-1 shows the phase states and Figure 2-23 [5] illustrates the resulting vectors.

Table 2-1: Phase States

	V0	V1	V2	V3	V4	V5	V6	V7
A	SA'	SA'	SA'	SA'	SA	SA	SA	SA
B	SB'	SB'	SB	SB	SB'	SB'	SB	SB
C	SC'	SC	SC'	SC	SC'	SC	SC'	SC

The author of Figure 2-23 uses lower-case d and q to denote the stationary reference frame and has nothing to do with FOC. With appropriate time averaging, the SVPWM can produce the same continuous rotation of the stator flux vector as the linear system, along with any desired magnitude. It was mentioned earlier that the virtual ground did not actually exist in a real system. Instead, we command a 50% duty cycle for a zero input. 100% corresponds to being hard against the positive rail (the upper-pole device is always on) and 0% corresponds to

being hard against the negative rail (the lower-pole device is always on). At 50% the average voltage is zero with respect to the virtual ground.

Fig. 2-23: SVPWM

. We mentioned in Chapter 1.1.2 that BLDC motors were similar to PMSM motors. A simple drive control that generates only the 6 distinct space vectors is known as a *6-step* or *trapezoidal* drive. The terminal voltage waveform has a trapezoidal shape, and a BLDC motor is wound to produce a BEMF that is more trapezoidal than sinusoidal. The motors are called brushless DC because in any given step the terminal voltage is DC.

Chapter 3: Proposed Polar FOC w/ 3rd Harmonic Injection

We propose to implement 3rd harmonic injection into a field-oriented control scheme. To accomplish this, we will need to design the field-oriented control using polar coordinates. We will use the well documented rectangular form to develop the polar form.

3.1: Polar Inverse Clarke and Park Transforms

Figure 3-1 shows the forward path of a rectangular FOC. We wish to replace the Inverse Park and Clarke transforms with Polar versions.

Fig. 3-1: Forward Path of Rectangular FOC

For the polar inverse Clarke, consider again the control algorithm for a voltage-mode drive with 3rd harmonic injection shown by equation 2-11, reprinted here:

$$\begin{aligned}
A &= V_1 + V_4 = M \cdot \sin(\phi) + \left[1 - \frac{\sqrt{3}}{2}\right] \cdot \sin(3 \cdot \phi) \\
B &= V_2 + V_4 = M \cdot \sin(\phi + 120) + \left[1 - \frac{\sqrt{3}}{2}\right] \cdot \sin(3 \cdot \phi) \\
C &= V_3 + V_4 = M \cdot \sin(\phi - 120) + \left[1 - \frac{\sqrt{3}}{2}\right] \cdot \sin(3 \cdot \phi)
\end{aligned} \tag{3-1}$$

We note that

$$\sin(\phi) + \sin(\phi - 120) + \sin(\phi + 120) = 0 \tag{3-2}$$

And therefore

$$V_3 = -(V_1 + V_2) \tag{3-3}$$

This saves us a trig operation that will save us some execution time when we go to actually program the algorithm. Equation 3-4 shows the final form of 3-1.

$$\begin{aligned}
A' &= M \cdot \sin(\phi) \\
B' &= M \cdot \sin(\phi + 120^\circ) \\
C' &= -(A' + B') \\
\zeta &= M \cdot \left[1 - \frac{\sqrt{3}}{2}\right] \sin(3 \cdot \phi) \\
A &= A' + \zeta \\
B &= B' + \zeta \\
C &= C' + \zeta
\end{aligned} \tag{3-4}$$

Equation 3-4 generates the ABC motor voltages but uses M and ϕ for inputs.

For the polar Inverse Park we note that the magnitude of a vector in different reference frames remains unchanged. The angle is found by simply adding the angle between the frames to the angle of the vector. This is illustrated in Figure 3-2 and equation 3-5:

$$\varphi = \Phi + \theta$$

3-5

Fig. 3-2: Polar Reference Frame Transformation

The error amplifiers remain unchanged except they now act on magnitude and phase data.

Figure 3-3 shows our final forward-path polar FOC:

Fig. 3-3 Final Forward Path of Polar FOC

3.2: Polar Park and Clarke Transforms

Now we turn our attention to the feedback path. Since the input to the error amplifier is now in polar form, we add a rectangular-to-polar transform to the rectangular feedback as shown in Figure 3-4:

Fig. 3-4: Polar FOC Feedback Path

As explained previously, the polar magnitude of a vector does not change from one reference frame to another. To implement the Park transform in polar coordinates, we need only subtract the angle between the frames as shown in equation 3-6:

$$\Phi = \varphi - \theta \quad 3-6$$

The resulting simplification is illustrated in Figure 3-5:

Fig. 3-5: Polar FOC Feedback with Polar Park

The equations for the rectangular-to-polar transformation are given by equation 3-7:

$$I_M = \sqrt{I_\alpha^2 + I_\beta^2} \quad 3-7$$

$$I_\phi = \arctan \left[\frac{I_\beta}{I_\alpha} \right]$$

Since the arctangent is only defined from $-\pi/2$ to $\pi/2$, and includes infinities, special techniques must be employed. These will be described later.

To complete the polar Clarke, we could absorb the rectangular Clarke into the rectangular to polar conversion. We see little point in doing so, since the rectangular Clarke would have to be absorbed into both equations, thus replicating effort. Therefore our polar Clarke will consist of the combination of a rectangular Clarke and a rectangular to polar conversion.

This completes our proposal for a fully polar FOC with 3rd harmonic injection. Figure 3-6 shows the final block-diagram representation.

Fig. 3-6: Complete Polar FOC with 3rd-Harmonic Injection

Before we close the proposal section, it should be noted that the idea of a polar FOC is not new. Several theoretical texts include conceptual drawings of how a polar FOC might be implemented. Never the less, no practical implementation has ever been published as far as our research can tell.

Figure 3-7 shows Bimal K. Bose's notion of a polar FOC [12]:

Fig. 3-7: Polar FOC Proposed by Bose

We note in particular the subtraction of the shaft angle θ from the current feedback angle α . Conversely, θ is added to the angle reference β^* to obtain the current angle reference α^* . This is identical to what we termed the Polar Park and inverse Park transforms.

A torque value T_e is computed (just a scaled version of the measured current magnitude), but the inclusion of the sine term indicates that the torque control loop is actually the equivalent

of the Q-axis current in a rectangular FOC. Never the less, the torque value is fed to an error amplifier with compensator $G2$ and the result is termed the current magnitude reference $|\hat{I}_s^*|$.

However, no attempt is made to perform closed-loop angle control. Instead this system estimates a required amount of phase-lead based upon the current reference.

The current reference and angle reference are fed to a vector rotator (VR) and 2ϕ to 3ϕ transform. This is our polar inverse Clarke given by equation 3.1.4, although Bose does not attempt 3rd-harmonic injection.

This diagram also features closed-loop position control, closed-loop velocity control, and field weakening. None of these features are directly concerned with our desire to produce a polar FOC with 3rd-harmonic injection, so they are omitted from our proposal.

Chapter 4: Implementation

In this chapter we present the nitty-gritty of a practical implementation.

4.1: DSP

We will be using the Analog Devices ADMC300 Digital Signal Processor (DSP) as our controller, so comments about its features are in order:

1. A 5-channel, 16-bit, Sigma-Delta Analogue-to-Digital Converter (ADC). The 5-channels are divided into two banks of 2 and 3. The sampling instant may be synchronized to the PWM output to minimize noise pickup caused by the dV/dt of the output switching. We will be using the ADC to measure the motor phase *A* & *B* currents.
2. 6-channel PWM outputs. These are divided into 3 pairs. The pairs are essentially complementary, except a small “dead time” may be inserted (a time when both outputs are off) to prevent “shoot through” as one output device turns off and the other turns on. In general, either the upper pole device or the lower pole device is on and the other off at all times to create the 8 possible space vector states (see the section on SVPWM).
3. Incremental Encoder Interface Unit (EIU). We will use an incremental encoder mounted to the motor shaft to track the shaft angle. This device outputs a pair of TTL signals that are in quadrature. The EIU tracks the shaft position from these signals.
4. 2-channel serial port (SPORT). These will be used to communicate in ASCII to a terminal window program on a PC to feed commands and read back data, as well as interface to a debugger running on the same PC.

5. Arithmetic Logic Unit (ALU) to perform various 16 bit arithmetic operations such as add, subtract, negate, not, and, or, xor, and control most conditional branches. There is also a special divide operation that can perform 16-bit division in 16 cycles.
6. Multiply Accumulate Chip (MAC). Performs 16-bit by 16-bit multiplication with a 32-bit result. The MAC can also perform single-cycle multiply and accumulate operations, and there is an extra 8-bit buffer to handle overflows during intermediate multiply and accumulate operations.
7. Barrel Shifter. The shifter can place a 16-bit word anywhere in a 32-bit result with sign-extension if desired. A shift and or operation allows double-precision shifts.
8. There are also several timers, a 12-bit I/O port, two auxiliary PWM outputs, two capture and compare inputs, 8 sophisticated address generators for indexed addressing, a second set of data registers that allows single-cycle context switching (instead of the usual “push” and “pop”, and various other hardware features common to modern microprocessors.
9. The instruction set features all single-cycle instructions and even multifunction instructions such as an arithmetic operation with simultaneous memory read or write.

4.2: Language

Although a ‘C’ compiler is available for the DSP, we choose not to use it. ‘C’ is a deceptive language. What appears to be a single ‘C’ instruction may compile into many cycles of machine code. A good example is division. Even though the ADMC 300 includes special hardware for division, it still requires at least 16 cycles. In ‘C’, division appears to take only one cycle, and that assumes integer division. These problems make it difficult to write highly efficient code that must execute in real-time in an embedded application. Even a carefully

written piece of ‘C’ code can never compile to anything smaller than the same code written in assembly. Therefore we choose to write directly in assembly.

4.3: Numeric Formatting

The ALU and MAC are specially optimized for use with fixed-point, 2’s compliment, fractional operations. We refer to this as 1.15 formatting, meaning that the most-significant bit (MSB) has a weight of 1, the next a weight of $\frac{1}{2}$, and so on. Since the MSB in 2’s compliment is also the sign bit, numbers from -1 (0x8000) to $+1 - 1$ LSB (0x7FFF) may be represented, where 0x denotes a hexadecimal number.

A 16-bit by 16-bit 2’s compliment multiplication results in a 2.30 result, i.e. there are redundant sign bits. The MAC can automatically left-shift the result to yield 1.31 format.

Similarly, the divide hardware is intended to accept a 1.31 formatted dividend, a 1.15 formatted divisor, and produce a 1.15 formatted quotient.

We will store most of our variables and coefficients as 1.15 “normalized” or “per-unit” values. For example, a 16-bit current with a value of “32” is not 32 amps, but $32 \div 32768^{\text{ths}}$ of some maximum value.

4.4: Special Routines

There are several routines to perform special operations we have written or researched that deserve special attention.

4.4.1: Square Root

We require a square-root routine for the rectangular-to-polar transform. Recall we will be solving the equation,

$$I_M = \sqrt{I_\alpha^2 + I_\beta^2} \quad 4-1$$

Also recall that a 1.15 multiplication (in this case a square) yields a 1.31 product, so the sum-of-squares may be as large as 2.31. However, the result of a normalized Clarke transform will only yield a vector that is less than 1. Therefore we assured of a 1.31 sum. This will be the input format to the square-root algorithm.

Research at Wikipedia [13] and Analog Devices [14] revealed several common methods for performing a square root:

1. The Babylonian method. This uses the approximation,

$$Y_{n+1} = \frac{1}{2} \left[Y_n + \frac{X}{Y_n} \right] \quad 4-2$$

The approximation must be repeated multiple times to achieve the desired degree of precision. This approximation requires division, and 16-bit division requires 16 cycles alone. Combined with the repetitions, this does seem to be an efficient method.

2. Power series. Analog Devices offers “canned” code to perform a square root using a power series equation of the following form:

$$\sqrt{X} = K_0 + K_1 X + \dots + K_5 X^5 \quad 4-3$$

It is valid for inputs in the range 0.5...1, therefore it requires pre- and post-manipulation to solve square roots over a wider range. Although the DSP is especially adept at sum-

of-products solutions, it requires 75 cycles and is still only an approximation. Worst of all, it is designed to accept an input format of 16.16, not 1.31.

3. Bit-by-bit solution. This is especially attractive in a digital application. Wikipedia offers a C solution that is valid for finding integer roots.

Method 3 inspired us to write our own bit-by-bit routine in assembly that is valid for 32 bit fractional roots. The method is illustrated in Fig. 4-1. Note that we begin the bit test at the second MSB since negative results are impossible. The use of a table rather than shifting allows us to take advantage of the multi-function capability of the DSP and save a cycle for each bit tested. The DSP also features a zero-overhead looping capability. Once the counter is loaded, it takes no extra cycles to execute the 15 loops. At 78 cycles it is no better than Analog Devices' method, but it is exact (not an approximation), and works over the desired range and resolution of the input.

4.4.2: Arctangent

The arctangent, also needed for the rectangular-to-polar transformation, poses special problems. It is only valid for -90° to 90° and even then the input is $\pm\infty$ at these angles. We can solve second problem by further restricting the angle to $\pm 45^\circ$. Over this range the input will be less than 1, and so it works with our 1.15 format. It works equally well over $180^\circ \pm 45^\circ$. We can distinguish these regions by noting that the α -axis is positive in the first, and negative in the second.

Fig. 4-1: Square Root Algorithm

We can solve for the remaining quadrants by using the arc cotangent. The arc cotangent is less than 1 in the regions $90^\circ \pm 45^\circ$, and $-90^\circ \pm 45^\circ$. Again we distinguish the regions by noting that the β -axis is positive in the first and negative in the second. These concepts are illustrated in Figure 4-2.

It is interesting to note that, in 16 bits, it does not matter if we consider $0 \dots 65536 = 0 \dots 360^\circ$, or $0 \dots \pm 32768 = 0 \dots \pm 180^\circ$, they are numerically identical. Therefore our 1.15 2's complement formatting works fine. Analog Devices provides us with canned assembly routines that accept and return 1.15 numbers for computing the arctangent and arc cotangent.

4.5: PWM Block

The PWM block is straightforward, but there are a few special considerations. The PWM is just a glorified timer that counts at $\frac{1}{2}$ the master clock rate of 26 MHz. This counter may be

Fig. 4-2: Computing the Arctangent

set to rollover at any desired 16-bit count (PWMTM register). This rollover value sets the PWM frequency given by,

$$f_{PWM} = \frac{f_{CLK}}{2 \cdot PWMTM}$$

4-4

So, for example, if we set PWM_{TM} = 1024, the PWM frequency will be 12.695 KHz. This is the lowest we should set the PWM frequency to keep it out of the audible range. This also sets the resolution of the PWM, in this case 10 bits.

A block diagram of the PWM unit is shown in Figure 4-3 [15].

Fig. 4-3: PWM Block Diagram

An interesting feature of the PWM unit is that it can be setup in ‘double update’ mode. This means that the ‘on’ time relative to center can be controlled independently, rather than just being symmetrical about center. This effectively doubles the bandwidth of the controller for a given PWM frequency.

The duty cycle is set by the ratio of input (PWMCHx) to PWMTM. The outputs are ‘center based’ as illustrated in Figure 4-4 [15]:

Fig. 4-4: Center-Based Double-Update PWM

Figure 4-4 also shows the automatic dead-band insertion that prevents shoot through. Also, take note of the sync pulse that is generated. This is used to start the ADC conversion at the center points to minimize noise generated by the dv/dt of the outputs switching.

The PWM registers are unsigned, but the output of equation 3-4 is signed. Therefore we offset the PWM output to 50%. As long as all outputs are equal, the differential voltage is zero. As we increase the duty cycle of one channel while decreasing another, we create a differential voltage.

If we setup so that 1024 (10 bits) is 100%, then the positive range (50% to 100% DC) is only 512. 512 will represent a normalized 1, while 0 will represent the normalized -1. This is easily handled by multiplying the normalized phase voltage by 512 prior to adding the 50% (also 512) offset.

The PWM unit also includes a feature to high-frequency chop the PWM which is useful for designs where magnetically, rather than optically, coupled gate drivers are used. It also includes features for emergency shutdown of the PWM. This is tied to the “desat” detector feature in the gate drivers.

4.6: Analog-to-Digital Converter

The Analog-Digital Converter (ADC) is used to read the phase currents. The transfer function for the converter is shown in Figure 4-5 [15].

The current sensors used have a built-in 2.5V offset so that negative currents (into the amplifier) are less than 2.5V, while positive currents (out of the amplifier) are greater than 2.5V. The A/D converter has differential inputs. By tying the (-) input to the 2.5V reference in the converter, the converter is nicely outputs 2’s complement numbers.

Fig. 4-5: ADC Transfer Function

Although the ADC includes a built-in calibration function, we do not use it because it does not take into account the offset between the sensor reference and the ADC reference, nor any other external offsets. Instead, during the power-up routine, we average 2^{16} samples while

the PWM is fixed at 50%. We save this measured offset and subtract from all future samples.

Figure 4-6 shows the converter block diagram [15].

Figure 5. Functional Block Diagram of ADC System of ADMC300

Fig. 4-6: ADC Block Diagram

Finally, it was mentioned in the PWM section that the PWM sync signal is used as the convert start signal. This is illustrated in Figure 4-7 for the single update mode [15]. There is an additional conversion in double update.

Fig. 4-7: PWM Sync and Convert Start

4.7: Position Sensing

We use an incremental optical encoder for rotor (and hence, rotor flux) angle sensing.

The encoder outputs a pair of quadrature signals as shown in Figure 4-8 [16].

Fig. 4-8: Encoder Signals

A sequence of 00-01-11-10-00... in the A and B signals indicates a CW rotation, while 00-10-11-01-00... indicates a CCW rotation. These signals are fed to the EIU (Encoder Interface Unit) which is a glorified up-down counter with a settable rollover. We set the counter to roll over at the number of counts per mechanical revolution. A block diagram of the EIU is shown in Figure 4-9 [15].

Fig. 4-9: EIU Block Diagram

The encoder also outputs a set of 3 absolute position signals, but our target drive hardware is not equipped to utilize them. We will discuss the purpose of these signals when we describe EIU initialization.

We define the normalized rotor angle as $\pm 32768 = \pm 180^\circ$, so we read the EIU counter and scale it up so that one mechanical rotation = 2^{16} . Equation 4-5 gives the scale value.

$$scale = \frac{2^{16}}{counts/revolution} \quad 4-5$$

We also need to note that this is an *electrical* rotor angle. Motors may have multiple sets of windings, and these are referred to as pole pairs. Likewise, for the permanent magnet motor, there are multiple permanent magnets attached to the rotor. The stator vector must complete N cycles for each mechanical revolution of the shaft, where N is the number of pole pairs. Therefore we multiply the now normalized mechanical angle by the number of pole pairs. Finally, we note that after each scaling the angle value is allowed to roll over at the 16 bit boundary.

The EIU unit also has a timer for measuring the time between the quadrature edges. The reciprocal of this time gives velocity very accurately at low speeds. We will not be using this feature since we are not attempting to close a velocity loop.

4.7.1: Initializing the EIU

As the name implies, an incremental encoder only supplies an incremental, not an absolute angle. This is fine for an induction motor, but for a synchronous motor we need to know the absolute angle. As we have defined our transforms, we need the electrical rotor angle to be zero when the rotor flux is pointing in the mechanical direction of the A stator phase. Therefore we need some sort of initialization of the angle. There are several ways of doing this.

An encoder such as this one also outputs a set of three signals that are absolute in nature and generate 6 unique codes. There are as many of these sets as there are pole-pairs in the motor. These signals can be used to directly commutate a 6-step (aka trapezoidal) drive ideally driving a BLDC motor. The absolute position is accurate to within $\pm 30^\circ$ electrical. This can be used to initialize the EIU counter. As the motor rotates past one of the code transitions, the angle is known exactly and the EIU counter can be set exactly. Note that to use these signals, the

encoder must be mechanically mounted to the rotor with the correct orientation. We cannot use this technique because our target drive does not have hardware to accept these signals.

Instead, we will use a technique known as “twanging”. We apply fairly heavy torque (current) command and a fixed angle of 90° . The motor may not rotate if it is near equilibrium or anti-equilibrium. Recall that there is zero torque in either of these situations (the vector cross-product). Now we force an angle of 0° . The motor will rotate to near 0° . We now set the EIU counter to zero degrees. Equation 3.4.1 utilizes sines, so the commutation angle will now be 90° .

4.8: The Target Drive

We will be using a PWM servo amplifier, designed and built by the author, as our test bed. A photo of the drive is shown in Figure 4-10.

Fig. 4-10: Target Drive

This drive was originally built to satisfy the Cal Poly undergraduate senior project requirement. This amplifier contains all the necessary hardware to interface to the user, motor, encoder, current sense, etc. The relevant hardware in the target system is shown in Figure 4-11.

Fig. 4-11: Model VSD-1 Motor Drive

4.9: Target Motor

A picture of the target motor is shown in Figure 4-12.

Fig. 4-12: Target Motor

This motor was purchased at ebay.com as a PMSM “mystery motor”, owing to the fact that there is no manufacturer’s label. The presence of the second MS connector suggested the existence of an encoder.

4.9.1: Target Encoder

Upon removing the rear cover, it was found that the motor is equipped with an Encoder Technologies LTD model OIH35-1000-P6-L6-5V incremental optical encoder. The 1000 indicates 1000 “line counts” per mechanical revolution resolution, and 6-pole commutation tracks. Note that by convention, a line count is rising edge to rising edge of one encoder channel. The total edges are 4 times this. The P6 suggests that the motor has 6 poles. These two pieces of information are essential for the calculations described previously.

4.10: Support Equipment

In addition to the target drive and motor, the following equipment, which is shown in Figure 4.13, was used in the development and evaluation of this project:

- Heathkit model IP-5220 “Variable Isolated AC Supply’. This power supply is capable of delivering 0 to the AC line voltage (typically 110-120 VAC) at 3A RMS. The author retrofitted this supply with 4 1N5406, 600V, 3A rectifiers in a bridge configuration; 4 paralleled 2700uF, 200V capacitors; and binding posts so as to provide up to about a 170 VDC bus (DC link) supply.
- Power Designs 5015-S “Regulated DC Power Supply”. This power supply is capable of 50V and 1.5A with current limiting. It was used to supply the 5V logic power.

- Tektronix TDS 2024C oscilloscope. This is a 200MHz, 2GSPs, 4 channel, color LCD 'scope used to measure the motor terminal voltages. It has math capability so as to provide phase-phase voltages and USB flash-drive storage of data.
- Acer PC loaded with Windows '98® OS and Analog Devices MCD (Motion-Control Debugger). It was used for code debugging and data acquisition.
- Dell PC loaded with Windows 7® OS and Microsoft Visual C++®. Visual C++ was used only for its code editing and file organizing capabilities. The code was compiled and linked using Analog Devices DOS assembler and linker software designed for the target DSP running on this same PC. In addition, this PC is loaded with Terra Term® terminal emulation software to communicate via ASCII over the target's RS232 link.
- Mastech M-830B DMM. Used for general purposes such as the precise setting of the 5V control voltage.

Fig. 4.13: Full Test Bench

Chapter 5: Evaluation

The system was programmed as described. At first, only the magnitude control loop was closed.

The first problem encountered was a reverse saturation of the magnitude integrator. The problem was evaluated as follows: All real-world systems exhibit offsets. Even the A/D converters themselves exhibit this. Let us postulate a positive offset. The magnitude control loop would need to drive negative to correct this. But even the smallest overshoot would again drive the error positive because of the absolute-value nature of the magnitude. This thus becomes positive feedback, and drives the loop into negative saturation.

This problem was solved by restricting the magnitude compensator to positive values only. This, in turn, means the offsets cannot, necessarily, be corrected. This leaves, not only an error in magnitude, but an error in phase as well: It is an error that the phase loop cannot correct, and so the phase loop saturates. It was thus determined that a bipolar magnitude would be more appropriate. Any negative angle in the rotating frame negates the magnitude, and angles are restricted to positive values. This relationship is demonstrated by,

$$\sin -\Phi = -\sin \Phi \quad 5.1$$

The magnitude control loop now closed stably. However, we could not close the phase control loop stably. In particular, at low currents it broke into violent oscillations. In considering this, we make several observations:

First, if the current is truly zero, the angle is undefined. Since we can't perform the division, we define this to be a zero angle. In reality, as current begins to flow, the actual angle

might be anything, depending on where the rotor flux is pointing. A better solution might be to freeze the PI compensator at the last known angle.

Second, imagine even a one-bit dither at zero current. This might exist in either the ADC, or from digital switching noise. The angle from +1 LSB to -1 LSB, on either X or Y, would be interpreted as 0° to 180° (dither in X) or 90° to -90° (dither in Y). Similarly, common-mode noise, also a common phenomenon, would be interpreted as 45° to 225° .

This last problem is formidable. Before tackling it, we chose to do a little blue-sky thinking.

Chapter 6: Proposal II

As a result of the problems encountered in the initial evaluation, we make some remarkable observations:

The magnitude of a vector does not change with the reference frame. Thus we may say,

$$M = \sqrt{D^2 + Q^2} = \sqrt{X^2 + Y^2} \quad 6.1$$

Where D and Q , and X and Y are the rectangular components of the current in the rotating and stationary reference frames respectively. But the goal of FOC is to drive D to zero. Thus, in normal operation,

$$M \cong \|Q\| \quad 6.2$$

Moreover, in the last chapter we found we need M to be bi-polar to overcome other difficulties.

We present no fewer than four reasons to use Q as M :

- $M \cong \|Q\|$ if the phase error is small.
- We need a bipolar M to overcome other difficulties.
- The reference current command is actually a torque command, and only the Q current produces torque.
- It eliminates the lengthy square-root algorithm.

In addition, we can obtain the angle error directly without first obtaining the angle thus:

$$\varepsilon = \hat{\Phi} - \Phi \cong \tan(\hat{\Phi} - \Phi) = \frac{\sin \hat{\Phi} \cdot \cos \Phi - \cos \hat{\Phi} \cdot \sin \Phi}{\cos \hat{\Phi} \cdot \cos \Phi + \sin \hat{\Phi} \cdot \sin \Phi} \quad 6.3$$

where,

ε = The angular error between the current vector and the Q axis.

$\hat{\Phi}$ = The desired angle in the rotating reference frame.

Φ = Actual angle in the rotating reference frame.

But,

$$\hat{\Phi} = \pm 90^\circ \text{ and, } \Phi = \varphi - \theta \quad 6.4$$

where

φ = Actual angle in stationary frame.

θ = Rotor flux angle.

Thus,

$$\begin{aligned} \varepsilon = \hat{\Phi} - \Phi &\cong \frac{\cos \varphi \cdot \cos \theta + \sin \varphi \cdot \sin \theta}{\sin \varphi \cdot \cos \theta - \cos \varphi \cdot \sin \theta} = \frac{M \cdot \cos \varphi \cdot \cos \theta + M \cdot \sin \varphi \cdot \sin \theta}{M \cdot \sin \varphi \cdot \cos \theta - M \cdot \cos \varphi \cdot \sin \theta} \quad 6.5 \\ &= \frac{X \cdot \cos \theta + Y \cdot \sin \theta}{Y \cdot \cos \theta - X \cdot \sin \theta} = \frac{D}{Q} \end{aligned}$$

But this is even worse than before: Now the angle error estimate can go to infinity.

