

**DALF – 1; Rev F      Motor Control Board**

**Owners' Manual**

Revision 0.37  
Aug 16, 2008

# Table of Contents

<b>1</b>	<b>MOTOR CONTROL BOARD OVERVIEW .....</b>	<b>6</b>
<b>2</b>	<b>DEVELOPMENT HISTORY .....</b>	<b>8</b>
2.1	BOARD DEVELOPMENT .....	8
2.2	FIRMWARE RELEASES .....	8
2.3	DOCUMENTATION.....	9
<b>3</b>	<b>OPERATING MODES (CMD, POT, R/C, RCSVO, POTSVO) .....</b>	<b>9</b>
3.1	TE INTERFACE .....	10
3.2	API INTERFACE.....	10
3.3	I2C2 INTERFACE.....	11
3.4	POT INTERFACES.....	11
3.5	R/C INTERFACE (3 CHANNELS).....	11
3.6	OTHER INTERFACES .....	11
<b>4</b>	<b>THE DALF-1 BOARD DETAIL .....</b>	<b>12</b>
4.1	PARTIAL FEATURE LIST .....	12
4.2	POWER SUPPLY.....	13
4.3	PIC18F6722 MICROCONTROLLER.....	13
4.4	MOTOR DRIVER CONNECTION .....	13
4.5	RS232 .....	15
4.6	EXTERNAL EEPROM/PARAMETER BLOCK .....	15
4.7	INTERNAL (BUILT-IN) EEPROM.....	16
4.8	FAN DRIVER OUTPUTS .....	16
4.9	EX0, EX1, EX2, EX3 .....	16
4.10	A/D INPUTS .....	17
4.11	OVER CURRENT SENSE .....	17
4.12	POSITION ENCODER INPUTS .....	17
4.13	I2C COMMUNICATION.....	18
4.14	I/O EXPANDERS.....	18
4.15	CONNECTORS.....	19
4.16	LEDs .....	19
4.17	SWITCHES .....	20
4.18	HEADERS .....	20
<b>5</b>	<b>CUSTOMIZATION AND CONFIGURATION .....</b>	<b>21</b>
5.1	EEPROM CONFIGURATION.....	21
5.2	FACTORY DEFAULTS, PARAMETER BLOCK, AND ERAM .....	21
5.3	CHANGING THE CONFIGURATION .....	22
5.4	PARAMETER BLOCK CHANGE GUIDE .....	22
5.5	SOME IMPORTANT PARAMETERS .....	23
5.6	PROGRAMMING CUSTOMIZATION.....	26
<b>6</b>	<b>DEVELOPMENT TOOLS.....</b>	<b>27</b>
6.1	COMPILER, ASSEMBLER, DEBUGGER .....	27
6.2	SOFTWARE DEVELOPMENT TOOL VERSION NUMBERS .....	28
	FLASHING THE MICROCONTROLLER USING THE BOOT LOADER .....	29
6.2.1	The Boot Block.....	29
6.2.2	Under The Hood .....	29
6.2.3	Installing the p1618qp.exe Application .....	30
6.2.4	About The Image .....	30
6.2.5	About The Parameter Block .....	31

6.2.6	The Process.....	31
<b>7</b>	<b>INTERRUPT DRIVEN FIRMWARE DESIGN .....</b>	<b>33</b>
7.1	TMR0 - HEARTBEAT .....	33
7.2	TMR1 - RTC .....	33
7.3	TMR2 - ADC STATE MACHINE .....	33
7.4	INT0 - MOTOR2 ENCODER .....	34
7.5	INT1 - MOTOR1 ENCODER .....	35
7.6	INT2 - MOTOR2 CURRENT LIMIT.....	35
7.7	INT3 - MOTOR1 CURRENT LIMIT.....	35
7.8	RB4 - EX2 .....	35
7.9	TX1 - USART1: RS232 SERIAL COMMAND RESPONSE.....	35
7.10	RX1 - USART1: RS232 SERIAL COMMAND RECEIVE.....	36
7.11	CCP1 - R/C CHANNEL 1 (EX0) .....	36
7.12	CCP2 - R/C CHANNEL 2 (EX1) .....	37
7.13	CCP3 - R/C CHANNEL 3 (EX3) .....	37
7.14	SSP2 - I2C2 SLAVE COMMAND INTERFACE (SC2, SD2) .....	37
7.15	//UNUSED INTERRUPTS// - RESOURCE DETAILS .....	37
<b>8</b>	<b>MOTOR ENCODER FEEDBACK.....</b>	<b>39</b>
8.1	OPTICAL ENCODERS .....	39
8.1.1	The CPR Specification .....	39
8.1.2	Optical Encoder Limitations.....	39
8.1.3	Optical Encoder Hookup .....	40
8.1.4	Optical Encoder Accuracy.....	41
8.2	ANALOG ENCODERS.....	41
8.2.1	Analog Encoder Limitations.....	41
8.2.2	The Position Algorithm (Under The Hood).....	42
8.2.3	The Digital Recursive Filter .....	43
8.2.4	Analog Encoder Hookup .....	44
8.2.5	Analog Encoder Accuracy.....	44
<b>9</b>	<b>CURRENT SENSOR FEEDBACK .....</b>	<b>45</b>
9.1	SENSOR SELECTION .....	45
9.2	SENSOR HOOKUP .....	45
9.3	SETUP.....	45
9.4	UNDER THE HOOD .....	46
9.5	LIMITATIONS .....	46
9.6	OVER CURRENT RESPONSE .....	47
9.7	ADJUSTING THE CURRENT LIMIT THRESHOLDS .....	47
<b>10</b>	<b>TERMINAL EMULATOR CMD INTERFACE .....</b>	<b>49</b>
10.1	RS232 SETUP .....	49
10.2	COMMAND SYNTAX (CMD D0 D1 ... DN).....	49
10.3	TERMINAL EMULATOR CMD CRIB SHEET .....	52
10.4	TERMINAL EMULATOR COMMAND SPECIFICATION .....	54
10.4.1	<b>Cmd_A PWM Freq Control .....</b>	54
10.4.2	<b>Cmd_B Fan Control .....</b>	55
10.4.3	<b>Cmd_C Get AtoD Reading (mV).....</b>	55
10.4.4	<b>Cmd_D Set/Get RTC .....</b>	56
10.4.5	<b>Cmd_E Get Motor Position .....</b>	56
10.4.6	<b>Cmd_F Set Encoder.....</b>	56
10.4.7	<b>Cmd_G // UNUSED // .....</b>	57
10.4.8	<b>Cmd_H // UNUSED // .....</b>	57

10.4.9	<b>Cmd_I Reset Board</b>	57
10.4.10	<b>Cmd_J IoExp Write</b>	57
10.4.11	<b>Cmd_K IoExp Read</b>	57
10.4.12	<b>Cmd_L Read Memory Block (RAM OR EEPROM)</b>	57
10.4.13	<b>Cmd_M Set Digital POT</b>	58
10.4.14	<b>Cmd_N R/C Snapshot</b>	58
10.4.15	<b>Cmd_O StopMotor(s)</b>	58
10.4.16	<b>Cmd_P Set/Get PID Parameters</b>	60
10.4.17	<b>Cmd_Q PID Step Response</b>	61
10.4.18	<b>Cmd_R Read Memory Byte</b>	64
10.4.19	<b>Cmd_S Move (Constant Velocity, Closed-Loop)</b>	65
10.4.20	<b>Cmd_T Trigger Move (Closed Loop)</b>	65
10.4.21	<b>Cmd_U Get Status</b>	66
10.4.22	<b>Cmd_V Get Motor Velocity</b>	66
10.4.23	<b>Cmd_W Write Memory Byte</b>	67
10.4.24	<b>Cmd_X Move (Constant Power, Open-Loop)</b>	67
10.4.25	<b>Cmd_Y Mtr Move (Closed-Loop Control)</b>	69
10.4.26	<b>Cmd_Z Upload To EEPROM</b>	70
<b>11</b>	<b>POT MODE INTERFACE</b>	<b>71</b>
11.1	POT MODES	71
11.2	POT CONNECTIONS	72
11.3	ENABLING POT MODES	72
11.4	IMPORTANT POT MODE PARAMETERS:	72
<b>12</b>	<b>R/C MODE INTERFACE</b>	<b>74</b>
12.1	RECEIVER CONNECTIONS	74
12.2	RECEIVER OUTPUT SIGNALS	74
12.3	R/C PWM MAPPING	74
12.4	R/C TUNING	75
12.4.1	Endpoint Adjustment At Transmitter:	75
12.4.2	Endpoint Adjustment On Dalf Board:	75
12.5	R/C MODES	75
<b>13</b>	<b>SERVO MODES</b>	<b>76</b>
13.1	CLOSED LOOP OPERATION	76
13.2	SERVO LIMITS	76
13.3	CONNECTIONS	77
13.4	POT SERVO MAPPING	77
13.5	POT SERVO PARAMETERS	77
13.6	R/C SERVO MAPPING	77
13.7	R/C TUNING	77
13.8	R/C SERVO MODE PARAMETERS	78
<b>14</b>	<b>TRAPEZOIDAL TRAJECTORY GENERATOR</b>	<b>79</b>
14.1	TRAJECTORY GENERATOR OVERVIEW	79
14.2	TRAJECTORY GENERATOR DETAILS	79
14.2.1	Trajectory Generator Timing	80
14.2.2	Trajectory Setup	81
14.2.3	Trajectory Routine	81
14.3	TRAJECTORY GENERATOR CASES	82
14.4	INTEGRAL WINDUP	83
<b>15</b>	<b>PID (CLOSED-LOOP) MOTOR CONTROL</b>	<b>84</b>

15.1 PROPORTIONAL-INTEGRAL-DERIVATIVE (PID) EQUATIONS .....	84
15.2 PID MOTOR CONTROL TUNING .....	85
<b>16 FIRMWARE LIBRARY ROUTINES.....</b>	<b>87</b>
16.1 IOEXP FUNCTIONS.....	88
16.1.1 <b>WriteIOExp1, WriteIOExp2 .....</b>	89
16.1.2 <b>ReadIOExp1, ReadIOExp2 .....</b>	89
16.2 DELAY AND TIMING FUNCTIONS .....	90
16.2.1 <b>GetTime .....</b>	90
16.2.2 <b>SetDelay .....</b>	91
16.2.3 <b>TimeOut.....</b>	91
16.3 EXTERNAL EEPROM FUNCTIONS .....	92
16.3.1 <b>WriteExtEE_Byte .....</b>	92
16.3.2 <b>ReadExtEE_Byte .....</b>	93
16.3.3 <b>WriteExtEE_Block .....</b>	93
16.3.4 <b>ReadExtEE_Block .....</b>	94
16.4 INTERNAL EEPROM FUNCTIONS.....	94
16.4.1 <b>WriteIntEE_Byte .....</b>	94
16.4.2 <b>ReadIntEE_Byte .....</b>	95
16.4.3 <b>WriteIntEE_Block .....</b>	95
16.4.4 <b>ReadIntEE_Block .....</b>	96
16.5 DIGITAL POT FUNCTIONS.....	97
16.5.1 <b>WritePot1 .....</b>	97
16.5.2 <b>WritePot2 .....</b>	98
<b>17 POTENTIAL FUTURE ENHANCEMENTS.....</b>	<b>99</b>
<b>18 APPENDIX A - STEP RESPONSE EXAMPLES.....</b>	<b>99</b>
<b>19 APPENDIX B - BOARD SCHEMATIC/LAYOUT.....</b>	<b>102</b>
<b>20 APPENDIX C - BOARD PARTS LIST.....</b>	<b>105</b>
<b>21 APPENDIX D - PARAMETER BLOCK DETAIL .....</b>	<b>107</b>
21.1 PARAMETER BLOCK TABLE.....	107
21.2 PARAMETER BLOCK RECOVERY.....	110
21.3 PARAMETER DESCRIPTIONS .....	110
<b>22 APPENDIX E - FIXED ADDRESS RAM VARIABLES .....</b>	<b>121</b>

## **Warranty**

The software libraries and tools are provided "as is" without warranty. The entire risk for the results and performance of these libraries and tools is assumed by the purchaser. Embedded Electronics LLC does not warrant, guarantee or make any representation regarding the use of this product. No other warranties are made, expressly or implied, including, but not limited to, the implied warranties of merchantability and suitability of products for a particular purpose. In no event will Embedded Electronics be held liable for additional damages, including lost profits, lost savings or other incidental or consequential damages arising from the use or inability to use Embedded Electronics LLC products.

## **Disclaimers**

Embedded Electronics LLC reserves the right to make changes without notice to this product. Changes made to improve reliability, performance, capabilities, design or ease of use, or to reduce size or cost could effect documentation, hardware, and firmware. Any Embedded Electronics LLC product may not be used as a component in life support devices of any description.

## **Copyright**

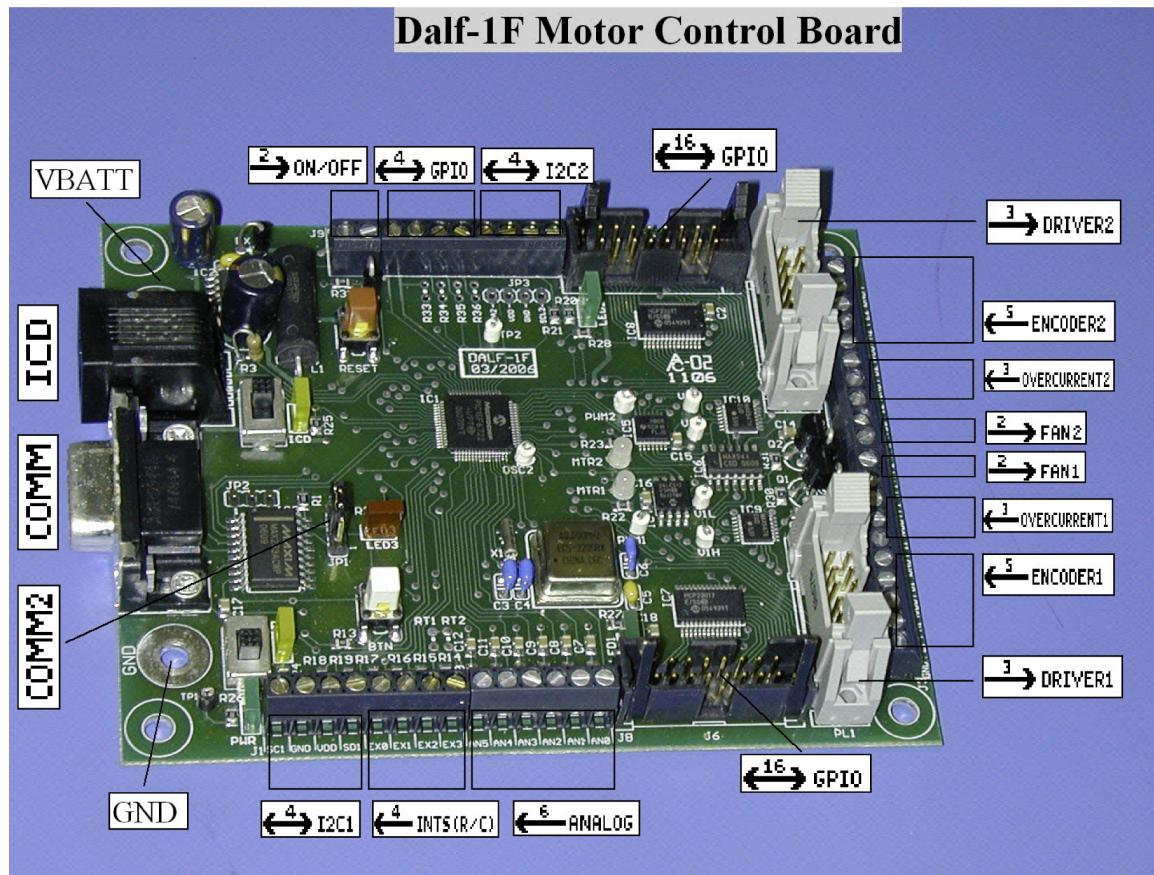
Copyright 2006 Embedded Electronics, LLC.

## **Software License**

The <main.c> and <dalf.lib> files provided on the CD for the purpose of encouraging additional software development are subject to a license agreement explained in the End User License Agreement (EULA). The EULA file is included on the CD that ships with the product and may also be read on the EE Website <http://www.embeddedelectronics.net/>

# 1 Motor Control Board Overview

This document describes the hardware and firmware features of the Dalf-1 Motor Control Board.  
[Board Size: 4.60" x 3.65"]



This board was designed as a dual motor controller for a reasonably autonomous mobile robot (Dalf) weighing approximately 150 pounds. Dalf is driven by a pair of PM brushed DC wheelchair motors using a differential drive arrangement. The board interfaces directly to a pair of electronic speed control (ESC) boards (Drivers) designed by the Open Source Motor Control (OSMC) group (1), but the board can be used with other motor drivers that expect standard Signed Magnitude PWM motor control inputs. **Because of the abundant, accessible I/O, large memories, and the development connector, the board is also quite suitable for a PIC based development platform or for other, non motor control, applications.** All of the features described in this manual come preprogrammed in the non-volatile memories for the PIC18F6722 (FLASH) and 24LC512 (EEPROM) parts. Assembled and tested boards are available for purchase thru resellers.

The PC board design and much of the software is provided as open source. The board comes with a CD containing documentation, a library file, and C source for the main loop. The library provides functions callable from either PIC Assembler or C. Included in the library are routines to access all of the on board parts. Sufficient detail is provided to make most of the firmware design and functionality transparent and extendable. See the appendices for the board schematic, board layout, and parts list.

(1) - The OSMC boards are reliable, high power, PM DC motor control drivers. They require PWM drive and other control signals to be supplied externally thru an on board connector. The OSMC boards with complete specifications are available from Robot Power ([robot-power.com](http://robot-power.com)).

The Dalf Board and built-in firmware maintain motor **position** and **velocity** information based on the inputs from standard **optical incremental motor encoders**. Optionally, **analog encoders** can replace the optical encoders in some designs. **Over current** response to off board sensors is immediate (interrupt driven) and the over current thresholds are digitally adjustable. Hardware to drive a pair of off board **fans**, or other high current devices is provided. The **board is configurable** to accept motor control commands from off board potentiometers, R/C, or one of several serial command interfaces.

Several **Open-Loop** motor control modes, as well as a robust **Trapezoidal Generator** and **PID** combine to provide **Closed-Loop** motor control operation. Closed-Loop “**move-to-position**” commands support both relative and absolute targets for the move destination. The trigger mode option can provide **synchronized motor movements**. Closed-Loop “**constant velocity**” is supported for applications that require constant velocity under varying load conditions. **Servo modes** (closed loop) may be used to control motor position with R/C or off-board potentiometers. **Pot and R/C interfaces** (open loop) may be used to control motor velocities. The three channel R/C interface provides both “**tank**” and “**mixed**” operating modes.

Adjustable **slew rate control** for both open and closed loop commands provides smooth motor velocity transitions.

Three different, **serial command interfaces** are provided to control all board features.

- TE - Terminal Emulator (RS232)
- API - Applications Programming Interface (RS232)
- I2C2 - Secondary I2C interface (second of two I2C interfaces)

The RS232 interfaces utilize the primary COMM channel, a standard modem cable, and a PC Application. The TE interface employs a standard terminal emulator application like Hyperterm or Teraterm running on the PC to issue commands and receive and display data. This interface provides the means to test all motor features and to customize motor control operations for the other operating modes. The API employs the same connector and cable and is used for similar operations, but is designed for a “smart” PC application like a Windows GUI (A Windows GUI is under development). The I2C2 interface utilizes a 3-wire (optionally 4) cable and the I2C2 screw terminal connectors to provide a command interface to an external board. The Dalf Board is configured as an I2C SLAVE device on the I2C2 interface, accepting commands and delivering data and status to the external device configured as the MASTER. The API and I2C2 interfaces are described in separate documents.

Non volatile storage of motor, motor encoder, and operating mode configuration information is provided via a **Parameter Block** within the serial EEPROM.

The board has **lots of unused I/O** which is routed to connectors for off board use. Analog inputs, GPIO pins, interrupt capable inputs, a power supply On/Off switch, and 32 GPIO lines from I/O Expander parts are examples. The second (unused) USART (**COMM2**) is routed to pads for either level shifted or +5V operation. Several screw terminal connectors provide +5V and GND for **BEC** usage. The primary I2C bus is configured as a MASTER and used for on board device communication, but both the primary and secondary **I2C buses** are routed to connectors for potential off board communications usage.

The **firmware is efficient**. All time critical operations are implemented with PIC assembler and all operations except the primary I2C access are “**interrupt driven**” as opposed to “polled”. The built-in interrupt and command handlers are written in PIC Assembler for efficiency. The main loop and the interrupt dispatcher are written in C. This design, with the provided C source code and PIC library file, makes for **easy customization** of the main loop or replacement of the existing interrupt handlers.

The <dalf.lib> **firmware library** provides routines, callable from C or PIC Assembler, that provide access to board features. Some examples are READ, and WRITE functions to provide access to the I/O extender parts, internal EEPROM, external EEPROM, and RAM. WRITE functions provide access to set the write only digital pots on the I2C bus. All I2C access is provided at the device level which means that you don't need to be aware of I2C bus details to access and control the parts. Access to all on-board devices is also provided thru commands in the serial interfaces.

A **boot loader application** <p1618qp.exe> that runs on your PC is provided on the CD. When used in conjunction with the boot loader firmware maintained in the boot block of flash program memory and the PGM Switch on the board, it allows firmware updates to be downloaded from your PC and flashed into the microcontroller.

## 2 Development History

### 2.1 Board Development

#### 08/28/2004: Versions (1A, 1B)

The first two versions of the Dalf-1 board used the Microchip PIC16F777 microcontroller and many of the features described in this document were fully functional in the 1B revision. The 8K x14 bits of flash and 384 bytes of RAM provided by the midrange 777 part were pretty much fully used up. The firmware was all written in PIC Assembler and the part was clocked by an external 20MHz resonator (5MIPS).

#### 01/29/2005: Version 1C

A fairly major redesign using the high end PIC18F6720 micro running at 25MHz (6.25MIPS) provides generous margins (RAM, FLASH, and EEPROM) for additional features. Starting with the 1C board, the code is a mix of C and PIC Assembler allowing faster development and easier customization.

#### 08/18/2005: Version 1D

This version provided a shrink of the 1C board and a few additional features.

#### 02/24/2006: Version 1E

This version replaces the PIC18F6720 micro with the follow on micro PIC18F6722. This part runs at 40MHz (10MIPS) and adds the secondary I2C bus. The MCP23016 parts are replaced with the improved MCP23017 parts. Manual trim pots (for over current detection) are replaced by digital pots accessed using the primary I2C bus. Most thru-hole R and C components are replaced with surface mount parts.

#### 03/15/2006: Version 1F

First board manufactured for production. Revisions include addition of pads for no-load parts and some minor routing improvements. Included are: a jumper for usage of the second serial port at logic voltage levels, pull-ups for some unused GPIO's, and voltage divider resistors for possible use with off board thermistors.

### 2.2 Firmware Releases

#### 09/11/2006: Version 1.40

First release for production.

#### 11/30/2006: Version 1.50

Second production release

#### 06/09/2007: Version 1.60

Third production release

## **2.3 Documentation**

### **9/11/2006: Documentation for the Version 1.40 Firmware Release**

- Owner's Manual (Ver 0.29)
- Getting Started Manual (Ver 0.05)
- I2C2 Interface (Ver 0.02)
- API Interface (Ver 0.09)

### **11/30/2006: Documentation for the Version 1.50 Firmware Release**

- Owner's Manual (Ver 0.30)
- Getting Started Manual (Ver 0.06)
- I2C2 Interface (Ver 0.03)
- API Interface (Ver 0.10)

### **06/09/2007: Documentation for the Version 1.60 Firmware Release**

- Owner's Manual (Ver 0.32)
- Getting Started Manual (Ver 0.07)
- I2C2 Interface (Ver 0.04)
- API Interface (Ver 0.11)

#### **Getting Started Manual**

This document describes hook ups, board configuration using a terminal emulator application, and general user interface information. More complete information can be found in the Owners' Manual, but this is the right place to start.

#### **Owner's Manual**

This document describes all features in detail. References to the separate API and I2C2 documents provide the detail for those features.

#### **API Interface**

This document describes a serial (RS232) protocol suitable for communication with the Dalf Board by a smart PC Application like a Windows GUI.

#### **I2C2 Interface**

This document describes a serial (I2C) protocol suitable for communication with an I2C MASTER.