Repeating the same derivation, but using the small angle sine approximation, we can at least confine the error to $\pm 180^\circ$ since,

$$\varepsilon \cong \frac{D}{M'} \quad 6.6$$

and $M \geq D$. (Recall that a normalized angle of $\pm 1 = \pm 180^\circ$.)

It should be noted that the roles of lead and lag reverse when the direction of rotation changes. The division by a bi-polar Q or M solves this, but any other method would need to include a provision for this.

But what if we choose D only as our angle-error approximation? Clearly, if Q equals the normalized value of 1, then $\varepsilon \approx D$ anyway. When $Q = 0$, the error estimate is infinite, so the error in the error estimate is infinite. This clearly makes the use of D alone a viable option.

We rationalize this another way: The type I control loop (PI) will drive the error to zero (in steady state) regardless of the linearity of the error estimate. These are also known as “null-seeking” controls. This rationalization has already been used in Chapter 2.6 to eliminate the voltage decoupling. It is also in another critical system component: The current sensors.

The current sensors use ferrous cores and Hall sensors. Both are notoriously non-linear. And yet, these sensors produce accurate results, because they act to null the core flux where the non-linearities do not exist. This is illustrated in Figure 6.1 [17].

Fig. 6.1: Principle of LEM Current Sensing Module

Current in the primary creates a non-linear core flux. The transconductance amplifier converts the non-linear Hall measurement into a secondary current. The counter-flux nulls the primary flux; a region at or near zero where both the core and Hall sensor are linear. The method works so well that these sensors have a non-linearity of less than 0.1%.

We conclude that the use of $\varepsilon = D$ is justified for at least 3 reasons:

- It does not exhibit the drastic angle errors as in the arctangent estimate.
- It does not exhibit the infinite errors in the D/Q method.
- The PI compensator will drive the error to zero regardless of the input non-linearity.

So our plan-B proposal is to perform a standard Park transform on the measured currents, using the Q axis as magnitude, and the D axis as the angular error. Figure 6.2 illustrates the plan.

Fig. 6.2: Proposal II

Chapter 7: Evaluation II

Proposal II was implemented and functioned almost perfectly.

7.1: Experimental Proof of 3rd Harmonic Injection

In Figure 7.1 the 3rd harmonic injection is disabled. We see that both the phase and phase-to-phase voltages are sinusoidal. Note the values of the phase and phase-to-phase voltages.

Fig. 7.1: Phase and Phase-to-Phase Voltages without 3rd Harmonic Injection

In Figure 7.2 the 3rd harmonic injection is enabled. Note the characteristic flattening of the phase voltages, but note also that the phase-to-phase voltage is still sinusoidal. Note again the values of the phase and phase-phase voltages.

Fig. 7.2: Phase and Phase-to-Phase Voltages with 3rd Harmonic Injection

Figures 7.1 and 7.2 were captured by placing a 100K Ω and 0.1 μ F RC low-pass filter from a phase output to bus – on two of the phases and using the oscilloscope’s averaging function with 128 samples. This filtering is necessary because the phase signals are actually PWM square waves swinging the full bus voltage of about 100V. Even with the filtering and

averaging, the waveforms are a little fuzzy. The phase-to-phase voltage was obtained using the oscilloscope's math function.

It is normal for the phase-phase voltages to vary slightly as the motor runs. This is due to heating in the windings and changes in the shaft seal and bearing friction. In this case the phase-to-phase voltage was 55.6V for the first pass and 53.2V for the second. Recall that it is the current we are controlling. In this test, the command was 200 out of 32768 or about 0.006 of the full-scale current which is 32A. Therefore we were commanding about 125mA.

What we are interested in is the reduction of phase voltage from 32 to 28, but to do this we first need to correct either of the phase voltages by the ratio of phase-to-phase voltages as shown in equation 7.1.

$$\frac{55.6}{53.2} \cdot 28 = 29.26 \quad 7.1$$

Now the reduction in phase voltage is calculated as

$$\frac{32 - 29.26}{32} \cdot 100\% = 8.5\% \quad 7.2$$

Therefore the increase in phase-phase voltage is

$$\frac{32}{29.26} \cdot 100\% = 109.4\% \quad 7.3$$

This is not as good as the 15% we expected as found from

$$\frac{1}{\frac{\sqrt{3}}{2}} \cdot 100\% = 115.4\% \quad 7.4$$

This might be measurement error from the fuzzy ripple in the signals, so we collect the data another way.

In Figure 7.3, we collect the data directly from the DSP. The *R*-phase PWM commands with and without 3rd harmonic injection are captured in a data memory queue and then dumped via the debugger. The data is then plotted using Excel.

We note at once that the 3rd harmonic is more pronounced than it should be, and more than is shown in the oscilloscope prints. As for the latter, the filtering and averaging functions are probably (as expected) rolling-off more of the 3rd harmonic than the fundamental. For the former, the numerical constants were double-checked and found to be accurate.

Fig. 7.3: Direct Measurement of R phase w/ and w/o 3rd Harmonic Injection

Never-the-less, we recognize that there are significant rounding errors. Recall that multiplication of 16-bit numbers yields a 32-bit result. The 3rd harmonic is multiplied by $1 - \frac{\sqrt{3}}{2}$, then by the magnitude of the current vector, and then by the PWM limit of 640 (0x280.)

We conclude then that the flattening of the oscilloscope waveform is from added roll-off the filter and averaging of the 3rd harmonic compared with the fundamental.

The actual improvement in range, however, is evident. Repeating calculation 7.3, we see that the improvement is

$$\frac{873 - 640}{846 - 640} \cdot 100\% = 113\% \quad 7.5$$

where 640 (0x280) is the bi-polar to unipolar offset.

Besides the rounding error, recall also that $1 - \frac{\sqrt{3}}{2}$ was not a precise re-creation of the neutral point voltage.

In conclusion, we aver that the effect of 3rd harmonic injection is evident.

7.2: Experimental Proof of Closed-Loop Magnitude Control

Figure 7.4 shows the magnitude error. The data was collected as in 7.3.

Fig. 7.4: Magnitude Error

The magnitude error is shown as a percentage of the command. The error is well under 1% worse-case, and the mean error is about 0.3%.

7.3: Experimental Proof of Closed-Loop Phase Control

The angle error was again collected by logging the D and Q axis values. The error was accurately computed as,

$$\varepsilon = \tan^{-1} \frac{D}{Q} \quad 7.6$$

In Figure 7.5, the phase loop is disabled. The error is displayed in degrees.

Fig. 7.5: Open-Loop Phase Error

The worse case error is nearly 50° and the mean is 34° .

Figure 7.6 shows the closed phase-loop error.

Fig. 7.6: Closed-Loop Phase Error

The worse-case error is now less than 15° and the mean error is less than 1° !

Chapter 8: Conclusions

The abstract states that our goal was to produce a field-oriented controller implemented in polar coordinates, and with 3rd harmonic injection. The introduction states that we were motivated to learn why FOC is, almost invariably, taught and implemented in rectangular coordinates. Both FOC and 3rd harmonic injection are well documented, but we believe the combination to be unique, and discussion of polar FOC is rare at best.

We have successfully implemented a polar FOC with 3rd harmonic injection, albeit something of a hybrid. Each of the principle features has been experimentally proven:

- 3rd harmonic injection has been shown to increase the dynamic range of the phase voltage.
- Closed-loop magnitude control has been proven to render a near-zero error.
- Closed-loop phase control has proven to greatly reduce phase error.

Most of all, we have discovered why FOC in rectangular components is universally taught and used. We have discovered and discussed the difficulties with obtaining a usable angle from the current measurements and it is clear that polar FOC requires more calculations than its rectangular counterpart. In total, there are 5 sine/cosine calculations in this implementation as compared to just 2 for the rectangular form.

The project is a “successful failure” in that we succeeded in all our goals but failed to improve upon existing rectangular FOC.

Epilog: Is there a future for polar FOC?

There probably is not a future for polar FOC as presented. However, there are possibilities for some variants.

Techniques used in place of 3rd harmonic injection in rectangular systems could be applied to the polar form, thus eliminating one trig operation.

We could define a coordinate system with axes that are 120° , rather than 90° , apart. Presumably this would eliminate the Clarke transform. That would mean the components would no longer be decoupled; but the terminal voltage vectors already aren't, as discussed.

Since we have shown that the non-linearities in the errors don't much matter for a null-seeking control, we might use the error squared to eliminate the square root.

And finally, we might use arcsine and arccosine techniques to eliminate the division associated with the arctangent.

These are just a few of the blue-sky thoughts that occur to us in our on-going effort to build a better mousetrap.

Martin Todd Hess

References

- [1] Chapman, Stephen J. *Electric Machinery Fundamentals*. 3rd Ed. Boston: McGraw-Hill, 1993.
- [2] Serway, Raymond A. *Physics for Scientists & Engineers*. Philadelphia: Saunder's College Publishing, 1982.
- [3] Fitzgerald, A.E. *Electric Machinery*. 5th Ed. Boston: McGraw-Hill, 1993.
- [4] Boldea, I. and Nasar, S. A. *Vector Control of AC Drives*. Boca Raton: CRC Press, 1992.
- [5] Trzynadlowski, Andrzej M. *Control of Induction Motors*. San Diego: Academic Press, 2001
- [6] Martín, H., R. Bargalló. *Voltage Waveform Comparison for Different PWM Modulation Strategies*. Barcelona: Escola Universitària d'Enginyeria Tècnica Industrial de Barcelona, Universitat Politècnica de Catalunya Compte d'Urgell
- [7] Fairchild Semiconductor. *Application Note 42037: ML4423 Application Guidelines*. Fairchild Semiconductor, April 1998.
- [8] Texas Instruments. *Application Report SPRA588: Implementation of a Speed Field Oriented Control of 3-Phase PMSM Motor using TMS320F240*. Texas Instruments, Rev 1.0 25 October 2000.
- [9] Ross, Dave, et al. *AN908 Using the dsPIC30F for Vector Control of an ACIM*. Microchip Technology, 2004.
- [10] Lepka, Jaroslav, Petr Stekl. *AN1930 3-Phase AC Induction Motor Vector control Using a 56F80x, 56F8100, or 56F8300 Device*. Freescale Semiconductor, Rev. 2 2/2005.
- [11] Dorf, Richard C., Robert H. Bishop. *Modern Control Systems*, 9th Ed. Upper Saddle River: Prentice Hall, 2001.
- [12] Bose, Bimal K. *Modern Power Electronics and AC Drives*. Upper Saddle River: Prentice Hall, 2002.
- [13] *Square Root*. http://en.wikipedia.org/wiki/Square_root
- [14] The Applications Engineering Staff of Analog Devices, DSP Division. *Digital Signal Processing Applications Using the ADSP-2100 Family*. Ed. Amy Mar. 2 vols. Englewood Cliffs: Prentice Hall, 1992.

- [15] *ADMC-300 High Performance DSP-Based Motor Controller*. Norwood: Microchip Technologies, 2000.
- [16] *Hollow Shaft Encoder OIH-35*. Leatherhead: Encoder Technology LTD.
- [17] *Current Transducer LPS 25-NP/SP10*. LEM USA.

Appendix A: Source Listing

Release 6.0 Assembler Version 2.0 11/7/97

C:\My Documents\thesis\code\adca_isr.dsp Wed Jan 18 17:44:21 2012

Page 1

addr inst source line

```
1  !*****
2  !
3  ! This is the ADC Bank A Interrupt Service routine.
4  ! It contains the entire closed control loop
5  ! The phase currents read, converted through Clarke and
6  ! Park transforms, fed through PI loops, and reconverted
7  ! to 3-phase. 3rd harmonic injection is added and the
8  ! signals are fed to the PWM hardware.
9  !
10 ! Also includes twang routine for initialising the EIU
11 ! and an ADC calibration routine.
12 !
13 ! Copyright 2012 MARTIN TODD HESS. All rights reserved
14 !
15 !*****
16
17 .module adca_isr;
18
19 #define ADMC300
20 #include <ADMC300.h>
20 #ifndef ADMC300_INCLUDE
20 #define ADMC300_INCLUDE
20
20 {
20     This include file defines important ADMC300 addresses. The names
20     defined below can be used in user programs by "including" this file.
```

```

20     This file defines:
20         - names for the peripheral registers of the ADC300
20         - names for the memory mapped core registers of the ADC300
20         - interrupt vector table addresses
20     }
20
20 { peripheral registers of the ADC300 }
20
20 .CONST PWMTM      = 0x2008;    {PWM timer register}
20 .CONST PWMDT      = 0x2009;    {PWM dead time register}
20 .CONST PWMPD      = 0x200A;    {PWM pulse deletion}
20 .CONST PWMGATE    = 0x200B;    {PWM gate register}
20 .CONST PWMCHA     = 0x200C;    {PWM channel A register}
20 .CONST PWMCHB     = 0x200D;    {PWM channel B register}
20 .CONST PWMCHC     = 0x200E;    {PWM channel C register}
20 .CONST PWMSEG     = 0x200F;    {PWM segment selection}
20 .CONST AUXTIM0    = 0x2010;    {Auxiliary PWM timer 0}
20 .CONST AUXTIM1    = 0x2011;    {Auxiliary PWM timer 1}
20 .CONST MODECTRL   = 0x2015;    {MODE control register}
20 .CONST SYSSTAT    = 0x2016;    {System status register}
20 .CONST WDTIMER    = 0x2018;    {Watchdog timer register}
20 .CONST PICVECTOR  = 0x201C;    {ISR address          }
20 .CONST PICMASK    = 0x201D;    {IRD mask regsiter     }
20 .CONST EIUCNT     = 0x2020;    {Encoder count register }
20 .CONST EIUMAXCNT  = 0x2021;    {Encoder max count register }
20 .CONST EIUSTAT    = 0x2022;    {Encoder status register  }
20 .CONST EIUCTRL    = 0x2023;    {Encoder control register  }
20 .CONST EIUPERIOD  = 0x2024;    {Encoder loop timer period register}
20 .CONST EIUSCALE   = 0x2025;    {Encoder loop timer scale register}
20 .CONST EIUTIMER   = 0x2026;    {Encoder loop timer}
20 .CONST EETCNT     = 0x2027;    {Latched value of EIUCNT register}
20 .CONST EETN       = 0x2028;    {EET pulse decimator register }
20 .CONST EETDIV     = 0x2029;    {EET timer decimator register }
20 .CONST EETDELTAT  = 0x202A;    {EET delta timer register   }
20 .CONST EETT       = 0x202B;    {EET timer period register   }

```

addr inst source line

```

20 .CONST EETSTAT = 0x202C; {EET overflow status register }
20 .CONST ADC1 = 0x2030; {ADC1 register }
20 .CONST ADC2 = 0x2031; {ADC2 register }
20 .CONST ADC3 = 0x2032; {ADC3 register }
20 .CONST ADC4 = 0x2033; {ADC4 register }
20 .CONST ADC5 = 0x2034; {ADC5 register }
20 .CONST ADCCTRL = 0x2036; {ADC control register }
20 .CONST ADCSTAT = 0x2037; {ADC busy status register }
20 .CONST ADCSYNC = 0x2038; {ADC synchronization register }
20 .CONST ADCDIVA = 0x2039; {ADC bank A clock divide register }
20 .CONST ADCDIVB = 0x203A; {ADC bank B clock divide register }
20 .CONST ADCCAL = 0x203B; {ADC calibration register }
20 .CONST PIOLEVEL = 0x2040; {PIO level control register }
20 .CONST PIOMODE = 0x2041; {PIO mode control register }
20 .CONST PIOPWM = 0x2042; {PIO PWMTRIP control register }
20 .CONST PIODIR = 0x2044; {PIO direction register }
20 .CONST PIODATA = 0x2045; {PIO data register}
20 .CONST PIOINTEN = 0x2046; {PIO interrupt enable register }
20 .CONST PIOFLAG = 0x2047; {PIO interupt flag register }
20 .CONST ETUA0 = 0x2050; {ETU0 Event A capture register }
20 .CONST ETUB0 = 0x2051; {ETU0 Event B capture register }
20 .CONST ETUAA0 = 0x2052; {ETU0 Event AA capture register }
20 .CONST ETUA1 = 0x2053; {ETU1 Event A capture regsiter}
20 .CONST ETUB1 = 0x2054; {ETU1 Event B capture register}
20 .CONST ETUAA1 = 0x2055; {ETU1 Event AA capture register}
20 .CONST ETUTIME = 0x2056; {ETU timer value}
20 .CONST ETUCONFIG = 0x205C; {ETU configuration register }
20 .CONST ETUDIVIDE = 0x205D; {ETU clock divide register }
20 .CONST ETUSTAT = 0x205E; {ETU status register }
20 .CONST ETUCTRL = 0x205F; {ETU control register }
20 .CONST PWMSYNCWT = 0x2060; {PWMSYNC width register}
20 .CONST PWMSWT = 0x2061; {PWM software trip register}
20

```

```

20 { memory mapped core registers of the ADMC300 }
20
20 .CONST SYSCNTL    =    0x3fff;
20 .CONST MEMWAIT    =    0x3ffe;
20 .CONST TPERIOD     =    0x3ffd;
20 .CONST TCOUNT     =    0x3ffc;
20 .CONST TSCALE      =    0x3ffb;
20 .CONST Sport0_Rx_Words1    =    0x3ffa;
20 .CONST Sport0_Rx_Words0    =    0x3ff9;
20 .CONST Sport0_Tx_Words1    =    0x3ff8;
20 .CONST Sport0_Tx_Words0    =    0x3ff7;
20 .CONST Sport0_Ctrl_Reg     =    0x3ff6;
20 .CONST Sport0_Sclkdiv      =    0x3ff5;
20 .CONST Sport0_Rfsdiv       =    0x3ff4;
20 .CONST Sport0_Autobuf_Ctrl =    0x3ff3;
20 .CONST Sport1_Ctrl_Reg     =    0x3ff2;
20 .CONST Sport1_Sclkdiv      =    0x3ff1;
20 .CONST Sport1_Rfsdiv       =    0x3ff0;
20 .CONST Sport1_Autobuf_Ctrl =    0x3fef;
20
20 { interrupt vector table addresses for ADMC300 }
20
20 .CONST ADCBANKA_INT_ADDR    = 0x30+1;      {ADC BANKA INTERRUPT }
20 .CONST PWMSYNC_INT_ADDR    = 0x34+1;      {PWMSYNC interrupt  }
20 .CONST ADCBANKB_INT_ADDR    = 0x38+1;      {ADC BANKB interrupt }
20 .CONST PIO_INT_ADDR        = 0x3C+1;      {PIO4 - PIO11 interrupt }
20 .CONST EIU_INT_ADDR        = 0x40+1;      {EIU interrupt  }
20 .CONST ETU_INT_ADDR        = 0x44+1;      {ETU interrupt  }

```

addr inst source line

```

20 .CONST PIO0_INT_ADDR      = 0x48+1;      {PIO0 interrupt }
20 .CONST PIO1_INT_ADDR      = 0x4C+1;      {PIO1 interrupt }
20 .CONST PIO2_INT_ADDR      = 0x50+1;      {PIO2 interrupt }
20 .CONST PIO3_INT_ADDR      = 0x54+1;      {PIO3 interrupt }
20 .CONST PWMTRIP_INT_ADDR   = 0x58+1;      {PWMTRIP interrupt}
20 .CONST TX0_INT_ADDR       = 0x10; {SPORT0 transmit interrupt}
20 .CONST RX0_INT_ADDR       = 0x14; {SPORT0 receive interrupt}
20 .CONST SW1_INT_ADDR       = 0x18;      {software interrupt 1}
20 .CONST SW0_INT_ADDR       = 0x1C;      {software interrupt 0}
20 .CONST TX1_INT_ADDR       = 0x20; {SPORT1 transmit interrupt}
20 .CONST RX1_INT_ADDR       = 0x24; {SPORT1 receive interrupt}
20 .CONST TIMER_INT_ADDR     = 0x28; {Timer interrupt}
20
20 #endif
21 #include <romutil.h>
21 {This include file contains the addresses for the ROM utility routines.
21  To use just include this file in the module where the routine will be called.
21  Then just call the routine name given below.
21
21 VERSION # 2.0      07/10/98
21
21 Name          Date      Description
21 ----          -
21 MOD/HISTORY:   TFH      07/10/98      Combined all targets into one file
21 }
21 #ifndef ROMUTIL_INCLUDE
21 #define ROMUTIL_INCLUDE
21
21 {-----ADMC300-----}
21 #ifdef ADMC300
21 {ROM Utility Start Locations}
21 .CONST PER_RST      = 0x07E4;
21 .CONST UMASK        = 0x0E21;

```



```

21  .CONST PUT_VECTOR= 0x0E28;
21  .CONST SMASK      = 0x0E35;
21  .CONST ADMC_COS   = 0x0E55;
21  .CONST ADMC_SIN   = 0x0E5C;
21  .CONST ARCTAN     = 0x0E72;
21  .CONST RECIP_NEW  = 0x0E94; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
21  .CONST SQRT       = 0x0EAA;
21  .CONST LOG        = 0x0EE7;
21  .CONST LN         = 0x0EE4;
21  .CONST FLTONE     = 0x0F03;
21  .CONST FIXONE     = 0x0F08;
21  .CONST FPA        = 0x0F0C;
21  .CONST FPS        = 0x0F1B;
21  .CONST FPM        = 0x0F2B;
21  .CONST FPD        = 0x0F34;
21  .CONST FPMACC     = 0x0F55;
21  .CONST REV_FOR_PARK = 0x0F77;
21  .CONST REV_CLARK   = 0x0F8B;
21  .CONST FOR_CLARK   = 0x0FA1;
21  .CONST SDIVQINT    = 0x0FAB;
21  .CONST SDIVQ       = 0x0FB4;
21  .CONST EXIT       = 0x0FC6;
21
21  {Monitor Variables}
21  .CONST I4_SAVE     = 0x3bbf;
21  #endif
21
21  {-----ADMC330-----}

```

addr inst source line

```

21  #ifdef ADMC330
21  {ROM Utility Start Locations}
21  .CONST  PUT_VECTOR      = 0x0B73;
21  .CONST  SIN12B          = 0x0DC4;
21  .CONST  COS12B          = 0x0DD3;
21  .CONST  CAL_INV_TAN     = 0x0E61;
21  .CONST  LOG1            = 0x0E14;
21  .CONST  ALOG1          = 0x0E20;
21  .CONST  SQRT1           = 0x0E27;
21  .CONST  DIVIDE          = 0x0DE2;
21  .CONST  X_Y_TO_ANG      = 0x0E35;
21  .CONST  APPROX10_ST     = 0x0DEC;
21  .CONST  APPROX_N_ST     = 0x0DFF;
21  .CONST  COS64           = 0x0B80;
21  .CONST  COS6_64        = 0x0C00;
21  .CONST  SIN6_64        = 0x0C40;
21  .CONST  ATN64           = 0x0D01;
21  .CONST  LOG64           = 0x0D42;
21  .CONST  ALOG64         = 0x0D83;
21
21  {Monitor Variables}
21  .CONST  HALT_FLAG       = 0x3B9A;    {used by exit library function}
21  .CONST  IDE_SP          = 0x79E;    {used by exit library function}
21  .CONST  IRQFLAG_SAVE    = 0x3BFE;    {IRQFLAG saved here each interrupt}
21  #endif
21
21  {-----ADMC331-----}
21  #ifdef ADMC331
21  {ROM Utility Start Locations}
21  .CONST  PER_RST        = 0x07F1;
21  .CONST  UMASK          = 0x0DED;
21  .CONST  PUT_VECTOR     = 0x0DF4;
21  .CONST  SMASK          = 0x0E06;

```

```

21  .CONST      ADMC_COS      = 0X0E26;
21  .CONST      ADMC_SIN      = 0X0E2D;
21  .CONST      ARCTAN        = 0X0E43;
21  .CONST      RECIP_NEW     = 0X0E65;  {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
21  .CONST      SQRT_NEW      = 0X0E7C;  {For SQRT - call SQRT in mathfix.dsp}
21  .CONST      LN_NEW        = 0X0EBA;  {For LN - call LN in mathfix.dsp}
21  .CONST      LOG_NEW       = 0X0EB9;  {For LOG - call LOG in mathfix.dsp}
21  .CONST      FLTONE        = 0X0ED4;
21  .CONST      FIXONE        = 0X0ED9;
21  .CONST      FPA           = 0X0EDD;
21  .CONST      FPS           = 0X0EEC;
21  .CONST      FPM           = 0X0EFC;
21  .CONST      FPD           = 0X0F05;
21  .CONST      FPMACC        = 0X0F26;
21  .CONST      PARK         = 0X0F48;
21  .CONST      REV_CLARK     = 0X0F5C;
21  .CONST      FOR_CLARK     = 0X0F72;
21  .CONST      COS64         = 0X0F80;
21  .CONST      ONE_BY_X      = 0X0FC0;
21  .CONST      SDIVQINT      = 0X0FD0;
21  .CONST      SDIVQ         = 0X0FD9;
21
21  {Monitor Variables}
21  .CONST HALT_FLAG = 0x3B68;  {used by exit library function}
21  .CONST IDE_SP     = 0x7F7;   {used by exit library function}
21  .CONST IRQFLAG_SAVE = 0x3B88; {IRQFLAG saved here each interrupt}
21  #endif
21

```

addr inst source line

```

21  {-----ADMC328 Emulation Board-----}
21  #ifdef ADMC328E
21  {ROM Utility Start Locations}
21  .CONST      PER_RST      = 0X07F1;
21  .CONST      UMASK = 0X0DED;
21  .CONST      PUT_VECTOR = 0X0DF4;
21  .CONST      SMASK = 0X0E06;
21  .CONST      ADMC_COS      = 0X0E26;
21  .CONST      ADMC_SIN      = 0X0E2D;
21  .CONST      ARCTAN        = 0X0E43;
21  .CONST      RECIP_NEW      = 0X0E65; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
21  .CONST      SQRT_NEW       = 0X0E7C; {For SQRT - call SQRT in mathfix.dsp}
21  .CONST      LN_NEW         = 0X0EBA; {For LN - call LN in mathfix.dsp}
21  .CONST      LOG_NEW        = 0X0EB9; {For LOG - call LOG in mathfix.dsp}
21  .CONST      FLTONE         = 0X0ED4;
21  .CONST      FIXONE         = 0X0ED9;
21  .CONST      FPA            = 0X0EDD;
21  .CONST      FPS            = 0X0EEC;
21  .CONST      FPM            = 0X0EFC;
21  .CONST      FPD            = 0X0F05;
21  .CONST      FPMACC         = 0X0F26;
21  .CONST      PARK           = 0X0F48;
21  .CONST      REV_CLARK       = 0X0F5C;
21  .CONST      FOR_CLARK       = 0X0F72;
21  .CONST      COS64 = 0X0F80;
21  .CONST      ONE_BY_X        = 0X0FC0;
21  .CONST      SDIVQINT        = 0X0FD0;
21  .CONST      SDIVQ = 0X0FD9;
21
21  {Monitor Variables}
21  .CONST HALT_FLAG = 0x3B68; {used by exit library function}
21  .CONST IDE_SP      = 0x7F7; {used by exit library function}
21  .CONST IRQFLAG_SAVE = 0x3B88; {IRQFLAG saved here each interrupt}

```

```

21 #endif
21
21 {-----ADMC328-----}
21 #ifdef ADMC328
21 {ROM Utility Start Locations}
21 .CONST    PER_RST      = 0X01F1;
21 .CONST    UMASK = 0X0F15;
21 .CONST    PUT_VECTOR   = 0X0F1C;
21 .CONST    SMASK = 0X0F29;
21 .CONST    SDIVQINT     = 0X0F30;
21 .CONST    SDIVQ = 0X0F39;
21
21 {Monitor Variables}
21 .CONST IRQFLAG_SAVE     = 0x3988; {IRQFLAG saved here each interrupt}
21 #endif
21
21 {-----ADMC201-----}
21 #ifdef ADMC201
21 {ROM Utility Start Locations}
21 .CONST    PUT_VECTOR   = 0x0573;
21 .CONST    COS64        = 0x0580;
21 .CONST    ONE_BY_X      = 0x05C0;
21
21 {Monitor Variables}
21 .CONST IRQFLAG_SAVE     = 0x3BFE; {IRQFLAG saved here each interrupt}
21 #endif
21
21 #endif

```

addr inst source line

```
21
21
21
22 #include <adca_isr.h>
22 .var ang_cos; .global ang_cos;      !Cosine of flux angle
22 .var ang_sin; .global ang_sin;      !Sine of flux angle
22 .var ang_comp; !Compensated angle error in rotating frame
22 .var ang_error; !Reference current angle - measured current angle
22 .var ang_flux; .global ang_flux;    !Rotor flux angle
22 .var ang_volt; .global ang_volt;    !Terminl voltage angle
22
22
22 .var cur_error; !Reference current - measured current magnitude
22 .var cur_mag; .global cur_mag;      !Magnitude of measured current vector
22 .var cur_r; .global cur_r; !R phase measured current
22 .var cur_r_offset;
22 .init cur_r_offset: 0;
22 .var cur_ref; .global cur_ref;      !Magnitude of reference (command) current
22 .init cur_ref: 0;
22 .var cur_t; .global cur_t; !T phase measred current
22 .var cur_t_offset;
22 .init cur_t_offset: 0;
22 .var cur_x; !X current component in stationary frame
22 .var cur_y; !Y current component in stationary frame
22
22 .var foc_integrators[4]; !PI compensator integrators
22 .global foc_integrators;
22 .var foc_coef[3]; .global foc_coef; !PI compensator coeffecients
22 .init foc_coef: 0x200, 0x2000, 0x4;
22
22 .var twang_loops;
22 .global twang_loops;
22
```

```

22 .var calibrate_loops;
22 .global calibrate_loops;
22
22 .var volt_mag; .global volt_mag; !Magnitude of terminal voltage
22 .var volt_r; .global volt_r; !R phase terminal voltage
22 .var volt_s; .global volt_s; !S phase terminal voltage
22 .var volt_t; .global volt_t; !T phase terminal voltage
22
22 .var plot_count;
22 .init plot_count: 0;
22 .global plot_count;
22 .var sample_count;
22
23 #include <defines.h>
23 #define pwmtm_default 0x500
23 #define magnitude_scale 0x280 /* sqrt(3)/3 * pwmtm_default / 2 */
23 #define third_harmonic_scale 0x55 /* magnitude_scale / 3 */
23 #define encoder_roll 3999
23 #define pole_pairs 3
23 #define encoder_scale 49 /* 65536 / encoder_roll * pole_pairs */
23 #define twang_angle 1000
24
25 .entry adca_int;
26
27 ADCA_INT:
0000 0C0030      28  ena sec_reg;
      29
0001 80000Au     30  ar = dm(calibrate_loops); !Perform ADC cal if not already done

```

addr inst source line

```

0002 2A7AAA      31  none = pass ar;
0003 180001u     32  if ne jump calibrate_adc;
      33
      34
      35  !*** Clarke ***
      36
0004 820300      37  ax0 = dm(adcl); !R phase measured current
0005 800004u     38  ay0 = dm(cur_r_offset);
0006 22E00F      39  ar = ax0 - ay0;
0007 233A0F      40  ar = -ar;
0008 0D002A      41  mx0 = ar;
      42
0009 900002u     43  dm(cur_r) = mx0;
000A 900002u     44  dm(cur_x) = mx0; !Save x-axis measured current
      45
000B 449E76      46  my0 = 18919;      !Normalised sqrt(3)/3
000C 20800F      47  mr = mx0 * my0 (ss);
000D 820310      48  ax0 = dm(adcl);
000E 800004u     49  ay0 = dm(cur_t_offset);
000F 22E00F      50  ar = ax0 - ay0;
0010 233A0F      51  ar = -ar;
      52
0011 90000Au     53  dm(cur_t) = ar;
0012 21020F      54  mr = mr + ar * my0 (ss);
0013 21020F      55  mr = mr + ar * my0 (ss);
0014 90000Cu     56  dm(cur_y) = mrl; !save y-axis measured current
0015 0D003C      57  mx1 = mrl;
      58
      59
      60  !***Compute Magnitude and angle error (Q & D) ***
      61
0016 800006u     62  my0 = dm(ang_cos);
0017 800007u     63  my1 = dm(ang_sin);

```



```

64
65
66  !*** Compute Magnitude ***
67
0018 20810F      68  mr = mx1 * my0 (ss);
0019 21880F      69  mr = mr - mx0 * my1 (ss);
001A 90000Cu     70  dm(cur_mag) = mrl;
71
72
73  !*** Compute Angle Error ***
74
001B 23E40F      75  ar = abs mrl;    !Only used to set CCR
76
001C 20800F      77  mr = mx0 * my0 (ss);
001D 21090F      78  mr = mr + mx1 * my1 (ss);
001E 0D00AC      79  ar = mrl;
001F 233A0A      80  if neg ar = -ar;
0020 90000Au     81  dm(ang_error) = ar;
82
83
84  ! *** Setup PI compensators ***
85
0021 800000u     86  ax0 = dm(foc_coef + 2);    !Power-of-two master gain
0022 22E81F      87  ar = ax0 - 16;
0023 0D009A      88  se = ar;
89
90
91  !*** Magnitude Error ***

```

addr inst source line

```

          92
0024 80000Au      93  ar = dm(cur_ref);      !Reference (commanded) current
          94
0025 800000u      95  ax0 = dm(twang_loops);
0026 2A78AA      96  none = pass ax0;
0027 1C0001u      97  if ne call twang;
          98
0028 800004u      99  ay0 = dm(cur_mag);
0029 22E20F     100  ar = ar - ay0;
002A 90000Au     101  dm(cur_error) = ar;
          102
002B 380000u     103  i4  = ^foc_integrators;
002C 380004     104  m4  = 0;
002D 380015     105  m5  = 1;
002E 3BFFF6     106  m6  = -1;
002F 800006u     107  my0 = dm(foc_coef);    !I gain
0030 800007u     108  my1 = dm(foc_coef + 1);    !P gain
          109
          110
          111  !*** Magnitude PI ***
          112
0031 7000B1     113  mr0 = dm(i4, m5);      !Last value of magnitude integrator
0032 7000C2     114  mr1 = dm(i4, m6);
0033 21020F     115  mr  = mr + ar * my0 (ss);    !ar still has magnitude error, my0 = I
0034 050000     116  if mv sat mr;    !Check for overflow
0035 7800B1     117  dm(i4, m5) = mr0;      !Next value of magnitude integrator
0036 7800C1     118  dm(i4, m5) = mr1;
          119
0037 210A0F     120  mr = mr + ar * my1 (ss);    !ar still has magnitude error, my1 = P
0038 050000     121  if mv sat mr;    !Check for overflow
          122
0039 0E240F     123  sr = ashift mr1 (hi); !Scale result
003A 0E1B0F     124  sr = sr or lshift mr0 (lo);

```

```

003B 0D00CE      125  mr1 = sr0;
003C 0D00DF      126  mr2 = srl;
003D 21180F      127  mr = mr;    !Check for overflow
003E 050000      128  if mv sat mr;
003F 90000Cu      129  dm(volt_mag) = mr1;
      130
      131
      132  !*** Angle PI ***
      133
0040 80000Au      134  ar = dm(ang_error);    !ar = angle error
0041 7000B1      135  mr0 = dm(i4, m5); !Last value of angle integrator
0042 7000C2      136  mr1 = dm(i4, m6);
0043 21020F      137  mr = mr + ar * my0 (ss);    !my0 = I
0044 050000      138  if mv sat mr;    !Check for overflow
0045 7800B1      139  dm(i4, m5) = mr0; !Next value of angle integrator
0046 7800C1      140  dm(i4, m5) = mr1;
      141
0047 210A0F      142  mr = mr + ar * my1 (ss);    !ar still has angle error, my1 = P
0048 050000      143  if mv sat mr;    !Check for overflow
      144
0049 0E240F      145  sr = ashift mr1 (hi); !Scale result
004A 0E1B0F      146  sr = sr or lshift mr0 (lo);
004B 0D00CE      147  mr1 = sr0;
004C 0D00DF      148  mr2 = srl;
004D 21180F      149  mr = mr;    !Check for overflow
004E 050000      150  if mv sat mr;
004F 0F24FF      151  sr = ashift mr1 by -1 (hi);
0050 90000Fu      152  dm(ang_comp) = srl;    !Compensated angle error in rotating frame

```

addr inst source line

```

153
154
155  !*** Polar Park ***
156
0051 800004u      157  ay0 = dm(ang_flux);      !Rotor flux angle
0052 22670F      158  ar = srl + ay0;
0053 90000Au      159  dm(ang_volt) = ar;      !Angle of terminal voltage
160
161
162  !*** Scale the magnitude ***
163
0054 800002u      164  mx0 = dm(volt_mag);      !Normalised magnitude
0055 402806      165  my0 = magnitude_scale;      !sqrt(3)/3 = max space vector * max PWM
0056 20800F      166  mr  = mx0 * my0 (ss);
167
0057 0D006C      168  my0 = mrl; !my0 preserved durring sine function
169
170
171  !*** Phase R bi-polar Magnitude ***
172
0058 0D000A      173  ax0 = ar; !ax0 is passed to function, and is preserved
0059 1CE5CF      174  call ADMC_SIN;
005A 20820F      175  mr  = ar * my0 (ss); !adjusted * sin (ang_volt)
005B 90000Cu      176  dm(volt_r) = mrl;      !Save R for monitoring
177
178
179  !*** Phase S bi-polar Magnitude ***
180
005C 455554      181  ay0 = 0x5555;      !120 degrees (65536 = 360 degrees)
005D 22600F      182  ar  = ax0 + ay0; !theta + 120
005E 0D000A      183  ax0 = ar;
005F 1CE5CF      184  call ADMC_SIN;
0060 20220F      185  mr  = ar * my0 (rnd); !M * cos (ang_volt + 120)

```

```

0061 90000Cu      186  dm(volt_s) = mrl;      !Save S^ for monitoring
      187
      188
      189  !*** Phase T bi-polar Magnitude ***
      190
0062 233C0F      191  ar  = -mrl;
0063 800004u      192  ay0 = dm(volt_r);
0064 22E20F      193  ar  = ar - ay0; !-(R^ + S^)
0065 90000Au      194  dm(volt_t) = ar; !Save T^ for monitoring
      195
      196
      197  !*** Compute 3-rd Harmanic to be added ***
      198
0066 800000u      199  ax0 = dm(ang_volt);    !Compute ang_volt * 3
0067 0D0040      200  ay0 = ax0;
0068 22600F      201  ar  = ax0 + ay0;
0069 22620F      202  ar  = ar + ay0;
006A 0D000A      203  ax0 = ar;
006B 1CE5CF      204  call ADMC_SIN;    !ar = 3-rd harmonic angle
      205
006C 400556      206  my0 = third_harmonic_scale;
006D 20820F      207  mr  = ar * my0 (ss); !negative normalised magnitude of the wobble
006E 800006u      208  my0 = dm(volt_mag);    !Then scale by the magnitude of the vector
006F 20840F      209  mr  = mrl * my0 (ss);
0070 402804      210  ay0 = magnitude_scale;    !50% offset for bi-polar -> unipolar
0071 26640F      211  af  = mrl + ay0;
      212
      213

```

addr inst source line

```

    214  !*** Adust phase voltages and output them ***
    215
0072 80000Au      216  ar = dm(volt_r);      !Bipolar R^
0073 22720F      217  ar = ar + af;    !Add 3rd harmonic + unipolar offset
0074 9200CA      218  dm(PWMCHA) = ar; !R Duty cycle register
    219
0075 80000Au      220  ar = dm(volt_s);      !Bipolar S^
0076 22720F      221  ar = ar + af;    !Add 3rd harmonic + unipolar offset
0077 9200DA      222  dm(PWMCHB) = ar; !S Duty cycle register
    223
0078 80000Au      224  ar = dm(volt_t);      !Bipolar T^
0079 22720F      225  ar = ar + af;    !Add 3rd harmonic + unipolar offset
007A 9200EA      226  dm(PWMCHC) = ar; !T Duty cycle register
    227
007B 80000Au      228  ar = dm(volt_r);
007C 22620F      229  ar = ar + ay0;
007D 0D00CA      230  mrl = ar;
    231
    232
    233  !*** Data logging routine ***
    234
007E 80000Au      235  ar = dm(sample_count);
007F 22621F      236  ar = ar + 1;
0080 90000Au      237  dm(sample_count) = ar;
0081 4000F4      238  ay0 = 0xf;
0082 23820F      239  ar = ar and ay0;
0083 180001u      240  if ne jump skip_plot;
    241
0084 80000Au      242  ar = dm(plot_count);
0085 2A7AAA      243  none = pass ar;
0086 180000u      244  if eq jump skip_plot;
0087 22E21F      245  ar = ar - 1;
0088 90000Au      246  dm(plot_count) = ar;

```

```

247
0089 8200CA      248  ar = dm(PWMCHA);
008A 6800A5      249  dm(i1, m1) = ar;
008B 8200DA      250  ar = dm(PWMCHB);
008C 6800A5      251  dm(i1, m1) = ar;
252
253  skip_plot:
008D 0C0020      254  dis sec_reg;
255
008E 0A001F      256  rti;
257
258  !*** Twang the motor for encoder setup ***
259
260  twang:
008F 4014D6      261  my0 = 333;
0090 22E01F      262  ar = ax0 - 1;
0091 90000Au      263  dm(twang_loops) = ar;
0092 20820F      264  mr = ar * my0 (ss);
0093 92020C      265  dm(EIUCNT) = mrl;
0094 403E8A      266  ar = 1000;
0095 221802      267  if gt ar = pass 0;
0096 000000      268  nop;
0097 000000      269  nop;
0098 000000      270  nop;
0099 000000      271  nop;
009A 0A000F      272  rts;
273
274

```

addr inst source line

```

    275  !*** Calibrate the ADC ***
    276
    277  calibrate_adc:
009B 400016      278  my0 = 1;
009C 380000u     279  i4 = ^foc_integrators;
009D 380004      280  m4 = 0;
009E 380015      281  m5 = 1;
009F 3BFFF6      282  m6 = -1;
    283
00A0 22E21F      284  ar = ar - 1;
00A1 90000Au     285  dm(calibrate_loops) = ar;
00A2 180000u     286  if eq jump end_calibrate;
    287
00A3 7000B1      288  mr0 = dm(i4, m5);
00A4 7000C2      289  mr1 = dm(i4, m6);
00A5 820302      290  mx0 = dm(ADC1);
00A6 21000F      291  mr = mr + mx0 * my0 (ss);
00A7 7800B1      292  dm(i4, m5) = mr0;
00A8 7800C1      293  dm(i4, m5) = mr1;
    294
00A9 7000B1      295  mr0 = dm(i4, m5);
00AA 7000C2      296  mr1 = dm(i4, m6);
00AB 820312      297  mx0 = dm(ADC2);
00AC 21000F      298  mr = mr + mx0 * my0 (ss);
00AD 7800B1      299  dm(i4, m5) = mr0;
00AE 7800C1      300  dm(i4, m5) = mr1;
    301
00AF 0C0020      302  dis sec_reg;
00B0 0A001F      303  rti;
    304
    305  end_calibrate:
00B1 400000      306  ax0 = 0;
    307
```



```

00B2 780001      308  dm(i4, m5) = ax0;
      309
00B3 7000A0      310  ar = dm(i4, m4);
00B4 780001      311  dm(i4, m5) = ax0;
00B5 90000Au     312  dm(cur_r_offset) = ar;
      313
00B6 780001      314  dm(i4, m5) = ax0;
      315
00B7 7000A0      316  ar = dm(i4, m4);
00B8 780001      317  dm(i4, m5) = ax0;
00B9 90000Au     318  dm(cur_t_offset) = ar;
      319
00BA 0C0020      320  dis sec_reg;
00BB 0A001F      321  rti;
      322
      323 .endmod;
      324

```

Release 6.0 Assembler Version 2.0 11/7/97

C:\My Documents\thesis\code\adcb_isr.dsp Wed Jan 18 17:44:21 2012

Page 1

addr inst source line

```

1
2 !*****
3 !
4 ! This is the ADC Bank B Interrupt Service routine.
5 ! This routine only monitors the bus voltage.
6 !
7 ! Copyright 2012 MARTIN TODD HESS. All rights reserved
8 !
9 !*****
10
11 .module adcb_isr;
12
13 #define ADMC300
14 #include <ADMC300.h>

```

```

14 #ifndef ADMC300_INCLUDE
14 #define ADMC300_INCLUDE
14
14 {
14     This include file defines important ADMC300 addresses. The names
14     defined below can be used in user programs by "including" this file.
14     This file defines:
14         - names for the peripheral registers of the ADMC300
14         - names for the memory mapped core registers of the ADMC300
14         - interrupt vector table addresses
14 }
14
14 { peripheral registers of the ADMC300 }
14
14 .CONST PWMTM      = 0x2008;    {PWM timer register}
14 .CONST PWMDT      = 0x2009;    {PWM dead time register}
14 .CONST PWMPD      = 0x200A;    {PWM pulse deletion}
14 .CONST PWMGATE     = 0x200B;    {PWM gate register}
14 .CONST PWMCHA      = 0x200C;    {PWM channel A register}
14 .CONST PWMCHB      = 0x200D;    {PWM channel B register}
14 .CONST PWMCHC      = 0x200E;    {PWM channel C register}
14 .CONST PWMSEG      = 0x200F;    {PWM segment selection}
14 .CONST AUXTIM0     = 0x2010;    {Auxiliary PWM timer 0}
14 .CONST AUXTIM1     = 0x2011;    {Auxiliary PWM timer 1}
14 .CONST MODECTRL    = 0x2015;    {MODE control register}
14 .CONST SYSSTAT     = 0x2016;    {System status register}
14 .CONST WDTIMER     = 0x2018;    {Watchdog timer register}
14 .CONST PICVECTOR   = 0x201C;    {ISR address          }
14 .CONST PICMASK     = 0x201D;    {IRD mask regsiter    }
14 .CONST EIUCNT      = 0x2020;    {Encoder count register }
14 .CONST EIUMAXCNT   = 0x2021;    {Encoder max count register }
14 .CONST EIUSTAT     = 0x2022;    {Encoder status register  }
14 .CONST EIUCTRL     = 0x2023;    {Encoder control register  }
14 .CONST EIUPERIOD   = 0x2024;    {Encoder loop timer period register}
14 .CONST EIUSCALE    = 0x2025;    {Encoder loop timer scale register}
14 .CONST EIUTIMER    = 0x2026;    {Encoder loop timer}
14 .CONST EETCNT      = 0x2027;    {Latched value of EIUCNT register}

```

```

14  .CONST  EETN      =    0x2028;  {EET pulse decimator register }
14  .CONST  EETDIV    =    0x2029;  {EET timer decimator register }
14  .CONST  EETDELTAT =    0x202A;  {EET delta timer register    }
14  .CONST  EETT      =    0x202B;  {EET timer period register   }
14  .CONST  EETSTAT   =    0x202C;  {EET overflow status register}
14  .CONST  ADC1      =    0x2030;  {ADC1 register               }
14  .CONST  ADC2      =    0x2031;  {ADC2 register               }
14  .CONST  ADC3      =    0x2032;  {ADC3 register               }
14  .CONST  ADC4      =    0x2033;  {ADC4 register               }
14  .CONST  ADC5      =    0x2034;  {ADC5 register               }

```

addr inst source line

```

14 .CONST ADCCTRL = 0x2036; {ADC control register }
14 .CONST ADCSTAT = 0x2037; {ADC busy status register }
14 .CONST ADCSYNC = 0x2038; {ADC synchronization register }
14 .CONST ADCDIVA = 0x2039; {ADC bank A clock divide register }
14 .CONST ADCDIVB = 0x203A; {ADC bank B clock divide register }
14 .CONST ADCCAL = 0x203B; {ADC calibration register }
14 .CONST PIOLEVEL = 0x2040; {PIO level control register }
14 .CONST PIOMODE = 0x2041; {PIO mode control register }
14 .CONST PIOPWM = 0x2042; {PIO PWMTRIP control register }
14 .CONST PIODIR = 0x2044; {PIO direction register }
14 .CONST PIODATA = 0x2045; {PIO data register}
14 .CONST PIOINTEN = 0x2046; {PIO interrupt enable register }
14 .CONST PIOFLAG = 0x2047; {PIO interrupt flag register }
14 .CONST ETUA0 = 0x2050; {ETU0 Event A capture register }
14 .CONST ETUB0 = 0x2051; {ETU0 Event B capture register }
14 .CONST ETUAA0 = 0x2052; {ETU0 Event AA capture register }
14 .CONST ETUA1 = 0x2053; {ETU1 Event A capture register}
14 .CONST ETUB1 = 0x2054; {ETU1 Event B capture register}
14 .CONST ETUAA1 = 0x2055; {ETU1 Event AA capture register}
14 .CONST ETUTIME = 0x2056; {ETU timer value}
14 .CONST ETUCONFIG = 0x205C; {ETU configuration register }
14 .CONST ETUDIVIDE = 0x205D; {ETU clock divide register }
14 .CONST ETUSTAT = 0x205E; {ETU status register }
14 .CONST ETUCTRL = 0x205F; {ETU control register }
14 .CONST PWMSYNCWT = 0x2060; {PWMSYNC width register}
14 .CONST PWMSWT = 0x2061; {PWM software trip register}
14
14 { memory mapped core registers of the ADMC300 }
14
14 .CONST SYSCNTL = 0x3fff;
14 .CONST MEMWAIT = 0x3ffe;
14 .CONST TPERIOD = 0x3ffd;
14 .CONST TCOUNT = 0x3ffc;

```

```

14  .CONST  TSCALE          =    0x3ffb;
14  .CONST  Sport0_Rx_Words1 =    0x3ffa;
14  .CONST  Sport0_Rx_Words0 =    0x3ff9;
14  .CONST  Sport0_Tx_Words1 =    0x3ff8;
14  .CONST  Sport0_Tx_Words0 =    0x3ff7;
14  .CONST  Sport0_Ctrl_Reg   =    0x3ff6;
14  .CONST  Sport0_Sclkdiv    =    0x3ff5;
14  .CONST  Sport0_Rfsdiv     =    0x3ff4;
14  .CONST  Sport0_Autobuf_Ctrl = 0x3ff3;
14  .CONST  Sport1_Ctrl_Reg   =    0x3ff2;
14  .CONST  Sport1_Sclkdiv    =    0x3ff1;
14  .CONST  Sport1_Rfsdiv     =    0x3ff0;
14  .CONST  Sport1_Autobuf_Ctrl = 0x3fef;
14
14  { interrupt vector table addresses for ADMC300 }
14
14  .CONST  ADCBANKA_INT_ADDR = 0x30+1;      {ADC BANKA INTERRUPT }
14  .CONST  PWMSYNC_INT_ADDR  = 0x34+1;      {PWMSYNC interrupt  }
14  .CONST  ADCBANKB_INT_ADDR = 0x38+1;      {ADC BANKB interrupt }
14  .CONST  PIO_INT_ADDR      = 0x3C+1;      {PIO4 - PIO11 interrupt }
14  .CONST  EIU_INT_ADDR      = 0x40+1;      {EIU interrupt  }
14  .CONST  ETU_INT_ADDR      = 0x44+1;      {ETU interrupt  }
14  .CONST  PIO0_INT_ADDR     = 0x48+1;      {PIO0 interrupt  }
14  .CONST  PIO1_INT_ADDR     = 0x4C+1;      {PIO1 interrupt  }
14  .CONST  PIO2_INT_ADDR     = 0x50+1;      {PIO2 interrupt  }
14  .CONST  PIO3_INT_ADDR     = 0x54+1;      {PIO3 interrupt  }
14  .CONST  PWMTRIP_INT_ADDR  = 0x58+1;      {PWMTRIP interrupt}
14  .CONST  TX0_INT_ADDR      = 0x10; {SPORT0 transmit interrupt}

```

addr inst source line

```

14  .CONST  RX0_INT_ADDR      = 0x14; {SPORT0 receive interrupt}
14  .CONST  SW1_INT_ADDR      = 0x18;      {software interrupt 1}
14  .CONST  SW0_INT_ADDR      = 0x1C;      {software interrupt 0}
14  .CONST  TX1_INT_ADDR      = 0x20; {SPORT1 transmit interrupt}
14  .CONST  RX1_INT_ADDR      = 0x24; {SPORT1 receive interrupt}
14  .CONST  TIMER_INT_ADDR    = 0x28; {Timer interrupt}
14
14  #endif
15  #include <romutil.h>
15  {This include file contains the addresses for the ROM utility routines.
15   To use just include this file in the module where the routine will be called.
15   Then just call the routine name given below.
15
15  VERSION # 2.0      07/10/98
15
15  Name          Date          Description
15  ----          -
15  MOD/HISTORY:   TFH      07/10/98      Combined all targets into one file
15  }
15  #ifndef ROMUTIL_INCLUDE
15  #define ROMUTIL_INCLUDE
15
15  {-----ADM300-----}
15  #ifdef ADM300
15  {ROM Utility Start Locations}
15  .CONST PER_RST      = 0x07E4;
15  .CONST UMASK        = 0x0E21;
15  .CONST PUT_VECTOR   = 0x0E28;
15  .CONST SMASK        = 0x0E35;
15  .CONST ADMC_COS     = 0x0E55;
15  .CONST ADMC_SIN     = 0x0E5C;
15  .CONST ARCTAN       = 0x0E72;
15  .CONST RECIP_NEW    = 0x0E94; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}

```

```

15  .CONST SQRT = 0x0EAA;
15  .CONST LOG  = 0x0EE7;
15  .CONST LN   = 0x0EE4;
15  .CONST FLTONE      = 0x0F03;
15  .CONST FIXONE      = 0x0F08;
15  .CONST FPA  = 0x0F0C;
15  .CONST FPS  = 0x0F1B;
15  .CONST FPM  = 0x0F2B;
15  .CONST FPD  = 0x0F34;
15  .CONST FPMACC      = 0x0F55;
15  .CONST REV_FOR_PARK      = 0x0F77;
15  .CONST REV_CLARK        = 0x0F8B;
15  .CONST FOR_CLARK        = 0x0FA1;
15  .CONST SDIVQINT  = 0x0FAB;
15  .CONST SDIVQ      = 0x0FB4;
15  .CONST EXIT = 0x0FC6;
15
15  {Monitor Variables}
15  .CONST I4_SAVE      = 0x3bbf;
15  #endif
15
15  {-----ADMC330-----}
15  #ifdef ADMC330
15  {ROM Utility Start Locations}
15  .CONST PUT_VECTOR      = 0x0B73;
15  .CONST SIN12B          = 0x0DC4;
15  .CONST COS12B          = 0x0DD3;
15  .CONST CAL_INV_TAN      = 0x0E61;