Updated documentation is maintained at <http://www.embeddedelectronics.net/>

## **3 Operating Modes (CMD, POT, R/C, RCSVO, POTSVO)**

To say that the firmware provides these 6 operating modes (and a few sub-modes if you count R/C mixing and different pot modes) is a bit misleading. The reason is that **most features of the Cmd Interfaces (TE, API, and I2C2) using the serial links can also be used in the other control modes**. For example, you can use the serial link to monitor motor speed and position while controlling the motors with pots or an R/C link. The TE, API, and I2C2 serial command Interfaces all have similar command sets and functionality. The TE and API utilize the same hardware interface (serial port; RS232), but the software communication protocol is quite different. Separate documents describe the API and I2C2 Interfaces in some detail.

**Throughout this document, references to the Cmd Interface refer explicitly to the TE Interface, but the general content will often apply to all three serial interfaces.** The operating mode is specific to each

motor. This means, for example, that there is nothing to prevent control of one of the motors with a POT and the other with an R/C channel. For a given motor, “motor movement” commands may generally come from only one source. For example, while operating a motor using the R/C Interface, an attempt to simultaneously control the motor by issuing a motor movement command over a serial command interface will be ignored (except for the STOP command!). All modes provide for an adjustable slew rate and the slew rate parameter is customizable. Even if your only interest is the POT or R/C control modes, the CMD interface is necessary for initial configuration.

#### **Motor Control Modes:**

Mode	Control Type	Required External Resources
TE - CMD (RS232)	Both	PC + Terminal Emulator Application + standard serial cable
API - CMD (RS232 )	Both	PC + Smart PC Application + standard serial cable
I2C2 - CMD (I2C)	Both	MASTER Application + 3 wire cable (SCL,SDA,GND)
POTF - Full Range	Open Loop	POT1, POT2, ON/OFF# switch, FWD/REV# switch
POTC - Center Zero	Open Loop	POT1, POT2, ON/OFF# switch
PITMIX - Pot Mix	Open Loop	POT1, POT2, ON/OFF# switch
POTSVO - Pot Servo	Closed Loop	POT1, POT2, ON/OFF# switch, motor position encoders
RCNRM - R/C Normal	Open Loop	Transmitter, Receiver
RCMIX - R/C Mixed	Open Loop	Transmitter, Receiver
RCSVO- R/C Servo	Closed Loop	Transmitter, Receiver, motor position encoders

The board powers up in a mode determined by values stored in the **Parameter Block** of non-volatile memory. Factory defaults stored in this parameter block may be customized using any of the serial mode interfaces. The factory default is to power up in TE command mode (R/C and POT modes disabled).

**This means that the only means of controlling the motors initially is thru one of the serial interfaces.**

**Generally the TE interface will be used for board configuration, but a soon to be released Motor Control Graphical User Interface (GUI) will offer a more user friendly alternative in the form of a Windows Application.** See the section in this manual labeled “Customization and Configuration” for a discussion of the Parameter Block and how it may be used to configure operations.

### **3.1 TE INTERFACE**

The TE command monitor runs over an RS232 link to a PC (running Hyperterm, TeraTerm, or some other terminal emulator application) and is a good way to test and demonstrate most board features. You should become familiar with this interface even if you plan on using one of the other interfaces for motor control because it provides commands for features such as “PID Tuning”, “R/C Switch Tuning”, “Operating Mode Selection”, etc. Other features include “A/D Snapshot” which can be used to verify correct POT operation, “R/C Snapshot” which shows actual pulse measurements on each of the R/C channels, Closed Loop (PID + Trajectory Generation) motor control, EEPROM and RAM access, and much more. The command monitor also provides the means for customizing values in the Parameter Block which (among other things) determine the power up operating mode of the board. See the chapter “Terminal Emulator CMD Interface” for a thorough discussion and listing of all commands.

### **3.2 API INTERFACE**

The API command monitor runs over an RS232 link to a smart PC Application such as a Graphical User Interface (GUI). A Windows GUI for communication with the Dalf Board is under development. Check the website for status on the GUI. The command set and capabilities of the API Interface are similar to the TE Interface, but the communication protocol is quite different. Commands and command arguments are

passed in binary to the board from the PC Application via message packets. Similarly, data and status are returned to the PC Host Application with message packets. All packets have check sums to verify data integrity. A complete description of the protocol, packet format, error handling, and possible future use for networking applications is described in the separate “**API Interface**” document.

### **3.3 I2C2 INTERFACE**

The I2C2 interface runs on the Secondary I2C Bus provided by the PIC and relies on an application running on an off-board device to deliver commands and request status. The off-board device will be configured as a MASTER and hosts the bus, while the Dalf Board is configured as a SLAVE device at an I2C address that is specified in the Parameter Block (default 0x60). The 2-wire connections (plus GND) are made using the provided screw terminals. The Dalf Board supplies the required pull-up resistors (2.7K) for this bus, but these may be removed if the pull-ups are instead supplied by the MASTER device. The Dalf Board is capable of I2C communications at the FAST rate (400 KHz). The command set and capabilities of the I2C2 interface are similar to that of the TE Interface, but as with the API, the communication protocol is quite different. Commands and command arguments are passed to the board from the Host Application via message packets (a somewhat reduced version of the API packets). Similarly, data and status are returned to the Host Application with message packets. All packets have check sums to verify data integrity. A complete description of the protocol, packet format, and error handling is described in the separate “**I2C2 Interface**” document.

### **3.4 POT INTERFACES**

Two open loop pot control methods (POTC, POTF) and one closed loop control method (POTSVO) are provided by the firmware. All three modes employ external potentiometers (one for each motor) and an On/Off switch for safety (POTF requires an external switch for directional control). The switches should be SPDT (or SPST + pull down) and the POTS should not exceed 2K Ohms. POTC and POTF modes control velocity and motor direction, while POTSVO is a closed loop method that maps the pot setting to a motor position (“Giant Servo”) using either optical or analog encoders for motor feedback. See the sections of this document devoted to using the POT Modes for additional details.

### **3.5 R/C INTERFACE (3 CHANNELS)**

Two open loop R/C control modes (RCNRM, RCMIX) and one closed loop control method (RCSVO) are provided by the firmware. Each of the two open loop modes use just 2 of the 3 available R/C channels to control motor velocity and direction. Channel 3 is currently unused **but the firmware still measures and converts the pulse width for all three channels**, so the hardware and firmware structure is in place to easily use the third channel for another R/C control application. If you decide to use channel 3 you may want (for safety) to include it in the signal loss detection function along with channels 1 and 2. The third R/C control mode is RCSVO which maps the R/C transmitter switch to motor position (“Giant Servo”). See the sections of this document devoted to using the R/C Interfaces for additional details.

### **3.6 OTHER INTERFACES**

There is an additional serial interface that can be developed without board changes:

- **RX2/TX2:** The second USART on the PIC18F6722 is currently unused(\*), but is level shifted and routed to a 3-pin header JP1: [T2OUT, GND, R2IN] on the board. There is also a provision to use this second serial channel with +5V TTL levels. A separate 3-pin header JP2: [TX2, GND, RX2] provides for this possible use. See schematic for details.

**(\*) - This interface could be developed in a future firmware revision.**

## 4 The DALF-1 Board Detail

The board is designed around the PIC18F6722 microcontroller running at 40MHz (10MIPS). This board features lots of available I/O routed to screw terminals and ribbon connectors for easy access. Some of the IO is dedicated to supporting off board features, but there is a lot of unused I/O for customization. This includes analog inputs, interrupt inputs, GPIO's directly from the microcontroller, and 32 GPIO's provided by I2C I/O expanders!

### 4.1 Partial Feature List

- Dual open loop motor controls via R/C, Pots, or Serial Interfaces.
- Dual closed loop motor positional and velocity controls via R/C, Pots, or Serial Interfaces.
- PID and Trajectory Generator employed for smooth velocity ramping in closed loop controls.
- Standard quadrature encoder inputs (2 motors) allow maintenance of position and velocity.
- Analog motor position feedback supported for some applications.
- Giant Servo mode using either Pot or R/C as inputs.
- Digitally adjustable voltage windows and interrupt handlers for fast over current response when used with off board current sensors.
- Support for PID “Tuning” via data capture using the “Step Response Command”.
- Adjustable Slew Rate for all motor control methods.
- Serial EEPROM (64K bytes) as well as internal EEPROM (1K bytes) for non-volatile storage.
- Parameter Block in EEPROM for customization of motor characteristics, operating mode, etc.
- Built in three channel R/C interface with optional mixing for open loop control.
- Two built in potentiometer motor control methods for open loop control.
- Terminal Emulator command line monitor using RS232 Channel 1.
- API cmd and monitor Interface using RS232 Channel 1 for use by a “smart” PC Application. (\*)
- I2C cmd and monitor Interface using secondary I2C Bus for use by a MASTER host Application.
- All features interrupt driven for efficiency.
- Robust 5V power supply, with external On/Off switch, provides ample power for BEC.
- RTC for timing and scheduling.
- Both primary and secondary I2C buses routed to connectors for potential off board use.
- Second RS232 interface (Channel 2; currently unused) routed to header.
- Firmware is updatable via a boot block loader using a serial cable to a PC.
- On board connector for additional code development using standard Microchip tools.
- Libraries of routines <dalf.lib>, callable from C or PIC Assembler, and the source code for the main loop and interrupt dispatcher <main.c> provide support for customization and enhancements.
- Lots of head room for code expansion both in Flash and RAM memories.
- Lots of unused I/O:
  1. 32 unused GPIO's from IO Extenders on the primary I2C bus.
  2. 3 (or 5 w/o use of POT Modes) unused digital I/O's from the micro.
  3. 1 (or 4 w/o use of R/C Modes) unused interrupt inputs.
  4. 4 (or 6 w/o use of POT Modes) unused analog inputs.

(\*) - A Windows GUI is under development. Check the website for status. One nice feature of the GUI is that it provides an alternative to the use of a terminal emulator for the board setup task.

## **4.2 Power Supply**

The board provides a +5V supply (VDD) derived from a VBATT DC source that may be in the range of [+8V ... +40V]. The switch mode power supply, based on the LM2672 part, provides up to 1 amp at VDD=+5V. This is a robust supply that provides significantly more power than is required by the board. As a result, a significant portion of this power (at least 500 mA at +5V) is available for off board functionality thru a battery eliminator circuit. It may be used to provide +5V power for an R/C receiver. A resistor divider from VBATT to ground is routed to a dedicated analog input pin on the micro to provide VBATT monitoring capability. A calibration parameter is provided in the Parameter Block to “tune” the measured value to compensate for the actual values of the resistor divider pair. A connector for an external switch (SPST) is provided which may be utilized to turn off the power supply. To use this feature, just wire the switch terminals to the screw terminal pair labeled [GND OFF#]. When the OFF# input is grounded, the power supply is disabled. When the OFF# input is disconnected, the supply output is enabled. It is sometimes necessary to add a small load resistor (270 Ohms) between VDD and GND to get a clean reset signal to the PIC microcontroller when using this feature.

For those using this board with the Microchip development tools (ICD2), the board may alternatively be powered thru the modular 6-pin programming/debugging connector. When using the ICD2 to power the board, a switch is provided to isolate the local supply and an LED indicates the switch position.

## **4.3 PIC18F6722 Microcontroller**

The microcontroller is the Microchip PIC18F6722 part. This is a 64 pin part provided in a TQFP package. The data sheet available from the Microchip website provides extensive detail on the part features, but here is a brief summary:

- 64K instruction words (128K bytes) provided in the on-board flash program space
- 3,936 bytes of RAM
- 1K bytes of built-in EEPROM.
- 7 I/O ports with 12 A/D inputs.
- 5 CCP Modules (Capture/Compare/PWM)
- 5 general purpose timers, 1 watchdog timer.
- 2 USARTS
- 2 I2C Modules
- 

This microcontroller permits several oscillator options including an internal, tunable, oscillator. Only the external oscillator option is supported on the Dalf-1 board with the design assuming an external oscillator running at 40 MHz to derive the system clock (**fOSC= 40 MHz**). The system clock is divided by 4 to obtain the instruction clock frequency (fOSC/4 = 10 MHz). As most instructions execute in one cycle, this provides an instruction rate close to **10 MIP's**. The board also uses an external 32.768 kHz crystal as an input for the 32-bit Timer1. The firmware uses Timer1 to support a RTC as well as timed delay routines.

## **4.4 Motor Driver Connection**

This section describes the connections to a Motor Driver like the OSMC ESC board. The discussion here is targeted specifically for OSMC Drivers, but the content (eg; pinouts) is needed for connection to other drivers as well. There are two 10-pin connectors (PL1, PL2) on the Dalf board suitable for direct connect to the corresponding connectors on two OSMC boards using a standard 10 pin ribbon cable. The cable connects the 10 pin OSMC connector (CNS) and the corresponding 10 pin connector (PL1 for Motor1, PL2 for Motor2) on the Dalf-1 Board.

The following table shows how the OSMC signals are used on the Dalf Board:

PIN#	OSMC Signal (CN5)	Dalf Signal (PLx)	Comment
1	+12 (pwr)	12V_x ;x=1,2 (pwr)	Optional fan driver pwr.
2	+12 (pwr)	12V_x ;x=1,2 (pwr)	
3	VBATTDIV (out)	NC	Unused
4	DISABLE (in)	DISx ;x=1,2 (out)	Disable Motor Drive
5	AHI (in)	12V_x ;x=1,2 (pwr)	Pulled High
6	ALI (in)	PWMx ;x=1,2 (out)	Motor Speed Control
7	BHI (in)	12V_x ;x=1,2 (pwr)	Pulled High
8	BLI (in)	DIRx ;x=1,2 (out)	Motor direction
9	GND (pwr)	GND (pwr)	
10	GND (pwr)	GND (pwr)	

In the table, “x” stands for motor#. For example, the DISABLE pin on the OSMC board controlling Motor1 is attached to the DIS1 line on the Dalf Board. The regulated +12V output from the OSMC board is used to tie the AHI and BHI lines high thru the ribbon cable attachment on the Dalf Board. The +12V supply from the OSMC board is also used to supply fan output power on the Dalf board. The outputs from the OSMC boards are kept separate on the Dalf Board, hence there are +12V\_1 and +12V\_2 signals. The VBATTDIV output from the OSMC board is not used.

The three main motor control outputs from the Dalf Board to the OSMC board are **DIR** (direction: 0=Fwd, 1=Rev), **PWM** (speed governed by Duty Cycle), and **DIS** (disable: 0=Enabled, 1=Disabled). DIS=1 will disable motor output from the OSMC board regardless of the other parameters.

### PWM Complement Trick

The H-bridge driver (HIP4081A) on the OSMC board allows motor direction reversal without the necessity of external logic to switch the PWM input to different legs of the drive FETs. The Dalf firmware, by default, uses a “trick” to take advantage of this feature. If your motor driver is not the OSMC product, you may need to disable the trick which is controlled by the “osmc” bit setting in the Parameter Block.

#### **osmc='1'**

This is the correct setting for the OSMC Motor Drivers. With this setting, to accomplish a move in the REV direction with a speed corresponding to a PWM duty cycle of 70%, the DIR line is driven hi, and the duty cycle is set to 100-70=30%. The firmware takes care of the duty cycle reversal details, so if the desired speed corresponds to a PWM duty cycle of 70%, the input is 70 regardless of motor direction. This “trick” is enabled for both motors by default (“osmc1=osmc2='1'”), but can be disabled by changing the parameter (for the motor in question) in the Parameter Block.

#### **osmc='0'**

Many non-OSMC motor drivers don’t require the “trick” and will require the osmc bit to be cleared for correct operation. With this setting, to move in the REV direction with a speed corresponding to a PWM duty cycle of 70%, the DIR line is driven hi, and the duty cycle is set to 70%.

You can easily determine what your driver expects by using the command interface to drive the motor slowly in reverse. If it goes fast instead, you will need to change the setting of the osmc bit in the Parameter Block.

**A note about MTR1 and MTR2 LED’s:** These non programmable, Bi-Color LED’s, are connected between the PWM and DIR signals to indicate motor direction and speed. Forward direction produces green, and reverse red, with LED intensity corresponding to speed. If your motor driver requires the osmc='0' setting, the LED intensity vs. speed relationship, for reverse direction only, will be reversed (that is; dim red will correspond to high reverse speed, while bright red will represent low reverse speed).

**A note about DIS polarity:** If you are using a motor driver other than the OSMC and it requires the DIS line to be asserted LO (that is; motor disabled whenever DIS = 0 Volts), you will need to set the “**dis\_activelo**” bit in the appropriate MODE byte in the Parameter Block. This will reverse the normal, default, polarity of the DIS signal to accommodate your driver.

## 4.5 RS232

The board provides a DSUB9 connector for an RS232 Interface to support a Command Line Interface to a Terminal Emulator Application running on a PC. The required RS232 level shifting is achieved with the MAX233 part. If you have a PC running Windows, you probably have the Hyperterm application which is generally supplied as part of a Windows OS. Alternatively you can download TeraTerm (free) from the web. Personally, I like TeraTerm better than Hyperterm, but either will work fine and they are both easy to configure: the default communication parameters are: 19,200 baud, 8 data bits, 1 stop bit, and no parity. The baud rate can be changed after powerup - see the nBR parameter in the Parameter Block. The command interface allows you to easily control the motors and obtain motor status from your PC. The command interface has many other uses as well including the ability to directly access (read/write) the RAM and special function registers (SFR’s) that control the processor peripheral modules.

The hardware (MAX233 and PIC18F6722) provide a second (currently unused) RS232 channel. This second channel is made available either level shifted or at +5V TTL levels by routing the signals to 3 pin headers JP1, JP2 (note usage of R1 in the schematic).

## 4.6 External EEPROM/Parameter Block

The Dalf board provides 2 separate non-volatile memories. The largest of these is a 64K Byte I2C serial EEPROM (24LC512) device mounted on the board. The low order address pins for this EEPROM part are tied to GND creating an **I2C bus address of: ‘0xA0’**. The write access to this part provides for both a single byte write as well as a 128 byte page write. With either a single byte write, or a page write, there is a 5 msec write cycle delay after sending the last byte before the device can again be accessed. The **Parameter Block** occupies the first 0x80 (128) bytes of this EEPROM and provides for non-volatile storage of motor parameters et al. The contents of the Parameter Block are explained in detail in the appendix. All that you need to know at this point is that:

- (1) During power-up, the values in the Parameter Block are copied to the ERAM portion of RAM, establishing the **Runtime Environment**. This run-time environment is then used by the system initialization code to configure and initialize the various devices (eg; PWM frequency, A/D behavior, PID parameters, operating mode, etc.).
- (2) Any value in the Parameter Block may be changed to affect the run time environment on all subsequent power-ups and there is more than one way to do this. See the Command Section and Appendix D for details. In addition, if you mess up and alter the data in the Parameter Block in such a way that you can’t power-up cleanly with the resulting run-time environment, there is a mechanism involving use of the push-button BTN to restore the original factory default data to the parameter block.

You may also change values in ERAM directly using the “Write Memory Byte” Command”. Changes to ERAM affect the current session only and some changes will affect behavior immediately. This is convenient for example during PID motor tuning.

#### **4.7 Internal (Built-In) EEPROM**

The second non-volatile memory is provided by a 1K Byte EEPROM built into the PIC18F6722 microcontroller. This memory is accessible thru the command interface and library routines, but is currently unused. Like the external eeprom, there is a write cycle delay.

#### **4.8 Fan Driver Outputs**

The Dalf Board has 2 high current outputs capable of driving cooling fans or other inductive type loads. If you are using OSMC Driver Boards, these come with PM DC brushless cooling fans. Each fan is rated at +12V, 170mA and may be connected directly to the OSMC board in which case it will always be on. The Dalf Board outputs provide an alternative connection for the fans allowing them to be controlled by the PIC processor. For each fan, the board provides: drive circuitry via an NPN 2N2222A transistor, base resistor, free wheeling diode, and a two pin screw terminal connector. The power source is routed from a pin on the 10-pin PLx connector and will be +12V if you are connected directly to the OSMC board thru the ribbon connector. Individual fan on/off controls are routed from the PIC to the base of the drive transistors. See the command interface for details of fan control.

There are a couple of noteworthy things to mention here about the fan drivers:

- 1) They may be used for other, high current, inductive load, applications. If you are not using the OSMC drivers, you can still use the Fan Driver Outputs, but you will have to provide your device supply voltage on the appropriate PLx pin. You should review the schematic, the PLx connector pinout, and the transistor specification to ensure that this will be suitable for your application.
- 2) If used for fan controls, they aren't much use without temperature sensing. The firmware does not provide temperature sensing, but I have provided a couple of ideas below that don't require board modification to achieve it. Both of these temperature solutions will require you to write some code.

##### **Temperature Sensing Suggestions:**

- The board provides pads (schematic: RT1, RT2) for a pair of thru-hole 10K resistors (not loaded) tied between VDD and analog inputs (AN4, AN5). Add the RTx resistor and mount an off board 10K NTC thermistor on the temperature source (eg; drive FET) with outputs routed to GND and the corresponding analog input on the Dalf Board. The A/D reading of this voltage divider will routinely be captured and stored by the existing firmware. The values stored will be a reasonably linear voltage response versus temperature.
- Alternatively, use I2C temperature sensors and connect the leads to the screw terminals for the primary I2C bus. Write code to access the sensors as I2C slave devices.

#### **4.9 EX0, EX1, EX2, EX3**

All of these signals are connected to interrupt capable inputs on the PIC microcontroller. They can all be used for standard GPIO's, or general purpose interrupts. EX0, EX1 and EX3 have additional capabilities because they are connected to CCP (Capture/Compare/PWM) microcontroller pins. With a bit of programming on your part, the signals on these pins can be quite versatile. Each of the EX0, EX1, and EX3 signals can become PWM outputs, or they can operate in capture mode to get accurate signal timing on the input, or they can generate outputs with accurately controlled timing. The default firmware uses EX0, EX1, and EX3 in capture mode for pulse width detection to support a 3 channel Radio Control Interface. EX2 is connected to the KBIO pin on the PIC and has interrupt-on-change capability, but is currently unused.

## **4.10 A/D Inputs**

The firmware enables 7 of the A/D inputs on the PIC processor (AN0... AN5, AN6). An external connector routes [AN0 ... AN5] thru RC filters to the A/D input pins. The dedicated AN6 analog input monitors the battery (VBATT) voltage. An optional, configurable, low pass, digital recursive filter can be applied to any of the raw A/D inputs - See the FENBL and DECAY parameters in the Parameter Block.

A mechanism using a hardware timer and an interrupt driven state machine is provided by the firmware to cyclically sample and store the A/D readings in RAM without the use of software delays. The timing for the acquisition, conversion, and between -channel-delays for the signals is controlled by 3 values in the Parameter Block - see AD\_ACQ, AD\_CNV, and AD\_GAP in the Parameter Block. A “snapshot” of the A/D readings can be viewed thru the command interface at any time. With the default timing values in the Parameter Block, the sequence of 7 readings completes in a bit less than 5 msec (about 220 times a second). Because the design is interrupt driven, the actual number of CPU cycles devoted to the A/D processing is small.

When external pots are used for open loop (POTF, POTC) or closed loop (POTSVO) control of the motors, 2 of the analog inputs [AN0, AN1] are dedicated to provide the motor control input signals for motor1 and motor2 respectively. The middle pins (wipers) of the external pots are routed to the AN0 and/or AN1 screw terminal inputs.

If analog feedback is used in place of optical encoders for motor position, 2 of the analog inputs [AN2, AN3] are dedicated for motor1 and motor2 position feed back respectively.

If you will be using one of the pot motor control methods, see the discussion on Pot Control Modes and the usage of the variables PMSP and AMINP.

## **4.11 Over Current Sense**

The Dalf Board provides hardware and firmware support for over current detection and response when used with compatible off board sensors. Two different, built-in, responses are possible if your application needs over current protection. See the section “Current Sensor Feedback” for details.

## **4.12 Position Encoder Inputs**

The Dalf Board has built-in support for use of either standard optical incremental motor encoders or absolute analog encoders for use in sensing the output shaft position. If you don't have motor encoders, the PID, Trajectory Generator, and closed loop operations will not be available to you. However, lack of encoder(s) will not keep you from controlling the motors in open loop modes with R/C, pots, or the serial interface(s).

The encoders may be mounted directly on the motor shaft or “downstream” after a gear reduction. If your application will use encoders you should study the section “Motor Encoder Feedback” in this manual which discusses the details of usage, hookup, and limitations of encoders.