```

addr inst source line

```

15 .CONST LOG1          = 0x0E14;
15 .CONST ALOG1         = 0x0E20;
15 .CONST SQRT1         = 0x0E27;
15 .CONST DIVIDE        = 0x0DE2;
15 .CONST X_Y_TO_ANG    = 0x0E35;
15 .CONST APPROX10_ST   = 0x0DEC;
15 .CONST APPROX_N_ST   = 0x0DFF;
15 .CONST COS64         = 0x0B80;
15 .CONST COS6_64       = 0x0C00;
15 .CONST SIN6_64       = 0x0C40;
15 .CONST ATN64         = 0x0D01;
15 .CONST LOG64         = 0x0D42;
15 .CONST ALOG64        = 0x0D83;
15
15 {Monitor Variables}
15 .CONST HALT_FLAG      = 0x3B9A;    {used by exit library function}
15 .CONST IDE_SP         = 0x79E;     {used by exit library function}
15 .CONST IRQFLAG_SAVE   = 0x3BFE;    {IRQFLAG saved here each interrupt}
15 #endif
15
15 {-----ADMC331-----}
15 #ifdef ADMC331
15 {ROM Utility Start Locations}
15 .CONST PER_RST        = 0X07F1;
15 .CONST UMASK          = 0X0DED;
15 .CONST PUT_VECTOR     = 0X0DF4;
15 .CONST SMASK          = 0X0E06;
15 .CONST ADMC_COS       = 0X0E26;
15 .CONST ADMC_SIN       = 0X0E2D;
15 .CONST ARCTAN         = 0X0E43;
15 .CONST RECIP_NEW      = 0X0E65;    {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
15 .CONST SQRT_NEW       = 0X0E7C;    {For SQRT - call SQRT in mathfix.dsp}
15 .CONST LN_NEW         = 0X0EBA;    {For LN - call LN in mathfix.dsp}

```



```

15 .CONST      LOG_NEW      = 0X0EB9;  {For LOG - call LOG in mathfix.dsp}
15 .CONST      FLTONE       = 0X0ED4;
15 .CONST      FIXONE       = 0X0ED9;
15 .CONST      FPA          = 0X0EDD;
15 .CONST      FPS          = 0X0EEC;
15 .CONST      FPM          = 0X0EFC;
15 .CONST      FPD          = 0X0F05;
15 .CONST      FPMACC       = 0X0F26;
15 .CONST      PARK         = 0X0F48;
15 .CONST      REV_CLARK    = 0X0F5C;
15 .CONST      FOR_CLARK    = 0X0F72;
15 .CONST      COS64        = 0X0F80;
15 .CONST      ONE_BY_X     = 0X0FC0;
15 .CONST      SDIVQINT     = 0X0FD0;
15 .CONST      SDIVQ        = 0X0FD9;
15
15 {Monitor Variables}
15 .CONST HALT_FLAG = 0x3B68;  {used by exit library function}
15 .CONST IDE_SP    = 0x7F7;   {used by exit library function}
15 .CONST IRQFLAG_SAVE = 0x3B88; {IRQFLAG saved here each interrupt}
15 #endif
15
15 {-----ADMC328 Emulation Board-----}
15 #ifdef ADMC328E
15 {ROM Utility Start Locations}
15 .CONST      PER_RST      = 0X07F1;
15 .CONST      UMASK        = 0X0DED;
15 .CONST      PUT_VECTOR   = 0X0DF4;

```

addr inst source line

```

15 .CONST      SMASK = 0X0E06;
15 .CONST      ADMC_COS      = 0X0E26;
15 .CONST      ADMC_SIN      = 0X0E2D;
15 .CONST      ARCTAN        = 0X0E43;
15 .CONST      RECIP_NEW     = 0X0E65; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
15 .CONST      SQRT_NEW      = 0X0E7C; {For SQRT - call SQRT in mathfix.dsp}
15 .CONST      LN_NEW        = 0X0EBA; {For LN - call LN in mathfix.dsp}
15 .CONST      LOG_NEW       = 0X0EB9; {For LOG - call LOG in mathfix.dsp}
15 .CONST      FLTONE        = 0X0ED4;
15 .CONST      FIXONE        = 0X0ED9;
15 .CONST      FPA           = 0X0EDD;
15 .CONST      FPS           = 0X0EEC;
15 .CONST      FPM           = 0X0EFC;
15 .CONST      FPD           = 0X0F05;
15 .CONST      FPMACC        = 0X0F26;
15 .CONST      PARK          = 0X0F48;
15 .CONST      REV_CLARK     = 0X0F5C;
15 .CONST      FOR_CLARK     = 0X0F72;
15 .CONST      COS64         = 0X0F80;
15 .CONST      ONE_BY_X      = 0X0FC0;
15 .CONST      SDIVQINT      = 0X0FD0;
15 .CONST      SDIVQ         = 0X0FD9;
15
15 {Monitor Variables}
15 .CONST HALT_FLAG = 0x3B68; {used by exit library function}
15 .CONST IDE_SP     = 0x7F7; {used by exit library function}
15 .CONST IRQFLAG_SAVE = 0x3B88; {IRQFLAG saved here each interrupt}
15 #endif
15
15 {-----ADMC328-----}
15 #ifdef ADMC328
15 {ROM Utility Start Locations}
15 .CONST      PER_RST      = 0X01F1;

```

```

15 .CONST      UMASK = 0X0F15;
15 .CONST      PUT_VECTOR = 0X0F1C;
15 .CONST      SMASK = 0X0F29;
15 .CONST      SDIVQINT  = 0X0F30;
15 .CONST      SDIVQ = 0X0F39;
15
15 {Monitor Variables}
15 .CONST IRQFLAG_SAVE      = 0x3988; {IRQFLAG saved here each interrupt}
15 #endif
15
15 {-----ADMC201-----}
15 #ifdef ADMC201
15 {ROM Utility Start Locations}
15 .CONST PUT_VECTOR      = 0x0573;
15 .CONST COS64           = 0x0580;
15 .CONST ONE_BY_X        = 0x05C0;
15
15 {Monitor Variables}
15 .CONST IRQFLAG_SAVE      = 0x3BFE; {IRQFLAG saved here each interrupt}
15 #endif
15
15 #endif
15
15
15
16
17 .var BUS;
18 .global bus;

```

addr inst source line

```

        19 .init BUS: 0;      !Bus voltage
        20
        21 .entry adcb_int;
        22 ADCB_INT:
0000 0C0030      23  ena sec_reg;
        24
0001 420006      25  my0=8192;
        26
0002 800002u     27  mx0=dm(BUS);      !S/A current above for bus
0003 820323      28  mx1=dm(adc3);
0004 20800F      29  mr=mx0*my0 (ss);
0005 21010F      30  mr=mr+mx1*my0 (ss);
0006 90000Cu     31  dm(BUS)=mrl;
        32
0007 0C0020      33  dis sec_reg;
        34
0008 0A001F      35  rti;
        36
        37 .endmod;
        38

```

addr inst source line

```

1  !*****
2  !
3  ! This routine contins the command subroutines. After a
4  ! user command line is parsed and the command determined,
5  ! one of these routines is called.
6  !
7  ! Copyright 2012 MARTIN TODD HESS. All rights reserved

```

```

8  !
9  !*****
10
11  .module command_functions;
12
13  #include <admc300.h>
13  #ifndef ADMC300_INCLUDE
13  #define ADMC300_INCLUDE
13
13  {
13      This include file defines important ADMC300 addresses.  The names
13      defined below can be used in user programs by "including" this file.
13      This file defines:
13          - names for the peripheral registers of the ADMC300
13          - names for the memory mapped core registers of the ADMC300
13          - interrupt vector table addresses
13  }
13
13  { peripheral registers of the ADMC300 }
13
13  .CONST PWMTM      = 0x2008;      {PWM timer register}
13  .CONST PWMDT      = 0x2009;      {PWM dead time register}
13  .CONST PWMPD      = 0x200A;      {PWM pulse deletion}
13  .CONST PWMGATE     = 0x200B;      {PWM gate register}
13  .CONST PWMCHA      = 0x200C;      {PWM channel A register}
13  .CONST PWMCHB      = 0x200D;      {PWM channel B register}
13  .CONST PWMCHC      = 0x200E;      {PWM channel C register}
13  .CONST PWMSEG      = 0x200F;      {PWM segment selection}
13  .CONST AUXTIM0     = 0x2010;      {Auxiliary PWM timer 0}
13  .CONST AUXTIM1     = 0x2011;      {Auxiliary PWM timer 1}
13  .CONST MODECTRL    = 0x2015;      {MODE control register}
13  .CONST SYSSTAT     = 0x2016;      {System status register}
13  .CONST WDTIMER     = 0x2018;      {Watchdog timer register}
13  .CONST PICVECTOR   = 0x201C;      {ISR address          }
13  .CONST PICMASK     = 0x201D;      {IRD mask regsiter   }
13  .CONST EIUCNT      = 0x2020;      {Encoder count register }
13  .CONST EIUMAXCNT   = 0x2021;      {Encoder max count register }

```

```

13 .CONST EIUSTAT   = 0x2022; {Encoder status register   }
13 .CONST EIUCTRL   = 0x2023; {Encoder control register  }
13 .CONST EIUPERIOD = 0x2024; {Encoder loop timer period register}
13 .CONST EIUSCALE  = 0x2025; {Encoder loop timer scale register}
13 .CONST EIUTIMER  = 0x2026; {Encoder loop timer}
13 .CONST EETCNT    = 0x2027; {Latched value of EIUCNT register}
13 .CONST EETN      = 0x2028; {EET pulse decimator register }
13 .CONST EETDIV    = 0x2029; {EET timer decimator register }
13 .CONST EETDELTAT = 0x202A; {EET delta timer register    }
13 .CONST EETT      = 0x202B; {EET timer period register   }
13 .CONST EETSTAT   = 0x202C; {EET overflow status register }
13 .CONST ADC1      = 0x2030; {ADC1 register              }
13 .CONST ADC2      = 0x2031; {ADC2 register              }
13 .CONST ADC3      = 0x2032; {ADC3 register              }
13 .CONST ADC4      = 0x2033; {ADC4 register              }
13 .CONST ADC5      = 0x2034; {ADC5 register              }
13 .CONST ADCCTRL   = 0x2036; {ADC control register       }

```

addr inst source line

```

13  .CONST  ADCSTAT    =    0x2037;    {ADC busy status register }
13  .CONST  ADCSYNC    =    0x2038;    {ADC synchronization register }
13  .CONST  ADCDIVA    =    0x2039;    {ADC bank A clock divide register }
13  .CONST  ADCDIVB    =    0x203A;    {ADC bank B clock divide register }
13  .CONST  ADCCAL     =    0x203B;    {ADC calibration register }
13  .CONST  PIOLEVEL   =    0x2040;    {PIO level control register }
13  .CONST  PIOMODE    =    0x2041;    {PIO mode control register }
13  .CONST  PIOPWM     =    0x2042;    {PIO PWMTRIP control register }
13  .CONST  PIODIR     =    0x2044;    {PIO direction register }
13  .CONST  PIODATA    =    0x2045;    {PIO data register}
13  .CONST  PIOINTEN   =    0x2046;    {PIO interrupt enable register }
13  .CONST  PIOFLAG    =    0x2047;    {PIO interrupt flag register }
13  .CONST  ETUA0      =    0x2050;    {ETU0 Event A capture register }
13  .CONST  ETUB0      =    0x2051;    {ETU0 Event B capture register }
13  .CONST  ETUAA0     =    0x2052;    {ETU0 Event AA capture register }
13  .CONST  ETUA1      =    0x2053;    {ETU1 Event A capture register}
13  .CONST  ETUB1      =    0x2054;    {ETU1 Event B capture register}
13  .CONST  ETUAA1     =    0x2055;    {ETU1 Event AA capture register}
13  .CONST  ETUTIME    =    0x2056;    {ETU timer value}
13  .CONST  ETUCONFIG  =    0x205C;    {ETU configuration register }
13  .CONST  ETUDIVIDE  =    0x205D;    {ETU clock divide register }
13  .CONST  ETUSTAT    =    0x205E;    {ETU status register }
13  .CONST  ETUCTRL    =    0x205F;    {ETU control register }
13  .CONST  PWMSYNCWT  =    0x2060;    {PWMSYNC width register}
13  .CONST  PWMSWT     =    0x2061;    {PWM software trip register}
13
13  { memory mapped core registers of the ADMC300 }
13
13  .CONST  SYSCNTL    =    0x3fff;
13  .CONST  MEMWAIT    =    0x3ffe;
13  .CONST  TPERIOD     =    0x3ffd;
13  .CONST  TCOUNT     =    0x3ffc;
13  .CONST  TSCALE     =    0x3ffb;

```

```

13  .CONST  Sport0_Rx_Words1      =    0x3ffa;
13  .CONST  Sport0_Rx_Words0      =    0x3ff9;
13  .CONST  Sport0_Tx_Words1      =    0x3ff8;
13  .CONST  Sport0_Tx_Words0      =    0x3ff7;
13  .CONST  Sport0_Ctrl_Reg        =    0x3ff6;
13  .CONST  Sport0_Sclkdiv         =    0x3ff5;
13  .CONST  Sport0_Rfsdiv          =    0x3ff4;
13  .CONST  Sport0_Autobuf_Ctrl    =    0x3ff3;
13  .CONST  Sport1_Ctrl_Reg        =    0x3ff2;
13  .CONST  Sport1_Sclkdiv         =    0x3ff1;
13  .CONST  Sport1_Rfsdiv          =    0x3ff0;
13  .CONST  Sport1_Autobuf_Ctrl    =    0x3fef;
13
13  { interrupt vector table addresses for ADMC300 }
13
13  .CONST  ADCBANKA_INT_ADDR      = 0x30+1;      {ADC BANKA INTERRUPT }
13  .CONST  PWMSYNC_INT_ADDR       = 0x34+1;      {PWMSYNC interrupt  }
13  .CONST  ADCBANKB_INT_ADDR      = 0x38+1;      {ADC BANKB interrupt }
13  .CONST  PIO_INT_ADDR           = 0x3C+1;      {PIO4 - PIO11 interrupt }
13  .CONST  EIU_INT_ADDR           = 0x40+1;      {EIU interrupt  }
13  .CONST  ETU_INT_ADDR           = 0x44+1;      {ETU interrupt  }
13  .CONST  PIO0_INT_ADDR          = 0x48+1;      {PIO0 interrupt  }
13  .CONST  PIO1_INT_ADDR          = 0x4C+1;      {PIO1 interrupt  }
13  .CONST  PIO2_INT_ADDR          = 0x50+1;      {PIO2 interrupt  }
13  .CONST  PIO3_INT_ADDR          = 0x54+1;      {PIO3 interrupt  }
13  .CONST  PWMTRIP_INT_ADDR       = 0x58+1;      {PWMTRIP interrupt}
13  .CONST  TX0_INT_ADDR           = 0x10; {SPORT0 transmit interrupt}
13  .CONST  RX0_INT_ADDR           = 0x14; {SPORT0 receive interrupt}

```


addr inst source line

```

13 .CONST SW1_INT_ADDR      = 0x18;      {software interrupt 1}
13 .CONST SW0_INT_ADDR      = 0x1C;      {software interrupt 0}
13 .CONST TX1_INT_ADDR      = 0x20; {SPORT1 transmit interrupt}
13 .CONST RX1_INT_ADDR      = 0x24; {SPORT1 receive interrupt}
13 .CONST TIMER_INT_ADDR    = 0x28; {Timer interrupt}
13
13 #endif
14 #include <command_list.h>
14 command_list:
0000 1800Fu      14  jump function_ignore; !0x00 NUL    Null
0001 1800Fu      14  jump function_ignore; !0x01 SOH
0002 1800Fu      14  jump function_ignore; !0x02 STX
0003 1800Fu      14  jump function_ignore; !0x03 ETX
0004 1800Fu      14  jump function_ignore; !0x04 EOT
0005 1800Fu      14  jump function_ignore; !0x05 ENQ
0006 1800Fu      14  jump function_ignore; !0x06 ACK
0007 1800Fu      14  jump function_ignore; !0x07 BEL
0008 1800Fu      14  jump function_bs;!0x08 BS      Back Space
0009 1800Fu      14  jump function_ignore; !0x09 HT
000A 1800Fu      14  jump function_ignore; !0x0A LF
000B 1800Fu      14  jump function_ignore; !0x0B VT
000C 1800Fu      14  jump function_ignore; !0x0C FF
000D 1800Fu      14  jump function_cr;!0x0D CR      Carriage Return
000E 1800Fu      14  jump function_ignore; !0x0E SO
000F 1800Fu      14  jump function_ignore; !0x0F SI
0010 1800Fu      14  jump function_ignore; !0x10 DLE
0011 1800Fu      14  jump function_ignore; !0x11 DC1
0012 1800Fu      14  jump function_ignore; !0x12 DC2
0013 1800Fu      14  jump function_ignore; !0x13 DC3
0014 1800Fu      14  jump function_ignore; !0x14 DC4
0015 1800Fu      14  jump function_ignore; !0x15 NAK
0016 1800Fu      14  jump function_ignore; !0x16 SYN
0017 1800Fu      14  jump function_ignore; !0x17 ETB

```

0018	18000Fu	14	jump function_ignore;	!0x18 CAN
0019	18000Fu	14	jump function_ignore;	!0x19 EM
001A	18000Fu	14	jump function_ignore;	!0x1A SUB
001B	18000Fu	14	jump function_ignore;	!0x1B ESC
001C	18000Fu	14	jump function_ignore;	!0x1C FS
001D	18000Fu	14	jump function_ignore;	!0x1D GS
001E	18000Fu	14	jump function_ignore;	!0x1E RS
001F	18000Fu	14	jump function_ignore;	!0x1F US
	14			
0020	18000Fu	14	jump function_ignore;	!0x20 Space
0021	18000Fu	14	jump function_ignore;	!0x21 !
0022	18000Fu	14	jump function_ignore;	!0x22 Quote
0023	18000Fu	14	jump function_ignore;	!0x23 #
0024	18000Fu	14	jump function_ignore;	!0x24 \$
0025	18000Fu	14	jump function_ignore;	!0x25 %
0026	18000Fu	14	jump function_ignore;	!0x26 &
0027	18000Fu	14	jump function_ignore;	!0x27 Apsotrophie
0028	18000Fu	14	jump function_ignore;	!0x28 (
0029	18000Fu	14	jump function_ignore;	!0x29)
002A	18000Fu	14	jump function_ignore;	!0x2A *
002B	18000Fu	14	jump function_ignore;	!0x2B +
002C	18000Fu	14	jump function_ignore;	!0x2C ;
002D	18000Fu	14	jump function_ignore;	!0x2D -
002E	18000Fu	14	jump function_ignore;	!0x2E .
002F	18000Fu	14	jump function_ignore;	!0x2F /
	14			
0030	18000Fu	14	jump function_number;	!0x30 0
0031	18000Fu	14	jump function_number;	!0x31 1

addr inst source line

```

0032 1800Fu      14  jump function_number; !0x32 2
0033 1800Fu      14  jump function_number; !0x33 3
0034 1800Fu      14  jump function_number; !0x34 4
0035 1800Fu      14  jump function_number; !0x35 5
0036 1800Fu      14  jump function_number; !0x36 6
0037 1800Fu      14  jump function_number; !0x37 7
0038 1800Fu      14  jump function_number; !0x38 8
0039 1800Fu      14  jump function_number; !0x39 9
003A 1800Fu      14  jump function_ignore; !0x3A :
003B 1800Fu      14  jump function_ignore; !0x3B ;
003C 1800Fu      14  jump function_ignore; !0x3C <
003D 1800Fu      14  jump function_ignore; !0x3D =
003E 1800Fu      14  jump function_ignore; !0x3E >
003F 1800Fu      14  jump function_ignore; !0x3F ?
      14
0040 1800Fu      14  jump function_ignore; !0x40 @
0041 1800Fu      14  jump function_ignore; !0x41 A
0042 1800Fu      14  jump function_cap_b; !0x42 B
0043 1800Fu      14  jump function_cap_c; !0x43 C
0044 1800Fu      14  jump function_ignore; !0x44 D
0045 1800Fu      14  jump function_ignore; !0x45 E
0046 1800Fu      14  jump function_cap_f; !0x46 F
0047 1800Fu      14  jump function_cap_g; !0x47 G
0048 1800Fu      14  jump function_ignore; !0x48 H
0049 1800Fu      14  jump function_cap_i; !0x49 I
004A 1800Fu      14  jump function_ignore; !0x4A J
004B 1800Fu      14  jump function_ignore; !0x4B K
004C 1800Fu      14  jump function_ignore; !0x4C L
004D 1800Fu      14  jump function_ignore; !0x4D M
004E 1800Fu      14  jump function_ignore; !0x4E N
004F 1800Fu      14  jump function_ignore; !0x4F O
      14
0050 1800Fu      14  jump function_cap_p; !0x50 P

```

0051	18000Fu	14	jump function_ignore;	!0x51 Q
0052	18000Fu	14	jump function_cap_r;	!0x52 R
0053	18000Fu	14	jump function_cap_s;	!0x53 S
0054	18000Fu	14	jump function_cap_t;	!0x54 T
0055	18000Fu	14	jump function_ignore;	!0x55 U
0056	18000Fu	14	jump function_ignore;	!0x56 V
0057	18000Fu	14	jump function_ignore;	!0x57 W
0058	18000Fu	14	jump function_ignore;	!0x58 X
0059	18000Fu	14	jump function_ignore;	!0x59 Y
005A	18000Fu	14	jump function_ignore;	!0x5A Z
005B	18000Fu	14	jump function_ignore;	!0x5B [
005C	18000Fu	14	jump function_ignore;	!0x5C fwd slash
005D	18000Fu	14	jump function_ignore;	!0x5D]
005E	18000Fu	14	jump function_ignore;	!0x5E jump
005F	18000Fu	14	jump function_ignore;	!0x5F _
	14			
0060	18000Fu	14	jump function_ignore;	!0x60 `
0061	18000Fu	14	jump function_hex;	!0x61 a
0062	18000Fu	14	jump function_hex;	!0x62 b
0063	18000Fu	14	jump function_hex;	!0x63 c
0064	18000Fu	14	jump function_hex;	!0x64 d
0065	18000Fu	14	jump function_hex;	!0x65 e
0066	18000Fu	14	jump function_hex;	!0x66 f
0067	18000Fu	14	jump function_ignore;	!0x67 g
0068	18000Fu	14	jump function_ignore;	!0x68 h
0069	18000Fu	14	jump function_ignore;	!0x69 i
006A	18000Fu	14	jump function_j;	!0x6A j
006B	18000Fu	14	jump function_k;	!0x6B k

addr inst source line

```
006C 1800Fu      14  jump function_l; !0x6C l
006D 1800Fu      14  jump function_ignore; !0x6D n
006E 1800Fu      14  jump function_ignore; !0x6E n
006F 1800Fu      14  jump function_ignore; !0x6F o
      14
0070 1800Fu      14  jump function_p; !0x70 p
0071 1800Fu      14  jump function_ignore; !0x71 q
0072 1800Fu      14  jump function_r; !0x72 r
0073 1800Fu      14  jump function_ignore; !0x73 s
0074 1800Fu      14  jump function_t; !0x74 t
0075 1800Fu      14  jump function_ignore; !0x75 u
0076 1800Fu      14  jump function_ignore; !0x76 v
0077 1800Fu      14  jump function_ignore; !0x77 w
0078 1800Fu      14  jump function_ignore; !0x78 x
0079 1800Fu      14  jump function_ignore; !0x79 y
007A 1800Fu      14  jump function_ignore; !0x7A z
007B 1800Fu      14  jump function_ignore; !0x7B Open brace
007C 1800Fu      14  jump function_ignore; !0x7C |
007D 1800Fu      14  jump function_ignore; !0x7D Close brace
007E 1800Fu      14  jump function_ignore; !0x7E ~
007F 1800Fu      14  jump function_ignore; !0x7F Del
      15  #include <command_functions.h>
      15
      15  .external ang_flux;
      15  .external ang_volt;
      15  .external bus;
      15  .external cur_mag;
      15  .external cur_r;
      15  .external cur_ref;
      15  .external cur_t;
      15  .external foc_coef;
      15  .external plot_count;
      15  .external volt_mag;
```

```

15 .external volt_r;
15 .external volt_s;
15 .external volt_t;
15
15 .external rd_queue;
15
15
15 .var index_j; .init index_j: 0;
15 .var index_k; .init index_k: 0;
15 .var index_l; .init index_l: 0;
15
15 .entry process_control_character;
15
16
17 !*****
18
19 process_control_character:
0080 380000u      20  i4 = ^command_list;
0081 0D084A      21  m4 = ar;
0082 090010      22  modify (i4, m4);
0083 0B000F      23  jump (i4); !Returns to main from from there
      24
      25 !*****
      26
      27 function_bs:
0084 400001u      28  ax1=^rd_queue;
0085 0D0143      29  ay0=i3;
0086 2AE1AA      30  none=ax1-ay0;

```

addr inst source line

```

0087 0A0000      31  if eq rts;
0088 090002      32  modify(i0, m2); !backup one location
0089 680009      33  dm(i2, m1)=ax0;
008A 0A000F      34  rts;
      35
      36  !*****
      37
      38  function_ignore:
008B 18000Fu     39  jump send_cr_lf_prompt;
008C 0A000F      40  rts;
      41
      42  !*****
      43
      44  function_cr:
008D 40000E     45  sr0 = 0;    !Used to construct argument
008E 400009     46  se = 0;    !Initial shift is zero
008F 400011     47  ax1 = 1;    !Read not write flag
      48
      49  parse:
0090 0D0100     50  ax0=i0;    !If at start, send prompt
0091 400004u     51  ay0=^rd_queue;
0092 2AE0AA     52  none=ax0-ay0;
0093 180000u     53  if eq jump SEND_CR_LF_PROMPT;
      54
0094 090002     55  modify(i0, m2);
0095 380000u     56  i4 = ^command_list;
0096 6000A0     57  ar = dm(i0, m0);
0097 4007F4     58  ay0 = 0x7F;
0098 23820F     59  ar = ar and ay0;
0099 0D084A     60  m4 = ar;
009A 090010     61  modify (i4, m4);
009B 0B000F     62  jump (i4); !Returns to main via there
      63

```

```

        64  !*****
        65  function_hex:
009C  400274      66  ay0 = 0x27;
009D  22E20F      67  ar = ar - ay0;
        68
        69  function_number:
009E  400001      70  ax1 = 0;  !Argument received, so clear r/w flag
        71
009F  400304      72  ay0 = 0x30;
00A0  22E20F      73  ar = ar - ay0;
        74
00A1  0E1A0F      75  sr = sr or lshift ar (lo);
00A2  0D00A9      76  ar = se;
00A3  22629F      77  ar = ar + 4;
00A4  0D009A      78  se = ar;
        79
00A5  18000Fu     80  jump parse;
        81
        82  !*****
        83  function_j:
00A6  380000u     84  i4 = ^index_j;
00A7  18000Fu     85  jump read_write;
        86
        87  !*****
        88
        89  function_k:
00A8  380000u     90  i4 = ^index_k;
00A9  18000Fu     91  jump read_write;

```


addr inst source line

```

    92
    93 !*****
    94
    95 function_l:
00AA 380000u    96 i4 = ^index_l;
00AB 18000Fu    97 jump read_write;
    98
    99 !*****
   100
   101 function_p:
00AC 380000u   102 i4 = ^plot_count;
00AD 18000Fu   103 jump read_write;
   104
   105 !*****
   106 function_r:
   107
00AE 80000Eu   108 sr0 = dm(cur_r);
00AF 18000Fu   109 jump CONVERT_TO_ASCII;    !Returns to main from there
   110
   111 !*****
   112
   113 function_t:
00B0 80000Eu   114 sr0 = dm(cur_t);
00B1 18000Fu   115 jump CONVERT_TO_ASCII;    !Returns to main from there
   116
   117 !*****
   118
   119 function_cap_a:
00B2 80000Eu   120 sr0 = dm(ang_volt);
00B3 18000Fu   121 jump CONVERT_TO_ASCII;    !Returns to main from there
   122
   123 !*****
   124
```

```

125 function_cap_c:
00B4 380000u      126  i4 = ^cur_ref;
00B5 18000Fu      127  jump read_write; !Returns to main from there
128
129  !*****
130
131 function_cap_b:
00B6 80000Eu      132  sr0=dm(bus);
00B7 18000Fu      133  jump CONVERT_TO_ASCII;      !Returns to main from there
134
135  !*****
136
137 function_cap_f:
00B8 80000Eu      138  sr0=dm(ang_flux);
00B9 18000Fu      139  jump CONVERT_TO_ASCII;      !Returns to main from there
140
141  !*****
142
143 function_cap_g:
00BA 380000u      144  i4 = ^foc_coef + 2;
00BB 18000Fu      145  jump read_write;
146
147
148  !*****
149
150 function_cap_i:
00BC 380000u      151  i4 = ^foc_coef;
00BD 18000Fu      152  jump read_write;

```

addr inst source line

```

153
154
155 !*****
156
157 function_cap_p:
00BE 380000u      158  i4 = ^foc_coef + 1;
00BF 18000Fu      159  jump read_write;
160
161
162 !*****
163
164 function_cap_r:
00C0 80000Eu      165  sr0=dm(volt_r);
00C1 18000Fu      166  jump CONVERT_TO_ASCII;      !Returns to main from there
167
168 !*****
169
170 function_cap_s:
00C2 80000Eu      171  sr0=dm(volt_s);
00C3 18000Fu      172  jump CONVERT_TO_ASCII;      !Returns to main from there
173
174 !*****
175
176 function_cap_t:
00C4 80000Eu      177  sr0=dm(volt_t);
00C5 18000Fu      178  jump CONVERT_TO_ASCII;      !Returns to main from there
179
180
181 !*****
182
183 SEND_CR_LF_PROMPT:
00C6 A000D9      184  dm(i2, m1)=0x000d;      !CR
00C7 A000A9      185  dm(i2, m1)=0x000a;      !LF
```

```

00C8 A003E9      186  dm(i2, m1)=0x003E;    !> prompt
00C9 0A000F      187  rts;
      188
      189  !*****
      190
      191  CONVERT_TO_ASCII:
00CA 3C0045      192  cntr=4;    !Send 4 ASCII characters to represent Hex #
00CB 14000Eu     193  do ASCII_LOOP until CE;
00CC 0F1604      194  sr=lshift sr0 by 4 (lo);    !Argument was passed in sr0
      195
00CD 400304      196  ay0=0x30; !Add ASCII offset of 0x30
00CE 22670F      197  ar=sr1+ay0;
00CF 400394      198  ay0 = 0x39;
00D0 400275      199  ay1 = 0x27;
00D1 2AE2AA      200  none = ar - ay0;
00D2 226A02      201  if gt ar = ar + ay1;
      202  ASCII_LOOP:
00D3 6800A9      203  dm(i2, m1)=ar;    !Put in write queue
      204
00D4 18000Fu     205  jump SEND_CR_LF_PROMPT;
      206
      207  !*****
      208
      209  read_write:
00D5 2A79AA      210  none = pass ax1;
00D6 180000u     211  if eq jump dm_write;
      212
      213  dm_read:

```

Release 6.0 Assembler Version 2.0 11/7/97

C:\My Documents\thesis\code\command_functions.dsp Wed Jan 18 17:52:01 2012

Page 9

addr inst source line

```
00D7 7000E0      214  sr0 = dm(i4, m4);
00D8 1800Fu      215  jump convert_to_ascii;
      216
      217 dm_write:
00D9 7800E0      218  dm(i4, m4) = sr0;
00DA 1800Fu      219  jump SEND_CR_LF_PROMPT;
      220
      221 !*****
      222
      223 .endmod;
```

Release 6.0 Assembler Version 2.0 11/7/97

C:\My Documents\thesis\code\host_communications.dsp Wed Jan 18 17:57:45 2012

Page 1

addr inst source line

```
1  !*****
2  !
3  ! This routine receives incoming characters from the user
4  ! RS-232 port and assembles them in a queue.  When a CR is
5  ! received, the string is parsed and the appropriate
6  ! subroutine is called.
7  !
8  ! This routine also cheques for charaters in the outgoing
9  ! queue and transmits them.
10 !
11 ! Copyright 2012 MARTIN TODD HESS.  All rights reserved
12 !
13 !*****
14
15 .module host_communications;
16
17 #include <host_communications.h>
```

```

17 #define ADMC300
17 #include <ADMC300.h>
17 #ifndef ADMC300_INCLUDE
17 #define ADMC300_INCLUDE
17
17 {
17     This include file defines important ADMC300 addresses. The names
17     defined below can be used in user programs by "including" this file.
17     This file defines:
17         - names for the peripheral registers of the ADMC300
17         - names for the memory mapped core registers of the ADMC300
17         - interrupt vector table addresses
17 }
17
17 { peripheral registers of the ADMC300 }
17
17 .CONST PWMTM      = 0x2008;    {PWM timer register}
17 .CONST PWMDT      = 0x2009;    {PWM dead time register}
17 .CONST PWMPD      = 0x200A;    {PWM pulse deletion}
17 .CONST PWMGATE    = 0x200B;    {PWM gate register}
17 .CONST PWMCHA     = 0x200C;    {PWM channel A register}
17 .CONST PWMCHB     = 0x200D;    {PWM channel B register}
17 .CONST PWMCHC     = 0x200E;    {PWM channel C register}
17 .CONST PWMSEG     = 0x200F;    {PWM segment selection}
17 .CONST AUXTIM0    = 0x2010;    {Auxiliary PWM timer 0}
17 .CONST AUXTIM1    = 0x2011;    {Auxiliary PWM timer 1}
17 .CONST MODECTRL   = 0x2015;    {MODE control register}
17 .CONST SYSSTAT    = 0x2016;    {System status register}
17 .CONST WDTIMER    = 0x2018;    {Watchdog timer register}
17 .CONST PICVECTOR  = 0x201C;    {ISR address          }
17 .CONST PICMASK    = 0x201D;    {IRD mask regsiter   }
17 .CONST EIUCNT     = 0x2020;    {Encoder count register }
17 .CONST EIUMAXCNT  = 0x2021;    {Encoder max count register }
17 .CONST EIUSTAT    = 0x2022;    {Encoder status register  }
17 .CONST EIUCTRL    = 0x2023;    {Encoder control register  }
17 .CONST EIUPERIOD  = 0x2024;    {Encoder loop timer period register}
17 .CONST EIUSCALE   = 0x2025;    {Encoder loop timer scale register}

```

```
17 .CONST EIUTIMER = 0x2026; {Encoder loop timer}
17 .CONST EETCNT = 0x2027; {Latched value of EIUCNT register}
17 .CONST EETN = 0x2028; {EET pulse decimator register }
17 .CONST EETDIV = 0x2029; {EET timer decimator register }
17 .CONST EETDELTAT = 0x202A; {EET delta timer register }
17 .CONST EETT = 0x202B; {EET timer period register }
17 .CONST EETSTAT = 0x202C; {EET overflow status register }
```

addr inst source line

```

17 .CONST ADC1      = 0x2030; {ADC1 register      }
17 .CONST ADC2      = 0x2031; {ADC2 register      }
17 .CONST ADC3      = 0x2032; {ADC3 register      }
17 .CONST ADC4      = 0x2033; {ADC4 register      }
17 .CONST ADC5      = 0x2034; {ADC5 register      }
17 .CONST ADCCTRL   = 0x2036; {ADC control register }
17 .CONST ADCSTAT   = 0x2037; {ADC busy status register }
17 .CONST ADCSYNC   = 0x2038; {ADC synchronization register }
17 .CONST ADCDIVA   = 0x2039; {ADC bank A clock divide register }
17 .CONST ADCDIVB   = 0x203A; {ADC bank B clock divide register }
17 .CONST ADCCAL    = 0x203B; {ADC calibration register }
17 .CONST PIOLEVEL  = 0x2040; {PIO level control register }
17 .CONST PIOMODE   = 0x2041; {PIO mode control register }
17 .CONST PIOPWM    = 0x2042; {PIO PWMTRIP control register }
17 .CONST PIODIR    = 0x2044; {PIO direction register }
17 .CONST PIODATA   = 0x2045; {PIO data register}
17 .CONST PIOINTEN  = 0x2046; {PIO interrupt enable register }
17 .CONST PIOFLAG   = 0x2047; {PIO interupt flag register }
17 .CONST ETUA0     = 0x2050; {ETU0 Event A capture register }
17 .CONST ETUB0     = 0x2051; {ETU0 Event B capture register }
17 .CONST ETUAA0    = 0x2052; {ETU0 Event AA capture register }
17 .CONST ETUA1     = 0x2053; {ETU1 Event A capture regsiter}
17 .CONST ETUB1     = 0x2054; {ETU1 Event B capture register}
17 .CONST ETUAA1    = 0x2055; {ETU1 Event AA capture register}
17 .CONST ETUTIME   = 0x2056; {ETU timer value}
17 .CONST ETUCONFIG = 0x205C; {ETU configuration register }
17 .CONST ETUDIVIDE = 0x205D; {ETU clock divide register }
17 .CONST ETUSTAT   = 0x205E; {ETU status register }
17 .CONST ETUCTRL   = 0x205F; {ETU control register }
17 .CONST PWMSYNCWT = 0x2060; {PWMSYNC width register}
17 .CONST PWMSWT    = 0x2061; {PWM software trip register}
17
17 { memory mapped core registers of the ADMC300 }

```



```

17
17 .CONST SYSCNTL = 0x3fff;
17 .CONST MEMWAIT = 0x3ffe;
17 .CONST TPERIOD = 0x3ffd;
17 .CONST TCOUNT = 0x3ffc;
17 .CONST TSCALE = 0x3ffb;
17 .CONST Sport0_Rx_Words1 = 0x3ffa;
17 .CONST Sport0_Rx_Words0 = 0x3ff9;
17 .CONST Sport0_Tx_Words1 = 0x3ff8;
17 .CONST Sport0_Tx_Words0 = 0x3ff7;
17 .CONST Sport0_Ctrl_Reg = 0x3ff6;
17 .CONST Sport0_Sclkdiv = 0x3ff5;
17 .CONST Sport0_Rfsdiv = 0x3ff4;
17 .CONST Sport0_Autobuf_Ctrl = 0x3ff3;
17 .CONST Sport1_Ctrl_Reg = 0x3ff2;
17 .CONST Sport1_Sclkdiv = 0x3ff1;
17 .CONST Sport1_Rfsdiv = 0x3ff0;
17 .CONST Sport1_Autobuf_Ctrl = 0x3fef;
17
17 { interrupt vector table addresses for ADMC300 }
17
17 .CONST ADCBANKA_INT_ADDR = 0x30+1; {ADC BANKA INTERRUPT }
17 .CONST PWMSYNC_INT_ADDR = 0x34+1; {PWMSYNC interrupt }
17 .CONST ADCBANKB_INT_ADDR = 0x38+1; {ADC BANKB interrupt }
17 .CONST PIO_INT_ADDR = 0x3C+1; {PIO4 - PIO11 interrupt }
17 .CONST EIU_INT_ADDR = 0x40+1; {EIU interrupt }
17 .CONST ETU_INT_ADDR = 0x44+1; {ETU interrupt }
17 .CONST PIO0_INT_ADDR = 0x48+1; {PIO0 interrupt }

```

addr inst source line

```

17  .CONST PIO1_INT_ADDR      = 0x4C+1;      {PIO1 interrupt }
17  .CONST PIO2_INT_ADDR      = 0x50+1;      {PIO2 interrupt }
17  .CONST PIO3_INT_ADDR      = 0x54+1;      {PIO3 interrupt }
17  .CONST PWMTRIP_INT_ADDR   = 0x58+1;      {PWMTRIP interrupt}
17  .CONST TX0_INT_ADDR       = 0x10; {SPORT0 transmit interrupt}
17  .CONST RX0_INT_ADDR       = 0x14; {SPORT0 receive interrupt}
17  .CONST SW1_INT_ADDR       = 0x18;      {software interrupt 1}
17  .CONST SW0_INT_ADDR       = 0x1C;      {software interrupt 0}
17  .CONST TX1_INT_ADDR       = 0x20; {SPORT1 transmit interrupt}
17  .CONST RX1_INT_ADDR       = 0x24; {SPORT1 receive interrupt}
17  .CONST TIMER_INT_ADDR     = 0x28; {Timer interrupt}
17
17  #endif
17  #include <romutil.h>
17  {This include file contains the addresses for the ROM utility routines.
17   To use just include this file in the module where the routine will be called.
17   Then just call the routine name given below.
17
17  VERSION # 2.0      07/10/98
17
17  Name          Date      Description
17  ----          -
17  MOD/HISTORY:   TFH      07/10/98      Combined all targets into one file
17  }
17  #ifndef ROMUTIL_INCLUDE
17  #define ROMUTIL_INCLUDE
17
17  {-----ADMC300-----}
17  #ifdef ADMC300
17  {ROM Utility Start Locations}
17  .CONST PER_RST      = 0x07E4;
17  .CONST UMASK        = 0x0E21;
17  .CONST PUT_VECTOR= 0x0E28;

```

```

17  .CONST SMASK      = 0x0E35;
17  .CONST ADMC_COS   = 0x0E55;
17  .CONST ADMC_SIN   = 0x0E5C;
17  .CONST ARCTAN     = 0x0E72;
17  .CONST RECIP_NEW  = 0x0E94; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
17  .CONST SQRT       = 0x0EAA;
17  .CONST LOG        = 0x0EE7;
17  .CONST LN         = 0x0EE4;
17  .CONST FLTONE     = 0x0F03;
17  .CONST FIXONE     = 0x0F08;
17  .CONST FPA        = 0x0F0C;
17  .CONST FPS        = 0x0F1B;
17  .CONST FPM        = 0x0F2B;
17  .CONST FPD        = 0x0F34;
17  .CONST FPMACC     = 0x0F55;
17  .CONST REV_FOR_PARK = 0x0F77;
17  .CONST REV_CLARK   = 0x0F8B;
17  .CONST FOR_CLARK   = 0x0FA1;
17  .CONST SDIVQINT   = 0x0FAB;
17  .CONST SDIVQ      = 0x0FB4;
17  .CONST EXIT       = 0x0FC6;
17
17  {Monitor Variables}
17  .CONST I4_SAVE     = 0x3bbf;
17  #endif
17
17  {-----ADMC330-----}
17  #ifdef ADMC330

```

addr inst source line

```

17 {ROM Utility Start Locations}
17 .CONST PUT_VECTOR      = 0x0B73;
17 .CONST SIN12B          = 0x0DC4;
17 .CONST COS12B          = 0x0DD3;
17 .CONST CAL_INV_TAN     = 0x0E61;
17 .CONST LOG1            = 0x0E14;
17 .CONST ALOG1          = 0x0E20;
17 .CONST SQRT1          = 0x0E27;
17 .CONST DIVIDE         = 0x0DE2;
17 .CONST X_Y_TO_ANG     = 0x0E35;
17 .CONST APPROX10_ST    = 0x0DEC;
17 .CONST APPROX_N_ST    = 0x0DFF;
17 .CONST COS64          = 0x0B80;
17 .CONST COS6_64        = 0x0C00;
17 .CONST SIN6_64        = 0x0C40;
17 .CONST ATN64          = 0x0D01;
17 .CONST LOG64          = 0x0D42;
17 .CONST ALOG64         = 0x0D83;
17
17 {Monitor Variables}
17 .CONST HALT_FLAG       = 0x3B9A;    {used by exit library function}
17 .CONST IDE_SP          = 0x79E;    {used by exit library function}
17 .CONST IRQFLAG_SAVE    = 0x3BFE;    {IRQFLAG saved here each interrupt}
17 #endif
17
17 {-----ADMC331-----}
17 #ifdef ADMC331
17 {ROM Utility Start Locations}
17 .CONST PER_RST        = 0X07F1;
17 .CONST UMASK          = 0X0DED;
17 .CONST PUT_VECTOR     = 0X0DF4;
17 .CONST SMASK          = 0X0E06;
17 .CONST ADMC_COS       = 0X0E26;

```

```

17 .CONST      ADMC_SIN      = 0X0E2D;
17 .CONST      ARCTAN       = 0X0E43;
17 .CONST      RECIP_NEW    = 0X0E65; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
17 .CONST      SQRT_NEW     = 0X0E7C; {For SQRT - call SQRT in mathfix.dsp}
17 .CONST      LN_NEW       = 0X0EBA; {For LN - call LN in mathfix.dsp}
17 .CONST      LOG_NEW      = 0X0EB9; {For LOG - call LOG in mathfix.dsp}
17 .CONST      FLTONE       = 0X0ED4;
17 .CONST      FIXONE       = 0X0ED9;
17 .CONST      FPA         = 0X0EDD;
17 .CONST      FPS         = 0X0EEC;
17 .CONST      FPM         = 0X0EFC;
17 .CONST      FPD         = 0X0F05;
17 .CONST      FPMACC      = 0X0F26;
17 .CONST      PARK        = 0X0F48;
17 .CONST      REV_CLARK    = 0X0F5C;
17 .CONST      FOR_CLARK    = 0X0F72;
17 .CONST      COS64       = 0X0F80;
17 .CONST      ONE_BY_X     = 0X0FC0;
17 .CONST      SDIVQINT     = 0X0FD0;
17 .CONST      SDIVQ       = 0X0FD9;
17
17 {Monitor Variables}
17 .CONST HALT_FLAG = 0x3B68; {used by exit library function}
17 .CONST IDE_SP    = 0x7F7; {used by exit library function}
17 .CONST IRQFLAG_SAVE = 0x3B88; {IRQFLAG saved here each interrupt}
17 #endif
17
17 {-----ADMC328 Emulation Board-----}

```

addr inst source line

```
17 #ifdef ADMC328E
17 {ROM Utility Start Locations}
17 .CONST PER_RST = 0X07F1;
17 .CONST UMASK = 0X0DED;
17 .CONST PUT_VECTOR = 0X0DF4;
17 .CONST SMASK = 0X0E06;
17 .CONST ADMC_COS = 0X0E26;
17 .CONST ADMC_SIN = 0X0E2D;
17 .CONST ARCTAN = 0X0E43;
17 .CONST RECIP_NEW = 0X0E65; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
17 .CONST SQRT_NEW = 0X0E7C; {For SQRT - call SQRT in mathfix.dsp}
17 .CONST LN_NEW = 0X0EBA; {For LN - call LN in mathfix.dsp}
17 .CONST LOG_NEW = 0X0EB9; {For LOG - call LOG in mathfix.dsp}
17 .CONST FLTONE = 0X0ED4;
17 .CONST FIXONE = 0X0ED9;
17 .CONST FPA = 0X0EDD;
17 .CONST FPS = 0X0EEC;
17 .CONST FPM = 0X0EFC;
17 .CONST FPD = 0X0F05;
17 .CONST FPMACC = 0X0F26;
17 .CONST PARK = 0X0F48;
17 .CONST REV_CLARK = 0X0F5C;
17 .CONST FOR_CLARK = 0X0F72;
17 .CONST COS64 = 0X0F80;
17 .CONST ONE_BY_X = 0X0FC0;
17 .CONST SDIVQINT = 0X0FD0;
17 .CONST SDIVQ = 0X0FD9;
17
17 {Monitor Variables}
17 .CONST HALT_FLAG = 0x3B68; {used by exit library function}
17 .CONST IDE_SP = 0x7F7; {used by exit library function}
17 .CONST IRQFLAG_SAVE = 0x3B88; {IRQFLAG saved here each interrupt}
17 #endif
```

```

17
17 {-----ADMC328-----}
17 #ifdef ADMC328
17 {ROM Utility Start Locations}
17 .CONST      PER_RST      = 0X01F1;
17 .CONST      UMASK = 0X0F15;
17 .CONST      PUT_VECTOR = 0X0F1C;
17 .CONST      SMASK = 0X0F29;
17 .CONST      SDIVQINT      = 0X0F30;
17 .CONST      SDIVQ = 0X0F39;
17
17 {Monitor Variables}
17 .CONST IRQFLAG_SAVE      = 0x3988; {IRQFLAG saved here each interrupt}
17 #endif
17
17 {-----ADMC201-----}
17 #ifdef ADMC201
17 {ROM Utility Start Locations}
17 .CONST PUT_VECTOR      = 0x0573;
17 .CONST COS64           = 0x0580;
17 .CONST      ONE_BY_X      = 0x05C0;
17
17 {Monitor Variables}
17 .CONST IRQFLAG_SAVE      = 0x3BFE; {IRQFLAG saved here each interrupt}
17 #endif
17
17 #endif
17

```

addr inst source line

```

17
17
17
17 !External functions
17 .external read_uart0;
17 .external write_uart0;
17 .external process_control_character;
17
17 !External memory
17 .external UART0_Status;
17 .external UART0_Status_TX;
17
17 !Make visible
17 .entry host_receive;
17 .entry host_transmit;
17
18
19 host_receive:
20
0000 80000Au      21  ar=dm(UART0_STATUS); !Test receive buffer empty flag
0001 23829F      22  ar=tstbit 2 of ar;
0002 0A0000      23  if eq rts;
24
0003 1C000Fu      25  call READ_UART0; !On return, ar contains character
26
0004 26EA5F      27  af = ar - 0x20; !Test if control character
0005 180004u      28  if lt jump process_control_character; !Returns to main from there
29
0006 6800A1      30  dm(i0, m1)=ar; !Place in read queue
0007 6800A9      31  dm(i2, m1)=ar; !Echo to write queue
32
0008 0A000F      33  rts;
34
```



```

35  !*****
36
37  host_transmit:
38
0009  80000Au      39  ar=dm(UART0_STATUS_TX);      !Test if transmit buffer empty
000A  238A9F      40  ar=TSTBIT 6 OF AR;
000B  0A0001      41  if ne rts;
42
000C  0D0102      43  ax0=i2;      !Test if anything to send
000D  0D0143      44  ay0=i3;
000E  2AE0AA      45  none=ax0-ay0;
000F  0A0000      46  if eq rts;
47
0010  60000D      48  ax0=dm(i3, m1); !Get character to send
0011  18000Fu     49  jump WRITE_UART0;      !Returns to main from there
50
51  !*****
52  .endmod;
53
54

```

Release 6.0 Assembler Version 2.0 11/7/97

C:\My Documents\thesis\code\init_vars.dsp Wed Jan 18 18:00:14 2012

Page 1

addr inst source line

```

1  !*****
2  !
3  ! This routine merely initialises important varibales
4  !
5  ! Copyright 2012 MARTIN TODD HESS. All rights reserved
6  !
7  !*****
8  .module init_vars_module;
9
10 #include <init_vars.h>
10 .entry init_vars;

```

```

10
10 .external calibrate_loops;
10 .external foc_integrators;
10 .external twang_loops;
10
11
12 init_vars:
0000 40000A      13  ar = 0;
0001 90000Au     14  dm(foc_integrators) = ar;
0002 90000Au     15  dm(foc_integrators + 1) = ar;
0003 90000Au     16  dm(foc_integrators + 2) = ar;
0004 90000Au     17  dm(foc_integrators + 3) = ar;
18
0005 48000A      19  ar = 32768;      !Executes 32768 loops.  On zero, loop exits.
0006 90000Au     20  dm(calibrate_loops) = ar;
21
0007 4FFFFFFA    22  ar = 65535;
0008 90000Au     23  dm(twang_loops) = ar;
24
0009 0A000F      25  rts;
26
27 .endmod;

```

Release 6.0 Assembler Version 2.0 11/7/97

C:\My Documents\thesis\code\main.dsp Wed Jan 18 18:05:40 2012 Page 1

addr inst source line

```

1  !*****
2  !
3  ! This routine performs power up initialisation of the
4  ! various hardware modules, ramps up the boot-strap supply,
5  ! and then enters the main loop.
6  !
7  ! Copyright 2012 MARTIN TODD HESS. All rights reserved
8  !
9  !*****

```

```

10 .module/ABS=0x60 main_module;
11
12 #include <main.h>
12
12 #define ADMC300
12 #include <ADMC300.h>
12 #ifndef ADMC300_INCLUDE
12 #define ADMC300_INCLUDE
12
12 {
12     This include file defines important ADMC300 addresses. The names
12     defined below can be used in user programs by "including" this file.
12     This file defines:
12         - names for the peripheral registers of the ADMC300
12         - names for the memory mapped core registers of the ADMC300
12         - interrupt vector table addresses
12 }
12
12 { peripheral registers of the ADMC300 }
12
12 .CONST PWMTM      = 0x2008;    {PWM timer register}
12 .CONST PWMDT      = 0x2009;    {PWM dead time register}
12 .CONST PWMPD      = 0x200A;    {PWM pulse deletion}
12 .CONST PWMGATE    = 0x200B;    {PWM gate register}
12 .CONST PWMCHA     = 0x200C;    {PWM channel A register}
12 .CONST PWMCHB     = 0x200D;    {PWM channel B register}
12 .CONST PWMCHC     = 0x200E;    {PWM channel C register}
12 .CONST PWMSEG     = 0x200F;    {PWM segment selection}
12 .CONST AUXTIM0    = 0x2010;    {Auxiliary PWM timer 0}
12 .CONST AUXTIM1    = 0x2011;    {Auxiliary PWM timer 1}
12 .CONST MODECTRL   = 0x2015;    {MODE control register}
12 .CONST SYSSTAT    = 0x2016;    {System status register}
12 .CONST WDTIMER    = 0x2018;    {Watchdog timer register}
12 .CONST PICVECTOR  = 0x201C;    {ISR address          }
12 .CONST PICMASK    = 0x201D;    {IRD mask regsiter   }
12 .CONST EIUCNT     = 0x2020;    {Encoder count register }
12 .CONST EIUMAXCNT  = 0x2021;    {Encoder max count register }

```

```

12 .CONST EIUSTAT    = 0x2022; {Encoder status register    }
12 .CONST EIUCTRL    = 0x2023; {Encoder control register   }
12 .CONST EIUPERIOD  = 0x2024; {Encoder loop timer period register}
12 .CONST EIUSCALE   = 0x2025; {Encoder loop timer scale register}
12 .CONST EIUTIMER   = 0x2026; {Encoder loop timer}
12 .CONST EETCNT     = 0x2027; {Latched value of EIUCNT register}
12 .CONST EETN       = 0x2028; {EET pulse decimator register }
12 .CONST EETDIV     = 0x2029; {EET timer decimator register }
12 .CONST EETDELTAT  = 0x202A; {EET delta timer register    }
12 .CONST EETT       = 0x202B; {EET timer period register   }
12 .CONST EETSTAT    = 0x202C; {EET overflow status register }
12 .CONST ADC1       = 0x2030; {ADC1 register              }
12 .CONST ADC2       = 0x2031; {ADC2 register              }
12 .CONST ADC3       = 0x2032; {ADC3 register              }
12 .CONST ADC4       = 0x2033; {ADC4 register              }