Depending on the type of encoder, the hookup is different. For optical encoders the board provides two 5-pin screw terminal connectors (VDD,GND,A1,B1,Z1), (VDD,GND,A2,B2,Z2) - one connector for each motor. For analog encoders the board provides (VDD,GND,AN2) screw terminals for motor1 and (VDD,GND,AN3) for motor2. Any available VDD and GND terminals may be used for analog encoder hookup, but it probably makes sense to use those provided for the optical encoders since they won't be otherwise used.

### **4.13 I2C Communication**

The board provides two 4-pin connectors (+5V, GND, SDA, SCL) for off board communication using either the primary or secondary I2C busses. The I2C Interface developed by Phillips may be used to communicate with a variety of peripheral devices (LCD's, EEPROM's, Extended I/O Devices, Smart Batteries, Microcontrollers, Temperature Sensors, navigational sensors, et al.). The primary I2C bus (SD1, SC1) connects the micro to the on-board devices in a MASTER/SLAVE configuration running at a clock rate of 400 KHz. The table below describes the I2C SLAVE devices on the primary I2C bus.

Address	Device	Access	Description
0xA0	IC4 - 24LC512	R/W	256 KByte EEPROM
0x42	IC7 - MCP23017	R/W	IOEXP1
0x40	IC8 - MCP23017	R/W	IOEXP2
0x52	IC9 - MAX5478	W	50K Digital Pot1
0x50	IC10 - MAX5478	W	50K Digital Pot2

Access to the parts is provided thru library routines and the CMD Interface. The 7 high order bits of the address are the actual I2C device address. Bit 0 of the address is the read/write# control (R/W#).

The secondary I2C bus (SD2, SC2) is configured so that the PIC18F6722 microcontroller is a SLAVE Device, allowing an off board MASTER Device to host this I2C Bus. The Master issues commands and receive status from the Dalf Board. This interface is described in more detail in the "I2C2 Interface" document.

### **4.14 I/O Expanders**

The board has two I/O Expander Parts (MCP23017) which provide a total of 32 configurable GPIO's for potential off-board use (sensors, memory, led's, or whatever). The 16 GPIO signals from each device are routed to 16-pin ribbon connectors (J5 and J6) and accessed by reading and writing registers on the I2C device. Each GPIO pin can be configured as either input or output with a drive level of 25 mA, but the device has a total current limit of 200 mA which must be observed. The firmware and the board routing does not support the interrupt on change feature of these parts. Device level read/write operations can be performed thru the command interface and library functions.

#### **J5 and J6 Connector Pin-Outs:**

Connectors J5 and J6 have identical pin-outs as described below. Pin 1 is clearly marked on the board layout, but is unfortunately covered by the connector on an assembled board. Note that the low order bits of PortA and PortB are at diagonally opposite corners of the connector.

Pin#	Name
1	GPIOA.0
2	GPIOA.1
3	GPIOA.2
4	GPIOA.3
5	GPIOA.4
6	GPIOA.5
7	GPIOA.6
8	GPIOA.7
9	GPIOB.7
10	GPIOB.6
11	GPIOB.5
12	GPIOB.4
13	GPIOB.3
14	GPIOB.2
15	GPIOB.1
16	GPIOB.0

## 4.15 Connectors

The board provides the following connectors (Listed logically rather than literally (eg; the 4 pin Interrupt connector and 4 pin I2C connector live on a single 8-pin connector). In addition see the pads provided for the 2<sup>nd</sup> USART Interface and the TP2 pad.

- PortD: [D0...D3]
- Extended I/O 1: [Y0...Y15]
- Extended I/O 2: [X0...X15]
- External Interrupts: [EX0...EX3]
- I2C\_1: [GND, VDD, SD1, SC1]
- I2C\_2: [GND, VDD, SD2, SC2]
- ICD2 (modular connector for development/debug): [MCLR#, VDDEXT, GND, PGD, PGC, NC]
- DSUB9 (cmd interface and bootloader): [... GND, T1OUT, R1IN, ..]
- Analog: [AN0...AN5]
- PL1 (Motor1 Control): [..., PWM1, DIR1, DIS1, +12V\_1, GND ...]
- PL2 (Motor2 Control): [..., PWM2, DIR2, DIS2, +12V\_2, GND ...]
- PwrSupply (On/Off): [GND, OFF#]
- Current Sense1: [GND, VDD, IS1]
- Current Sense2: [GND, VDD, IS2]
- Quadrature Encoder1: [GND, Z1, A1, VDD, B1]
- Quadrature Encoder2: [GND, Z2, A2, VDD, B2]
- Fan1: [F1-, F1+]
- Fan2: [F2-, F2+]

## 4.16 LEDs

There are a total of 8 LED's on the Dalf-1 board. Three of these are programmable as indicated in the following table.

Name	Programmable?	Usage	Color
MTR1	No	Motor1 Speed & Direction	Bicolor: RED/GRN
MTR2	No	Motor2 Speed & Direction	Bicolor: RED/GRN
PWR	No	VDD Indicator	RED
PGM	No	Boot loader mode active	YEL (normally off)
ICD	No	Powered by ICD2	YEL (normally off)
LED1	Yes	Motor 1 Status	GRN
LED2	Yes	Motor 2 Status	GRN
LED3	Yes	Error Indicator	RED

LED1, LED2, and LED3 are programmable with different blink patterns to indicating status.

LED	State	Usage
LED1 (Mtr1) LED2 (Mtr2)	OFF FAST BLINK SLOW BLINK FULL ON	No Pwr Pwr, TGA active Pwr, TGA inactive, Vel>0 Pwr, TGA inactive, Vel=0 (stall)
LED3 (Error)	OFF FAST BLINK SLOW BLINK FULL ON	No Error Over current (possibly transient) R/C signal loss (possibly transient) Low Battery (VBATT < VBWARN)

#### 4.17 Switches

- **PGM** - This switch puts the firmware in boot loader mode on power-up. This is used only to download and flash an image using the serial modem link to a PC. The PGM LED will be on while the boot load mode is active. In the normal (non boot load) mode of operation, this switch will force the PGM LED to be off.
- **ICD** - This switch isolates the local +5V power supply from the +5V power supplied by the ICD2 tool used during code development/debug. The ICD LED will be on to indicate when the ICD2 Device is supplying power to the board. In the normal (non ICD2 development) mode of operation, the switch position will force the ICD LED to be off.

#### 4.18 Headers

- JP1 - This 3-pin header connector provides access to the level shifted version of the second serial channel. Note the use of zero-ohm R1 in the schematic.
- JP2 - This 3-pin header connector provides access to the +5V TTL level signals of the second serial channel. Note the use of zero-ohm R1 in the schematic.
- JP3 - This 4-pin header is supplied for potential use to easily daisy chain the secondary I2C bus.

## 5 Customization and Configuration

Customization can be achieved thru program changes to the contents of the FLASH or data changes (configuration) to the contents of the EEPROM. Files are provided on the CD (dalf.lib, main.c, et. al) to assist in code customization.

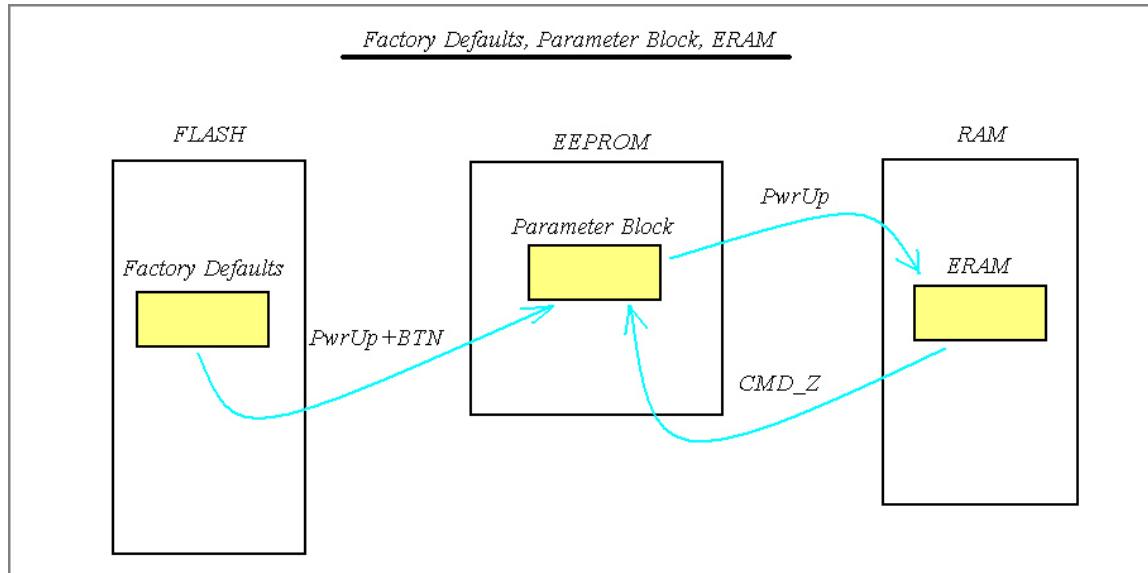
### 5.1 EEPROM Configuration

Detailed description of the Parameter Block Contents and commands available in the TE Command Interface to alter the contents appear in later sections of this document. For now, I want to describe the general function of the Parameter Block and the role it plays in determining operating conditions for the Dalf Board. During the early part of power up and system initialization, the entire Parameter Block is copied into a portion of microcontroller RAM that I refer to in this document as **ERAM** (Eeprom RAM Copy). The Parameter Block (and EEPROM in general), is not accessed after power-up (unless a command is issued to do so using one of the command interfaces). ERAM constitutes the **Runtime Environment** and is used to configure the devices and operating mode of the board.

**Unless you make changes to ERAM or the Parameter Block after power up, the content of ERAM and that of the Parameter Block will always be identical.** A reasonable way to think of these two memory areas is that ERAM is the current working copy of the Parameter Block.

### 5.2 Factory Defaults, Parameter Block, and ERAM

The diagram below describes the relationship between the Parameter Block (in EEPROM), ERAM (in RAM), and the Factory Defaults (in FLASH):



When you receive the board from the factory, the factory defaults will have been stored in the Parameter Block so that when the board is powered up it will use the defaults.

### 5.3 Changing the Configuration

Using the CMD Interface (any of the 3 serial links), there are two ways to change values in the Parameter Block (first 128 bytes of the external 64K serial EEPROM) to customize operation of the board:

- Write the data directly into the location(s) in the Parameter Block. This capability is provided by the “Write Memory Byte” command.
- Write the data first into ERAM using the “Write Memory Byte” command and then later upload (copy) the entire ERAM block into the Parameter Block using Cmd\_Z. This option is useful when evaluating those changes (eg; PID Tuning) that take effect immediately upon ERAM update. In this way you can, in some cases, observe the effect of changes without requiring a board reset.

To restore the factory default values to the Parameter Block, simply power up with the BTN switch depressed.

Regardless of the method chosen to make changes, you will need to know where to make the changes in ERAM or the Parameter Block. See Appendix D for a layout of the Parameter Block and the corresponding locations in ERAM. See the section in the CMD Interface for details of the commands needed to make the changes.

What you need to change, if anything, in the Parameter Block will depend on your application and usage. The table below may help you focus on the parameters that are important to your application. Across the top of the table are modes and activities and in the first column is a list of all of the parameters in the Parameter Block. An “X” in the table indicates that the parameter may apply to the column operation. A blank box indicates that the parameter probably doesn’t apply to the specified operation. The “System” heading is for general operations that don’t naturally fit in the other categories (eg; VBATT monitoring, communications, etc.). If the parameter does apply to your intended usage, I recommend that you search the documentation for that parameter to find the appropriate details. Even if the parameter applies to your application, the default value may be fine for you. Some parameters are motor specific and there is a parameter for each motor (eg; VMAX1, VMAX2). In these cases, I have listed the parameter only once in the table (eg; VMAX) - see Appendix D.

### 5.4 Parameter Block Change Guide

Parameter	CMD Open Lp	CMD Closed Lp	POTF, POTC	RCNRM, RCMIX	POT SERVO	RC SERVO	ANALOG Feedback	I/O	System
fPWM	X	X	X	X	X	X			X
AD_ACQ			X		X		X		X
AD_CNV			X		X		X		X
AD_GAP			X		X		X		X
SYSMODE									X
X1_Regs								X	
X2_Regs								X	
VBCAL									X
VBWARN									X
AMINP	X		X	X					
MAXERR		X			X	X			
MAXSUM		X			X	X			

### Parameter Block Change Guide (Cont'd)

Parameter	CMD Open Lp	CMD Closed Lp	POTF, POTC	RCNRM, RCMIX	POT SERVO	RC SERVO	ANALOG Feedback	I/O	System
MODE1	X	X	X	X	X	X	X		X
MODE2	X	X	X	X	X	X			X
MODE3	X	X	X	X	X	X	X		X
ACC		X			X	X			
VMID		X			X	X			
VSP	X	X	X	X	X	X			
KP		X			X	X			
KI		X			X	X			
KD		X			X	X			
VMIN		X			X	X			
VMAX	X	X			X	X			
TPR (1)									X
MIN					X	X			
MAX					X	X			
RC1MIN				X		X			
RC1MAX				X		X			
RC2MIN				X		X			
RC2MAX				X		X			
RC3MIN				X					
RC3MAX				X					
RCD				X					
POT1A (2)									X
POT1B (2)									X
POT2A (2)									X
POT2B (2)									X
Nbr									X
NID									X
RX1TO									X
NPID (3)		X							
DALFA									X
RCSP				X		X			
PSP			X		X				
DMAX							X		
FENBL					X	X	X		X
DECAY					X	X	X		X

- (1) - Assuming that you have motor encoders, TPR is necessary to get reliable motor velocity in units of RPM , but this is informative only and not used in any motor control features.  
 (2) - The on board pot settings are important only if over-current protection is enabled.  
 (3) - NPID is only used by Command Q (“PID Tuning”).

### 5.5 Some Important Parameters

A few parameters are of fundamental importance. This section provides a bit of guidance on how to make the adjustments to these parameters. If you have not read the “Getting Started Manual”, it would be a good idea to do so before making the changes. I recommend making all of these changes using the CMD Interface with a Terminal Emulator Application. When the Windows GUI (in development) is available it will make the setup process a bit more straightforward.

### **AMINP**

This parameter is the acceleration (slew rate) governor for all open loop motor operations. When using the CMD Interface, it is also the default slew rate if you don't enter that parameter in the command. For a dramatic illustration of its effect, you can issue the same motor movement command but with different slew rates. Example: With motor1 stopped try "X1 0 64 0" to run the motor at maximum forward velocity. This will actually use AMINP as the slew rate instead of the zero that was entered (AMINP is the minimum value for the actual slew rate). Now stop the motor and issue the same command but with a different slew rate: "X1 0 64 FF". You should notice a dramatic difference in the acceleration to final speed.

To set a reasonable value for AMINP, run your motor at full velocity and then use Cmd\_O to stop the motor (eg; "O1"; that is Oh, not zero). If the transition is too abrupt, increase the value of AMINP. If the transition is sluggish, or you believe that the motor should be stopped more quickly, decrease the value of AMINP. This value affects all open loop motor control commands (sets the maximum acceleration rate) and all situations (including PID) where the motor must be stopped because of an error situation.

### **VSP**

This is an important parameter for closed loop motor operations. When the Trajectory Generator is active, a new positional target for the motor is generated every VSP msec. When PID is active, new motor control signals are generated every VSP msec. See the Trajectory and PID sections of this document for a discussion of how to determine an appropriate value for your application.

### **VMAX**

Set VMAX to 0x64 (the default), unless you want to restrict the effective voltage to your motor. The value of VMAX is the maximum duty cycle that will be applied to the motor. A smaller value may be appropriate if you do not want the effect of full VBATT power applied to your motor. Do not set VMAX to any value over 0x64=100%.

### **VMIN**

VMIN should be set to something small. The factory default is VMIN=0x02=2% which will probably be fine for you. In closed loop operations, when PID is active, the value of the PWM duty cycle will never go below VMIN. VMIN is useful to deal with frictional forces (deadband startup) and to maintain some force on the motor even when stopped to keep it at the current position.

### **TPR**

The Ticks Per Revolution parameter affects the velocity (shown in units of RPM) when using Cmd\_V to "Get Motor Velocity". If you do not have motor encoders it is not important to set it. Otherwise the value to record is 4\*CPR where CPR is the "Cycles Per Revolution" specification for your encoder. Example: CPR=22 → TPR=88, so you should record 0x0058 (88) in the TPR word parameter. If you are using an analog motor encoder instead of an optical encoder, you should set TPR=0x100 (256) to corresponding to the [0...360] degree range (0x00 to 0xFF on the ADC readings).

### **VBCAL**

This is a scaled calibration constant used in computing the VBATT voltage based on the ADC measurement on pin AN6 (voltage divider input). It is designed to allow you to "tune" the VBATT battery monitoring gauge and in particular to compensate for inaccuracies in the R3/R4 resistor divider pair used to capture the VBATT voltage. See Appendix D for additional detail and a recommended adjustment procedure.

### **VBWARN**

When set properly, this parameter will make LED3 useful as a low battery warning. In addition, setting this Low Batt Threshold to a reasonable value so that the LED is off in normal operation allows the LED to show other error indications (eg; over-current). Example: Your nominal VBATT is 18V and you want the low batt warning to occur at 16.5 Volts (=16,500 mV). Convert 16,500 to hex (0x4074) and record this in the word parameter VBWARN.

### **VMID**

If you do not specify a mid course velocity for a closed loop motor command, this parameter will be used instead. While not essential, it is quite convenient for routine motor commands to just use the default value. Of course that makes sense only if the default is reasonable. The value of VSP should be established before using this procedure to set VMID.

Procedure: Use Cmd\_X to run your motor at various speeds (eg; "X1 0 21" for 33% power) to find one that you want to use as the default for the midcourse speed in closed loop commands. Then use Cmd\_V (eg; "V1") to get the corresponding motor velocity in units of ticks/vsp. Use the result to set the VMID word parameter. For example, assume that your result is V=0x000019 (ticks/vsp). To set VMID to use this velocity as the default for midcourse velocity in closed loop commands, multiply the observed value by 256 (VMID is divided by 256 in closed loop usage). In this example, you would record  $256 * 0x19 = 0x1900$  for VMID.

### **ACC**

Similar to VMID, this parameter is the default acceleration input for closed loop commands if you don't supply it as part of the command. It is probably a good idea to "PID Tune" your motors (determine appropriate values for KP, KI, and KD) before following the suggestion here on setting ACC.

Procedure: Use Cmd\_Y to repeatedly issue closed loop motor commands and supply various values of the "a" input parameter until you are satisfied with the observed acceleration rate. Then simply record this value into the ACC word parameter for use as the default. I recommend starting out with small values for "a" (eg; 0x0005) and repeating the command with incrementally larger values (eg; increase by 0x0002) until you are satisfied with the results; 0x0005, 0x0007, 0x0009, 0x000B, ... and so forth. Once you are satisfied, simply record this value into the ACC word parameter.

### **MODE1**

You are probably fine with the default here, at least initially. Possible exceptions might be the need to change the state of the "aleadsb" or the "dis\_activelo" bits. If only the "aleadsb" bit is in the wrong state, you can avoid a change by simply reversing the connection of the encoder A and B signals at the Dalf screw terminal connectors. Later you may want to change some of the other control bits in this parameter - See Appendix D for details.

### **MODE2**

Bits in the MODE2 parameter control operating mode on power up. For example, if you want to control the motors with R/C or Pots, this is the place to make those changes. See Appendix D for details.

### **MODE3**

If your motor driver is something other than the OSMC Driver, then the "osmc" bit may need to be cleared (default is "osmc"=1). If this bit needs to be changed, you should do it right away to avoid confusion on commands that involve motor reversal.

### **nBR**

This is the default index that governs the serial (RS232) baud rate. The default index produces 19.2K, but slower and faster rates have been tested without issues. See the table in Appendix D for details.

### **KP, KI, KD**

Only important for closed loop motor control operation. See "PID Motor Control Tuning" for details.

## **5.6 Programming Customization**

For customization via code development you use the 6-pin modular connector with a tool like the ICD2. The other alternative uses the boot loader option to download the binary image over the serial port. For development or debug, I recommend ICD2. For a code upgrade only, the boot loader is fine.

- **Modular Connector**

The 6-pin modular connector fits a cable coming from the Microchip ICD2 development tool. The ICD2 tool works in conjunction with the MPLAB IDE (Integrated Development Environment) and allows for code development, on-board programming of the flash part, and real time debugging.

When using the ICD2, the Dalf-1 board is powered thru the connector. An on-board switch provides isolation of the local power supply from that provided by the ICD2 tool and an LED indicates the switch position. For details on the tools, see the “Development Tools” section of this document and of course the Microchip web site.

- **DSUB9 connector (RS232 serial)**

With this method the binary image is downloaded and self flashed into the PIC18F6722 part using the RS232 serial line with a standard modem cable connected to a PC. This Boot-load mode is selected via an on-board switch with an LED indicating the switch position. A boot loader Application Program (running on the PC) and special power-up boot loader code running in the PIC18F6722 (because of switch detection) act together to self-flash the binary image. When the process is complete, the PC application is terminated, the switch on the board is put back in normal mode, and the board is rebooted (reset) to power-up running the newly flashed image. This programming method would be used by those wanting a flash update (from me or someone else), but not having the ICD2 tool. See the section “Programming Using the Boot Block Feature” for additional details.

## 6 Development Tools

### 6.1 Compiler, Assembler, Debugger

The firmware for the Dalf-1 board is a combination of code written in PIC Assembler (MPASM Assembler) and the C Language (MCC18 Compiler). The low-level firmware consisting of system initialization, device level support, command handlers and the Interrupt Service Routines is written in PIC Assembler. The main loop, interrupt dispatcher, and the services scheduled by interrupts are written in C. Additionally, much of the serial output to the Terminal Emulator Screen is written in C to take advantage of the printf() functionality. The source code for all of the C code is on the CD provided in the box with the product. In addition, the assembler code is provided in library format allowing it to be linked with other files Dalf board customization.

All of the code for the Dalf-1 firmware was developed and tested using standard Microchip tools. The prices for the development tools, current at the time this document was publicized, are listed below. The tools are all available for download or ordering directly from the Microchip Web Site.

- |  |                              |
|--|------------------------------|
| • MPLAB IDE: Integrated Development Environment. | [FREE]                       |
| • ICD2: Flash programming, Real time debugging.  | [\$159.99]                   |
| • MCC18.EXE: C compiler.                         | [\$495.00; Free for 60 days] |

The ICD2 product is a cheap, but very functional, alternative to an In-Circuit-Emulator (ICE). Other alternatives to the mcc18.exe compiler include products from HiTech, IAR, and CCS. The HiTech compiler is more expensive than the mcc18. The CCS compiler is less expensive.

I recommend the free version of the mcc18 over the other alternatives. The mcc18 product is a reliable compiler with ANSI extensions and permits 24 bit data values. If you choose a different compiler for customization of the existing firmware you will have some issues to deal with. First you should know that the supplied library functions depend on the runtime environment of the mcc18. For example, calls from C to Assembler functions assume parameters are passed on the software stack. The other compilers have different approaches and will require code modifications to deal with issues like parameter passing, and returning results to the caller. These are “solvable” issues, but they will result in some additional work if you expect to customize the existing firmware.

Perhaps the most compelling reason for use of the mcc18.exe compiler is that **Microchip offers a free (for 60 days), full featured, download of this product from their website. I was told (Microchip sales rep) that the free version, remains fully functional after the 60 day trial with the exception that the code optimization features will no longer be available.** Frankly, when used with the PIC18F6722, that makes the choice of a compiler pretty much a “no-brainer” to me. The code size of the existing firmware is only about 35% of the flash capacity, leaving lots of headroom for customization using a non-optimized image.

If you are doing C development using this board and the Microchip Compiler <mcc18.exe>, you will need to use a modified version of the startup code <c018i.c>. The only required change to the <c018i.c> file supplied with the compiler is the relocation of the reset and interrupt vectors to addresses after 0x0800 (instead of 0x0200). The <c018i.c> file (and the compiled c018i.o file) with the required changes is included on the CD. See the discussion of the Boot Loader for additional details. The <dalf.lib> file was created using the {Large code, Large Data, Multibank Stack} model settings.

## **6.2 Software Development Tool Version Numbers**

Tool	Tool Description	Version#
MPLAB IDE	Integrated Development Environment	8.00
MCC18	C Compiler	3.10
MPLINK	Linker	4.10
MPLIB	Librarian	4.10
MPASM	Assembler	5.10

Beginning with Dalf firmware version 1.71, the Microchip Development Tools listed in the table above are required to guarantee successful flash image creation when using the development files provided on the CD. Earlier releases of these Dalf files were compiled with the Microchip Tool Suite accompanying mcc18 version 3.0. Those earlier Dalf files have symbol names which conflict with the tools in the table above.