```

addr inst source line

```

12  .CONST  ADC5      =  0x2034;  {ADC5 register          }
12  .CONST  ADCCTRL   =  0x2036;  {ADC control register   }
12  .CONST  ADCSTAT   =  0x2037;  {ADC busy status register }
12  .CONST  ADCSYNC   =  0x2038;  {ADC synchronization register }
12  .CONST  ADCDIVA   =  0x2039;  {ADC bank A clock divide register }
12  .CONST  ADCDIVB   =  0x203A;  {ADC bank B clock divide register }
12  .CONST  ADCCAL    =  0x203B;  {ADC calibration register }
12  .CONST  PIOLEVEL  =  0x2040;  {PIO level control register }
12  .CONST  PIOMODE   =  0x2041;  {PIO mode control register }
12  .CONST  PIOPWM    =  0x2042;  {PIO PWMTRIP control register }
12  .CONST  PIODIR    =  0x2044;  {PIO direction register  }
12  .CONST  PIODATA   =  0x2045;  {PIO data register}
12  .CONST  PIOINTEN  =  0x2046;  {PIO interrupt enable register }
12  .CONST  PIOFLAG   =  0x2047;  {PIO interrupt flag register }
12  .CONST  ETUA0     =  0x2050;  {ETU0 Event A capture register }
12  .CONST  ETUB0     =  0x2051;  {ETU0 Event B capture register }
12  .CONST  ETUAA0    =  0x2052;  {ETU0 Event AA capture register }
12  .CONST  ETUA1     =  0x2053;  {ETU1 Event A capture register}
12  .CONST  ETUB1     =  0x2054;  {ETU1 Event B capture register}
12  .CONST  ETUAA1    =  0x2055;  {ETU1 Event AA capture register}
12  .CONST  ETUTIME   =  0x2056;  {ETU timer value}
12  .CONST  ETUCONFIG =  0x205C;  {ETU configuration register }
12  .CONST  ETUDIVIDE =  0x205D;  {ETU clock divide register }
12  .CONST  ETUSTAT   =  0x205E;  {ETU status register }
12  .CONST  ETUCTRL   =  0x205F;  {ETU control register }
12  .CONST  PWMSYNCWT =  0x2060;  {PWMSYNC width register}
12  .CONST  PWMSWT    =  0x2061;  {PWM software trip register}
12
12  { memory mapped core registers of the ADMC300 }
12
12  .CONST  SYSCNTL   =  0x3fff;
12  .CONST  MEMWAIT   =  0x3ffe;
12  .CONST  TPERIOD   =  0x3ffd;

```

```

12  .CONST  TCOUNT          =    0x3ffc;
12  .CONST  TSCALE           =    0x3ffb;
12  .CONST  Sport0_Rx_Words1  =    0x3ffa;
12  .CONST  Sport0_Rx_Words0  =    0x3ff9;
12  .CONST  Sport0_Tx_Words1  =    0x3ff8;
12  .CONST  Sport0_Tx_Words0  =    0x3ff7;
12  .CONST  Sport0_Ctrl_Reg    =    0x3ff6;
12  .CONST  Sport0_Sclkdiv     =    0x3ff5;
12  .CONST  Sport0_Rfsdiv      =    0x3ff4;
12  .CONST  Sport0_Autobuf_Ctrl =    0x3ff3;
12  .CONST  Sport1_Ctrl_Reg    =    0x3ff2;
12  .CONST  Sport1_Sclkdiv     =    0x3ff1;
12  .CONST  Sport1_Rfsdiv      =    0x3ff0;
12  .CONST  Sport1_Autobuf_Ctrl =    0x3fef;
12
12  { interrupt vector table addresses for ADMC300 }
12
12  .CONST  ADCBANKA_INT_ADDR  = 0x30+1;      {ADC BANKA INTERRUPT }
12  .CONST  PWMSYNC_INT_ADDR   = 0x34+1;      {PWMSYNC interrupt  }
12  .CONST  ADCBANKB_INT_ADDR  = 0x38+1;      {ADC BANKB interrupt }
12  .CONST  PIO_INT_ADDR       = 0x3C+1;      {PIO4 - PIO11 interrupt }
12  .CONST  EIU_INT_ADDR       = 0x40+1;      {EIU interrupt  }
12  .CONST  ETU_INT_ADDR       = 0x44+1;      {ETU interrupt  }
12  .CONST  PIO0_INT_ADDR      = 0x48+1;      {PIO0 interrupt  }
12  .CONST  PIO1_INT_ADDR      = 0x4C+1;      {PIO1 interrupt  }
12  .CONST  PIO2_INT_ADDR      = 0x50+1;      {PIO2 interrupt  }
12  .CONST  PIO3_INT_ADDR      = 0x54+1;      {PIO3 interrupt  }
12  .CONST  PWMTRIP_INT_ADDR   = 0x58+1;      {PWMTRIP interrupt}

```

addr inst source line

```

12  .CONST  TX0_INT_ADDR      = 0x10; {SPORT0 transmit interrupt}
12  .CONST  RX0_INT_ADDR      = 0x14; {SPORT0 receive interrupt}
12  .CONST  SW1_INT_ADDR      = 0x18;      {software interrupt 1}
12  .CONST  SW0_INT_ADDR      = 0x1C;      {software interrupt 0}
12  .CONST  TX1_INT_ADDR      = 0x20; {SPORT1 transmit interrupt}
12  .CONST  RX1_INT_ADDR      = 0x24; {SPORT1 receive interrupt}
12  .CONST  TIMER_INT_ADDR    = 0x28; {Timer interrupt}
12
12  #endif
12  #include <romutil.h>
12  {This include file contains the addresses for the ROM utility routines.
12   To use just include this file in the module where the routine will be called.
12   Then just call the routine name given below.
12
12  VERSION # 2.0      07/10/98
12
12  Name          Date      Description
12  ----          -
12  MOD/HISTORY:   TFH      07/10/98      Combined all targets into one file
12  }
12  #ifndef ROMUTIL_INCLUDE
12  #define ROMUTIL_INCLUDE
12
12  {-----ADM300-----}
12  #ifdef ADM300
12  {ROM Utility Start Locations}
12  .CONST PER_RST      = 0x07E4;
12  .CONST UMASK        = 0x0E21;
12  .CONST PUT_VECTOR   = 0x0E28;
12  .CONST SMASK        = 0x0E35;
12  .CONST ADMC_COS     = 0x0E55;
12  .CONST ADMC_SIN     = 0x0E5C;
12  .CONST ARCTAN       = 0x0E72;

```

```

12  .CONST RECIP_NEW = 0x0E94; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
12  .CONST SQRT = 0x0EAA;
12  .CONST LOG = 0x0EE7;
12  .CONST LN = 0x0EE4;
12  .CONST FLTONE = 0x0F03;
12  .CONST FIXONE = 0x0F08;
12  .CONST FPA = 0x0F0C;
12  .CONST FPS = 0x0F1B;
12  .CONST FPM = 0x0F2B;
12  .CONST FPD = 0x0F34;
12  .CONST FPMACC = 0x0F55;
12  .CONST REV_FOR_PARK = 0x0F77;
12  .CONST REV_CLARK = 0x0F8B;
12  .CONST FOR_CLARK = 0x0FA1;
12  .CONST SDIVQINT = 0x0FAB;
12  .CONST SDIVQ = 0x0FB4;
12  .CONST EXIT = 0x0FC6;
12
12  {Monitor Variables}
12  .CONST I4_SAVE = 0x3bbf;
12  #endif
12
12  {-----ADMC330-----}
12  #ifdef ADMC330
12  {ROM Utility Start Locations}
12  .CONST PUT_VECTOR = 0x0B73;
12  .CONST SIN12B = 0x0DC4;
12  .CONST COS12B = 0x0DD3;

```


addr inst source line

```

12  .CONST  CAL_INV_TAN      = 0x0E61;
12  .CONST  LOG1             = 0x0E14;
12  .CONST  ALOG1           = 0x0E20;
12  .CONST  SQRT1           = 0x0E27;
12  .CONST  DIVIDE          = 0x0DE2;
12  .CONST  X_Y_TO_ANG      = 0x0E35;
12  .CONST  APPROX10_ST     = 0x0DEC;
12  .CONST  APPROX_N_ST     = 0x0DFF;
12  .CONST  COS64           = 0x0B80;
12  .CONST  COS6_64        = 0x0C00;
12  .CONST  SIN6_64        = 0x0C40;
12  .CONST  ATN64           = 0x0D01;
12  .CONST  LOG64           = 0x0D42;
12  .CONST  ALOG64         = 0x0D83;
12
12  {Monitor Variables}
12  .CONST  HALT_FLAG        = 0x3B9A;    {used by exit library function}
12  .CONST  IDE_SP           = 0x79E;     {used by exit library function}
12  .CONST  IRQFLAG_SAVE     = 0x3BFE;    {IRQFLAG saved here each interrupt}
12  #endif
12
12  {-----ADMC331-----}
12  #ifdef ADMC331
12  {ROM Utility Start Locations}
12  .CONST    PER_RST        = 0X07F1;
12  .CONST    UMASK = 0X0DED;
12  .CONST    PUT_VECTOR    = 0X0DF4;
12  .CONST    SMASK = 0X0E06;
12  .CONST    ADMC_COS      = 0X0E26;
12  .CONST    ADMC_SIN      = 0X0E2D;
12  .CONST    ARCTAN        = 0X0E43;
12  .CONST    RECIP_NEW     = 0X0E65;    {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
12  .CONST    SQRT_NEW      = 0X0E7C;    {For SQRT - call SQRT in mathfix.dsp}

```

```

12  .CONST      LN_NEW      = 0X0EBA;  {For LN   - call LN in mathfix.dsp}
12  .CONST      LOG_NEW     = 0X0EB9;  {For LOG  - call LOG in mathfix.dsp}
12  .CONST      FLTONE      = 0X0ED4;
12  .CONST      FIXONE      = 0X0ED9;
12  .CONST      FPA        = 0X0EDD;
12  .CONST      FPS        = 0X0EEC;
12  .CONST      FPM        = 0X0EFC;
12  .CONST      FPD        = 0X0F05;
12  .CONST      FPMACC      = 0X0F26;
12  .CONST      PARK       = 0X0F48;
12  .CONST      REV_CLARK   = 0X0F5C;
12  .CONST      FOR_CLARK   = 0X0F72;
12  .CONST      COS64      = 0X0F80;
12  .CONST      ONE_BY_X    = 0X0FC0;
12  .CONST      SDIVQINT    = 0X0FD0;
12  .CONST      SDIVQ      = 0X0FD9;
12
12  {Monitor Variables}
12  .CONST HALT_FLAG = 0x3B68;  {used by exit library function}
12  .CONST IDE_SP     = 0x7F7;   {used by exit library function}
12  .CONST IRQFLAG_SAVE = 0x3B88; {IRQFLAG saved here each interrupt}
12  #endif
12
12  {-----ADMC328 Emulation Board-----}
12  #ifdef ADMC328E
12  {ROM Utility Start Locations}
12  .CONST      PER_RST      = 0X07F1;
12  .CONST      UMASK       = 0X0DED;

```

addr inst source line

```

12 .CONST PUT_VECTOR = 0X0DF4;
12 .CONST SMASK = 0X0E06;
12 .CONST ADMC_COS = 0X0E26;
12 .CONST ADMC_SIN = 0X0E2D;
12 .CONST ARCTAN = 0X0E43;
12 .CONST RECIP_NEW = 0X0E65; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
12 .CONST SQRT_NEW = 0X0E7C; {For SQRT - call SQRT in mathfix.dsp}
12 .CONST LN_NEW = 0X0EBA; {For LN - call LN in mathfix.dsp}
12 .CONST LOG_NEW = 0X0EB9; {For LOG - call LOG in mathfix.dsp}
12 .CONST FLTONE = 0X0ED4;
12 .CONST FIXONE = 0X0ED9;
12 .CONST FPA = 0X0EDD;
12 .CONST FPS = 0X0EEC;
12 .CONST FPM = 0X0EFC;
12 .CONST FPD = 0X0F05;
12 .CONST FPMACC = 0X0F26;
12 .CONST PARK = 0X0F48;
12 .CONST REV_CLARK = 0X0F5C;
12 .CONST FOR_CLARK = 0X0F72;
12 .CONST COS64 = 0X0F80;
12 .CONST ONE_BY_X = 0X0FC0;
12 .CONST SDIVQINT = 0X0FD0;
12 .CONST SDIVQ = 0X0FD9;
12
12 {Monitor Variables}
12 .CONST HALT_FLAG = 0x3B68; {used by exit library function}
12 .CONST IDE_SP = 0x7F7; {used by exit library function}
12 .CONST IRQFLAG_SAVE = 0x3B88; {IRQFLAG saved here each interrupt}
12 #endif
12
12 {-----ADMC328-----}
12 #ifdef ADMC328
12 {ROM Utility Start Locations}

```

```

12 .CONST      PER_RST      = 0X01F1;
12 .CONST      UMASK = 0X0F15;
12 .CONST      PUT_VECTOR  = 0X0F1C;
12 .CONST      SMASK = 0X0F29;
12 .CONST      SDIVQINT    = 0X0F30;
12 .CONST      SDIVQ = 0X0F39;
12
12 {Monitor Variables}
12 .CONST IRQFLAG_SAVE      = 0x3988; {IRQFLAG saved here each interrupt}
12 #endif
12
12 {-----ADMC201-----}
12 #ifdef ADMC201
12 {ROM Utility Start Locations}
12 .CONST PUT_VECTOR      = 0x0573;
12 .CONST COS64           = 0x0580;
12 .CONST ONE_BY_X        = 0x05C0;
12
12 {Monitor Variables}
12 .CONST IRQFLAG_SAVE      = 0x3BFE; {IRQFLAG saved here each interrupt}
12 #endif
12
12 #endif
12
12
12
12
12 .external adca_int;

```

[illegible]

```

13  #define twang_angle 1000
14
15
16  POWER_UP:
17
18  !*** Init variables ***
19
0000 1C000Fu      20  call init_vars;
21
22
23  !*** Init SPORT0 as UART ***
24
0001 1C000Fu      25  call INIT_UART0;
26
27  !*** Ramp up boot-strap supply ***
28
0002 40000A      29  ar = 0;    !ar will count up to duty cycle
0003 400334      30  ay0 = 51; !defined in ay0 as 20% (256*0.2)
31
0004 140001u     32  do RAMP_BOOT_STRAP until eq;
0005 3C0085      33  CNTR = 8; !Delay 8*16383 between each step in duty cycle
0006 14000Eu     34  do BOOT_DELAY_1 until CE;
0007 3FFFF5      35  CNTR = 16383;
0008 14000Eu     36  do BOOT_DELAY_2 until CE;
0009 000000      37  nop;
38
39  BOOT_DELAY_2:
000A 000000      40  nop;

```

addr inst source line

```

      41
      42 BOOT_DELAY_1:
000B 000000      43  nop;
000C 22621F      44  ar = ar + 1;
000D 2AE2AA      45  none = ar - ay0;
      46
      47 RAMP_BOOT_STRAP:
000E 92011A      48  dm(AUXTIM1) = ar;      !This is the duty cycle register
      49
      50
      51 !*** Initilize I/O ***
      52
000F 40000A      53  ar = 0;      !Disable PWM trips from PIO
0010 92042A      54  dm(PIOPWM) = ar;
      55
      56
      57 !*** Initilize main PWM output ***
      58
0011 380350      59  i4 = PWMSYNC_INT_ADDR;      !Initilize interupt vector
0012 40000Bu      60  mr0 = ^PWMSYNC_INT;
0013 1CE28F      61  call PUT_VECTOR;
      62
0014 82036A      63  ar = dm(ADCCTRL);      !Clear desat faults
0015 23AA1F      64  ar = setbit 4 of ar;
0016 92036A      65  dm(ADCCTRL) = ar;
      66
0017 0D03A3      67  ar = IMASK;      !Enable pwmsync interupts
0018 23B25F      68  ar = setbit 9 of ar;
0019 0D0C3A      69  IMASK = ar;
001A 8201DA      70  ar = dm(PICMASK);
001B 23A25F      71  ar = setbit 1 of ar;
001C 9201DA      72  dm(PICMASK) = ar;
      73

```

```

001D 4000CA      74  ar=12;      !Dead time = 1/12.5MHz*12 = 960ns
001E 92009A      75  dm(PWMDT)=ar;
      76
001F 82015A      77  ar=dm(MODECTRL);!Set bit 5 to enable double update mode
0020 23AA5F      78  ar=setbit 5 of ar;
0021 92015A      79  dm(MODECTRL)=ar;
      80
0022 40500A      81  ar=pwmtm_default;      !PWM interrupt runs at clkin (12.5MHz) / 640 = 19.531KHz
0023 92008A      82  dm(PWMTM)=ar;
      83
0024 402804      84  ay0 = pwmtm_default>>1;      !Write each once to start the PWM at 50
0025 9200C4      85  dm(PWMCHA)=ay0;
0026 9200D4      86  dm(PWMCHB)=ay0;
0027 9200E4      87  dm(PWMCHC)=ay0;
      88
      89
      90  !*** Initilize Encoder Interface ***
      91
0028 40F9FA      92  ar = encoder_roll;
0029 92021A      93  dm(EIUMAXCNT) = ar;
002A 40001A      94  ar = 1;
002B 92023A      95  dm(EIUCTRL) = ar;
      96
      97  !*** Set the regen clamp to 50% to test ***
      98
002C 40080A      99  ar=128;
002D 92010A     100  dm(AUXTIM0)=ar;
      101

```


addr inst source line

```

    102
    103  !***  Initilize ADC ***
    104
002E 380310      105  i4=ADCBANKA_INT_ADDR; !Initilize interupt vector table
002F 40000Bu     106  mr0=^ADCA_INT;
0030 1CE28F      107  call PUT_VECTOR;
0031 380390      108  i4=ADCBANKB_INT_ADDR;
0032 40000Bu     109  mr0=^ADCB_INT;
0033 1CE28F      110  call PUT_VECTOR;
    111
    112
    113  !***  Setup read and write queues ***
    114
0034 340004      115  m0=0;      !Default values for DAG modifiers
0035 340015      116  m1=1;
0036 37FFF6      117  m2=-1;
    118
0037 340000u     119  i0=^RD_QUEUE;  !Pointers to next empty cell.
0038 340408      120  l0=%RD_QUEUE;
0039 340002u     121  i2=^WR_QUEUE;
003A 34040A      122  l2=%WR_QUEUE;
003B 0D0532      123  i3=i2;
003C 0D05BA      124  l3=l2;
    125
003D 340001u     126  i1 = ^data_queue;
003E 342009      127  l1 = %data_queue;
    128
    129
003F 8201DA      130  ar=dm(PICMASK); !Enable interrupts
0040 23A21F      131  ar=setbit 0 of ar;
0041 23A29F      132  ar=setbit 2 of ar;
0042 9201DA      133  dm(PICMASK)=ar;
    134

```

```

0043 82036A      135  ar=dm(ADCCTRL);
0044 401804      136  ay0=0x0180;      !Force PWM sync
0045 23A20F      137  ar=ar or ay0;
0046 4F3FF4      138  ay0=0xF3FF;      !Turn on
0047 23820F      139  ar=ar and ay0;
0048 92036A      140  dm(ADCCTRL)=ar;
0049 40280A      141  ar=640;      !Run at same rate as PWM
004A 92039A      142  dm(ADCDIVA)=ar;
004B 9203AA      143  dm(ADCDIVB)=ar;
      144
      145
      146
      147
      148  !*****
      149  !*      *
      150  !* This is the main loop.  Processes serial data.  *
      151  !*      *
      152  !*****
      153
      154  main:
004C 1C00Fu      155  call host_receive;
004D 1C00Fu      156  call host_transmit;
004E 1800Fu      157  jump main;
      158
      159  !*****
      160  !*      *
      161  !* End of main      *
      162  !*      *

```

Release 6.0 Assembler Version 2.0 11/7/97

C:\My Documents\thesis\code\main.dsp Wed Jan 18 18:05:40 2012 Page 9

addr inst source line

```
163 !*****
164
165 .endmod;
166
167
168
169
```

Release 6.0 Assembler Version 2.0 11/7/97

C:\My Documents\thesis\code\pwm_isr.dsp Wed Jan 18 18:08:23 2012 Page 1

addr inst source line

```
1 !*****
2 !
3 ! This is the PWM module Interrupt Service routine.
4 ! It reads the EIU, scales appropriately, and then computes
5 ! the sine and cosine of electrical angle
6 !
7 ! Copyright 2012 MARTIN TODD HESS. All rights reserved
8 !
9 !*****
10 .module pwm_isr;
11
12 #define ADMC300
13 #include <ADMC300.h>
13 #ifndef ADMC300_INCLUDE
13 #define ADMC300_INCLUDE
13
13 {
13 This include file defines important ADMC300 addresses. The names
13 defined below can be used in user programs by "including" this file.
13 This file defines:
```

```

13      - names for the peripheral registers of the ADMC300
13      - names for the memory mapped core registers of the ADMC300
13      - interrupt vector table addresses
13  }
13
13  { peripheral registers of the ADMC300 }
13
13  .CONST PWMTM      = 0x2008;      {PWM timer register}
13  .CONST PWMDT      = 0x2009;      {PWM dead time register}
13  .CONST PWMPD      = 0x200A;      {PWM pulse deletion}
13  .CONST PWMGATE     = 0x200B;      {PWM gate register}
13  .CONST PWMCHA      = 0x200C;      {PWM channel A register}
13  .CONST PWMCHB      = 0x200D;      {PWM channel B register}
13  .CONST PWMCHC      = 0x200E;      {PWM channel C register}
13  .CONST PWMSEG      = 0x200F;      {PWM segment selection}
13  .CONST AUXTIM0     = 0x2010;      {Auxiliary PWM timer 0}
13  .CONST AUXTIM1     = 0x2011;      {Auxiliary PWM timer 1}
13  .CONST MODECTRL    = 0x2015;      {MODE control register}
13  .CONST SYSSTAT     = 0x2016;      {System status register}
13  .CONST WDTIMER     = 0x2018;      {Watchdog timer register}
13  .CONST PICVECTOR   = 0x201C;      {ISR address          }
13  .CONST PICMASK     = 0x201D;      {IRD mask regsiter    }
13  .CONST EIUCNT      = 0x2020;      {Encoder count register }
13  .CONST EIUMAXCNT   = 0x2021;      {Encoder max count register }
13  .CONST EIUSTAT     = 0x2022;      {Encoder status register  }
13  .CONST EIUCTRL     = 0x2023;      {Encoder control register  }
13  .CONST EIUPERIOD   = 0x2024;      {Encoder loop timer period register}
13  .CONST EIUSCALE    = 0x2025;      {Encoder loop timer scale register}
13  .CONST EIUTIMER    = 0x2026;      {Encoder loop timer}
13  .CONST EETCNT      = 0x2027;      {Latched value of EIUCNT register}
13  .CONST EETN        = 0x2028;      {EET pulse decimator register }
13  .CONST EETDIV      = 0x2029;      {EET timer decimator register }
13  .CONST EETDELTAT   = 0x202A;      {EET delta timer register   }
13  .CONST EETT        = 0x202B;      {EET timer period register   }
13  .CONST EETSTAT     = 0x202C;      {EET overflow status register }
13  .CONST ADC1        = 0x2030;      {ADC1 register            }
13  .CONST ADC2        = 0x2031;      {ADC2 register            }

```

```
13  .CONST  ADC3      =    0x2032;  {ADC3 register      }
13  .CONST  ADC4      =    0x2033;  {ADC4 register      }
13  .CONST  ADC5      =    0x2034;  {ADC5 register      }
13  .CONST  ADCCTRL   =    0x2036;  {ADC control register }
```

addr inst source line

```

13  .CONST  ADCSTAT    =    0x2037;    {ADC busy status register }
13  .CONST  ADCSYNC    =    0x2038;    {ADC synchronization register }
13  .CONST  ADCDIVA    =    0x2039;    {ADC bank A clock divide register }
13  .CONST  ADCDIVB    =    0x203A;    {ADC bank B clock divide register }
13  .CONST  ADCCAL     =    0x203B;    {ADC calibration register }
13  .CONST  PIOLEVEL   =    0x2040;    {PIO level control register }
13  .CONST  PIOMODE    =    0x2041;    {PIO mode control register }
13  .CONST  PIOPWM     =    0x2042;    {PIO PWMTRIP control register }
13  .CONST  PIODIR     =    0x2044;    {PIO direction register }
13  .CONST  PIODATA    =    0x2045;    {PIO data register}
13  .CONST  PIOINTEN   =    0x2046;    {PIO interrupt enable register }
13  .CONST  PIOFLAG    =    0x2047;    {PIO interrupt flag register }
13  .CONST  ETUA0      =    0x2050;    {ETU0 Event A capture register }
13  .CONST  ETUB0      =    0x2051;    {ETU0 Event B capture register }
13  .CONST  ETUAA0     =    0x2052;    {ETU0 Event AA capture register }
13  .CONST  ETUA1      =    0x2053;    {ETU1 Event A capture register}
13  .CONST  ETUB1      =    0x2054;    {ETU1 Event B capture register}
13  .CONST  ETUAA1     =    0x2055;    {ETU1 Event AA capture register}
13  .CONST  ETUTIME    =    0x2056;    {ETU timer value}
13  .CONST  ETUCONFIG  =    0x205C;    {ETU configuration register }
13  .CONST  ETUDIVIDE  =    0x205D;    {ETU clock divide register }
13  .CONST  ETUSTAT    =    0x205E;    {ETU status register }
13  .CONST  ETUCTRL    =    0x205F;    {ETU control register }
13  .CONST  PWMSYNCWT  =    0x2060;    {PWMSYNC width register}
13  .CONST  PWMSWT     =    0x2061;    {PWM software trip register}
13
13  { memory mapped core registers of the ADMC300 }
13
13  .CONST  SYSCNTL    =    0x3fff;
13  .CONST  MEMWAIT    =    0x3ffe;
13  .CONST  TPERIOD     =    0x3ffd;
13  .CONST  TCOUNT      =    0x3ffc;
13  .CONST  TSCALE     =    0x3ffb;