## **Flashing the Microcontroller Using the Boot Loader**

The boot loader firmware in the boot block section of the PIC FLASH operates in conjunction with the Windows Application **p1618qp.exe** (PIC16/PIC18 Quick Programmer). This application uses a serial cable to download the flash image (\*.hex) from the PC to the PIC Microcontroller. The p1618qp.exe controls this process and directs the PIC which runs the boot loader code to self-flash the program data that is down loaded from the flash image program file. The p1618qp application expects the flash image file to be in INHX32 format.

### **Use [file/Export]**

If you have created the image with the MPLAB IDE Development Environment, it is a good idea to use the ‘export’ feature of MPLAB to export the .hex file to the actual file that you will supply to the p1618qp.exe tool. If you are upgrading from a file supplied by EE, this step will already have been done for you. The export function does a bit of “housekeeping” on the INHX32 file to make all data lines contain 16 bytes. This can avoid some error messages that p1618qp.exe may generate otherwise. Using the export feature also allows you to specify the starting address of 0x00800 for the output file and in this way avoid any possibility of the flash process overwriting the boot block code.

### **6.2.1 The Boot Block**

The Dalf Board comes with a firmware boot loader located in the first 0x800 bytes of flash memory (**boot block**). This boot loader code can be utilized with the p1618qp.exe PC application to download and flash a program image into the microcontroller. On a normal board reset the boot block code in the Dalf Firmware quickly determines that a power up is required and the code branches to perform system initialization. If instead boot loader operation has been requested, the boot loader code works in conjunction with the p1618qp.exe application to self flash a new firmware image. Boot loader operation is requested manually by placing the PGM switch in the position that turns on the yellow LED (\*).

(\*) - Alternatively, beginning with version 1.73, boot loader operation may be requested programmatically by writing 0xFF into the internal EEPROM of the microcontroller at location 0x03FF before the board is reset. The primary use for this new feature is to allow firmware upgrades to enclosed boards without having to open the enclosures to gain access to the PGM Switch.

The boot loader code supplied in the Dalf Board Flash Memory is a slightly modified version of the one found on the Microchip web site and described in the App Note: **AN851**. This application note is not specific to the PIC18F6722, but most of its content applies to the process described here. **It would be worthwhile for you to review the application note (which has considerably more detail than what is presented here) and the following discussion before attempting to use the boot loader feature.** The only boot loader modifications for this product are:

- Boot mode request is detected by either the switch or the last byte of internal EEPROM.
- The reset vector for normal run mode is redirected to 0x800, instead of 0x200.
- The interrupt vectors are redirected to 0x808 and 0x818 (instead of 0x208 and 0x218).

**The boot block section of program memory ([0x0000 - 0x07FF]) is reserved for the boot loader and the content of this section of FLASH memory should remain unchanged to allow for future code update support.**

### **6.2.2 Under The Hood**

When the microcontroller is reset, execution always starts at address 0x0000 in the boot loader. The first thing that the code does is examine the state of the **PGM** Switch to determine whether or not this will be a normal power up (Yellow LED off) or if the code should remain in the Boot Loader section to support self flash of a new firmware version (Yellow LED on). If the Yellow LED is off, the code branches to begin system initialization, but first examines the last byte (at 0x03FF) of the internal EEPROM. If the byte at

this address is 0xFF, it is interpreted as a programmatic request for boot mode operation. Note that the Yellow LED indicates the PGM Switch position only and will not indicate whether or not a programmatic boot loader request is being processed. In the event of a programmatic request, the request byte is cleared (0x00) and the code branches back to boot mode operation where it awaits communication with the p1618qp.exe application just as with the manual request.

**The procedure for updating the flash image is fairly simple, but success will depend on carefully following the procedure (“The Process”) described a bit later.** Assuming that you are using the p1618qp.exe application and the existing boot loader firmware on the part to flash a new image, you should never flash the first 0x0800 bytes. Corruption of the boot loader portion of the flash will result in a situation where you will be unable to use the boot loader to re-flash the part. If this were to happen, the flash image (including boot loader) would have to be restored using a device programmer (Eg; ICD2 using the modular 6-pin connector). Actually, in practice, it is difficult to make a mistake here even if you supply a flash image that includes code that resides in the boot block portion of memory. This is because the p1618qp.exe application has some safeguards inherited from the p1618qp.ini file that avoids flashing that area of memory.

### 6.2.3 Installing the p1618qp.exe Application

First install the p1618qp application on your PC if you don’t already have it. During installation, if the installer asks if you want to replace your system files (.dll’s, etc.) with older versions from the application, I recommend that you keep your newer files. **After installation of <P1618QP> is complete you will need to replace one of the files installed for the application with one that is provided on the CD.**

When you run the p1618qp.exe application it reads a p1618qp.ini (text) file that lists supported PIC devices. The version of the p1618qp.ini file installed by the application is somewhat “dated”, and will not recognize the newer Dalf PIC18F6722 part. To remedy this, the CD that comes with the Dalf Board has a modified p1618qp.ini file which will enable the application to communicate with the Dalf microcontroller part. After installing the P1618QP Application on your PC, the p1618qp.ini file from the CD should be copied into the same directory with the p1618qp.exe application. You can simply overwrite the old p1618qp.ini file that came with the application, or if you prefer, save the old version to a different directory first.

### 6.2.4 About The Image

The image file should have a .hex extension with the code starting at address 0x0800. This is one sure way to avoid changes to the content of the Boot Block.

If you are doing C development using the mcc18.exe compiler and will be supplying your own image, you should use a modified version of the normal startup code <c018i.c> to compile with your application so that the application begins at 0x0800. However, unless you need to make additional changes to <c018i.c>, you should simply link in the pre-assembled <c018i.o> located on the CD. The only difference between the modified <c018i.c> source file and the one that is distributed in the Microchip MCC18 Compiler Source Library is the relocation of the reset and interrupt vectors. The modified <c018i.c> source file is included on the CD for reference only. **One other thing is important, but not obvious: The hex image file should be copied to your hard drive before you begin the update process with p1618qp.** The terminology used by the p1618qp.exe application for selecting the hex file for the flash process is called “importing the file”. For some reason the application doesn’t like to import the file from the CD, but if you try, you won’t get the error until the actual flash operation is started.

### 6.2.5 About The Parameter Block

A firmware upgrade will often require corresponding changes to the Parameter Block (a section of memory in the external EEPROM), but the process of upgrading the flash by itself does not modify the Parameter Block!

Here is how this is handled: First, the flash image always includes a copy of the factory default values for the Parameter Block. Second, on initial power up after a firmware upgrade, you should hold down on the BTN Switch during the reset in order to request that the factory defaults be written into the Parameter Block. This forces the system initialization code to first copy the factory defaults into the Parameter Block before they are used to initialize the system.

It is not a bad idea to make a hardcopy of the current Parameter Block contents before proceeding to flash the new image (eg; use Cmd\_L with the TE interface with logging enabled). This will enable you to later easily restore previous settings (eg; PID constants, operating mode, etc.) that you had made to the Parameter Block. Be aware that if you are flashing a Dalf image that is a firmware upgrade from Embedded Electronics some settings may have moved to new locations in the Parameter Block or have been eliminated entirely.

### 6.2.6 The Process

In general, the boot load process itself can be described as a series of steps:

- 1) Board and PC Setup
- 2) Connection
- 3) Import Hex File
- 4) Erase
- 5) Flash
- 6) Disconnect
- 7) Reset Dalf Board while holding the BTN switch.
- 8) Restore any customizations previously made to the Parameter Block

Before proceeding you should have the image (.hex file) that you want to flash **on your hard drive**.

#### Board and PC Setup:

- Connect a standard serial modem cable between the board and your PC.
- Apply power to the Dalf board.
- Move the slider on the switch labeled PGM to the position that turns on the yellow LED (**BOOT LOADER POSITION**). The yellow PGM LED in the On state indicates that the board will enter boot mode when next reset.
- Reset the Dalf Board by pressing the RESET button. This will start the boot loader firmware which will wait for communication with the Windows application. You will not see any visible indication that anything has happened, but the reset is important.
- Start the p1618qp.exe application on your PC. You will see a device selection menu.

#### Connection:

- Scroll down in the device selection drop down menu and highlight the PIC18F6722 part (1). Press the [SELECT] button. The PIC18F/PIC16F Quick Programming Toolbar appears.

- Right click on the [COM] and [BAUDRATE] buttons if necessary to select COM1 and 57,600 baud . Left click on the [CONNECT] button. You should see a response in the status line that says “Device Found”.

**NOTES:**

- (1) If you don't see the PIC18F6722 part in the list, there is a problem with access to the proper <p1618qp.ini > file.

**Import Hex File:**

- Left click on the [OPEN HEX FILE] button to locate and then open your flash image (.HEX) file. You should see the status response: “HEX File Imported..”.

**Erase:**

- Left click on the [ERASE] button to erase the flash memory. This is necessary before programming the part. You should see a progress meter and when finished, the status line response: “Finished Operation..”.

**Flash:**

- Left click on the [WRITE DEVICE] button to start the flash process. Again you should see a progress meter and when finished, the status response: “Finished Operation..”.

**Disconnect:**

- Close the p1618qp.exe application on your PC.
- Leave the modem cable attached.

**Reset Dalf Board:**

- Start a terminal emulator application. This will be used to test the newly flashed code.
- Move the PGM slider to the position in which the yellow LED is off (**NORMAL POSITION**).
- Press and hold the BTN switch (request new factory defaults be copied to Parameter Block).
- Press and then release RESET button to reset the Dalf Board. This will cause the newly flashed firmware to begin running on the microcontroller.
- When you see the terminal emulator greeting, you can release the BTN switch. This will have copied the new factory default values into the Parameter Block and performed system initialization using the new firmware and factory default Parameter Block values..

**Restore any customizations to Parameter Block**

- Use Cmd\_W to restore any special settings to the Parameter Block. Use Cmd\_W, and Cmd\_L to verify your changes. If you have previously saved a hardcopy (using Cmd\_L with terminal emulator logging enabled) of your settings, it will be easy to compare the current values with the previous ones (remember - some parameters may have changed locations).

Alternatively, the Dalf GUI can be used to restore settings, but be sure that the GUI version is compatible with the new Dalf firmware.

## 7 Interrupt Driven Firmware Design

The firmware design is interrupt driven. Currently, the single exception is access to the on-board I2C parts using the Primary I2C Bus. Some of the interrupt handlers (ISR's) schedule services within the main loop to perform actions that if executed within the interrupt handler itself could create interrupt latency issues. The main loop consists primarily of processing service requests from the interrupts. Much of the firmware design is apparent from this document, a review of the interrupt system, and an inspection of the main loop.

The PIC18F6722 part supports a 2-level priority interrupt system. At the time of this writing all interrupts are configured to use the high priority handler only. The actual interrupt handler is referred to as the **ISR** (Interrupt Service Request) routine. The ISR routine dispatches to individual service routines by testing the Interrupt Flags. This code and mechanism can be reviewed in the file <main.c>. New or different interrupt handlers may be installed easily by modifications to the <main.c> file. The existing Interrupt Handlers generally, but not always, request a **Main Loop Service (Svcx)** to handle processing better done outside of the actual interrupt handler. What follows is a description of the various interrupts, their handlers, their main loop Svcx Services, and their general function.

### 7.1 TMR0 - Heartbeat

#### Main Loop Service: Svc0

Timer0 is configured as an 8-bit count-up timer that generates an interrupt on overflow every 1 msec. The TMR0 Handler is responsible for maintaining various software counters that are used to schedule other actions (eg; sampling). TMR0\_ISR is also responsible for maintaining the 24-bit, 2's complement position counter when an analog feedback device is used as a motor position sensor. As with most of the interrupt routines, the ISR schedules a main loop service request (Svc0) that executes roughly every 1msec (after return to the main loop).

Svc0 has several functions. One important example is velocity slew rate (ramping) control during motor state changes. This service is also responsible for initiating open loop motor commands generated by R/C and or POT control modes.

### 7.2 TMR1 - RTC

#### Main Loop Service: Svc3

Timer1 is a 16-bit timer/counter configured as an Asynchronous Counter driven by an external 32.768 kHz crystal time base. The interrupt is configured to occur once every second and the ISR maintains a 24-hour real time clock (HOURS, MINS, SECS) which can be used as a time-stamp or to initiate scheduled events. In addition the 32-bit variable "Seconds" is incremented every second. This variable represents a running second count from the time of power-up and is used by some library routines for extended delays. Main loop service (svc3) is currently used only to periodically compute the measured (and calibrated) battery voltage derived from the stored ADC reading and calibration constant in the Parameter Block.

### 7.3 TMR2 - ADC State Machine

#### Main Loop Service: Svc4 (UNUSED)

Timer2 is an 8-bit count up timer with a clock source of fOSC/4 = 10MHz. The firmware dedicates Timer2 to driving the ADC State Machine that is responsible for the acquisition, conversion, and result storage, for each of the 7 ADC inputs [AN0...AN6]. Only the most significant 8 bits for each 10 bit conversion is stored. When the TMR2 Interrupt is serviced, the current value of the ADC State Machine variable "adc\_state" determines one of 3 timing parameters that controls the next TMR2 interrupt timing. It is the value of the timing parameters in ERAM that control the TMR2 interrupt timing for the next state.

After power-up, the ADC module runs autonomously using TMR2 interrupts to provide appropriate delays and sequencing of the ADC State Variable. At each TMR2 interrupt, the interrupt handler code dispatches

based on the state variable to one of 3 routines. The role of TMR2 in this process is to generate the appropriately timed interrupts to sequence the operations in a way that doesn't involve software delay loops to provide the necessary delays. Because the delays are accomplished with interrupts, the number of CPU cycles devoted to processing the analog inputs is small. Here is a rough description of how this state machine works:

adc_state	action
0	Start acquisition on current channel. Set TMR2 for [acquisition] delay. adc_state $\leftarrow$ 1. Exit
1	Start conversion on current channel. Set TMR2 for [conversion] delay. adc_state $\leftarrow$ 2. Exit
2	Store result (*). Advance channel. If sequence complete, request main loop svc. Set TMR2 for between-channel [gap] delay. adc_state $\leftarrow$ 0. Exit.

(\*) If the digital filter is enabled for this channel, the reading will be filtered before storage. See the section of this manual that discusses the digital filter for details.

**Required minimum timing:** You can control the timing for each of the states thru the values stored in the Parameter Block. However, proper **signal acquisition** requires a minimum charging time for the internal sampling capacitor. The required minimum signal acquisition time primarily depends on the signal input impedance (higher impedance requires more time). A typical minimum acquisition time might be 20 uSec. The time required for a successful 10-bit conversion is at least 12\*TAD where TAD is the period of the ADC Module Sample Clock. The ADC sample clock is configured for  $1^*TAD = 64*tOSC = 64*(1/40E6) = 1.6$  uSecs, and the minimum **conversion time** is  $12*TAD = 19.2$  uSecs.

**Defaults:**      Acquisition: 20uS + Interrupt latency = 30 uS  
                   Conversion: 35uS + Interrupt latency = 45 uS  
                   Between channel (gap) delay: 500uS + Interrupt latency = 540 uS

**With the above defaults, the sequence of 7 measurements completes every 4.4 msec (results for a specific channel are updated about 227 times each second).**

The 3 delays (Acquisition, Conversion, Gap) can be customized by altering the ADC Timing indices in the Parameter Block (See the Timing Table in the Parameter Block Section of this document for details).

## 7.4 INT0 - Motor2 Encoder

### Main Loop Service: Svc1 (UNUSED)

The Int0 interrupt handler services the motor2 encoder input signal AB2INT. The quadrature [A,B] signals from the motor2 encoder are routed thru an XOR gate to produce an Int0 interrupt for transitions on either the A or B signals. The A and B signals are also routed to input pins that allow their actual state to be read by the handler. The role of the Int0 ISR is to read the AB state and compare it with the previous AB state to determine if the motor has moved forward or reverse. A 24-bit, 2's complement position counter, which is cleared on power-up, is maintained by this interrupt handler. The counter is incremented for forward motion and decremented for reverse motion.

Here is a rough description of how the counter is used (outside of the interrupt handler):

Every Velocity Sampling Period (**VSP**) msec, the old value of the counter is compared with the new value to determine a velocity in units of ticks/VSP msec. The value of the encoder count and the velocity are maintained in RAM and can be viewed at any time thru the command interface. The timing of the velocity sampling is determined by the VSP parameter. The VSP value is an important environment variable that you can/should change to suit your application. It also plays an important role in the timing of the

Trajectory Generation and PID control algorithms. Eg; With VSP2 = 0x14 = 20 msec, a fresh value for the motor2 velocity would be determined every 20 msec. When a closed loop motor2 control command is active it would also result in a call to the Trajectory Generator and the PID control filter every 20 msec to generate a new “waypoint” and a motor speed correction as the PID response.

The **Velocity Sampling Period (VSP msec)** controls how frequently the motor velocity is updated. The velocity is determined via the counter difference on each successive sampling period. In the case of closed loop motor control commands, VSP is also used as the timing interval for the Trajectory Generator and the PID Controller firmware. The Trapezoidal Trajectory Generator produces a “waypoint” every VSP msec for use as the target location in the PID computations. Since the speed of the motor and the CPR of the encoder will vary with different applications, the sampling period VSP (msec) for velocity computations is stored in the parameter block to allow easy customization. See the Trajectory Generator section of this document for a discussion of how to determine an appropriate value of VSP for your application.

## **7.5 INT1 - Motor1 Encoder**

### **Main Loop Service: Svc2 (UNUSED)**

The Int1 interrupt handler services the motor1 encoder input signal AB1INT. See INT0 for details.

## **7.6 INT2 - Motor2 Current Limit**

### **Main Loop Service: Svc5**

This interrupt is disabled by default - see the “ocie” bit in the Parameter Block.

The Int2 interrupt handler services the motor2 over current sense input from the I2INT signal. The interrupt indicates that the motor2 current sensor output voltage is outside the limits determined by the digital pots located near the comparator device. If the interrupt is enabled and the “FastOff” response option has been selected (see the “oc\_fastoff” bit in the Parameter Block), the ISR will immediately remove motor power and disable the motor control interface. Otherwise, the only action taken by the ISR is to toggle the edge detection circuitry, record the state of over-current (or not), and request the main loop service. Svc5 is responsible for taking action to deal with the over current notification. See the section in this document “Current Sensor Feed back” for additional details.

## **7.7 INT3 - Motor1 Current Limit**

### **Main Loop Service: Svc6**

The Int3 interrupt handler services the motor1 over current sense input from the I1INT signal. See INT2.

## **7.8 RB4 - EX2**

### **Main Loop Service: None**

The RB4 interrupt pin is dedicated to the off-board signal EX2 (UNUSED). The RB4 interrupt is a bit different than other PIC interrupts in that it is one of a group of 4 pins (RB4... RB7) that when enabled, cause interrupts to occur on signal transitions (“interrupt-on-change”). Pins RB5, RB6, and RB7 are dedicated for other uses and may not be used for interrupts. Currently the RB4 interrupt is not enabled. There is no ISR handler for RB4, and there is no main loop service.

## **7.9 TX1 - USART1: RS232 Serial Command Response**

### **Main Loop Service: Svc8 (UNUSED)**

The firmware configures USART1 as a full-duplex, asynchronous system, with a default baud rate of 19,200. The source for the baud rate generator is derived from the system clock (fOSC). Transmittal of a string of data is interrupt driven with the string first loaded into a circular buffer Tx1\_Buff in RAM.

Transmittal begins by enabling the transmitter circuitry and the TX1 interrupt. This will generally cause an immediate TX1 interrupt (because the Transmit Buffer Register TXREG1 is empty).

The TX1 Interrupt handler fetches the next byte from Tx1\_Buff and writes it into the TXREG1 hardware register. The hardware then transfers this into the Transmit Shift Register TSR1 (when the TSR1 register is empty). This action “empties” TXREG1 (making it available for the next byte) and causing the next interrupt. This process continues with an interrupt for every byte to be transferred until the ISR detects that the Tx1\_Buff buffer is empty. At this point further TX1 interrupts are disabled.

Main loop service is requested on every byte transfer, but the service is a NOP.

### **7.10 RX1 - USART1: RS232 Serial Command Receive**

#### **Main Loop Service: Svc9**

Receipt of serial data via USART1 is also interrupt driven. The data byte received by the hardware interface is placed into the Receive Shift Register (RSR1) and then transferred into the double buffered Receive register RCREG1. The transfer into the RCREG1 causes an RX1 interrupt. The RX1 Interrupt handler copies the character received into the circular Rx1\_Buff buffer in RAM.

When using the Terminal Emulator mode for the serial interface, the receipt of a CR character signals end of transmission and the interrupt service routine schedules a main loop service to process the data (1). The main loop service parses the expected command string into CMD and ARG variables, and dispatches to the appropriate command handler.

- (1) – Actually, the RX1 ISR is a bit more complicated than this. Since the service is designed to communicate with a Terminal Emulator Application running on a PC, the service must echo back characters to the PC, ignore certain control characters, and provide special treatment of others (eg; BS char). In addition, certain characters are allowed as data delimiters (for readability), hence discarded.

When using the API mode for the serial interface, it is the detection of end of message packet that signals an end of transmission causing a request for main loop service to process the data.

### **7.11 CCP1 - R/C Channel 1 (EX0)**

#### **Main Loop Service: SvcA**

The discussion here applies to CCP1 (EX0), but similar comments apply to CCP2 (EX1) and CCP3 (EX3).

Because the EX0, EX1, and EX3 signals are connected to CCP (Capture/Compare/PWM) modules on the PIC microcontroller, they can be configured for different uses. With a little programming, these signals can become PWM outputs, or special purpose capture inputs to time events, or (in compare mode) to generate carefully timed signal outputs. Finally, they can be general purpose interrupt inputs or simply digital GPIO's. The Dalf firmware does not support all of these features, but it is good to know that it is possible if your application requires it and you are feeling adventuresome with the code.

The firmware provides a 3 channel R/C interface using signals EX0, EX1, and EX3. An interrupt handler for each of these signals provide very precise pulse width measurement for receiver outputs which would normally drive servos but are instead connected to EX0, EX1, and EX3. Timer3 and the Capture/Compare/PWM modules are the key to the accuracy. Timer3 is configured as a free running, 16-bit counter with clock source = fOSC/4 and a prescaler of 1:8. With fOSC = 40E6 Hz, Timer3 increments every 0.8 uSecs and wraps every  $65536 * 0.8 \text{uSecs} = 52.428 \text{msecs}$ . When used in conjunction with the CCP1 module configured in **capture mode**, this provides a signal pulse measuring tool with a resolution of 0.8 uSecs and a range of about 52 msecs.

#### **EX0 ISR Operation:** (Similar comments apply to EX1 and EX3)

The CCP1 interrupt is alternately configured to interrupt on the rising edge and the falling edge of the input signal. On the rising edge, the value of Timer3 is latched by the hardware into the 16-bit special function register CCPR1 and transferred by the Interrupt handler to the 16-bit RAM value PulseStart1. On the signal falling edge, the newly latched value of Timer3 is used together with PulseStart1 to determine the elapsed time for the pulse in units of Timer3 ticks. This elapsed time is stored in the 16-bit RAM value PulseWidth1. Once the signal pulse width is captured, the ISR requests main loop service to convert the measured pulse width from units of 0.8 uSecs to units of 1.0 uSecs (multiplies by 0.8).

#### **7.12 CCP2 - R/C Channel 2 (EX1)**

##### **Main Loop Service: SvcB**

This is similar to CCP1.

#### **7.13 CCP3 - R/C Channel 3 (EX3)**

##### **Main Loop Service: SvcC**

This is similar to CCP1. Note that while the above described measurement operations take place for all 3 channels, the results for channel 3 are currently unused.

#### **7.14 SSP2 - I2C2 SLAVE Command Interface (SC2, SD2)**

##### **Main Loop Service: SvcD**

Interrupts are generated on both MASTER read and write operations with data received (command and arguments) stored in an input buffer and data to be transmitted retrieved from a transmit buffer. Clock stretching is employed to provide setup time to allow the Dalf Firmware to prepare data for transmission to the Host. Reception of the last byte (CKSUM) of the command packet signals end of transmission and the interrupt service routine schedules a main loop service to process the data. The main loop service parses the command string into CMD and ARG variables, and dispatches to the command handler. If there is data to be returned to the host, the command handler is responsible for starting the process (again interrupt driven) to deliver the data.