```

```

13  .CONST  Sport0_Rx_Words1      =   0x3ffa;
13  .CONST  Sport0_Rx_Words0      =   0x3ff9;
13  .CONST  Sport0_Tx_Words1      =   0x3ff8;
13  .CONST  Sport0_Tx_Words0      =   0x3ff7;
13  .CONST  Sport0_Ctrl_Reg        =   0x3ff6;
13  .CONST  Sport0_Sclkdiv         =   0x3ff5;
13  .CONST  Sport0_Rfsdiv          =   0x3ff4;
13  .CONST  Sport0_Autobuf_Ctrl    =   0x3ff3;
13  .CONST  Sport1_Ctrl_Reg        =   0x3ff2;
13  .CONST  Sport1_Sclkdiv         =   0x3ff1;
13  .CONST  Sport1_Rfsdiv          =   0x3ff0;
13  .CONST  Sport1_Autobuf_Ctrl    =   0x3fef;
13
13  { interrupt vector table addresses for ADMC300 }
13
13  .CONST  ADCBANKA_INT_ADDR      = 0x30+1;      {ADC BANKA INTERRUPT }
13  .CONST  PWMSYNC_INT_ADDR       = 0x34+1;      {PWMSYNC interrupt  }
13  .CONST  ADCBANKB_INT_ADDR      = 0x38+1;      {ADC BANKB interrupt }
13  .CONST  PIO_INT_ADDR           = 0x3C+1;      {PIO4 - PIO11 interrupt }
13  .CONST  EIU_INT_ADDR           = 0x40+1;      {EIU interrupt  }
13  .CONST  ETU_INT_ADDR           = 0x44+1;      {ETU interrupt  }
13  .CONST  PIO0_INT_ADDR          = 0x48+1;      {PIO0 interrupt  }
13  .CONST  PIO1_INT_ADDR          = 0x4C+1;      {PIO1 interrupt  }
13  .CONST  PIO2_INT_ADDR          = 0x50+1;      {PIO2 interrupt  }
13  .CONST  PIO3_INT_ADDR          = 0x54+1;      {PIO3 interrupt  }
13  .CONST  PWMTRIP_INT_ADDR       = 0x58+1;      {PWMTRIP interrupt}
13  .CONST  TX0_INT_ADDR           = 0x10; {SPORT0 transmit interrupt}
13  .CONST  RX0_INT_ADDR           = 0x14; {SPORT0 receive interrupt}

```

addr inst source line

```

13  .CONST  SW1_INT_ADDR      = 0x18;      {software interrupt 1}
13  .CONST  SW0_INT_ADDR      = 0x1C;      {software interrupt 0}
13  .CONST  TX1_INT_ADDR      = 0x20; {SPORT1 transmit interrupt}
13  .CONST  RX1_INT_ADDR      = 0x24; {SPORT1 receive interrupt}
13  .CONST  TIMER_INT_ADDR    = 0x28; {Timer interrupt}
13
13  #endif
14  #include <romutil.h>
14  {This include file contains the addresses for the ROM utility routines.
14   To use just include this file in the module where the routine will be called.
14   Then just call the routine name given below.
14
14  VERSION # 2.0      07/10/98
14
14  Name          Date      Description
14  ----          -
14  MOD/HISTORY:   TFH      07/10/98      Combined all targets into one file
14  }
14  #ifndef ROMUTIL_INCLUDE
14  #define ROMUTIL_INCLUDE
14
14  {-----ADMC300-----}
14  #ifdef ADMC300
14  {ROM Utility Start Locations}
14  .CONST PER_RST      = 0x07E4;
14  .CONST UMASK        = 0x0E21;
14  .CONST PUT_VECTOR= 0x0E28;
14  .CONST SMASK        = 0x0E35;
14  .CONST ADMC_COS     = 0x0E55;
14  .CONST ADMC_SIN     = 0x0E5C;
14  .CONST ARCTAN       = 0x0E72;
14  .CONST RECIP_NEW    = 0x0E94; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
14  .CONST SQRT = 0x0EAA;

```



```

14  .CONST LOG = 0x0EE7;
14  .CONST LN  = 0x0EE4;
14  .CONST FLTONE      = 0x0F03;
14  .CONST FIXONE      = 0x0F08;
14  .CONST FPA  = 0x0F0C;
14  .CONST FPS  = 0x0F1B;
14  .CONST FPM  = 0x0F2B;
14  .CONST FPD  = 0x0F34;
14  .CONST FPMACC      = 0x0F55;
14  .CONST REV_FOR_PARK    = 0x0F77;
14  .CONST REV_CLARK      = 0x0F8B;
14  .CONST FOR_CLARK      = 0x0FA1;
14  .CONST SDIVQINT  = 0x0FAB;
14  .CONST SDIVQ      = 0x0FB4;
14  .CONST EXIT = 0x0FC6;
14
14  {Monitor Variables}
14  .CONST I4_SAVE  = 0x3bbf;
14  #endif
14
14  {-----ADMC330-----}
14  #ifdef ADMC330
14  {ROM Utility Start Locations}
14  .CONST PUT_VECTOR      = 0x0B73;
14  .CONST SIN12B          = 0x0DC4;
14  .CONST COS12B          = 0x0DD3;
14  .CONST CAL_INV_TAN      = 0x0E61;
14  .CONST LOG1            = 0x0E14;

```

addr inst source line

```

14  .CONST  ALOG1          = 0x0E20;
14  .CONST  SQRT1          = 0x0E27;
14  .CONST  DIVIDE         = 0x0DE2;
14  .CONST  X_Y_TO_ANG     = 0x0E35;
14  .CONST  APPROX10_ST    = 0x0DEC;
14  .CONST  APPROX_N_ST    = 0x0DFF;
14  .CONST  COS64          = 0x0B80;
14  .CONST  COS6_64        = 0x0C00;
14  .CONST  SIN6_64        = 0x0C40;
14  .CONST  ATN64          = 0x0D01;
14  .CONST  LOG64          = 0x0D42;
14  .CONST  ALOG64         = 0x0D83;
14
14  {Monitor Variables}
14  .CONST  HALT_FLAG       = 0x3B9A;    {used by exit library function}
14  .CONST  IDE_SP          = 0x79E;     {used by exit library function}
14  .CONST  IRQFLAG_SAVE    = 0x3BFE;    {IRQFLAG saved here each interrupt}
14  #endif
14
14  {-----ADMC331-----}
14  #ifdef ADMC331
14  {ROM Utility Start Locations}
14  .CONST    PER_RST       = 0X07F1;
14  .CONST    UMASK = 0X0DED;
14  .CONST    PUT_VECTOR    = 0X0DF4;
14  .CONST    SMASK = 0X0E06;
14  .CONST    ADMC_COS      = 0X0E26;
14  .CONST    ADMC_SIN      = 0X0E2D;
14  .CONST    ARCTAN        = 0X0E43;
14  .CONST    RECIP_NEW     = 0X0E65;    {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
14  .CONST    SQRT_NEW      = 0X0E7C;    {For SQRT - call SQRT in mathfix.dsp}
14  .CONST    LN_NEW        = 0X0EBA;    {For LN - call LN in mathfix.dsp}
14  .CONST    LOG_NEW       = 0X0EB9;    {For LOG - call LOG in mathfix.dsp}

```

```

14  .CONST      FLTONE      = 0X0ED4;
14  .CONST      FIXONE      = 0X0ED9;
14  .CONST      FPA        = 0X0EDD;
14  .CONST      FPS        = 0X0EEC;
14  .CONST      FPM        = 0X0EFC;
14  .CONST      FPD        = 0X0F05;
14  .CONST      FPMACC      = 0X0F26;
14  .CONST      PARK       = 0X0F48;
14  .CONST      REV_CLARK   = 0X0F5C;
14  .CONST      FOR_CLARK   = 0X0F72;
14  .CONST      COS64      = 0X0F80;
14  .CONST      ONE_BY_X    = 0X0FC0;
14  .CONST      SDIVQINT    = 0X0FD0;
14  .CONST      SDIVQ      = 0X0FD9;
14
14  {Monitor Variables}
14  .CONST HALT_FLAG = 0x3B68;    {used by exit library function}
14  .CONST IDE_SP     = 0x7F7;    {used by exit library function}
14  .CONST IRQFLAG_SAVE = 0x3B88; {IRQFLAG saved here each interrupt}
14  #endif
14
14  {-----ADMC328 Emulation Board-----}
14  #ifdef ADMC328E
14  {ROM Utility Start Locations}
14  .CONST      PER_RST      = 0X07F1;
14  .CONST      UMASK      = 0X0DED;
14  .CONST      PUT_VECTOR  = 0X0DF4;
14  .CONST      SMASK      = 0X0E06;

```

addr inst source line

```

14  .CONST      ADMC_COS      = 0X0E26;
14  .CONST      ADMC_SIN      = 0X0E2D;
14  .CONST      ARCTAN        = 0X0E43;
14  .CONST      RECIP_NEW     = 0X0E65; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
14  .CONST      SQRT_NEW      = 0X0E7C; {For SQRT - call SQRT in mathfix.dsp}
14  .CONST      LN_NEW        = 0X0EBA; {For LN - call LN in mathfix.dsp}
14  .CONST      LOG_NEW       = 0X0EB9; {For LOG - call LOG in mathfix.dsp}
14  .CONST      FLTONE        = 0X0ED4;
14  .CONST      FIXONE        = 0X0ED9;
14  .CONST      FPA           = 0X0EDD;
14  .CONST      FPS           = 0X0EEC;
14  .CONST      FPM           = 0X0EFC;
14  .CONST      FPD           = 0X0F05;
14  .CONST      FPMACC        = 0X0F26;
14  .CONST      PARK          = 0X0F48;
14  .CONST      REV_CLARK     = 0X0F5C;
14  .CONST      FOR_CLARK     = 0X0F72;
14  .CONST      COS64         = 0X0F80;
14  .CONST      ONE_BY_X      = 0X0FC0;
14  .CONST      SDIVQINT      = 0X0FD0;
14  .CONST      SDIVQ         = 0X0FD9;
14
14  {Monitor Variables}
14  .CONST HALT_FLAG = 0x3B68; {used by exit library function}
14  .CONST IDE_SP     = 0x7F7; {used by exit library function}
14  .CONST IRQFLAG_SAVE = 0x3B88; {IRQFLAG saved here each interrupt}
14  #endif
14
14  {-----ADMC328-----}
14  #ifdef ADMC328
14  {ROM Utility Start Locations}
14  .CONST      PER_RST       = 0X01F1;
14  .CONST      UMASK         = 0X0F15;

```

```

14  .CONST      PUT_VECTOR = 0X0F1C;
14  .CONST      SMASK = 0X0F29;
14  .CONST      SDIVQINT  = 0X0F30;
14  .CONST      SDIVQ = 0X0F39;
14
14  {Monitor Variables}
14  .CONST IRQFLAG_SAVE    = 0x3988; {IRQFLAG saved here each interrupt}
14  #endif
14
14  {-----ADMC201-----}
14  #ifdef ADMC201
14  {ROM Utility Start Locations}
14  .CONST PUT_VECTOR      = 0x0573;
14  .CONST COS64           = 0x0580;
14  .CONST ONE_BY_X        = 0x05C0;
14
14  {Monitor Variables}
14  .CONST IRQFLAG_SAVE    = 0x3BFE; {IRQFLAG saved here each interrupt}
14  #endif
14
14  #endif
14
14
15  #include <defines.h>
15  #define pwmtm_default 0x500
15  #define magnitude_scale 0x280 /* sqrt(3)/3 * pwmtm_default / 2 */
15  #define third_harmonic_scale 0x55 /* magnitude_scale / 3 */

```

addr inst source line

```

15 #define encoder_roll 3999
15 #define pole_pairs 3
15 #define encoder_scale 49      /* 65536 / encoder_roll * pole_pairs */
15 #define twang_angle 1000
16 #include <pwm_isr.h>
16 .external ang_flux;
16
16 .external ang_cos;
16 .external ang_sin;
16
17
18
19 .entry pwmsync_int;
20 PWMSYNC_INT:
0000 0C0030      21  ena sec_reg;      !Context switch
      22
0001 820202      23  mx0 = dm(eiucnt);
0002 400316      24  my0 = encoder_scale;
0003 0C3000      25  ena m_mode;
0004 20E00F      26  mr  = mx0 * my0 (uu);
0005 0C2000      27  dis m_mode;
0006 90000Bu     28  dm(ang_flux) = mr0;
      29
0007 0D000B      30  ax0 = mr0;
0008 1CE55F      31  call ADMC_COS;
0009 90000Au     32  dm(ang_cos) = ar;
000A 1CE5CF      33  call ADMC_SIN;
000B 90000Au     34  dm(ang_sin) = ar;
      35
000C 0C0020      36  dis sec_reg;
      37
000D 0A001F      38  rti;
      39

```

```
40 .endmod;
41
```

Release 6.0 Assembler Version 2.0 11/7/97

C:\My Documents\thesis\code\uart0.dsp Wed Jan 18 18:08:23 2012 Page 1

addr inst source line

```
1
2 .module uart0;
3
4 #define ADMC300
5 #include <ADMC300.h>
5 #ifndef ADMC300_INCLUDE
5 #define ADMC300_INCLUDE
5
5 {
5     This include file defines important ADMC300 addresses. The names
5     defined below can be used in user programs by "including" this file.
5     This file defines:
5         - names for the peripheral registers of the ADMC300
5         - names for the memory mapped core registers of the ADMC300
5         - interrupt vector table addresses
5 }
5
5 { peripheral registers of the ADMC300 }
5
5 .CONST PWMTM      = 0x2008;      {PWM timer register}
5 .CONST PWMDT      = 0x2009;      {PWM dead time register}
5 .CONST PWMPD      = 0x200A;      {PWM pulse deletion}
5 .CONST PWMGATE    = 0x200B;      {PWM gate register}
5 .CONST PWMCHA     = 0x200C;      {PWM channel A register}
5 .CONST PWMCHB     = 0x200D;      {PWM channel B register}
5 .CONST PWMCHC     = 0x200E;      {PWM channel C register}
5 .CONST PWMSEG     = 0x200F;      {PWM segment selection}
5 .CONST AUXTIM0    = 0x2010;      {Auxiliary PWM timer 0}
5 .CONST AUXTIM1    = 0x2011;      {Auxiliary PWM timer 1}
```

```

5 .CONST MODECTRL = 0x2015;    {MODE control register}
5 .CONST SYSSTAT = 0x2016;    {System status register}
5 .CONST WDTIMER = 0x2018;    {Watchdog timer register}
5 .CONST PICVECTOR = 0x201C;  {ISR address          }
5 .CONST PICMASK = 0x201D;    {IRD mask regsiter    }
5 .CONST EIUCNT = 0x2020;     {Encoder count register }
5 .CONST EIUMAXCNT = 0x2021;  {Encoder max count register }
5 .CONST EIUSTAT = 0x2022;    {Encoder status register }
5 .CONST EIUCTRL = 0x2023;    {Encoder control register }
5 .CONST EIUPERIOD = 0x2024;  {Encoder loop timer period register}
5 .CONST EIUSCALE = 0x2025;   {Encoder loop timer scale register}
5 .CONST EIUTIMER = 0x2026;   {Encoder loop timer}
5 .CONST EETCNT = 0x2027;     {Latched value of EIUCNT register}
5 .CONST EETN = 0x2028;       {EET pulse decimator register }
5 .CONST EETDIV = 0x2029;     {EET timer decimator register }
5 .CONST EETDELTAT = 0x202A;  {EET delta timer register  }
5 .CONST EETT = 0x202B;       {EET timer period register  }
5 .CONST EETSTAT = 0x202C;    {EET overflow status register }
5 .CONST ADC1 = 0x2030;       {ADC1 register           }
5 .CONST ADC2 = 0x2031;       {ADC2 register           }
5 .CONST ADC3 = 0x2032;       {ADC3 register           }
5 .CONST ADC4 = 0x2033;       {ADC4 register           }
5 .CONST ADC5 = 0x2034;       {ADC5 register           }
5 .CONST ADCCTRL = 0x2036;    {ADC control register    }
5 .CONST ADCSTAT = 0x2037;    {ADC busy status register }
5 .CONST ADCSYNC = 0x2038;    {ADC synchronization register }
5 .CONST ADCDIVA = 0x2039;    {ADC bank A clock divide register }
5 .CONST ADCDIVB = 0x203A;    {ADC bank B clock divide register }
5 .CONST ADCCAL = 0x203B;     {ADC calibration register }
5 .CONST PIOLEVEL = 0x2040;   {PIO level control register }
5 .CONST PIOMODE = 0x2041;    {PIO mode control register }
5 .CONST PIOPWM = 0x2042;    {PIO PWMTRIP control register }

```


addr inst source line

```

5  .CONST  PIODIR      =    0x2044;    {PIO direction register      }
5  .CONST  PIODATA     =    0x2045;    {PIO data register}
5  .CONST  PIOINTEN    =    0x2046;    {PIO interrupt enable register }
5  .CONST  PIOFLAG     =    0x2047;    {PIO interupt flag register  }
5  .CONST  ETUA0       =    0x2050;    {ETU0 Event A capture register }
5  .CONST  ETUB0       =    0x2051;    {ETU0 Event B capture register }
5  .CONST  ETUAA0      =    0x2052;    {ETU0 Event AA capture register }
5  .CONST  ETUA1       =    0x2053;    {ETU1 Event A capture regsiter}
5  .CONST  ETUB1       =    0x2054;    {ETU1 Event B capture register}
5  .CONST  ETUAA1      =    0x2055;    {ETU1 Event AA capture register}
5  .CONST  ETUTIME     =    0x2056;    {ETU timer value}
5  .CONST  ETUCONFIG   =    0x205C;    {ETU configuration register }
5  .CONST  ETUDIVIDE   =    0x205D;    {ETU clock divide register }
5  .CONST  ETUSTAT     =    0x205E;    {ETU status register }
5  .CONST  ETUCTRL     =    0x205F;    {ETU control register }
5  .CONST  PWMSYNCWT   =    0x2060;    {PWMSYNC width register}
5  .CONST  PWMSWT      =    0x2061;    {PWM software trip register}
5
5  { memory mapped core registers of the ADMC300 }
5
5  .CONST  SYSCNTL     =    0x3fff;
5  .CONST  MEMWAIT     =    0x3ffe;
5  .CONST  TPERIOD      =    0x3ffd;
5  .CONST  TCOUNT     =    0x3ffc;
5  .CONST  TSCALE      =    0x3ffb;
5  .CONST  Sport0_Rx_Words1 =    0x3ffa;
5  .CONST  Sport0_Rx_Words0 =    0x3ff9;
5  .CONST  Sport0_Tx_Words1 =    0x3ff8;
5  .CONST  Sport0_Tx_Words0 =    0x3ff7;
5  .CONST  Sport0_Ctrl_Reg =    0x3ff6;
5  .CONST  Sport0_Sclkdiv =    0x3ff5;
5  .CONST  Sport0_Rfsdiv =    0x3ff4;
5  .CONST  Sport0_Autobuf_Ctrl =    0x3ff3;

```

```

5  .CONST  Sport1_Ctrl_Reg      =    0x3ff2;
5  .CONST  Sport1_Sclkdiv      =    0x3ff1;
5  .CONST  Sport1_Rfsdiv       =    0x3ff0;
5  .CONST  Sport1_Autobuf_Ctrl =    0x3fef;
5
5  { interrupt vector table addresses for ADMC300 }
5
5  .CONST  ADCBANKA_INT_ADDR    = 0x30+1;      {ADC BANKA INTERRUPT }
5  .CONST  PWMSYNC_INT_ADDR     = 0x34+1;      {PWMSYNC interrupt  }
5  .CONST  ADCBANKB_INT_ADDR    = 0x38+1;      {ADC BANKB interrupt }
5  .CONST  PIO_INT_ADDR        = 0x3C+1;      {PIO4 - PIO11 interrupt }
5  .CONST  EIU_INT_ADDR        = 0x40+1;      {EIU interrupt  }
5  .CONST  ETU_INT_ADDR        = 0x44+1;      {ETU interrupt  }
5  .CONST  PIO0_INT_ADDR       = 0x48+1;      {PIO0 interrupt  }
5  .CONST  PIO1_INT_ADDR       = 0x4C+1;      {PIO1 interrupt  }
5  .CONST  PIO2_INT_ADDR       = 0x50+1;      {PIO2 interrupt  }
5  .CONST  PIO3_INT_ADDR       = 0x54+1;      {PIO3 interrupt  }
5  .CONST  PWMTRIP_INT_ADDR     = 0x58+1;      {PWMTRIP interrupt}
5  .CONST  TX0_INT_ADDR        = 0x10; {SPORT0 transmit interrupt}
5  .CONST  RX0_INT_ADDR        = 0x14; {SPORT0 receive interrupt}
5  .CONST  SW1_INT_ADDR        = 0x18;      {software interrupt 1}
5  .CONST  SW0_INT_ADDR        = 0x1C;      {software interrupt 0}
5  .CONST  TX1_INT_ADDR        = 0x20; {SPORT1 transmit interrupt}
5  .CONST  RX1_INT_ADDR        = 0x24; {SPORT1 receive interrupt}
5  .CONST  TIMER_INT_ADDR      = 0x28; {Timer interrupt}
5
5  #endif
6  #include <romutil.h>

```

addr inst source line

```

6 {This include file contains the addresses for the ROM utility routines.
6 To use just include this file in the module where the routine will be called.
6 Then just call the routine name given below.
6
6 VERSION # 2.0      07/10/98
6
6 Name          Date      Description
6 ----          -
6 MOD/HISTORY:   TFH      07/10/98      Combined all targets into one file
6 }
6 #ifndef ROMUTIL_INCLUDE
6 #define ROMUTIL_INCLUDE
6
6 {-----ADMC300-----}
6 #ifdef ADMC300
6 {ROM Utility Start Locations}
6 .CONST PER_RST      = 0x07E4;
6 .CONST UMASK        = 0x0E21;
6 .CONST PUT_VECTOR= 0x0E28;
6 .CONST SMASK        = 0x0E35;
6 .CONST ADMC_COS     = 0x0E55;
6 .CONST ADMC_SIN     = 0x0E5C;
6 .CONST ARCTAN       = 0x0E72;
6 .CONST RECIP_NEW    = 0x0E94; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
6 .CONST SQRT = 0x0EAA;
6 .CONST LOG  = 0x0EE7;
6 .CONST LN   = 0x0EE4;
6 .CONST FLTONE    = 0x0F03;
6 .CONST FIXONE    = 0x0F08;
6 .CONST FPA  = 0x0F0C;
6 .CONST FPS  = 0x0F1B;
6 .CONST FPM  = 0x0F2B;
6 .CONST FPD  = 0x0F34;

```

```

6  .CONST FPMACC      = 0x0F55;
6  .CONST REV_FOR_PARK    = 0x0F77;
6  .CONST REV_CLARK      = 0x0F8B;
6  .CONST FOR_CLARK      = 0x0FA1;
6  .CONST SDIVQINT      = 0x0FAB;
6  .CONST SDIVQ         = 0x0FB4;
6  .CONST EXIT = 0x0FC6;
6
6  {Monitor Variables}
6  .CONST I4_SAVE      = 0x3bbf;
6  #endif
6
6  {-----ADMC330-----}
6  #ifdef ADMC330
6  {ROM Utility Start Locations}
6  .CONST PUT_VECTOR      = 0x0B73;
6  .CONST SIN12B          = 0x0DC4;
6  .CONST COS12B          = 0x0DD3;
6  .CONST CAL_INV_TAN      = 0x0E61;
6  .CONST LOG1            = 0x0E14;
6  .CONST ALOG1           = 0x0E20;
6  .CONST SQRT1           = 0x0E27;
6  .CONST DIVIDE          = 0x0DE2;
6  .CONST X_Y_TO_ANG       = 0x0E35;
6  .CONST APPROX10_ST      = 0x0DEC;
6  .CONST APPROX_N_ST      = 0x0DFF;
6  .CONST COS64            = 0x0B80;
6  .CONST COS6_64         = 0x0C00;

```

addr inst source line

```

6  .CONST  SIN6_64          = 0x0C40;
6  .CONST  ATN64            = 0x0D01;
6  .CONST  LOG64            = 0x0D42;
6  .CONST  ALOG64           = 0x0D83;
6
6  {Monitor Variables}
6  .CONST  HALT_FLAG        = 0x3B9A;    {used by exit library function}
6  .CONST  IDE_SP           = 0x79E;     {used by exit library function}
6  .CONST  IRQFLAG_SAVE     = 0x3BFE;    {IRQFLAG saved here each interrupt}
6  #endif
6
6  {-----ADMC331-----}
6  #ifdef ADMC331
6  {ROM Utility Start Locations}
6  .CONST  PER_RST          = 0X07F1;
6  .CONST  UMASK            = 0X0DED;
6  .CONST  PUT_VECTOR       = 0X0DF4;
6  .CONST  SMASK            = 0X0E06;
6  .CONST  ADMC_COS         = 0X0E26;
6  .CONST  ADMC_SIN         = 0X0E2D;
6  .CONST  ARCTAN           = 0X0E43;
6  .CONST  RECIP_NEW        = 0X0E65;    {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
6  .CONST  SQRT_NEW         = 0X0E7C;    {For SQRT - call SQRT in mathfix.dsp}
6  .CONST  LN_NEW           = 0X0EBA;    {For LN - call LN in mathfix.dsp}
6  .CONST  LOG_NEW          = 0X0EB9;    {For LOG - call LOG in mathfix.dsp}
6  .CONST  FLTONE           = 0X0ED4;
6  .CONST  FIXONE           = 0X0ED9;
6  .CONST  FPA              = 0X0EDD;
6  .CONST  FPS              = 0X0EEC;
6  .CONST  FPM              = 0X0EFC;
6  .CONST  FPD              = 0X0F05;
6  .CONST  FPMACC           = 0X0F26;
6  .CONST  PARK             = 0X0F48;

```

```

6  .CONST      REV_CLARK    = 0X0F5C;
6  .CONST      FOR_CLARK    = 0X0F72;
6  .CONST      COS64 = 0X0F80;
6  .CONST      ONE_BY_X     = 0X0FC0;
6  .CONST      SDIVQINT     = 0X0FD0;
6  .CONST      SDIVQ = 0X0FD9;
6
6  {Monitor Variables}
6  .CONST HALT_FLAG = 0x3B68;    {used by exit library function}
6  .CONST IDE_SP     = 0x7F7;    {used by exit library function}
6  .CONST IRQFLAG_SAVE      = 0x3B88; {IRQFLAG saved here each interrupt}
6  #endif
6
6  {-----ADMC328 Emulation Board-----}
6  #ifdef ADMC328E
6  {ROM Utility Start Locations}
6  .CONST      PER_RST      = 0X07F1;
6  .CONST      UMASK = 0X0DED;
6  .CONST      PUT_VECTOR   = 0X0DF4;
6  .CONST      SMASK = 0X0E06;
6  .CONST      ADMC_COS     = 0X0E26;
6  .CONST      ADMC_SIN     = 0X0E2D;
6  .CONST      ARCTAN       = 0X0E43;
6  .CONST      RECIP_NEW    = 0X0E65; {For RECIPROCAL - call RECIPROCAL in mathfix.dsp}
6  .CONST      SQRT_NEW     = 0X0E7C; {For SQRT - call SQRT in mathfix.dsp}
6  .CONST      LN_NEW       = 0X0EBA; {For LN - call LN in mathfix.dsp}
6  .CONST      LOG_NEW      = 0X0EB9; {For LOG - call LOG in mathfix.dsp}
6  .CONST      FLTONE       = 0X0ED4;