#### **7.15 //Unused Interrupts// - Resource Details**

There are a number of unused interrupts, but in many cases the resources (eg; timers) that would generate the interrupt are in use for features which may preclude the use of the interrupt. Here is a list:

##### **TMR3**

The Timer3 interrupt is unused, but the Timer3 resource is dedicated to the role of pulse width timing capture if the R/C interface is used.

##### **TMR4**

The Timer4 interrupt is unused, but the Timer4 resource is dedicated to the role of providing the PWM frequencies used by the two motors for speed control.

Timer4 is configured to provide an 8-bit counter with a clock source of fOSC/4 = 10E6 Hz. Timer4 operation depends on 3 factors; a period register (PR4), a prescaler, and a postscaler. Only the first two of these (the postscaler is not used) affect the PWM operation. For the prescaler there are 3 choices: 1:1, 1:4, or 1:16. The prescaler setting is defined by the low order first two bits in the T4CON control register.

Timer4 increments at a rate of prescaler\*(4\*tOSC) until a comparator match with the 8-bit PR4 register occurs. On the next count {(PR4)+1}, the Timer4 value is reset to 0, restarting the counting sequence. It is this timing that determines the PWM frequency. For example; with the default values for T4CON and PR4 determined by the parameter block value fPWM, the prescaler is 1:4 and the value of PR4 is 124. This sets the default PWM frequency to

$$f_{PWM} = (f_{OSC}/4)/\text{PreSc}/(PR4+1) = 20.000 \text{ KHz.}$$

**I2C (BCL)**

The PIC I2C hardware supports true Multi-Master I2C communication with the use of 2 interrupts (SSP, BCL), but this is currently not supported by the firmware. The firmware **-does-** support non interrupt driven, I2C Master/Slave access to on-board devices using the Primary I2C Bus {EEPROM, I/O Expanders, Digital Pots} at 400 kHz. The Secondary I2C Bus (I2C2), configured as a Slave, **-is-** interrupt driven, and is used as a Command Interface. Neither interface uses the BCL interrupt.

**PSP**

The parallel slave port interrupt is unused, and the PortD pin resources that it would require are used for other purposes.

**CM**

The comparator interrupt is unused, and the pins RF1-RF6 that it would use as resources are instead used for other features on the Dalf board.

**AD**

The Analog-To-Digital interrupt is unused. Instead, TMR2 is used to sequence a state machine for cyclic sampling and storage of analog inputs.

**EE**

The internal EEPROM “write complete” interrupt is unused.

**TX2**

The second USART TX interrupt is currently unused.

**RX2**

The second USART RX interrupt is currently unused.

**LVD**

The low voltage detect interrupt is currently unused.

## 8 Motor Encoder Feedback

Beginning with the Dalf firmware release 1.50, support is provided for both incremental optical encoders and absolute analog encoders. The incremental encoders will generally be more accurate and I recommend their use for everything but servo operations. **If you are going to be using one of the Servo Modes (new feature in release 1.50) you will probably want to use absolute analog encoders.**

### 8.1 Optical Encoders

For each motor, the board provides a 5-pin connector (VDD, GND, A, B, Z) for standard quadrature encoder inputs. The VDD and GND signals power the encoder logic and the A,B, and Z signals are inputs to the Dalf Board. The firmware determines the direction of rotation from the relative phase of the A and B signals. The Z signal (Index) is routed to a digital input, but is currently unused.

#### 8.1.1 The CPR Specification

The **CPR** (“Cycles Per Revolution”) is an important encoder specification. CPR is the number of cycles of either the A or B signal when the shaft on which the encoder is mounted rotates one revolution. This is not the same as “Counts Per Revolution” which I will refer to as “**Ticks Per Revolution**” (TPR) to avoid confusion. Because the A and B signals are 90 degrees out of phase, the Dalf firmware obtains an actual encoder resolution that is finer by a factor of 4;  $\text{TPR}=4*\text{CPR}$ . TPR is one of the parameters stored in the Parameter Block and enables correct display of motor velocity in units of RPM. If you are using encoders you should record the value of TPR1 and TPR2 for your encoders into the Parameter Block.

#### 8.1.2 Optical Encoder Limitations

The interrupt handler maintains 24-bit, forward/reverse, 2’s complement counters which represent motor position. The interrupt handler increments or decrements these counters on every encoder tick. This design imposes a limitation on high speed, high resolution, motor/encoder combinations in terms of the maximum encoder interrupt frequency. If the interrupt frequency is too large, it will introduce errors in other processes. For example, it can interfere with serial communication; produce encoder count errors; etc.

**If you have not chosen a motor/encoder combination and are planning on using the Dalf Board, this discussion may help you make your choice. If you already have a motor/encoder setup, this discussion will help you to determine if it is a reasonable system for use with the Dalf Board.**

Assume **Vm** is the maximum no load velocity (RPM) that you intend to drive the shaft to which the encoder is attached. Note that Vm will depend on your operating voltage. If your encoder is attached directly to the motor shaft, Vm will be the maximum motor RPM. Let **fINT** be the frequency of encoder interrupts that the Dalf Board uses to maintain motor position.

At maximum RPM, the parameters fINT, CPR, and Vm are related by the equation:

$$\text{fINT} = \text{TPR} * (\text{Vm}/60) = \text{CPR} * \text{Vm}/15 \text{ (ticks/s)}$$

**I don't recommend the Dalf Board for closed loop applications in which  
fINT > 40,000.**

An application that has fINT = 40,000 will produce encoder interrupts every 25 microseconds when running at full velocity.

**EXAMPLE 1:**

The encoder is the E6M part from US Digital. This part comes as an easy to assemble kit that is mounted directly to the motor shaft. For this example we assume that the E6M was specified to have resolution **CPR=64** (TPR=256). The motor is a geared, PM DC, wheelchair motor rated at +12VDC that will be overdriven with a +18VDC power source. The no-load velocity when driven at +18V is **V<sub>m</sub>=320 RPM**.

$$f_{INT} = 64 * 320 / 15 \rightarrow f_{INT} = 1,365$$

In this application, at maximum velocity, an encoder interrupt occurs every 732 microseconds. This is well within the specification - no problem.

**EXAMPLE 2:**

Pittman +12VDC Motor. Encoder (built in) with **CPR=500**. At full +12V power, the no load velocity is **V<sub>m</sub>=4,000 RPM**.

$$f_{INT} = 500 * 4000 / 15 \rightarrow f_{INT} = 133,333$$

In this application, at maximum velocity, an encoder interrupt occurs every 7.5 microseconds. This exceeds the f<sub>INT</sub> specification for the Dalf Board by a large margin. You would still be able to drive this motor and control velocity in open loop modes using the Dalf Board without any problem. In order to successful use the encoder inputs for closed loop operations you would have to limit the maximum velocity (see VMAX parameter in the parameter block) to roughly 1,200 RPM.

**TIP:** CPR is an encoder characteristic which should be considered carefully in conjunction with your maximum motor velocity and desired position resolution. With large values of CPR, you will gain finer position and velocity resolution, but this advantage has a cost. The larger the value of CPR that you use, the more time will be spent handling the interrupts to maintain position and velocity. In extreme cases, you may be forced to limit motor operation to some fraction of full power (see VMAX), or mount the encoder “downstream” after a gear reduction.

### 8.1.3 Optical Encoder Hookup

The encoders will generally have 5 wires to hook up (4 if missing the index wire). The wires are frequently labeled as shown below. The “Type” field refers to things from the point of view of the encoder (ie; Signal “A” is an output from the encoder to the Dalf Board). The “Z” signal is routed to an input pin on the microcontroller, but current Dalf firmware does not monitor it.

Signal	Type	Comments
+5V	Power	Encoder power.
A	Output	With motor movement a square wave.
B	Output	With motor movement a square wave, 90 deg out of phase with Signal A.
Z (or I)	Output	Index or Home. With motor movement, a single pulse every revolution.
GND	Power	Encoder power.

The “analogfb” bit in the MODE1 byte(s) in the parameter block must be set to ‘0’, but this is the default so you shouldn’t have to make a change. You may need to change the state of the “aleadsb” bit in the Parameter Block. Signal hookup is quite simple. Dalf has two 5-pin screw terminals (VDD, A1,B1,Z1,GND) and (VDD,A2,B2,Z2,GND) - one for each encoder. Be sure to change the values for TPR in the Parameter Block to match your encoder.

### 8.1.4 Optical Encoder Accuracy

Accuracy is one advantage of optical sensors over analog sensors. Assuming that your application meets the fLINT Specification, the counter will accurately track your encoder. Actual position of the output shaft will depend on the resolution of your encoder and mechanical factors like “shaft play”. In closed loop operations accuracy will also depend on successful PID tuning. If your application is “tuned”, you can typically achieve position accuracy for closed loop operations to be within +/- 1 tick of your position counter. Depending on the CPR of your encoder, this positioning accuracy is better than you can expect to achieve with analog encoders.

## 8.2 Analog Encoders

Beginning with firmware version 1.50 there is built-in support for an analog motor position encoder that could optionally replace an optical incremental encoder for use in closed loop operations. This feature has been tested with the **MA2 analog sensor** from US Digital on a geared wheelchair motor with maximum RPM of 320 and has been found to function quite satisfactorily in full range (not just [0,360] degrees) operation. The feature has not been tested with other sensors or faster motors.

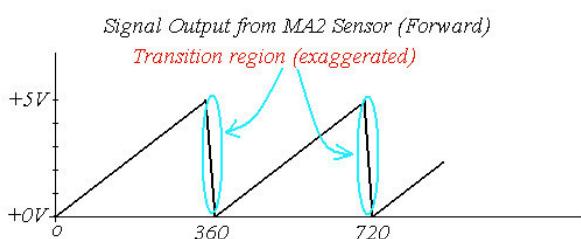
### 8.2.1 Analog Encoder Limitations

The nature of the sensor output (0 to +5V sawtooth ramp with abrupt transition on each revolution) makes ADC sampling and interpretation a challenge. Periodically, the ADC Module samples the sensor output to maintain the position counter. This sampling is asynchronous to the shaft movement and inevitably sampling will sometimes occur during the sensor transition period resulting in a bogus position measurement. The algorithm that processes the ADC samples must deal with this situation robustly to maintain an accurate position counter. Here are some recommendations:

- 1) **Don't** use analog sensors if your maximum motor RPM exceeds 500.
- 2) **Do** use the MA2 sensor (preferably the ball bearing version). It has been tested.
- 3) **Don't** use a sensor with a rotational limitation (eg; a single or multi-turn pot).
- 4) **Don't** use the digital filter for analog encoder inputs for full range (not just [0..360]) operation.
- 5) **Do** consider using digital filter for analog encoder inputs in POTSVO and RCSVO [0...360] operation.

#### A bit of discussion on the Do's and Don'ts

Usage of the analog sensor in closed loop operations assumes reasonable PID parameters. If the sensor cannot rotate freely for several motor revolutions, you will not be able to use Cmd\_Q to assist in PID tuning. This is one, but not the only, reason for (3). Once the motor(s) are PID tuned, it may be reasonable to replace the MA2 sensor with a standard pot (use your own judgement on this one). The algorithm that processes the analog readings rejects readings that differ significantly from the last reading. This is necessary to identify bogus readings obtained during the signal transition that indicates a complete revolution. This places a limitation on the maximum motor RPM (one of the reasons for 1).



The ADC Module captures the sensor reading asynchronous to the motor movement.

This results in occasional ADC sampling during the brief transition period, resulting in a bogus indication of true motor position.

In addition to the R/C filter on the [AN0..AN5] analog inputs, you have the option of applying a digital filter to any of the [AN0..AN6] inputs. This digital filter is discussed in detail in a later section. It can be quite useful to eliminate positional “chatter” due to analog noise not related to actual shaft movement, but it can get in the way of the algorithm used to detect full revolutions (see the graph). I don’t recommend using the digital filter for analog position feedback for applications that operate outside of the [0..360] range, but it is quite appropriate for usage in typical [0..360] closed loop servo operations. If you decide to use it, be sure to disable it during PID tuning operations.

### 8.2.2 The Position Algorithm (Under The Hood)

Knowledge is power, so how does it work? If you are not interested, you can probably skip this section, but be sure to follow the above recommendations with regard to specifications. As explained in an earlier section of the document, the ADC Module functions as a state machine driven by TMR2 to periodically sequence thru all 7 channels AN0 .. AN6 and record 8-bit measurements in RAM (ADC0[0..6]). The overall process time for a complete cycle is primarily controlled by the AD\_GAP parameter in the Parameter Block. With the default setting, each channel is updated roughly every 4.4 msec. We are only concerned here with channels AN2 and AN3 which are dedicated for use in measuring the analog feedback signals for motors 1 and 2. So the measurements are being recorded in RAM at ADC0[2] and ADC0[3], but what is the connection between a measured analog reading, say “A” (range [0..255]) to the 24-bit, 2’s complement, position counter?

If an analog sensor is being used for motor position feedback, the TMR0 Interrupt Handler (int every 1 msec) maintains the counter. If there is no change between the current measured value “A” in RAM and the previous value “LastA”, the counter is not changed. Otherwise, the counter is either incremented or decremented by **Diff = LastA - A**. Note that this is an 8-bit, 2’s complement subtract. For example; LastA=0x03, A=0xFE → Diff=0x05 (the motor has just passed thru the transition region in the reverse direction). If the transition happened instead in the forward direction with LastA=0x03, A=0x00 → Diff=0xFB (=5). In a perfect world this would be the end of the story. Unfortunately, there is a problem.

#### The problem:

The signal transitions abruptly between +0V and +5V with every revolution. The signal transition is “falling” for forward movement and “rising” for reverse. Ideally the transition would occur instantaneously, but in the real world there is a brief interval where the signal can be sampled by the ADC Module and obtain **-ANY-** value in the range [0..255].

#### The solution:

We couple a fast ADC sampling rate with limited motor RPM to ensure that the value for |Diff| will not be large unless the reading is bogus. This allows us to identify as bogus, all readings in which |Diff| exceeds a threshold, say **DMAX** (see Parameter Block). This strategy raises two important issues! What do we do about clearly identified bogus readings ( $|Diff| \geq DMAX$ ) and what do we do about bogus readings that we fail to identify because  $|Diff| < DMAX$ ?

- $|Diff| \geq DMAX$   
In this case, the algorithm assumes sampling occurred during the transition and uses the value of LastA to determine if the signal is rising or falling. If falling (fwd motion) the low byte of the counter is set to 0xFF (corresponds to +5V, just before the transition). If the signal is rising (rev motion), the low byte of the counter is set to 0x00 (corresponds to +0V, just before the transition).  
**In short, we treat this situation as if we had really sampled a tiny bit earlier in time.**
- $|Diff| < DMAX$   
In this case, we can’t distinguish the bogus reading from a real one, and the counter is updated with a temporarily incorrect count. The error is always less than DMAX, but typically will be smaller. In any case, because the analog sensor generates an absolute position (within the [0..360] interval) and not an incremental one, **the very next counter update fully corrects the count.**

The value for DMAX is located in the Parameter Block. It shouldn't be necessary to change the default values for DMAX or AD\_GAP unless you want to experiment with faster motors than I have recommended.

### 8.2.3 The Digital Recursive Filter

In addition to the hardware RC filter on analog inputs [AN0...AN5], you can optionally enable a **Digital Low Pass Recursive Software Filter** to be applied to any of the [AN0...AN6] inputs.

The filter is enabled thru use of the bits in the **FENBL** byte in the Parameter Block. The characteristic of the filter is controlled by the **DECAY** byte in the Parameter Block. The cutoff frequency is inversely proportional to the DECAY parameter (increasing the value of DECAY decreases the cutoff frequency). Here are the equations that define the filter:

$Y_n = (a_0 * X_n) + (b_1 * Y_{n-1})$	The Y's are outputs and the X's are the inputs.  So, in words, the current filter output $Y_n$ is the sum of two terms. The first term (the $X_n$ term) is derived from the current input (ADC reading). The second term (the $Y_{n-1}$ term) is derived from the previous output (the last value stored in the ADC0[] memory array).
---------------------------------------	---

The DECAY parameter determines the coefficients a0 and b1 as follows:

$$\begin{aligned} d &= \text{DECAY}/256 & (0 \leq d < 1) \\ a_0 &= 1 - d \\ b_1 &= d \end{aligned}$$

Restated in terms of the DECAY parameter, the above equation becomes more computationally convenient in terms of fixed point arithmetic.

$Y_n = [(256-\text{DECAY})X_n + \text{DECAY}Y_{n-1}] / 256$	Each time the ADC State Machine stores it's reading, the FENBL byte is examined to determine whether or not to employ the filter. If the filter is not enabled for the current channel, the value of the raw reading ( $X_n$ ) is stored in the ADC0[] array.  If the filter is enabled, the filter computation occurs and the filtered value ( $Y_n$ ) is stored in the ADC0[] array.
---	--

The value of the decay constant d is related to the filters time constant T by the equation:  $d = e^{(-1/T)}$ . For example if  $d = 0.80$ , then after  $T=4.48$  samples of  $X=0$ , the filter output will have decayed to 36.8%.

Note that there is only one DECAY parameter (not one for every channel), so the same filter characteristic applies to all ADC channels for which it is enabled.

### 8.2.4 Analog Encoder Hookup

When using the analog encoder option, the analog voltage signals from motors 1 and 2 are wired to the AN2 and AN3 screw terminals respectively. Any available Vdd and GND can be used to supply the sensor with the +5V and Gnd signal references. The “analogfb” bit must be set in the appropriate MODE1 byte(s) in the Parameter Block. The TPR parameter for the motor(s) that will use analog feedback should be set to 256 (0x0100).

#### Mounting the sensor

Your application may require mounting the analog sensor in such a way that forward motor movement actually results in the sensor output indicating reverse direction. This situation is analogous to the optical encoder mounting issue (see the aleadsb bit in the optical encoder section), but requires a slightly different solution. To see if you have this issue, connect the sensor and use the open loop command (Cmd X) to move the motor slowly in the forward direction while monitoring the position using Cmd E. If the counter increases, you are fine and will not need to make a change. If instead the counter decreases with forward motor movement you have 3 possible options to fix things up:

- 1) Mechanical Change: If your motor is a dual shaft motor, remount the sensor on the opposing shaft. This change to the sensor orientation will fix things up.
- 2) Electrical Change: Reverse your motor leads. This change will fix things up. One side effect of this change is that forward motor commands will now cause the motor to move in the opposite direction to that previously. Depending on your application, this may not be acceptable.
- 3) Software Change: Beginning in firmware version 1.60 there is a “software” change to fix things up. The software solution requires you to change the state of the **analogdir** bit in the **MODE3** parameter in the Parameter Block. After making the change and resetting the board the sensor output and the motor movement should be in sync. Here is how this works: When the ADC Module samples the channel associated with the sensor output it will complement the value that it reads if the analogdir bit is set to “1”. The effect is that falling voltage ramps from the sensor are converted to rising ramps and vice versa.

### 8.2.5 Analog Encoder Accuracy

Accuracy is one advantage of optical sensors over analog sensors. Errors in maintaining the position counter from the analog input contribute to position inaccuracy. The errors in maintaining the counter come from the analog sensor itself, the conversion performed by the ADC Module, and the algorithm used in updating the counter. Testing with the MA2 analog sensor in parallel with a US Digital optical encoder (TPR=256) gave satisfactory results. The analog count typically synched with the optical count to within +/- 2 ticks over full range (not just [0..360]) operation. Position of the output shaft will depend on the resolution of your encoder and mechanical factors like “shaft play”. In closed loop operations, positioning will also depend on successful PID tuning. If your application is “tuned”, you can typically achieve position accuracy for closed loop operations to be within +/- 1 tick of your position counter. This error is in addition to the error in maintaining the counter. So, if your application can tolerate a total position error that is no more than 4 or 5 degrees, the analog sensor may work well for you.

## 9 Current Sensor Feedback

This section describes the built-in support for current sensing. Over current detection is disabled by default - see the “**ocie**” bit in the Parameter Block. If your application needs protection against over current, you will have to supply compatible off-board current sensors and perform these setup steps:

- Adjust the digital pots to set your current limits.
- Select one of the two built-in responses to the over current condition.
- Enable over current detection.

### 9.1 Sensor Selection

The hardware and built-in firmware provide support for over current detection and response when used with a compatible off board sensor. I recommend Hall-Effect type sensors that generate analog “center-offset” voltages proportional to the measured motor current and direction (\*). The 3-pin (+5V, VOUT,GND) **AMPLOC current sensor** with a sensitivity of 37mV/A is such a sensor and has been tested. The AMPLOC devices generate an output voltage VOUT which is an offset from 2.5V. The sign of (Vout - 2.5V) indicates the current direction and the magnitude of (Vout - 2.5V) indicates current magnitude.

(\*) -Use of non “center-offset” sensors whose output reads zero volts for zero current is possible, but there are limitations and the pot threshold settings are somewhat non-intuitive. A white paper will be posted on the EE website to explain the limitations and usage of this type of sensor with the Dalf Board.

### 9.2 Sensor Hookup

There is a 3-pin connector on the Dalf board for each sensor with labels (VDD, IS, GND). The IS (I Sense) pin of the connector is connected to the sensor VOUT signal. Attach the sensor leads to the connector for the appropriate motor [VDD, ISx, GND].

### 9.3 Setup

#### Digital Pot Settings

First you will want to adjust the digital pots that define the over current trip levels (the procedure is described a bit later in this section). There are two pot settings (a low voltage threshold (VL), and a high voltage threshold (VH) ) to be set for each motor.

#### Over Current Response

After you are satisfied with the current limit settings, the next step is to select the over current response. The selection for the over current response is determined by the setting of the “**oc\_fastoff**” bit in one of the MODE bytes (See the Parameter Block in Appendix D). The two choices for the over current response are “*FastOff*” and “*SlowRun*”. Rapid blinking of the red LED indicates the over current condition. The Over Current Mode setting can be verified with the “Get Status” Command (Cmd\_U).

#### FastOff: (“**oc\_fastoff**” = ‘1’)

This option is the default and results in the immediate removal of motor power. It also leaves the motor interface (for this motor) disabled and you will have to reset the board before you will again be able to control this motor. This is the appropriate response in situations where the over current condition is unexpected and you want to cease all motor activity until the cause can be determined. This setting is also appropriate for any over current testing that you might want to perform.

#### SlowRun: (“**oc\_fastoff**=’0’)

This setting can roughly be described as a reduction in the motor power level to maintain the current limits set by the pots. This is appropriate if you want your motor to “keep trying”, but at possibly reduced power levels. An example might be a manually controlled robot competition.

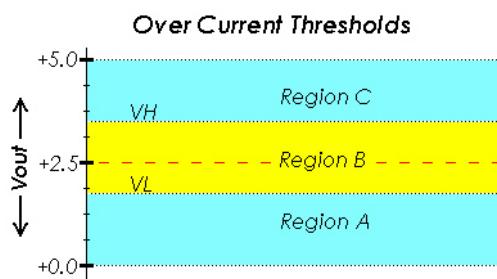
## Enabling Over Current Detection

Now that you have the current thresholds set, and the response selected, the only thing left to do is enable the feature. This is controlled by the “**ocie**” bit in the Parameter Block. You will need to reset the board (the “**ocie**” bit setting is detected during power up) before the over current detection will be enabled.

## 9.4 Under The Hood

The sensor voltages are **-not-** routed to analog inputs on the microcontroller, but instead are inputs to voltage comparator circuitry thru an RC filter (see schematic). The voltage thresholds for the comparators are determined by digital pots (see schematic). The pots are adjusted to the proper “trip levels” by writing the desired level (wiper tap position) using the primary I2C bus. Typically the pot levels would be set just once during initial setup. This over current circuitry drives a pair of interrupt capable pins on the microcontroller and the interrupt handlers respond to the over-current situation.

To use this circuitry in your application, you will need to adjust the pots to provide the desired upper and lower voltages for the voltage window  $[VL, VH]$  for each of the IS inputs. For each sensor, a PIC interrupt is generated when the output voltage from the sensor transitions into or out of the voltage window defined by the pots. The values for the voltage window endpoints should be determined by the application (motor stall current, sensor sensitivity, etc.) and can be set directly from the command interface. For example, a reasonable voltage window might be  $[1.5V, 3.5V]$  which allows a 1000 mV excursion in either current direction from the null offset ( $2.5V$ ) before triggering an interrupt. The sensor that has been tested has a sensitivity of  $37mV/A$ , so this example would translate to a current limit of about  $27A$ .



In the picture to the left, Regions A and C represent overcurrent. Region B is the normal operating region.

## 9.5 Limitations

Generally, the over current condition will be due to too much current in your drive electronics (driver or motors). The response from the Dalf Board is to remove (or reduce) the control signals to your driver (stopping the motor). **You will -not- be protected against situations like a VBATT/GND short** - there is nothing the board can control to help with that.

**Initial over current detection requires a transition out of the operating region** (Region B in the diagram) to detect over current. Thereafter all transitions across the VH and VL thresholds are detected with appropriate responses. When you initially power up the board the motor control signals will be in the off state. If for some reason you initially power up in a high current state (Regions A or C), the over current situation won’t be detected (with the motor controls already off, the board can’t remedy it anyway). This could happen for example as the result of a motor control wiring error.

After the sensor delivers a  $V_{out}$  voltage that represents an over current situation there is a finite response time before the board firmware can respond. With the FastOff response, power will typically be removed from your drive electronics within about **6 microseconds**. This is about as good as it gets, but it is not instantaneous. Whether it will protect you from a hard motor short is something you may not want to explore.

## **9.6 Over Current Response**

There are two choices for over current response. The selection is governed by the setting of the “**oc\_fastoff**” bit as mentioned earlier. You must decide which one is most appropriate for your application.

### **CASE1: SlowRun (**oc\_fastoff** = ‘0’)**

This response effectively protects your motor by degrading the requested power. When the over current condition is detected, the firmware responds by issuing a command to ramp the motor to a stop. While in the over current state the R/C and POT controls are disabled. Typically, the process of ramping the motor speed to zero reduces the current to an acceptable level (Vout is again in Region B), and the R/C and POT mode interfaces are re-enabled. Depending on the interface you are using to control the motors, you will have different observations of the behavior when using this choice for the response.

#### **Pot or RC Modes**

If the over current situation occurs while operating the control switch at full throttle (eg; motor stalled), the observed effect might simply be reduced drive to the motor with perhaps some flickering on LED3. An attempt will be made to ramp the motor to a full stop, but commonly this goal is not achieved because the reduction in power resets the over current situation which allows the next R/C (or POT) command to again drive the motor too hard, which results in over current which ...I think you get the picture. This “struggle” persists until the condition causing the over current response is removed, but meanwhile, your motor and drive electronics are protected.

#### **CMD modes**

If the over current situation occurs while operating the motor from the TE interface, the observed effect will likely be that the motor will simply be stopped. The difference between this and the other operating modes is that the command stream is slow (typing thru a terminal emulator is slow) and the motor has time to stop before the next command is issued. The situation could be different in the “programmable” API and I2C2 interfaces. In these interfaces, it is possible for the application to repeatedly, and rapidly, issue new motor commands. In this situation, the response to over current would be similar to that of the POT and R/C modes.

### **CASE2: FastOff (**oc\_fastoff** = ‘1’)**

This response protects your motor and driver by disabling all motor commands until you reset the board. When the over current condition is detected, power to the motor will be immediately removed without ramping, and the motor control interface will -NOT- be automatically restored when the measured current is again acceptable (Vout in Region B). A board reset is required before you will again be able to operate the motor. The Serial CMD Interfaces will still function, but will not respond to motor movement requests.

## **9.7 Adjusting the Current Limit Thresholds**

Each MAX5478 Digital Pot Device provides dual 50K Ohm, non-volatile, 256 tap position, pots. There is one device (and two pots) per motor. The board has 4 test points ({V1H,V1L} for motor1, {V2H,V2L} for motor2) which provide convenient access (meter or scope) for the voltage divider outputs from the pots. You will hook up your meter (or scope) between these test points and GND to measure the voltage thresholds.

The POT devices are programmed over the primary I2C bus to set the wiper positions to one of the 256 taps. Increasing the value written to the tap position [0x00...0xFF] will increase the output voltage at the corresponding test point. There are 12 registers on each device (see table below), but it is only necessary to program registers VREGA and VREGB on each device. As indicated in the schematic, **the VxH test points correspond to wiper A and the VxL test points correspond to wiper B**. The discussion and table below should help with the adjustment.

### MAX5478 Registers

Reg Name	Reg#	Function
<b>VREGA</b>	<b>0x11</b>	<b>Wiper A tap position (VH)</b>
NVREGA	0x21	Non volatile default for Wiper A
NVREGAtoVREGA	0x61	Copy NVREGA to VREGA
VREGAtoNVREGA	0x51	Copy VREGA to NVREGA
<b>VREGB</b>	<b>0x12</b>	<b>Wiper B tap position (VL)</b>
NVREGB	0x22	Non volatile default for Wiper B
NVREGBtoVREGB	0x62	Copy NVREGB to VREGB
VREGBtoNVREGB	0x52	Copy VREGB to NVREGB
VREG	0x13	Both Wiper tap positions
NVREG	0x23	Both Non volatile defaults for Wipers
NVREGtoVREG	0x63	Copy NVREG to VREG for both Wipers
VREGtoNVREG	0x53	Copy VREG to NVREG for both Wipers

In the absence of any configuration code on startup, the wipers would be set to the NVREG values on power up. However, shortly after power up commences, the Dalf firmware reconfigures the pots to values held in the Parameter Block. This means that the non-volatile memories on the MAXIM parts are essentially unused and don't need to be changed. Simply record the desired power up values in the Parameter Block along with the other usual power up defaults. Similarly, it should not be necessary to access any of the copy functions (which use the other registers) for these parts. Follow this procedure to adjust the pots and record the settings into the Parameter Block:

1. Attach a meter or scope to the appropriate VxH or VxL test point (see schematic).
2. Use the command interface (Cmd\_M = SET Digital POT) to adjust the wiper and the corresponding wiper voltage until you have the measured voltage set as desired.
3. Repeat for all 4 pots.
4. Use the command interface (Cmd\_Z = UPLOAD To EEPROM) to upload the current values to the Parameter Block. Alternatively, you can write the desired pot settings directly into the Parameter Block using the command interface (Cmd\_W = Write Memory Byte).

Reset the board and retest the pot voltage levels. The settings should match what you recorded above.

EXAMPLE:

```
M 1 11 A0      ; V1H
M 1 12 60      ; V1L
M 2 11 B0      ; V2H
M 2 12 50      ; V2L
Z              ; Upload
I              ; Reset the board
```

The voltage at V1H after power up will be roughly -----  $(0xA0/0xFF)*5 = (160/255)*5 = 3.1V$

## 10 Terminal Emulator CMD Interface

This section applies to the TE Command Interface, however the discussion of command functionality applies equally well to all three serial interfaces (TE, API, I2C2). The command protocol (syntax) will differ a bit between the 3 interfaces, but the functionality will in general be identical. For example, a “Read Memory Byte” operation from any of the 3 board memories (RAM, Internal EEPROM, External EEPROM) is achieved with the “R” command and is provided in all 3 interfaces.

The Terminal Emulator Interface (**TE**) is an ASCII RS232 interface suitable for control and monitor of your motors with the use of any of the many generic Terminal Emulator Applications available for your PC. While arguments and results are transmitted and received as ASCII characters, the user interface requires some familiarity with hex format.

The Applications Programming Interface (**API**) is a binary RS232 interface suitable for a smart PC Application like a Windows GUI. The API command protocol is described in a separate document.

The Secondary I2C Interface (**I2C2**) is a binary I2C interface suitable for communication with an off board device that hosts the Secondary I2C bus as a MASTER (The Dalf Board is configured as a SLAVE on this bus). The I2C2 command protocol is described in a separate document.

Beginning with Firmware Version 1.62, there is support for a Cmd Interface Timeout Feature. This feature when enabled, will shutdown all motors if a valid serial command is not received before the timeout period. This is of primary interest as a safety feature for motors driven by one of the two programmable interfaces (API, I2C2), so will not be described here. See the separate documents describing the API and I2C2 interfaces for details.

### 10.1 RS232 Setup

The TE interface, accepts most inputs, and delivers some outputs, in ASCII Hex character format. While this is not extremely user friendly, it is reasonably efficient. All you need to use the TE interface is a standard RS232 cable and a terminal emulator application running on your PC. The command interface is implemented with an RS232, 19,200 baud, 8bit, no parity, 1 stop bit interface. The commands and data are expected to be transmitted and received as ASCII characters via a Terminal Emulator Application running on a PC. The default baud rate may be changed (after powerup) by changing the **nBR** value in the Parameter Block.

### 10.2 Command Syntax (**CMD D0 D1 ... Dn**)

A command string is composed of a single first byte which identifies the command. The CMD byte is optionally followed by command parameters that are transmitted as data byte pairs.

The CMD byte may be any ASCII character ‘A’-‘Z’, ‘a’-‘z’, but the lower case entry is converted to upper case (there are really only 26 commands). Each data byte is sent and received as a byte pair that represents a single hex byte. For example to send the data byte 0xF3, the two ASCII characters ‘F’ and ‘3’ are transmitted.

Pressing ENTER on the keyboard generates a carriage return (**CR=0x0D**) which terminates the command string and initiates command processing by the firmware. The data byte pairs may be separated by **delimiters (SP, COMMA, or SEMI)** to improve readability in the terminal emulator window. Most data received will be echoed back for display on the terminal emulator window. Delimiter characters are not different in this sense, but the interrupt handler which receives data discards them - they are not stored in the Receive input buffer. Most control characters are simply ignored, but there are a few exceptions. The backspace character (**BS=0x08**) functions as a backspace (ie; is echoed back to the terminal emulator),

allowing you to do error recovery before sending the command string. If the particular command is designed to return data, it generally sends the data back in the same format.

**Example:** The command “A 0B” stops both motors, disables R/C and POT control modes, and resets the PWM freq to 16,026 Hz. For a discussion of the details of how this works, see [Cmd\\_A](#).

### **Command syntax tips:**

All of the examples in the command section which follows show the data byte inputs as 2 characters separated by the SP delimiter. For example the command to read a byte from the external EEPROM at address 0x000C is shown as “R 02 00 0C”. The ‘R’ is the command byte, the 0x02 is the memory type (in this case external EEPROM), and the remaining two bytes 0x00 and 0x0C indicate the high and low order bytes of the address. Notice that the command string actually consists of 10 bytes (including spaces), but I have referred to the “byte pairs” as bytes, because that is how they will be used by the command handler. In this particular example the syntax would be indicated by [CMD D0 D1 D2] to indicate the four fields.

The TE Command Parser allows some short-cuts that you might want to use after you are familiar with the non-abbreviated form. Hex bytes which may be represented as a single hex character (values 0 thru 15) may be entered as single characters provided that they are separated from other variables with a delimiter. Delimiters are not required, but if not used, you must enter all data bytes in the paired form even if the value is a single digit. For example: “R02000C”, “R2 0 0C”, “R 2 0 C”, and “R2 0 C” are all acceptable alternatives to the command parser for the above command.

### **Error handling:**

The command parser will generally catch and respond to most syntax errors by echoing an appropriate error message and aborting the command. Be aware that while the parser will verify the expected number of arguments for the command, the commands themselves may not always do complete error checking. So if a particular command specifies that an argument is range limited, it is up to you to ensure that you supply a value that is in range.

### **A word about logging:**

Both the TeraTerm and Hyperterm terminal emulators provide a mechanism to record your session to a text file. This can be very useful. One example would be usage in conjunction with the “Read Memory Block” command to record your Parameter Block settings to a file. Another example is usage with features like the PID Tuning Command to capture data for graphical analysis with other programs (eg; Excel). With TeraTerm, use the File/Log.. menu to select the log file. With Hyperterm, use the Transfer/Capture Text tabs.

### **A word about the motor movement commands:**

The **open loop** motor control command is Cmd\_X. This command allows you to specify motor speed as a PWM duty cycle percentage [0-100%]. Because the command is open loop, the parameter is not really a motor speed control, but really a motor power control. As with all of the open loop control modes, there is no feedback from the motor and therefore no guarantee that the motor will actually achieve the desired speed. In fact, lugging down the motor(s) after execution of this command will result in a noticeable speed decrease. Take a look at the roles of the parameters **VSP** and **AMINP** in motor control.

There are two different **closed loop** motor control commands. The first of these is Cmd\_S which allows closed loop velocity control. The second is Cmd\_Y which allows you to specify a trapezoidal trajectory speed profile for moving the motor to a target position under PID control. Before you try the closed loop commands, you should read the material discussing **PID Tuning**. Besides the VSP and AMINP parameters, pay attention to the role that some of the **PID parameters** play. You will probably need to adjust some or even all of these for optimum functionality.

All of the commands are explained in the Command Specification Section.

### 10.3 Terminal Emulator Cmd Crib Sheet

Cmd	Description	Syntax
A	Set PWM freq	A <index>
B	Fan Control	B <fan#> <on/off>
C	Get A/D	C <adc_ch>
D	Set/Get RTC	D <hh><mm><ss>
E	Get Motor Position(s)	E <mtr#>
F	Set Encoder	F <mtr#> <eHi> <eMid> <eLo>
G,H	//UNUSED//	
I	Reset	I
J	IO Expander Write	J <ioexp#><ioreg#><datatype>
K	IO Expander Read	K <ioexp#><ioreg#>
L	Read Memory Block	L <memtype><adrshi><adrslo>
M	Set Digital POT	M <pot#><reg#><byte>
N	Get R/C Pulse Widths	N <ch>
O	Stop Motor(s)	O <mtr#>
P	Set/Get PIDParms	P <mtr#> <KPH><KPL> <KIH><KIL> <KDH><KDL>
Q	PID Step Response	Q <mtr#> <TgtHi> <TgtMid><TgtLo><LimHi><LimLo>
R	Read Memory Byte	R <memtype><adrshi><adrslo>
S	Move (constant velocity; closed loop)	S <mtr#> <dir> <VmHi> <VmLo> <AccHi> <AccLo>
T	Trigger Move (Closed loop)	T <mtr#>
U	Motor Status	U <mtr#>
V	Motor Velocity	V <mtr#>
W	Write Memory Byte	W <memtype><adrshi><adrslo><datatype>
X	Move (constant velocity; open loop)	X <mtr#> <dir><spd><tSlew>
Y	Move (to position; closed loop)	Y <mtr#> <TgtHi><TgtMid><TgtLo> <VmHi><VmLo> <AccHi><AccLo>
Z	Upload to EEPROM	Z

**Notes:**

- A: index: (0x00.. 0x18) selects pre-programmed PWM frequencies.
- B: fan#: (1,2), on/off: (0=off, 1=On)
- C: adc\_ch: ADC Channel (0-6) optional
- D: Decimal arguments.
- E: mtr#(1,2) optional.
- F: mtr#(1,2), e=24 bit, 2's complement encoder position (default: 0).
- J,K: ioexp#:(1,2), ioreg#:(0x00..0x15).
- L: memtype{1=RAM, 2=Ext EEPROM, 3=Int EEPROM}, block size=128 bytes.
- M: Pot# (1,2); reg#(0x11, 0x12, .. others - see data sheet); Two devices, 4 pots.
- N: ch: R/C Channel (1,2,3) optional
- O: mtr#(1,2) optional.
- P: Enter mtr# for View. Set requires all 7 args. KP, KI, KD 16 bits.
- Q: Tgt is 24 bits, Lim controls output to Terminal Emulator.
- R: memtype:(1,2,3)
- U,V: mtr#(1,2) optional.
- W: memtype(1,2,3).
- X: tSlew: (0..255) msec. Default = AMINP.

- Y: 8,6, or 4 args (24 bit Tgt is required. Acc default is ACC, Vm default is VMID.
- Z: Uploads 128 bytes starting at RAM: 0x0100 to Ext EPROM: 0x0000.

## **10.4 Terminal Emulator Command Specification**

Each of the command handlers has expectations with regard to the number and type of arguments. Some arguments represent single hex bytes, while others are multi-byte arguments. In the case of multi-byte arguments, the high order bytes are entered first, just as you would write them (this is one difference between this interface and the API and I2C2 interfaces). For example; the two byte address of 0x01C would be entered as “01 5C”. Some arguments are optional and this is indicated in the CMD description. In some cases, optional arguments, which are not supplied, are replaced with defaults. In other cases, the lack of the argument signals “both” as in some commands that expect an argument for motor#. See each command for details.

In the command descriptions, D0 will refer to the first data byte, D1 to the second and so forth. Remember that each of these data bytes is actually an ASCII byte pair that represents a single **HEX** byte.

### **10.4.1 Cmd\_A      PWM Freq Control**

**Syntax: “A D0”**

**D0 = PWM\_TBL Index**

The index selects the PWM frequency from the table below. If the motors are running they are first ramped to a stop (using AMINP for deceleration) and the disable (DIS) lines are asserted. This command will also disable any Pot or R/C Control Modes. The value of D0 is then copied into the ERAM variable corresponding to fPWM (see the Parameter Block) and then used as an index into the following table to initialize the PWM hardware control registers: {T4CON, PR4} to produce the requested frequency.

$$f_{\text{PWM}} = 1/[(PR4+1)*4*T_{\text{osc}}*\text{prescale}]; \quad T_{\text{osc}}=1/40E6, \quad T4CON \text{ defines a Prescale of 1,4, or 16.}$$

The command provides a PWM frequency range described by the following table:

PWM Frequency Table

D0 (=fPWM)	FREQ (Hz)	T4CON	PR4
0x00	2,441	0x7F	255
0x01	3,005	0x7F	207
0x02	4,006	0x7F	155
0x03	5,000	0x7F	124
0x04	6,010	0x7F	103
0x05	7,022	0x7F	88
0x06	8,013	0x7F	77
0x07	9,058	0x7F	68
0x08	10,000	0x7D	249
0x09	12,019	0x7D	207
0x0A	13,966	0x7D	178
0x0B	16,026	0x7D	155
0x0C	17,986	0x7D	138
0x0D	20,000	0x7D	124

(Table continued next page)

(Continued)

D0 (=fPWM)	FREQ (Hz)	T4CON	PR4
0x0E	25,000	0x7D	99
0x0F	30,120	0x7D	82
0x10	35,211	0x7D	70
0x11	40,000	0x7C	249
0x12	45,045	0x7C	221
0x13	50,000	0x7C	199
0x14	59,880	0x7C	166
0x15	69,930	0x7C	142
0x16	80,000	0x7C	124
0x17	90,090	0x7C	110
0x18	100,000	0x7C	99

Comments:

- 1) **The PWM frequency selection provided by this command governs the current session only.**  
To make the selection “permanent”, you can write the fPWM value directly into the Parameter Block to affect the default power-up frequency, or use the Upload feature. (See commands “Write Memory Byte”, “Upload To EEPROM”).

#### 10.4.2 Cmd\_B      Fan Control

**Syntax: “B D0 D1”**

**D0 =Fan# (1,2)**

**D1 = Fan On/Off control . . . . . (0=off,    1=on)**

This command controls a pair of high current drivers (minimum 200mA for each) suitable for inductive loads. Each driver consists of an NPN transistor, base resistor, and a free-wheeling diode to handle inductive loads. If using the OSMC driver boards and the standard 10 pin connector, the supply for the fans will be the +12V regulated output from the OSMC boards (See schematic). Alternatively, you may supply your own “fan” power supply by connecting your power source to the +12V and GND pins on the PL1 and PL2 connectors. The high side of the load is connected to the power source thru the connector labeled F1+ (or F2+). The low side of the load is connected to the collector of the drive transistor using the F1- (or F2-) connector.

The D1 value is used to control the microcontroller output pin that delivers current to the base of the drive transistor selected by D0. The state of this control is recorded in a bit in the ERAM variable MODE1 and affects the current session only. Write your selection directly into the Parameter Block to affect the default power-up state (See commands “Write Memory Byte”, “Upload To EEPROM”).

#### 10.4.3 Cmd\_C      Get AtoD Reading (mV)

**Syntax: “C D0”**

**D0 = Channel (0-6)**

**Returns: Channel reading in mV**

**If this command is given without the D0 argument, readings from all channels are returned.**

The Analog-To-Digital-Converter State Machine provided in the firmware cyclically samples and records the values of the seven analog inputs (signals AN0 - AN6) in memory starting at the RAM location ADC0. Timing control of the ADC State Machine can be customized by altering parameters in the Parameter Block which act as indices into a timing table. The measured values are stored in an 8-bit hex format with the value proportional to the input voltage. This command converts the requested reading(s) to units of millivolts and returns the converted value(s).

Channel 6 is dedicated to monitoring the VBATT voltage and the ADC measurement corresponds to that portion of VBATT that is produced by the voltage divider between VBATT and GND. The measurement of the voltage at this divider is maintained in ADC0[6] in a manner identical to the other values. However, periodically (once every second) in the main loop service Svc3, the value is converted to a VBATT reading and stored in the RAM variable “vbatt”. The conversion of ADC0[6] to vbatt can be “tuned” by changing the resistor divider calibration constant **VBCAL** in the Parameter Block.

#### 10.4.4 Cmd\_D      Set/Get RTC

**Syntax:** “D”

**D0 = HH (Hours: 0-23)**

**D1 = MM (Mins: 0-59)**

**D2 = SS (Secs: 0-59)**

**If no Arguments returns: “HH:MM:SS” (current time)**

Unlike most others, this command accepts and returns results in decimal. The results and inputs assume the 24 hour clock format ( $0 \leq \text{HH} \leq 23$ ).

If no arguments are provided (**Get RTC**), this command simply displays the RTC: HH/MM/SS by converting the hex values stored in the RAM variables HOURS, MINS, and SECS to BCD.

If the 3 arguments are supplied (**Set RTC**), the BCD input values are converted to Hex and stored in the RAM variables HOURS, MINS, and SECS.

#### 10.4.5 Cmd\_E      Get Motor Position

**Syntax:** “E D0”

**D0 = Motor# (1,2)**

**Returns: [D0,D1,D2] = encoder position (24-bit, 2's complement format).**

**If the command is issued with no argument, encoder positions for both motors are returned.**

The result represents the current value of the position counter maintained by the firmware and associated with the optical encoder on the selected motor shaft. This is a 24 bit, 2's complement value and the result is presented as a 6 digit hex string. Optionally (TE Interface only), you can select decimal display by setting the “pos\_d” bit in the MODE3 parameter in the Parameter Block.

Eg; “0079F3” – Motor currently at forward position 0x79F3 = 31219 ticks.

Eg; “FFF624” – Motor currently at reverse position 0x9DC = -2524 ticks.

Each tick represents an edge on the optical encoder signals, so there are  $4 * \text{CPR}$  ticks per shaft revolution (CPR is the optical encoder spec: “Cycles Per Revolution”). An optical encoder that has CPR=64 will have TPR=256 (“Ticks Per Revolution”). With such an encoder in the first example, the motor would have rotated forward (from the home position) by  $31219/256 = 121.9$  revolutions. In the second example, the motor position would correspond to a reverse rotation position of  $2524/256 = 9.9$  revolutions.

#### 10.4.6 Cmd\_F      Set Encoder

**Syntax:** “F D0 D1 D2 D3”

**D0=Mtr#(1,2)**

**[D1, D2, D3] = Encoder Value**

This command sets the encoder count on the specified motor. The Encoder Value argument is assumed zero if it is not supplied (resets encoder).

First the command disables POT or R/C Control Modes and stops (removes power from) the motor. It is possible for motor momentum to change the encoder count after the value has been set by this command.

To avoid this, stop the motor and wait a reasonable settling time before issuing the command. Alternatively, check the result using the “Get Motor Position” command. If not equal to the desired setting, you can reissue the “Set Encoder” command.

**10.4.7 Cmd\_G**      // UNUSED //

**10.4.8 Cmd\_H**      // UNUSED //

**10.4.9 Cmd\_I**      **Reset Board**

**Syntax:** “I”

This command waits briefly for the terminal emulator “echo” of the chr “I” to complete and then executes the **RESET Instruction**. Generally a software reset is not as robust as a full hardware reset, but the RESET instruction purports to be just as good.

**10.4.10 Cmd\_J**      **IoExp Write**

**Syntax:** “J D0 D1 D2”

**D0=IOEXP# (1,2)**

**D1=Reg Address (0x00 ... 0x15)**

**D2=Data Byte**

There are two IO Expanders on the Dalf Board and each provides 16 GPIO’s which are routed to connectors on the board. The microcontroller communicates with these devices over the primary I2C bus at a bit rate of 400 KHz. This command gives write access to the registers internal to the IO Expanders. It is used to provide configuration and output control of the GPIO’s.

The byte value D2 is written to the register number D1 on the IO Expander specified by D0. See the section “IOEXP Functions” in the Firmware Library for details concerning the IO Expanders and their registers.

**10.4.11 Cmd\_K**      **IoExp Read**

**Syntax:** “K D0 D1”

**Returns:** D2

**D0=IOEXP# (1,2)**

**D1=Reg Address (0x00 ... 0x15)**

**D2=Data Byte**

This command provides Read Access to the registers internal to the IO Expanders. This includes input pin status for those GPIO’s configured as inputs. The byte value D2 is read from the register number D1 on the IO Expander specified by D0. See the section “IOEXP Functions” in the Firmware Library for details concerning the IO Expanders and their registers.