```

addr inst source line

```

6  .CONST      FIXONE      = 0X0ED9;
6  .CONST      FPA      = 0X0EDD;
6  .CONST      FPS      = 0X0EEC;
6  .CONST      FPM      = 0X0EFC;
6  .CONST      FPD      = 0X0F05;
6  .CONST      FPMACC      = 0X0F26;
6  .CONST      PARK      = 0X0F48;
6  .CONST      REV_CLARK      = 0X0F5C;
6  .CONST      FOR_CLARK      = 0X0F72;
6  .CONST      COS64      = 0X0F80;
6  .CONST      ONE_BY_X      = 0X0FC0;
6  .CONST      SDIVQINT      = 0X0FD0;
6  .CONST      SDIVQ      = 0X0FD9;
6
6  {Monitor Variables}
6  .CONST HALT_FLAG = 0x3B68;    {used by exit library function}
6  .CONST IDE_SP      = 0x7F7;    {used by exit library function}
6  .CONST IRQFLAG_SAVE      = 0x3B88; {IRQFLAG saved here each interrupt}
6  #endif
6
6  {-----ADMC328-----}
6  #ifdef ADMC328
6  {ROM Utility Start Locations}
6  .CONST      PER_RST      = 0X01F1;
6  .CONST      UMASK      = 0X0F15;
6  .CONST      PUT_VECTOR      = 0X0F1C;
6  .CONST      SMASK      = 0X0F29;
6  .CONST      SDIVQINT      = 0X0F30;
6  .CONST      SDIVQ      = 0X0F39;
6
6  {Monitor Variables}
6  .CONST IRQFLAG_SAVE      = 0x3988; {IRQFLAG saved here each interrupt}
6  #endif

```

```

6
6 {-----ADMC201-----}
6 #ifdef ADMC201
6 {ROM Utility Start Locations}
6 .CONST PUT_VECTOR      = 0x0573;
6 .CONST COS64           = 0x0580;
6 .CONST ONE_BY_X        = 0x05C0;
6
6 {Monitor Variables}
6 .CONST IRQFLAG_SAVE     = 0x3BFE; {IRQFLAG saved here each interrupt}
6 #endif
6
6 #endif
6
6
6
7
8 !*****
9 !* The following routine for using SPORT0 as a UART *
10 !* is from Analog Devices web @ ftp://ftp.analog.com/pub/*
11 !* www/marketSolutions/motorControl/applicationCode/*
12 !* admc300/uart0.exe *
13 !* *
14 !*****
15 #include <UART0.h>;
15 {***** definition of external labels *****}
15
15 .EXTERNAL Init_UART0;

```


addr inst source line

```
15 .EXTERNAL Write_UART0;
15 .EXTERNAL Read_UART0;
15
15 {***** definition of constants *****}
15
15 #ifndef Baudrate
15 #define UART0_Baudrate 115200
15 #endif
15
15 #ifndef Crystal_Freq
15 #define Crystal_Freq 12500000
15 #endif
15
15
15 .CONST UART0_Clk = Crystal_Freq/3/UART0_Baudrate; { Ratio between Crystal frequency
and frequency of the serial port clock }
15
16
17 {*****Program variable definitions*****}
18
19 .VAR/DM RX0_DATA;
20 .VAR/DM TX0_DATA;
21 .VAR/DM RX0A;
22 .VAR/DM RX0B;
23 .VAR/DM TX0A;
24 .VAR/DM TX0B;
25 .VAR/DM UART0_Status; .global UART0_Status;
26 .VAR/DM UART0_Status_TX; .global UART0_Status_TX;
27
28 .VAR/DM TX0_Isr_context_backup; {only ar needs to be saved }
29 .VAR/DM RX0_Isr_context_backup; {only ar needs to be saved }
30
31 #define UART0_FirstWordReceived 0
```

```

32 #define UART0_SecondWordReceived      1
33 #define UART0_ReceiveBufferFull      2
34
35 #define UART0_FirstWordTransmitted    4
36 #define UART0_SecondWordTransmitted  5
37 #define UART0_TransmitBufferNotEmpty  6
38
39
40 {*****}
41 {
42 {          Macro definitions          }
43 {
44 {*****}
45
46 .MACRO init_dm_scalar(%0,%1); { initialises a data memory location}
47     ar=%1; dm(%0)=ar;
48 .ENDMACRO;
49
50 {*****}
51 {
52 {          Program code of UART0 subroutines          }
53 {
54 {*****}
55
56 .entry init_uart0;
57
58 Init_UART0:    { initialisation of serial port 0 for UART mode    }
59
60 .CONST  UART0_Configuration  = b#0101111001001110; {0x5E4E }

```

addr inst source line

```

61 { Configuration:
62     Multichannel Mode Enable OFF      : MCE  =0
63     Internally generated serial clock : ISCLK =1
64     Receive frame sync NOT required  : RFSR  =0
65     Receive alternate framing mode    : RFSW  =1
66     Transmit frame sync required     : TFSR  =1
67     Transmit alternate framing mode   : TFSW  =1
68     Internal transmit frame sync ON   : ITFS  =1
69     Internal receive frame sync OFF   : IRFS  =0
70     Transmit frame sync active HIGH   : INVTFS=0
71     Receive frame sync active LOW     : INVRFS=1
72     Data format right just., 0 filled : DTYPE =b#00
73     Word length 15 bits               : SLEN  =0xe }
74
75
76     init_dm_scalar(sport0_ctrl_reg, UART0_Configuration);
77
78     {set serial clock frequency}
79     init_dm_scalar(sport0_sclkdiv, UART0_Clk - 1);
80
81     {clear all status flags}
82     init_dm_scalar(UART0_Status, 0x0000);
83     init_dm_scalar(UART0_Status_TX, 0x0000);
84
0008 380140      85         I4=RX0_INT_ADDR;          {RX0_Vector location      }
0009 40000Bu     86         mr0=^UART0_RX_Isr;          {pointer to Receive interrupt routine}
000A 1CE28F     87         CALL PUT_VECTOR;          {set RX0_Vector          }
88
000B 380100     89         I4=TX0_INT_ADDR;          {TX0_Vector location      }
000C 40000Bu     90         mr0=^UART0_TX_Isr;          {pointer to Transmit interrupt routine}
000D 1CE28F     91         CALL PUT_VECTOR;          {set TX0_Vector          }
92
000E 3C060C     93         ifc = 0x0060;          {clear pending SPORT0 interrupts }

```

```

000F 0D03A3      94      ar = IMASK;
0010 23AA5F      95      ar = setbit 5 of ar; {enable receive and ...      }
0011 23AA9F      96      ar = setbit 6 of ar; {... transmit interrupts for SPORT0 }
0012 0D0C3A      97      IMASK = ar;
      98
0013 83FFFA      99      ar=dm(SYSCNTL);
0014 23BA1F     100      ar= setbit 12 of ar;
0015 93FFFA     101      dm(SYSCNTL)=ar;      {enable SPORT0      }
      102
0016 0A000F     103      rts;
      104
      105
      106 {
      107     waits for transmitter buffer to be empty and then starts the
      108     transmission of the data in the lower byte of ax0
      109     Input:  ax0 (low byte) = databyte to be transmitted
      110     Output: ax0 is unaltered
      111     Called Routines: PROCESS_TX0, (Interrupt Service routine)
      112     Modified registers : ALU except ax0
      113     transfer is completed
      114 }
      115
      116 .entry Write_UART0;
      117 Write_UART0:
0017 80000Au     118      ar=dm(UART0_Status_TX);
0018 238A3F     119      ar=CLRBIT UART0_FirstWordTransmitted of ar;
0019 238A7F     120      ar=CLRBIT UART0_SecondWordTransmitted of ar;
      121      { None of the 30 Bits has been transmitted yet}

```

addr inst source line

```

001A 23AA9F      122          ar=SETBIT UART0_TransmitBufferNotEmpty of ar;
      123      { Indicate that Transmission is in progress}
001B 90000Au     124          dm(UART0_Status_TX)=ar;
      125
001C 1C000Fu     126          call PROCESS_TX0;
      127      { Build the two words to be sent from the data  }
      128
001D 80000Au     129          ar=dm(UART0_Status_TX);
001E 23AA1F      130          ar=SETBIT UART0_FirstWordTransmitted of ar;
001F 040040      131          dis INTS;
      132      { following two instructions uninterruptible }
0020 90000Au     133          dm(UART0_Status_TX)=ar;
      134      { update status and          }
0021 8C0009u     135          TX0=dm(TX0A);
      136      { send the 1st half (the 2nd is handled by  }
0022 040060      137          ena INTS;
      138      { the interrupt service routine)      }
0023 0A000F      139  rts;
      140
      141  .entry read_uart0;
      142  Read_UART0:
      143      { waits for a data-byte to be received from the serial port
      144      and stores the byte into the lower half of ax0      }
      145      { Input:  none
      146      Output: ax0 (low byte) = received databyte
      147      Called Routines: PROCESS_RX0, (Interrupt Service routine)
      148      Modified registers : ALU      }
      149
0024 1C000Fu     150          call PROCESS_RX0;          { extract byte from the 30 bit word}
      151
0025 80000Au     152          ar=dm(UART0_Status);
0026 23823F      153          ar=CLRBIT UART0_FirstWordReceived of ar;
0027 23827F      154          ar=CLRBIT UART0_SecondWordReceived of ar;

```

0028	2382BF	155	ar=CLRBIT UART0_ReceiveBufferFull of ar;	
	156			
0029	90000Au	157	dm(UART0_Status)=ar;	{ restore status to ready for new reception}
	158			
002A	80000Au	159	ar=dm(RX0_DATA);	{ store data into ar}
002B	0A000F	160	rts;	
	161			
	162		PROCESS_TX0:	
	163		{ forms the 30-bit word to be sent from the databyte	}
	164		{ Input: ax0 (low byte) = databyte to be transmitted	
	165		Output: ax0 is unaltered, variable TX0A and TX0B contain 1st	
	166		and 2nd half (15 bits aligned to the right) of the word	
	167		Called Routines: none	
	168		Modified registers : ALU except ax0}	
	169			
002C	900000u	170	dm(TX0_DATA)=ax0;	{ save content of ax0}
002D	0D00A0	171	ar =ax0;	{ ar=byte to be transmitted}
002E	0F1208	172	sr = LSHIFT ar by 8 (LO);	{ shift data byte into high-byte}
002F	0D000E	173	ax0= sr0;	
	174			
0030	40E00F	175	srl=0x0e00;	{stop bit in position 25-27, will be shifted down when inserting data bits}
0031	40000E	176	sr0=0x0;	
	177			
0032	3C0085	178	cntr = 8;	{insert 8 bits into sr}
0033	14000Eu	179	do Insert_Bit until CE;	
0034	0D0040	180	ay0=ax0;	
0035	22600F	181	ar = ax0 + ay0;	{left shift 1 bit, highest bit into carry flag}
0036	0D000A	182	ax0= ar;	{store shifted value in ax0}

addr inst source line

```

0037 0D00AF      183          ar=sr1;          {store higher word in ar, preserve previous
status}
0038 180009u     184          if NOT AC jump Bit_0;          {test bit shifted into carry}
0039 470004      185 Bit_1:    ay0=0x7000;          {if 1, expand it to 111 (aligned to positions 28-
30)}
003A 23A20F     186          ar= ar or ay0;          {ar=111 ored into current higher word}
003B 0F16FD     187 Bit_0:    sr= LSHIFT sr0 by -3 (LO);    {shift 32 bit word down three positions
for next bit}
003C 0F0AFD     188 Insert_Bit: sr=sr or LSHIFT ar by -3 (HI);{and insert current bit}
189
190          { after 8 bits, the word is aligned to bits 1-30 of sr}
191          { start bit is automatically in positions 28-30 (000)}
192
003D 90000Fu     193          dm(TX0A)=sr1;          {TX0A gets 15 MSBs = highword of sr}
003E 0F16FF     194          sr= LSHIFT sr0 by -1 (LO);
003F 90000Eu     195          dm(TX0B)=sr0;          {TX0B gets 15 LSBs from sr0 aligned to the right}
196
0040 800000u     197          ax0=DM(TX0_DATA);    {restore content of ax0}
0041 0A000F     198          rts;
199
200 {-----}
201
202 UART0_TX_Isr:
203          { Interrupt service routine, sends 2nd word if required  }
204          { Input:  variable TX0B contains 2nd word
205          Output: none
206          Called Routines: none
207          Modified registers : none
208          Cycles 13 (first word sent), 15 second word sent, 6 buffer empty DAB 08Nov01 }
209
0042 90000Au     210          DM(TX0_Isr_context_backup) = ar; {save context}
211
0043 80000Au     212          ar = dm(UART0_Status_TX);

```

```

0044 238A9F      213      ar = TSTBIT UART0_TransmitBufferNotEmpty of ar;
0045 180000u     214      if eq jump Exit_TX0_Isr;          { If the buffer is empty do nothing}
      215
0046 80000Au     216      ar = dm(UART0_Status_TX);
0047 238A5F      217      ar = TSTBIT UART0_SecondWordTransmitted of ar;
0048 180000u     218      if eq jump SendSecondWord;      { If the First word has been
transmitted send the second}
      219
0049 80000Au     220      ar = dm(UART0_Status_TX);
004A 238ABF      221      ar = CLRBIT UART0_TransmitBufferNotEmpty of ar;
004B 90000Au     222      dm(UART0_Status_TX)=ar;          { If the second word has been
transmitted reset status}
      223
004C 18000Fu     224      jump Exit_TX0_Isr;
      225
      226 SendSecondWord:
      227
004D 80000Au     228      ar = dm(UART0_Status_TX);
004E 23AA5F      229      ar=SETBIT UART0_SecondWordTransmitted of ar;
004F 040040      230      dis INTS;                      { following two instructions
uninterruptible}
0050 90000Au     231      dm(UART0_Status_TX)=ar;          { update status and}
0051 8C0009u     232      TX0=dm(TX0B);                    { send the 2nd half}
0052 040060      233      ena INTS;
      234
      235 Exit_TX0_Isr:
0053 80000Au     236      ar = DM(TX0_Isr_context_backup); {restore context}
0054 0A001F      237      rti;
      238
      239 {-----}
      240 {-----}
      241
      242 UART0_RX_Isr:
      243      { Interrupt service routine, stores 1st and 2nd word

```


addr inst source line

```

244          as required by the current status
245
246      During uart init, receive frame sync required after every word, RFSR
247      is set to false (0) so after the first word is received, the second
248      word is immediately clocked in without waiting for a frame sync.
249      During this ISR, RFSR is set to 1 while the first word is being read
250      and the second word is being clocked in, thus forcing the receiver
251      to stop after the second word is clocked in and wait for a new rfs.
252      When the 2nd word is received and read, RFSR is set back to 0 to allow
253      two words to be read following the next RFS
254  }
255
256      { Input:  none
257      Output: variables RX0A and RX0B contain received words
258      Called Routines: none
259      Modified registers : none
260      Cycles 14 (first word), 18 second word DAB 08Nov01
261  }
262
263
264
265
0055 90000Au      266      DM(RX0_Isr_context_backup) = ar; { save context}
0056 80000Au      267      ar = dm(UART0_Status);
0057 23821F       268      ar = TSTBIT UART0_FirstWordReceived of ar;
0058 180001u      269      if ne Jump SecondWord;          { first word was already received}
270
0059 83FF6A       271      ar=dm(Sport0_Ctrl_Reg);
005A 23BA5F       272      ar=SETBIT 13 of ar;
005B 93FF6A       273      dm(Sport0_Ctrl_Reg)=ar;          { set SPORT receive mode for 2nd half}
274
005C 9C0008u     275      dm(RX0A)=RX0;          { store first word}
276

```

005D	80000Au	277	ar = dm(UART0_Status);
005E	23A21F	278	ar=SETBIT UART0_FirstWordReceived of ar;
005F	90000Au	279	dm(UART0_Status)=ar; { update status flags}
	280		
0060	18000Fu	281	jump Exit_RX0_Isr;
	282		
	283	SecondWord:	
0061	80000Au	284	ar = dm(UART0_Status);
0062	23825F	285	ar = TSTBIT UART0_SecondWordReceived of ar;
0063	180001u	286	if ne Jump FlushWord;
	287		
	288	{ this is second half}	
0064	83FF6A	289	ar=dm(Sport0_Ctrl_Reg);
0065	239A7F	290	ar=CLRBIT 13 of ar;
0066	93FF6A	291	dm(Sport0_Ctrl_Reg)=ar; { prepare SPORT0 for the next 1st half}
	292		
0067	9C0008u	293	DM(RX0B)=RX0; { store second word}
	294		
0068	80000Au	295	ar = dm(UART0_Status);
0069	23A25F	296	ar = SETBIT UART0_SecondWordReceived of ar;
006A	23A29F	297	ar = SETBIT UART0_ReceiveBufferFull of ar;
006B	90000Au	298	dm(UART0_Status)=ar; { indicate that complete byte has been
	299	received}	
006C	18000Fu	300	jump Exit_RX0_Isr;
	301		
	302	FlushWord:	
006D	0D03A8	303	ar=RX0;
	304		

addr inst source line

```

        305 Exit_RX0_Isr:
        306
        307
006E 80000Au      308          ar = DM(RX0_Isr_context_backup); {restore context}
006F 0A001F      309          rti;
        310
        311 {-----}
        312
        313 PROCESS_RX0:
        314          { forms the databyte from the received 30-bit word      }
        315          { Input:  variable RX0A and RX0B contain 1st and 2nd half
        316                  (15 bits aligned to the right) of the word
        317          Output: Variable RX0_DATA (low byte) = databyte
        318          Called Routines: none
        319          Modified registers : ALU
        320          Cycles, 68 DAB 11/14/2002 }
        321
0070 80000Cu      322          mr1=DM(RX0A);          {1st half received in RX0A}
0071 80000Bu      323          mr0=DM(RX0B);          {2nd half received in RX0B}
        324
0072 0F13FC      325          sr= LSHIFT mr0 by -4 (LO);    { shift 3 stop bits + 1 bit out}
        326                  { to get to middle of 1st bit}
0073 0F0CFB      327          sr=sr or LSHIFT mr1 by -5 (HI);    { MSW shifted down and or'd in}
        328
0074 400005      329          ayl=0x0;          { ayl contains final byte}
        330
0075 400014      331          ay0=0x1;
0076 3C0075      332          CNTR = 0x7;          { extract seven bits}
0077 14000Eu      333          do Extract_Bit until CE;
0078 23860F      334          ar= sr0 and ay0 ;          { extract bit}
0079 23AA0F      335          ar= ar or ayl ;          { OR into byte}
007A 0D005A      336          ayl=ar;          { also create mirror image}
007B 226A0F      337          ar= ar + ayl ;          { shift it left 1}

```

```

007C 0D005A      338          ayl=ar;
007D 0D00BF      339          mr0=sr1;
007E 0F16FD      340          sr= LSHIFT sr0 by -3 (LO);      { shift 30 bit word down to}
007F 0F0BFD      341 Extract_Bit:  sr=sr or LSHIFT mr0 by -3 (HI); {next bit}
      342
0080 23860F      343          ar= sr0 and ay0;          { extract 8th bit}
0081 23AA0F      344          ar= ar or ayl;          { final byte created}
0082 90000Au      345          DM(RX0_DATA)=ar;      { store in RX0_DATA buffer}
      346
0083 0A000F      347          rts;
      348
      349
      350 .endmod;
      351

```

194

Phase R Output Section

Martin Hess		California Polytechnic State University, San Luis Obispo	
Title			
Induction Motor ASD: Phase R Output			
Size	Document Number	Rev	
A	ASD-1	A	
Date:		Sunday, August 17, 2003	Sheet 2 of 7

Phase S Output Section

Martin Hess		California Polytechnic State University, San Luis Obispo	
Title			
Induction Motor ASD: Phase S Output Section			
Size	Document Number	Rev	
A	ASD-1	A	
Date:		Sunday, August 17, 2003	Sheet 3 of 7

Phase T Output Section

Martin Hess		California Polytechnic State University, San Luis Obispo	
Title			
Induction Motor ASD: Phase T Output			
Size	Document Number	Rev	
A	ASD-1	A	
Date:	Sunday, August 17, 2003	Sheet	4 of 7

Martin Hess		California Polytechnic State University, San Luis Obispo	
Title		Induction Motor ASD: Bus Monitor and Gate Drive Supply	
Size	Document Number	Rev	
A	ASD-1	A	
Date		Monday, November 24, 2003	Sheet 6 of 7

Regen Clamp
(Bus Shunt Regulator)

Martin Hess		California Polytechnic State University, San Luis Obispo	
Title			
Induction Motor ASD: Regen Clamp			
Size	Document Number	Rev	
A	ASD-1	A	
Date:		Sunday, August 17, 2003	Sheet 7 of 7

Table I Bill of Materials				
Item	Qa	Reference	Description	Mfr
1	2	A1,A2	Current Sensor	LEM
2	40	C1,C2,C3,C4,C5,C6,C9,C10,C11,C12,C13,C14,C17,C18,C19,C20,C21,C22,C23,C24,C26,C28,C29,C30,C31,C32,C45,C48,C50,C52,C55,C58,C61,C62	0.1uF 50V Ceramic	Yageo
3	2	C8,C7	18pF 50V Ceramic	Yageo
4	3	C15,C16,C35	0.047uF 50V Ceramic	Kemit
5	8	C25,C40,C43,C46,C49,C53,C56,C59	4.7uF 20V Tantalum	Kemit
6	1	C27	220uF 6.3V Tantalum	Kemit
7	7	C38,C41,C44,C47,C51,C54,C57	100pF 50V Ceramic	Yageo
8	1	C60	47uF 450V Electrolytic	Illinois Capacitor
9	2	D2,D1	1A 100V Schottky	IR
10	14	D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16	1A 600V Ultra Fast	Diodes Inc
11	2	J2,J1	DB9 Female	Norcomp
12	1	J3	10 pos header	Amp/Tyco
13	1	J4	2 POS Header	Phoenix
14	2	J5,J6	3 POS Header	Phoenix
15	3	Q1,Q3,Q5	PNP Signal	Fairchild
16	2	Q4,Q2	NPN Signal	Fairchild
17	1	Q6	N-MOSFET	IR
18	7	Q7,Q8,Q9,Q10,Q11,Q12,Q13	600V 28A NPN IGBT	IR
19	2	R13,R1	1.00K 1/8W	Panasonic
20	6	R2,R6,R9,R10,R11,R12	4.99K 1/8W	Panasonic
21	3	R3,R4,R5	150 1/8W	Panasonic
22	11	R7,R8,R15,R21,R24,R27,R30,R33,R36,R39,R42	100 1/8W	Panasonic
23	9	R14,R16,R23,R26,R29,R32,R35,R38,R41	10.0K 1/8W	Panasonic
24	1	R17	3.32K 1/8W	Panasonic
25	10	R18,R19,R20,R22,R25,R28,R31,R34,R37,R40	10.0 1/8W	Panasonic
26	2	R44,R43	499K 1/8W	Panasonic

Table I con't					
Item	Mfr P/N	Vendor	Vendor P/N	Each	Ext
1	LTS25-NP				
2	12062R104K9B20D	Digikey	311-1179-1-ND	0.098	3.92
3	1206CG180J9B200	Digikey	311-1152-1-ND	0.090	0.18
4	C1206C473K5RACTU	Digikey	399-1246-1-ND	0.165	0.50
5	T491B475K020AS	Digikey	399-1612-1-ND	0.240	1.92
6	T520D227M006ASE050	Digikey	399-1760-1-ND	1.850	1.85
7	1206CG101J9B200	Digikey	311-1161-1-ND	0.105	0.74
8					
9	MBRS1100TR	Digikey	MBRS1100CT-ND	0.400	0.80
10	MURS160-13	Digikey	MURS160DICT-ND	0.512	7.17
11	172-009-282-031	Digikey	15609F-ND	5.220	10.44
12	104433-5	Digikey	104433-5-ND	0.950	0.95
13	1803426	Digikey	277-1221-ND	0.560	0.56
14	1804690	Digikey	277-1470-ND	3.780	7.56
15	MMBT2907A	Digikey	MMBT2907AFSCT-ND	0.170	0.51
16	MMBT2222A	Digikey	MMBT2222AFSCT-ND	0.170	0.34
17	IRLL3303	Digikey	IRLL3303-ND	1.800	1.80
18	IRG4PC30KD	Digikey	IRG4PC30KD-ND	4.300	30.10
19	ERJ-8ENF1001V	Digikey	P1.00KFCT-ND	0.117	0.23
20	ERJ-8ENF4991V	Digikey	P4.99KFCT-ND	0.117	0.70
21	ERJ-8ENF1500V	Digikey	P150FCT-ND	0.117	0.35
22	ERJ-8ENF1000V	Digikey	P100FCT-ND	0.063	0.69
23	ERJ-8ENF1002V	Digikey	P10.0KFCT-ND	0.117	1.05
24	ERJ-8ENF3321V	Digikey	P3.32KFCT-ND	0.117	0.12
25	ERJ-8ENF10R0V	Digikey	P10.0FCT-ND	0.117	1.17
26	ERJ-8ENF4993V	Digikey	P499KFCT-ND	0.117	0.23

Table I con't				
Item	Qa	Reference	Description	Mfr
27	1	S1	PUSHBUTTON	Alco
28	1	S2	3 pos header	Amp/Tyco
29	1	S3	Thermostat	Cantherm
30	1	T1	5:18 Toroid Transformer	
31	1	U1	RS232 Transceiver	TI
32	1	U2	Reset IC	National
33	1	U3	Line Receiver	TI
34	1	U4	admc300	Analog Devices
35	1	U5	EEPROM	Atmel
36	1	U6	Optocoupler	Clare
37	1	U7	GATE DRIVER	TI
38	7	U8,U9,U10,U11,U12,U13,U14	Opto Gatedriver	Agilent
39	1	U15	SINGLE SUPPLY OPAMP	Burr Brown
40	1	Y1	12.500MHz Crystal	Citizen
41	1		PCB	Mhess
42	1		2 Pos mating connector	Phoenix
43	2		3 Pos mating connector	Phoenix
44	1		Heat Sink	
45	8		4-40 x 3/8	
46	6		4-40 x 1/2	
47	4		#4 nylon washer	
48	4		#4 x 3/8 nylon spacer	

Table I con't					
Item	Mfr P/N	Vendor	Vendor P/N	Each	Ext
27	TP11CGPC2	Digikey	450-1124-ND	4.570	4.57
28	146225-3	Digikey	146225-3-ND	0.270	0.27
29	F11070151ZA0060	Digikey	317-1057-ND	8.120	8.12
30					0.00
31	MAX232DR	Digikey	296-14619-1-ND	0.750	0.75
32	LM3722IM5-4.63	Digikey	LM3722IM5-4.63CT-ND	2.360	2.36
33	SN75175DRT2	Digikey	296-14954-1-ND	1.080	1.08
34	ADMC300BST	Future	ADMC300BST	25.750	25.75
35	AT17C128-10NC	Digikey	AT17C128-10NC-ND	9.780	9.78
36	LOC111PTR	Digikey	CLA118CT-ND	5.270	5.27
37	TPS2829DBVR	Digikey	296-1987-1-ND	1.440	1.44
38	HCPL316J	Newark	92F648	4.290	30.03
39	OPA340NA/250	Digikey	OPA340NACT-ND	1.800	1.80
40	CM309S12.500MABJT R	Digikey	300-2043-1-ND	1.100	1.10
41	VSD-1	PCB Express		290.000	290.00
42	1803578	Digikey	277-1161-ND	1.630	1.63
43	1848902	Digikey	277-1464-ND	5.850	11.70
44					
45					
46					
47					
48

TOP SILK

TOP SOLDERMASK

TOP CIRCUIT

POWER PLANE (NEGATIVE IMAGE)

GROUND PLANE (NEGATIVE IMAGE)

BOTTOM CIRCUIT

BOTTOM SOLDERMASK

Drill Symbols		
SYM	COUNT	SIZE(in)
+	119	0.020
X	46	0.035
Y	12	0.042
T	14	0.052
Z	21	0.067
⊠	2	0.081
□	10	0.125

DRILL TEMPLATE