**10.4.12 Cmd\_L**      **Read Memory Block (RAM OR EEPROM)**

**Syntax:** “L D0 D1 D2”

**D0=Memory Select (1=RAM, 2=EXT\_EEPROM, 3=INT\_EEPROM)**

**D1=AdrHi**

**D2=AdrLo**

This command provides a 128 byte window into any of the 3 memory devices. The address represents the location of the first byte in the 128 block. The data is sent to the terminal emulator screen in groups of 8

bytes per line. The address is range checked for validity (to make sure that the address +127 is in the device memory space):

**Valid [D1 D2] Address Range:**

RAM:	0x0000 - 0x0F80
Ext EEPROM:	0x0000 - 0xFF80
Int EEPROM:	0x0000 - 0x0380

These ranges permit a view of the full address space of the devices. You can use this command with terminal emulator logging enabled to capture a copy of your Parameter Block settings to a file. This will enable you to easily restore your settings at a later time if that becomes necessary.

**Note that the API and I2C2 versions of this command provide the same functionality but are a bit more general because they permit a variable block size for the amount of data to be read.**

#### 10.4.13 Cmd\_M

#### Set Digital POT

**Syntax: “M D0 D1 D2”**

**D0=Pot Select (1=Pot1, 2=Pot2)**

**D1=Pot Cmd (Generally VREGA=0x11, or VRGB=0x12)**

**D2=Tap Position (0-255)**

This command provides for adjustment of the voltage references used by the over current detect circuitry. Each Pot Device has dual 50K pots with 256 wiper positions for each. See the section “Current Sensor Feedback” section in this manual for details on how to adjust the pots for current sensing applications.

#### 10.4.14 Cmd\_N

#### R/C Snapshot

**Syntax: “N”**

**Returns: D0,D1,D2**

RtnVar	Pulse Width (uSec)
D0	Ch1
D1	Ch2
D2	Ch3

This command is used to view the pulse width outputs in units of microseconds from your receiver. The results are accurate to within a microsecond and can be useful for “R/C switch tuning”. See a discussion of this topic in the Radio Control section of this document. In a nutshell, use the results of this command to see exactly what your receiver is outputting for each of the corresponding transmitter switches.

#### 10.4.15 Cmd\_O

#### StopMotor(s)

**Syntax: “O D0”**

**D0 = Motor# (1=Motor1, 2=Motor2)**

**If no Motor# argument is supplied, this command will stop both motors.**

This command stops the specified motors using a slew rate determined by the AMINP parameter in ERAM. This command essentially stops the motors as fast as you have deemed safe by your choice of AMINP. If you have set AMINP to 0, this command will **immediately** (drive train ouch!) stop the motor(s).

**Note:**

This command disables POT and R/C Control Modes (ERAM based flags), and terminates the Trajectory Generator and PID Controller if these are active. The operating mode becomes CMD Mode. Here is a

potential “gotcha” to be aware of: If you use this command and then save your run-time environment (ERAM) with the “UPLOAD TO EEPROM” command (before re-asserting POT or R/C control modes), your next power-up will have CMD Mode as the default. If power is being supplied to the motor for braking (eg; steep hill), and this command is executed, the motor will free-wheel.

### 10.4.16 Cmd\_P

### Set/Get PID Parameters

Syntax: “P D0 D1 D2 D3 D4 D5 D6”

D0=Motor# (1=Motor1, 2=Motor2)

D1=KPH

D2=KPL

D3=KIH

D4=KIL

D5=KDH

D6=KDL

This command allows you to set the 16-bit constants {KP, KI, KD} which play a major role in PID control. It also permits you to view these 3 and a few other control variables affecting PID performance.

#### Set:

If you enter the command with the full argument list it will copy the arguments into the ERAM variables (for use during this session only) and do -NOT- alter the values stored in the Parameter Block. Typically this command is used as part of the “PID Tuning” process and the final values are later uploaded or stored directly into the Parameter Block to become the new power-up defaults.

The PID constants {KP,KI,KD} are not the total PID story by far, so be sure to read the Trajectory Generation and PID section of this document for a description of the other variables that may need to be changed to achieve optimum performance in your application.

#### Get:

If this command is issued with the Motor# as its single argument, the current PID constants and some other mostly PID related variables will be displayed. An example appears below:

```
“P 2”
MTR# : 2
  KP    : 0700      (7*256)
  KI    : 0400      (4*256)
  KD    : 0C00      (12*256)
  VSP   : 14        (20 msec)
  VMIN  : 03        (3% motor brake/deadband)
  VMAX  : 64        (100%: PWM not limited)
  MAXE  : 2000     (Stop motor if Err exceeds MAXERR=8,192)
  MAXS  : 0100     (Clip Integral Sum at +/- 256)
```

Note that all of the above variables may be changed for the current session, which can then be uploaded to the Parameter Block in EEPROM to become the new power-up defaults if desired. **However, note that this command allows you to change KP, KI, and KD only.** See the “Write Memory Byte” command for details on changing the other RAM variables. See the “Write Memory Byte” and “Upload To EEPROM” commands for changing the Parameter Block in Ext EEPROM.

### 10.4.17 Cmd\_Q

### PID Step Response

Syntax: “Q D0 D1 D2 D3 D4 D5”

D0=Motor# (1=Motor1, 2=Motor2)

D1=StepH                    (Target Destination; Hi Byte)

D2=StepM                    (Target Destination; Mid Byte)

D3=StepL                    (Target Destination; Lo Byte)

D4=PidLimitH              (Output Limit; Hi Byte)

D5=PidLimitL              (Output Limit; Lo Byte)

This command is used to evaluate the step response of the motor control system for use in closed loop operations. You will not need this command unless you are planning on using closed loop motor operations (Cmd\_S, Cmd\_Y, Servo Modes). Before executing this command, read the “**PID (Closed-Loop) Motor Control**” section of this document to get a brief description of PID and the role of the control parameters in determining the PWM duty cycle (power) output to the motor.

In practice, this command is used in close association with commands “P”, “W”, “R” and “L”, with the goal of obtaining closed loop control parameters that produce an acceptable step response. The process of identifying the proper control parameters is called “**PID Tuning**”. Manual PID Tuning describes a trial and error process which is facilitated by this command and your observations of the result. Typical usage involves 3 steps repeated over and over again until you obtain satisfactory results:

1. Make parameter changes
2. Execute this command (identical arguments each time)
3. Evaluate the results

This trial and error process is repeated until you are satisfied with the step response. The evaluation of the results consists of watching the behavior of the motor and analyzing the output to the terminal emulator window (assuming verbose enabled) after the motor has stopped. By enabling logging, you can capture the output to a file which can be imported for graphical analysis (eg; Excel). With a bit of experience, you will be able to determine the needed parameter changes simply from the terminal emulator output. When you have used this command to obtain a reasonable step response for your motor you will have made the coarse adjustments necessary to get a reasonable motor response in your application. Any further adjustments in your actual application should be minor and you will be ready for any of the closed loop operations.

#### What are the arguments?

- The **Step** argument governs how long the motor will run. It is used as the target destination. Typically, one or two seconds of runtime is sufficient to evaluate the motor control response. The **PidLimit** argument governs how much output is sent to the terminal emulator screen. For example, with PidLimit = 0x0200, there will be 512 lines of output. Typically, 20 to 40 observations after the motor has reached the target are sufficient. If the PidLimit argument is not supplied, then it will be defaulted to the value **NPID** in the Parameter Block. If you have terminal emulator logging enabled when you execute this command you will have captured the step response to a file which can be imported into a graphical application (eg; Excel) for easy analysis. However, after a bit of experience, eyeballing the results in the terminal emulator window is often sufficient. There are a couple of step response plots in Appendix A to give you an idea of what to expect.

#### What does the command do?

- After preliminary argument checks, the motors are stopped if necessary and PID is disabled. Any POT or Radio Control modes will also be disabled. Then the command resets the encoder position to [000000] (\*). This allows you to repeat this same command over and over again with

a constant value for the step. It also means that this particular command operates in the same way regardless of the rel/abs# setting (See the MODE1 flags).

(\*) - For an absolute analog encoder, the encoder is reset to [00 00 xx] where xx represents the position of the motor in the current revolution. If you keep the low byte of your step target = 0x00, subsequent command executions will likely start with an encoder position near [00 00 00] - similar to the optical encoder case.

- The Step argument is loaded into the 24 bit variable “**xpos**” which the PID controller uses and PID is enabled to rapidly drive your motor to the target destination. Assuming that “Verbose Mode” is enabled (default=enabled), the PID Err value during each VSP sampling period is output to the terminal emulator screen. It is this list of errors that will help you determine how well “tuned” your motor response is. The Err values will start at the target step value (that is the initial position error) and should rapidly converge toward zero with possibly some overshoot and oscillations. The error may never reach zero, may overshoot zero, or may oscillate around zero.

**You will probably want to issue the stop motor command (Cmd\_O) shortly after the motor reaches the target. This will remove power from the motor and stop any oscillations that may be present.**

#### What is the goal?

The goal is a set of PID parameters that will rapidly drive your motor to the step destination with little or no overshoot and with the error rapidly settling at or near zero without continuing oscillations.

A bit of pre-planning and setup will help to make this a smooth process. You will benefit from reading the sections in this document on Trajectory Generation and PID Control before attempting this command. This command replaces what the Trajectory Generator provides in normal closed loop motor moves with a fixed stationary target (your Step input), but otherwise most of the material in those sections apply.

#### A bit of caution:

1. This command causes an abrupt motor response even with perfectly tuned controls. So, be sure that your wheels (or whatever) are off the ground when you execute it. If you are concerned about your drive train, disconnect it. Be aware however that the best controls for a no-load system will probably not be optimal for a fully loaded system.
2. Stay close to the terminal keyboard, ready to issue the off command (Cmd\_O) in case things get out of hand. **The expected behavior for this command is that the motor will start rather violently at near full speed in an attempt to acquire the destination.** As the motor nears the destination, it should slow down, but there will quite possibly be some overshoot, and oscillations about the target before the motor settles at (or near) the destination. It is also possible that oscillations will continue, so be ready to stop the motor if necessary.

#### Some tips:

1. As a **starting point for the step value** in your testing, I would suggest a step magnitude that will give you no more than 2 seconds of runtime at maximum motor velocity

**Example:** While running motor2 forward at full (100%) duty cycle using the open loop motor move command (eg; “X2 0 64”), you find that the velocity (eg; “V2”) is 0x000019 = 25 ticks/VSP2. If VSP2 = 20msec, this represents a velocity of (25 ticks/20msec) or 1250 ticks/sec. So a value of 0x000500 (1,280) might be reasonable for the target value. Using a less “analytical” approach, just try the command a few times starting out with small values for the target until you get somewhere between 1 and 2 seconds of runtime before your motor arrives at the target.

2. Use **verbose mode** (the default). This will allow you to capture the PID Err to the display of the terminal emulator. Experiment with the PidLimit argument to get just enough output and possibly a bit more to capture the “essential portion” of the step response. You will want enough output to see the PID Err get to (or near) zero and a bit more -- perhaps as much as 20 to 40 samples more. Once you have settled on a value for PidLimit you might want to record it (NPID) so that you don’t have to keep entering it (if you record it in the Parameter Block remember that you will have to reset the board to make it usable).
3. If you have **logging enabled(\*)** in your terminal emulator application, you will have captured the motor step response in a log file which you can then easily import into a graphical application like Excel for a visual analysis. See the examples in Appendix A.

(\*) To record your session to a text file, use the File/Log.. feature in TeraTerm or the Transfer/Capture Text tabs in Hyperterm.

4. Since you will likely be changing parameters in a trial and error process, Use the “**Set/Get PID Parameters**” command to view the PID parameters just before executing the step response command. This will give you documentation in your terminal emulator window of the PID parameters used to produce the test results.

#### **Parameter Suggestions:**

- Begin by taking the ErrSum term and the ErrDiff terms out of the PID picture for initial tuning. You can do this by initially setting the KI and KD parameters to zero (see the PID equations).

It is quite possible that your application will not need all of the PID capabilities. For example, you might be just fine with a PD (no integral term) controller. **Don’t feel like you must use all of the PID terms just because they are available.** The I-term may be unnecessary unless you have an application where you have difficulty achieving and maintaining a zero steady-state error (eg; stopping a robot on an incline).

- Be patient and methodical.
- **KP:** Start with small numbers. Start the tuning process with something like {KP=0x0100, KI=0x0000, KD=0x0000}. Then gradually increase KP in small increments (say 0x0100) until you are satisfied. You can use the Set/Get PID Parameters command to change or view the run time values of these 3 constants. You can only use this command to change the primary 3 PID constants KP,KI, and KD, but the command can be used to VIEW pretty much all of the variables involved in the PID process. By setting KI and KD to zero, you will take the Integral and Derivative terms out of the picture initially allowing you to home in on a correct value for KP. With KI=0 you also won’t have to worry about MAXSUM or the sumho flag for a bit. If you have very fine resolution on your encoder (CPR large), or you have fast motors, you may need to increase MAXERR to avoid false triggering of an error condition which will abort the “STEP RESPONSE” command.

Your goal should be to have a sufficiently large value of KP that the motor reaches the target reasonably rapidly. A bit of overshoot is ok - the KD term can fix that up later. Your motor should settle at or near the target. Values of KP that are too large may cause oscillations that don’t stop. Back the value of KP off until that doesn’t happen. You may want to experiment a bit with the value of VMIN to make sure that you don’t have this parameter set too high.

- **KD:** Now try increasing increasing the value of KD in small increments (say 0x0100) to reduce overshoot and oscillations. If there was overshoot and/or oscillation in previous trials without the use of KD, you should begin to see a noticeable difference. When most or all of the oscillations and overshoot are gone, stop increasing KD.

- **KI:** OK, you are satisfied with your KP and KD parameters. If you decide that your application needs the integral term ( $KI > 0$ ) you will again want to start out with small test values for KI. You also need to know that the integral errors (the KI term) will begin to accumulate immediately when using Cmd\_Q, regardless of the state of the “sumho” flag. The issue here is that in normal PID operations initiated by Cmd\_Y, it is the Trajectory Generator which decides when to eventually enable ErrSum accumulation (when the trajectory is finished and presumably the motor is near the final target). However for the “STEP RESPONSE” command, the Trajectory Generator is not running. You will need to be sure to test final adjustment of KI using Cmd\_Y.
- **Additional testing:** This PID Tuning Process will have given you the necessary coarse adjustment to the PID parameters and with their use you can expect reasonable results from closed loop operations. You will need to do a bit of testing with Cmd\_Y (“Mtr Move (Closed-Loop Control”) to really ensure proper functionality. When you do this be sure to enter reasonable (achievable) values for velocity and acceleration variables {Vm, a}. Try motor movements with both slow and faster (but achievable) values for Vm. Try movements with final destinations that are both close and far from the starting position. If you find that moves to very close destinations don’t result in motor movement or leave the motor short of the destination, you may need to increase the gain (KP) a bit. If it becomes necessary to make additional changes, you might want to recheck the results of CMD\_Q with the new values.
- **Saving your values:** When you feel satisfied with the Step Response you will probably want to save your current parameters in the Parameter Block so that they will become the default. The easiest way to accomplish this is to immediately use Cmd\_Z (“Upload to EEPROM”). Remember that by using Cmd\_Q you will have reset the operating mode to CmdOnly (disabled R/C and POT operation) and this setting (unless first changed) will be copied as well. Alternatively you can copy your current settings individually to the Parameter Block using Cmd\_W.

#### 10.4.18 Cmd\_R

#### Read Memory Byte

**Syntax:** “R D0 D1 D2”

**Returns:** D3

**D0 = Device (1=RAM, 2=Ext EEPROM [24LC512], 3=Int EEPROM)**

**D1=AdrHi**

**D2=AdrLo**

**D3=Value of byte read at location [AdrHi, AdrLo].**

This command allows you to read a single data byte from any of the 3 memory areas:

- **RAM** [0x0000 - 0x0FFF] is internal to the microcontroller.
- **Ext EEPROM** [0x0000 - 0xFFFF] is an external device accessed on the primary I2C Bus.
- **Int EEPROM** [0x0000 - 0x03FF] is internal to the microcontroller.

The last 160 bytes of the RAM [0x0F60 - 0x0FFF], the special function registers (SFR’s), are memory mapped in the microcontroller to control the various hardware peripherals (timers, pwm, i2c, serial ports, etc.). The PIC18F6722 Data Sheet (Microchip web site) gives all of the details needed for successful access of the SFR’s. Details of general RAM usage depend on evolving firmware design, but there is an **important 128 byte block of RAM** (starting at address 0x0100) referred to in this document as **ERAM**. On power-up ERAM will be a copy of the first 128 bytes of the Ext EEPROM (the **Parameter Block**). It is useful to become familiar with ERAM since some changes to ERAM allow runtime testing without the need to change the Parameter Block and reboot.

Examples:	R1 1 0	; Read RAM location 0x0100
	R2 67 F0	; Read Ext EEPROM location 0x67F0
	R3 3 FF	; Read Int EEPROM location 0x03FF

#### 10.4.19 Cmd\_S Loop)

#### Move (Constant Velocity, Closed-

Syntax: "S D0 D1 D2 D3 D4 D5"

D0: Motor# (1,2)  
D1: Motor Direction; 0=Fwd, 1=Rev  
[D2 D3]: vm = 16 bit scaled (\*) int, desired Velocity (ticks/vsp)  
[D4 D5]: a = 16 bit scaled (\*) int, Acceleration (ticks/vsp<sup>2</sup>)

This command provides motor control for applications that require constant velocity under varying motor loading conditions. The command is implemented using PID and the Trajectory Generator. The discussion in the PID and Trajectory Generator sections of this document applies to this command. The main difference between this command and the closed loop move command, is that here no target is specified, and the motor will remain in the constant velocity portion of the velocity profile indefinitely (until stopped or another motor command is issued). When the motor would normally decelerate to the target, this command causes the constant velocity portion of the profile to be restarted (including resetting the encoder position).

Three possibilities for command syntax are supported by this command depending on whether all arguments are supplied or not. The optional arguments, if not supplied, will be replaced with defaults from the Parameter Block:

{mtr#, dir, vm, a} - All 6 arguments supplied.  
{mtr#, dir, vm} - 4 arguments supplied. ACC is used for the acceleration.  
{mtr#, dir} - 2 arguments supplied. VMID is used for vm, and ACC is used for acc.

This command will error if attempted while in one of the R/C or POT Operating Modes. The command will also error if either vm = 0, or a = 0. Please see the Trajectory Generator and PID sections in this document for additional details.

**It is important to use reasonable (achievable) Trajectory Generator cmd arguments for "vm" and "a" for closed loop operations. This applies to the commands: Cmd\_S, Cmd\_Y as well as servo operations.** Your motor system has characteristics that determine the range of achievable values for acceleration and velocity. Experiment with the open loop commands to determine these characteristics. If values are used that cannot be achieved by your motor application, you may experience abrupt motor transitions and/or error terminations prior to target acquisition. This applies as well to the values of **VMID** and **ACC** in the Parameter Block since these values may be used as defaults for the "vm" and "a" arguments.

See Cmd\_Y (Motor Move - Closed Loop) for a more detailed discussion of the "vm" and "a" scaled arguments.

#### 10.4.20 Cmd\_T

#### Trigger Move (Closed Loop)

Syntax: "T D0"

D0=Motor# (1=Motor1, 2=Motor2)

When trigmode is enabled, the closed loop motor move commands (See Cmd\_S, Cmd\_Y) do not commence immediately, but instead wait for a trigger to be issued by this command. In this way, synchronized closed loop motor movements are possible.

This command will generate an error if trigger mode is disabled (default=disabled). The trigmode flag is contained in the **MODE1** parameter (See Parameter Block). You can use the Write Memory Byte command to change either the ERAM copy or the Parameter Block copy.

**If no argument is given with this command, any pending closed loop moves for either motor will be started.**

#### 10.4.21 Cmd\_U

#### Get Status

**Syntax:** “U D0”

**D0 = Mtr# (1,2)**

This command sends operating status for the specified motor to the terminal emulator screen. If the Mtr# argument is not supplied, results are supplied for both motors. The display will look like the following:

MTR#	:	1	(possibles: 1, 2)
MODE	:	CMD	(possibles: CMD, POTF, POTC, RC, RCMIX)
PWM	:	FWD( 20%)	(possibles: FWD(xxx%), OFF, REV(xxx%))
TRIG	:	OFF	(possibles: ON, OFF)
SUMHO	:	ON	(possibles: ON, OFF)
TGT	:	ABS	(possibles: REL, ABS)
OCMODE	:	NONE	(possibles: NONE, FASTOFF,SLOWRUN)

MODE is self explanatory. If the selected motor is not in one of the POT or RC modes this will be CMD. The percentage shown for forward or reverse is **duty cycle**, not speed. If you have PWM power applied to the motor, this number will be positive even if your motor is locked up. The TRIG, SUMHO, and TGT settings apply only to closed loop motor control. TRIG indicates whether you are in closed-loop-trigger mode or not. That is; do closed-loop motor commands execute immediately, or do they await a trigger signal for execution? SUMHO indicates whether you are in Err-Sum Hold Off mode or not. In this mode, PID err summations are not applied (held off) to the PID equation until near target acquisition in order to avoid the “Integral Windup” issue. TGT indicates how the value entered for target in the motor move command will be treated. Is the target position to be entered as the absolute position, or is the target position to be given relative to the current motor encoder position? ODMODE indicates whether or not over current detect is enabled, and if yes, which response is selected.

**To change mode settings, see the parameters MODE1, MODE2 in Appendix D.** The runtime environment can be altered for the current session only, or the values in the Parameter Block may be changed in order to become the defaults for future power-ups.

#### 10.4.22 Cmd\_V

#### Get Motor Velocity

**Syntax:** “V D0”

**D0 = Mtr# (1,2)**

**Returns:** D1,D2,D3

**D1=VH**           **(Velocity High Byte)**

**D2=VM**           **(Velocity Mid Byte)**

**D3=VL**           **(Velocity Low Byte)**

**If this command is given without argument, both motor velocities are given.** The result represents the last calculation stored in the 24 bit, 2's complement RAM variable V1 (or V2 for motor2). The units are

encoder ticks/VSP msec. Optionally (TE Interface only), you can select decimal display by setting the “vel\_d” bit in the MODE3 parameter in the Parameter Block. This value is computed once per VSP sample period as the difference between the encoder count during the previous period and the current encoder count. The Velocity Sampling Period is controlled by the ERAM variable **VSP** (separate values possible for motor1 and motor2). The default values for VSP1 and VSP2 (see Appendix D) is appropriate for my application/motor/encoder combination, but may not be right for yours. A value of VSP=0x14 represents a velocity sample every 20 msec.

Example: VSP2 = 0x14 = 20 msec, CPR=64. Assume this Cmd delivers a result of “000019”. The interpretation here would be 0x19 = 25 ticks/20msec or 1,250 ticks/s. TPR = 4\*CPR = 256 ticks/rev. This translates to a motor angular velocity:

$$V2 = (1,250 \text{ ticks/s}) * (1\text{rev}/256\text{ticks}) * (60\text{s}/\text{min}) = 293.0 \text{ rpm.}$$

The **TPR** (Ticks-Per-Revolution) parameters (See Parameter Block) value is used by this command to convert velocity in ticks/VSP into units of RPM.

#### 10.4.23 Cmd\_W

#### Write Memory Byte

Syntax: “W D0 D1 D2 D3”

**D0** = Device (1=RAM, 2=Ext EEPROM [24LC512], 3=Int EEPROM)

**D1=AdrHi**

**D2=AdrLo**

**D3**=Value of byte read at location [AdrHi, AdrLo].

This command allows you to write a data byte to any of the 3 memory areas:

- **RAM** [0x0000 - 0x0FFF] is internal to the microcontroller.
- **Ext EEPROM** [0x0000 - 0xFFFF] is an external device accessed on the primary I2C Bus.
- **Int EEPROM** [0x0000 - 0x03FF] is internal to the microcontroller.

See the Cmd “Read Memory Byte” for a brief discussion of these memory areas.

Examples:

W1 1 0 12	; Write byte 0x12 to RAM location 0x0100
W2 67 F0 8	; Write byte 0x08 to Ext EEPROM location 0x67F0
W3 3 FF E2	; Write byte 0xE2 to Int EEPROM location 0x03FF

#### 10.4.24 Cmd\_X

#### Move (Constant Power, Open-Loop)

Syntax: “X D0 D1 D2 D3”

**D0: Motor# (1,2)**

**D1: DIR=Motor Direction Control ----- (0=Forward, 1=Reverse)**

**D2: SPD=Motor Speed Control ----- (0-100; PWM Duty Cycle: 0=Stop, 100=Full Speed)**

**D3: tSLEW=Motor Acceleration Control ---- (0-255 ms: 0=Immediate, 255=slowest)**

The tSLEW argument is for acceleration control in ramping the motor to the specified duty cycle. The units of tSLEW are milliseconds. If tSLEW is not provided, or tSLEW < AMINP(see Parameter Block), this command will use AMINP as the default value for tSLEW.

The command will error if attempted while in one of the R/C or POT Operating Modes.

#### **Open Loop:**

This command does not use the optical encoder feedback to affect the motor drive and is consequently open loop in nature. The actual acceleration is inversely related to the tSLEW input value. The acceleration as well as velocity will be affected by motor loading. While the duty cycle input directly controls the magnitude of the velocity, it does not allow you to maintain a given velocity under loading conditions. Even under no load conditions, the value of the input will yield a different velocity depending on battery voltage, motor characteristics, etc.

The duty cycle (0-100) represents a percentage of full voltage. The acceleration variable “tSLEW” allows for ramping the motor speed to the desired state, avoiding abrupt motor state changes. Emergency stops or other necessary immediate state changes can be achieved with tSLEW=0 (\*). The 8-bit variable “tSLEW” is actually a period and has units of milliseconds. Every “tSLEW” milliseconds, the firmware will adjust the duty cycle from the current state by 1% until the desired duty cycle (your entry) is achieved.

**Example:** Assume that motor1 is running in the forward direction with a 22% duty cycle and you want to increase this to 48% (0x30) somewhat gradually:

X 1 0 30 14 ; Motor1, Forward, 48% Duty Cycle, tSLEW = 20 milliseconds.

In this case, since the motor is currently running forward with a 22% duty cycle, the command calls for a 26% change (48%-22%). Because tSLEW is 20 milliseconds, this will occur at a rate of 1% every 20 milliseconds. The motor will reach the target speed in roughly  $26 * 20 \text{ msec} = 520 \text{ msec}$ .

- Direction changes require that the motor first be stopped. However, the firmware takes care of this for you by first stopping the motor (ramping down velocity at a rate given by tSLEW=AMINP) and then rescheduling your requested motor control operation after the motor is safely stopped.
- The motor may be stopped using either the “Stop Motor” command or by using this command with a duty cycle of 0%. In either case, the motor will be stopped immediately if “tSLEW”=0 (\*), or ramped to a stop if “tSLEW” is non-zero.

(\*) Actually, you will be able to stop the motors immediately only if you have “disarmed” the acceleration governor AMINP (set AMINP=0). This variable can be changed in the run-time environment for the current session or changed to become the default for tSLEW by changing the value in the Parameter Block. Setting AMINP to zero is probably not a good idea, but you may want to customize it for your application.

#### 10.4.25 Cmd\_Y

#### Mtr Move (Closed-Loop Control)

Syntax: "Y D0 D1 D2 D3 D4 D5 D6 D7"

D0: Motor# (1,2)

[D1 D2 D3]: x = 24 bit, 2's comp. Target Position (ticks)

[D4 D5]: vm = 16 bit scaled (\*) int, desired Midcourse Velocity (ticks/vsp)

[D6 D7]: a = 16 bit scaled (\*) int, Acceleration (ticks/vsp^2)

Three possibilities for command syntax are supported by this command. Some optional arguments, if not supplied, will be replaced with defaults from the Parameter Block:

{mtr#, x, vm, a} - All 4 arguments supplied.

{mtr#, x, vm} - 3 arguments supplied. ACC is used for "a".

{mtr#, x} - 2 arguments supplied. VMID is used for "vm", and ACC is used for "a".

This command will error if attempted while in one of the R/C or POT Operating Modes. The command will also error if either vm = 0, or a = 0.

This command starts the Trapezoidal Trajectory Generator which generates path waypoints for the motor to follow in moving to the destination. Simultaneously, a closed loop PID control process is started that drives the motor to follow the path to the target destination. The acceleration variable "a" controls the rate of change of the velocity during the constant acceleration and deceleration portion of the process. The variable "vm" controls the velocity during the constant midcourse velocity portion of the profile. The REL/ABS setting (default = ABS) for mode of target entry determines how the "x" argument is treated. If in ABS mode, "x" is treated as the target destination. If in REL mode, "x" is treated as a signed offset from the current motor encoder position in determining the target position.

Please see the Trajectory Generator and PID sections in this document for additional details.

#### Important Notes

1) **Here is a list of some other control parameters in the Parameter Block that influence the operation of this command.** In general, it will be necessary for you to first "Tune" the PID Controller (obtain optimum values for some of these parameters) before expecting good results from the closed loop motor control commands.

KP ..... Proportional constant  
KI ..... Integral constant  
KD ..... Derivative constant  
VSP ..... Velocity Sampling Period (msec)  
VMIN ..... PWM minimum (motor deadband)  
VMAX ..... PWM maximum (Speed governor)  
MAXERR ..... Maximum PID position error before automatic motor turnoff.  
MAXSUM ..... PID Error sum will be clipped to +/- MAXSUM  
MODE1 ..... See EEPROM section for bit definitions.  
VMID ..... Midcourse velocity (default for "vm")  
ACC ..... Trajectory Generator Acceleration (default for "a")

2) **If closed loop trigger mode is enabled** (disabled by default), this command will be held as pending awaiting a trigger command. See the Closed Loop Trigger command for details.

3) **It is important to use reasonable (achievable) Trajectory Generator cmd arguments for "vm" and "a" for closed loop operations. This applies to the commands: Cmd\_S, Cmd\_Y as well as servo operations.** Your motor system has characteristics that determine the range of achievable values for acceleration and velocity. Experiment with the open loop commands to determine these characteristics. If

values are used that cannot be achieved by your motor application, you may experience abrupt motor transitions and/or error terminations prior to target acquisition. This applies as well to the values of **VMID** and **ACC** in the Parameter Block since these values may be used as defaults for the “vm” and “a” arguments.

**The variables “a” and “vm” are given in units that involve VSP.** This means that if you use different values of VSP for each motor, you will need different arguments for Motor1 and Motor2 commands to achieve the same effective motor speed and acceleration.

**The values you enter for “a” and “vm” are treated as scaled inputs by a factor of 256** for use by the trajectory generator. The scaled inputs for these parameters provide an easy vehicle to allow fractional control values without actually having to deal with fractional inputs.

**Example:** You enter 0x680 for vm. The value used by the Trajectory Generator is:  $VM=vm/256=6.5$  (ticks/vsp). If you use the “Get Motor Velocity” command during the constant velocity course of this command, you would expect to generally see a value of  $V=6$  (ticks/vsp) or  $V=7$  (ticks/vsp); the velocity sampling period is asynchronous with encoder transitions. If the acceleration entry is “a”=0x0030, then the actual acceleration used by the Trajectory Generator is  $A=a/256=0.1875$  (ticks/vsp<sup>2</sup>).

#### 10.4.26 Cmd\_Z

#### Upload To EEPROM

**Syntax:** “Z”

Use this command to copy ERAM (the RAM data that represents the current run-time-environment) into the Parameter Block of the non-volatile EEPROM. In this way, the current environment will be restored to RAM (copied from the Parameter Block to ERAM) on the next power-up. For a list of the data that is copied, see Appendix D.

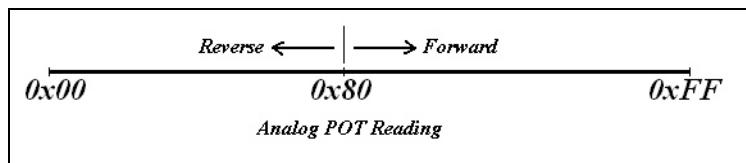
If necessary, the original factory defaults for the Parameter Block can be restored (losing any customizations) by simply holding down on the BTN switch during power up reset.

## 11 POT Mode Interface

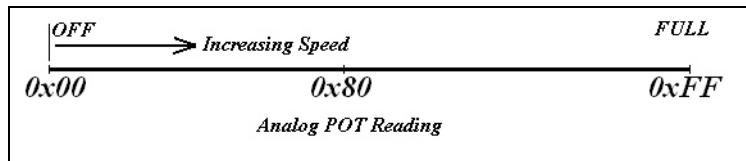
The pot control modes are disabled by default. Before you will be able to use these motor control methods, you will have to enable it using the command interface (See MTRx\_MODE2). You can enable the pot mode control for the current session, or you may make it the default operating mode for subsequent power-ups. In addition to the three open loop velocity control modes described here, there is a closed loop position control mode (PotSvo) described in a later section. See also the section of this document which describes the optional digital low pass filter.

### 11.1 Pot Modes

In **PotC Mode**, the center position of each pot is mapped to zero velocity and excursions in either direction produce increasing speed changes in either forward or reverse. POT analog readings of 0x80 thru 0xFF are mapped to motor velocity in the forward direction, and readings of 0x7F thru 0x00 are mapped to motor velocity in the reverse direction.



In **PotF Mode**, the full range of each POT is mapped to a 0-100% motor duty cycle and the setting of the FWD/REV# switch controls both motor directions.



As an example, in the PotC mode, the central location of the pot (2.5V) will correspond to “motor off” while in the PotF mode, that same position and voltage will result in ½ speed (either forward or reverse), with the separate switch controlling the motor direction.

In **PotMix Mode**, the signals from two external pots are “mixed” in order to use a single, 2-axis, control device known as a joystick. In this mode, forward stick travel (+Y) results in forward drive being applied to -both- motors. Reverse stick travel (-Y) results in reverse drive being applied to -both- motors. Left (-X) and right (+X) travel on the stick is used to modify -both- the motor drive levels for turning. The connections are not symmetric with this mode (see below).

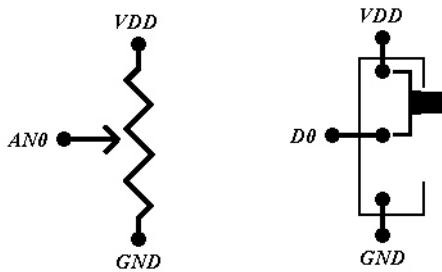
#### Recommended connections for a differential drive system:

Connect your motor 1 driver to the right motor (as seen from drivers’ seat), and motor 2 driver to left motor. Confirm that serial motor commands drive the proper motor in the desired direction before proceeding. Now configure both motors in POTC mode (without mixing). This will enable you to more easily get the joystick movements in sync with the desired motor directions. Connect the wiper signals from the joystick pots to the analog inputs. The Y-Pot wiper should be connected to AN0 screw terminal input (Motor 1 = right motor) and the X-Pot wiper should be connected to the AN1 screw terminal input (motor 2 = Left motor). Now connect the Vdd and GND connections to the X-Pot and Y-Pot such that +Y drives motor 1 forward and +X drives motor 2 forward. If the directions are wrong, switch Vdd and GND connections and

retest. Now change the operating modes for both motors to PotMix (like the R/C Mix mode, this operating mode must be selected for both motors in order to achieve the mixing operation) and test joystick operation.

## 11.2 Pot Connections

The pot range is not too important, but I recommend using pots in the 2K Ohm range. Smaller values waste power needlessly. Larger values can impact the response time for the A/D sampling circuitry. Regardless of which pot mode you have chosen, the POT hook-ups are identical. Connect the center pin of the 2 pots (the wipers) to the AN0 and AN1 analog inputs thru the Dalf on-board connector. Any available VDD and GND connections on the Dalf board can be taken off board for the other two pot pins. For safety, an external On/Off switch (On=VDD, Off=GND) is required to be connected to the D0 input. You can use SPST switches with current limiting resistors, but the simplest arrangement is to use SPDT switches (see the diagram below).



If PotF mode is used, connect the additional “Direction Control Switch” (Forward=VDD, Reverse=GND) to the D1 input on the Dalf board using a SPDT switch with similar hookup to that shown for D0.

## 11.3 Enabling Pot Modes

A pot control mode is enabled whenever the bit which enables it in the MODE2 variable is set. There are essentially two ways to enable the desired pot control mode. If you want the change to become the default operating mode on subsequent power ups, record the change in the Parameter Block in the parameter that corresponds to MODE2. If you want the pot mode to be active temporarily only, you can make the change to the ERAM variable MODE2, but if you do it this way you should stop the motors first.

- **To make the change to RAM:** Use the “Read Memory Byte” and “Write Memory Byte” commands to change and verify the value of the variable MODE2.
- **To make the change in the Parameter Block:** Use the same commands as above, but with the appropriate arguments for access to the Ext EEPROM data. Then use the RESET command to power up using the changed value to set the default operating mode to the specified pot control mode.

You can use Cmd\_U (Get Status) to confirm correct operating mode.

## 11.4 Important Pot Mode Parameters:

Two important parameters in the Parameter Block that affect the Pot operating modes are PMSP and AMINP. You should be fine with the defaults to begin with, but you may want to customize these variables later. Appendix D shows where these parameters are located in the Parameter Block.

The **PMSP** (Pot Mode Sampling Period) Parameter controls how frequently the firmware acts on your pot and switch inputs. This is distinct from the ADC rates which control how frequently the analog inputs from the pots are acquired and stored in RAM.

The **AMINP** (Acceleration Minimum Period) Parameter is an “acceleration governor” and is designed to limit the slew rate to safe levels during rapid velocity changes or abrupt stops. This parameter, whose units are milliseconds, is important for other motor control methods as well. AMINP is the minimum acceptable period (=fastest acceptable acceleration/deceleration) at which a 1% change will be made when ramping velocity changes.

## 12 R/C Mode Interface

The R/C Control Modes are disabled by default. Before you will be able to use either the RCNRM or RCMIX open loop motor control method, you will have to enable it using the CMD Interface (See MODE2 in the Parameter Block). You can enable radio control for the current session (changes to ERAM), or you may make it the default operating mode for subsequent power-ups (changes to Parameter Block).

### 12.1 Receiver Connections

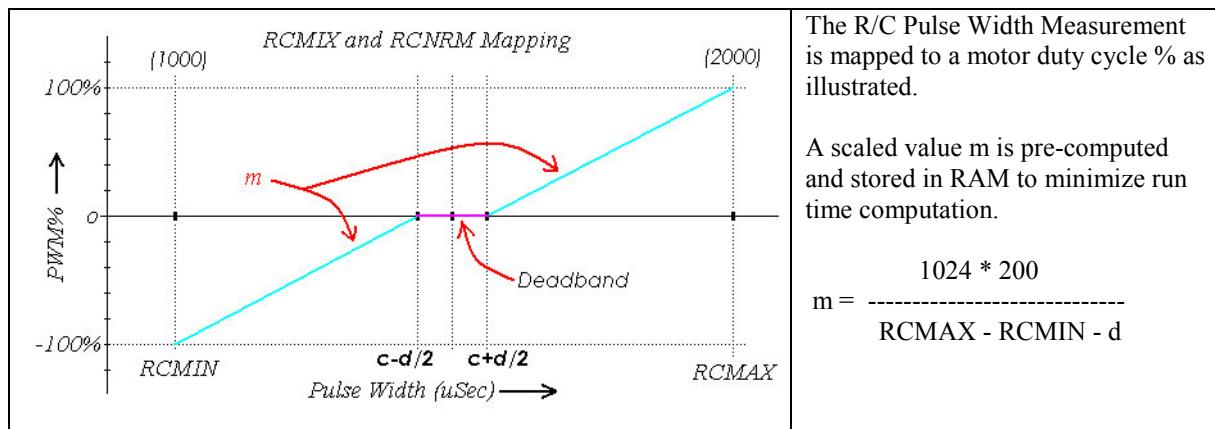
Any available VDD and GND connections on the Dalf board can be used to supply +5V receiver power. Connect the receiver outputs to the Dalf Board connector as follows:

EX0:ch1  
EX1:ch2  
EX2://unused//  
EX3:ch3

### 12.2 Receiver Output Signals

A receiver output is represented by a pulse whose time width conveys the transmitter switch position. Typically, full travel on the transmitter switch results in a pulse width output at the receiver with a nominal range of [1000 uSec, 2000 uSec] with the center position “c” (about 1500 microseconds) corresponding to “servo off”. The pulse is repeated by the receiver every 20 msec or so. The Dalf firmware provides an adjustable “deadband” parameter to ensure that the switch center position will truly represent off. Interrupt driven firmware and special hardware (CCP Modules) on the Dalf Board capture pulse width timing measured in microseconds for each of the 3 channels. There is essentially no latency and **0.80 microsecond resolution** in this process so the pulse width measurements are quite accurate. The pulse widths are then mapped (depending on R/C operating mode) to the appropriate motor control (PWM) responses.

### 12.3 R/C PWM Mapping



## **12.4 R/C Tuning**

Some R/C transmitters permit “endpoint adjustments” to properly match switch endpoints (end of switch travel on the transmitter) to the desired receiver output (pulse width) on the channel assigned to the switch. This process of adjusting each switch to its assigned servo can be thought of as “**R/C Switch Tuning**”. For example, in typical R/C operation, a desired position for a particular servo can be made to correspond to full forward on the transmitter switch that controls it. This adjustment is done on a channel by channel basis by adjusting the transmitter output so that the receiver generates the desired output pulse width for that channel when the switch is at full travel (forward and/or reverse).

The R/C interface on the Dalf Board controls a pair of DC motors whose velocity range [-100%, 100%] is mapped to the pulse width range [RCMIN uS, RCMAx uS] for channels 1 and 2. The defaults for RCMIN and RCMAx are 1000 uS and 2000 uS respectively. If the maximum travel of your channel 1 transmitter switch results in a pulse output from your receiver whose width is 1850 uSec (for example), this will not result in full motor1 speed without some adjustments. There are two ways to perform this tuning process depending on whether the change is made at the transmitter or on the Dalf Board:

### **12.4.1 Endpoint Adjustment At Transmitter:**

If your transmitter has this feature you may make the switch travel adjustment at the transmitter. With this procedure you adjust the transmitter until your receiver outputs have pulse widths that match the Dalf defaults [RCMIN, RCMAx]=[1000 uS, 2000 uS] stored in the Parameter Block. The full travel of the switch will then correspond to the [RCMIN, RCMAx] range that the Dalf firmware uses for the mapping. The “R/C SNAPSHOT” Command (Cmd\_N) can be very helpful here allowing you to make the adjustment easily and accurately without the necessity of any test equipment (eg; oscilloscope).

### **12.4.2 Endpoint Adjustment On Dalf Board:**

Use the “R/C SNAPSHOT” (Cmd\_N) command to determine the pulse widths corresponding to the existing switch endpoints for each channel you will use on the Dalf Board. Then record these values into the EEPROM Parameter Block (see the “WRITE EEPROM” command) in place of the existing RCMIN, RCMAx default values. These new settings will become the defaults on subsequent power-ups. This method has the advantage that if you use your transmitter with another device, your transmitter switches can remain “tuned” to that device, while still functioning optimally with the Dalf Board.

Either of the two methods above will result in matching the receiver outputs for full range of switch travel on your transmitter with the Dalf R/C Parameters, and the full range of switch travel on the transmitter should then correspond to the full range [-100%, 100%] of motor duty cycle (ie; you will be R/C Tuned”).

## **12.5 R/C Modes**

### **Mixed vs Normal Mode:**

There are two operating modes provided. There is a Mixed Mode and a Normal Mode. In **Normal Mode** (tank mode), channel 1 controls only motor1, and channel 2 controls only motor2. In this mode, the switch controlling channel 1 will be mapped to full forward and reverse operation of motor1 with the center position corresponding to off (analogous to PotC mode). In **Mixed Mode**, channel 1 is the main drive for both motors while channel 2 “modifies” the drive signals for both motors for turning. In this mode, the switch center position still represents off. Mixed mode is commonly used with a single stick on the transmitter (having both XY travel to operate 2 channels). In this mode, forward stick travel (+Y) results in forward drive being applied to -both- motors. Reverse stick travel (-Y) results in reverse drive being applied to -both- motors. Left (-X) and right (+X) travel on the stick is used to modify -both- the motor drive levels for turning. Do not confuse these 2 operating modes with the mixing which may be a feature provided by your transmitter.

## 13 Servo Modes

The two servo control modes are disabled by default. Before you will be able to use one of these motor control methods, you will have to enable it using the command interface (See MTRx\_MODE2). You can enable the servo mode for the current session by changing the parameter in ERAM, or you may make it the default operating mode for subsequent power-ups by changing the parameter in the Parameter Block.

### 13.1 Closed Loop Operation

The control inputs for the servo modes come from either an external potentiometer or an R/C Transmitter just as in the previously explained open loop R/C and POT modes. The primary difference between the servo modes discussed here and the previously discussed R/C and POT Modes is the nature of the motor control.

**In the open loop POT and R/C control modes, the *motor speed* was controlled by the inputs. In the Servo Modes discussed here, it is the *motor position* that is controlled.**

One other difference is that when using the servo modes the motor control is “closed loop”. The Servo Modes require motor position feedback - either with analog or optical encoders, and utilize PID and the Trajectory Generator code to achieve the desired motor position.

**I recommend use of an absolute analog encoder for feedback in servo operations.**

The problem with use of an incremental encoder for servo operations is that the shaft position is not known at startup. When using an incremental optical encoder the position counter is reset to 0x000000 at startup. If your pot or R/C switch requests a position that would require  $\frac{1}{2}$  revolution of the shaft from 0x000000, that is what you will get (even if the motor is already at the desired position on startup!).

Smooth and optimal performance in using the servo modes is also highly dependent on appropriate PID parameters for your particular motor application. If you are not satisfied with the default performance of this feature, and you have not performed “PID Tuning”, I recommend that as your next step. If you have ever driven a car with an engine badly out of tune, you can appreciate the huge difference this can make in servo performance. There is a terminal emulator command (Cmd\_Q) that can assist you in PID Tuning.

### 13.2 Servo Limits

Typically in a servo application the output shaft rotates in an operational region that is some fractional portion of a full revolution. For example, the servo motor might rotate between 10 degrees and 350 degrees. In this example, if the TPR value for your encoder was 256, you would want to set your operational range to [7 ... 249] (Hex: [0x07 ... 0xF9]) in units of encoder ticks.

$$10 \text{ deg} * (256 \text{ ticks/rev}) * (1\text{rev}/360 \text{ deg}) = 7.11 \text{ ticks}$$

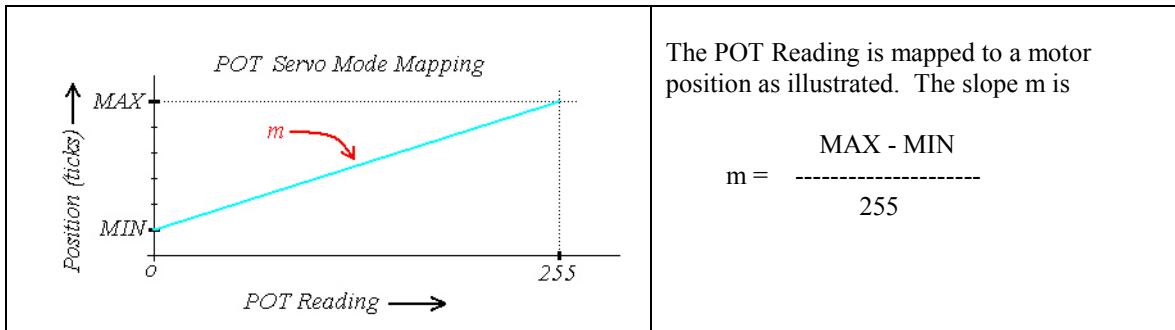
There are two parameters **MIN** and **MAX**, one set for each motor, in the Parameter Block to establish the operating limits for your motor position in servo control modes. You will notice that MIN and MAX are word values, so as you might suspect, the rotation limits are not limited to the [0,360] range, but this is atypical. In the above example, you would set MIN = 0x0007, and MAX = 0x00F9. Be aware that increasing the limits beyond one full rotation will decrease your positional resolution.

The [MIN...MAX] operating limits are “software limits”, not mechanical limits. There is nothing to prevent your motor from briefly moving past the MAX limit for example (especially if your system is not PID Tuned!). If this is a concern, be a bit conservative in setting the limits.

### 13.3 Connections

The POT and R/C connections are identical to that described in earlier sections. In Pot Servo Mode the pot inputs for motors 1 and 2 are connected respectively to the AN0 and AN1 screw terminal inputs and the pot reading is mapped linearly to the desired motor position. Pot Servo mode, like the other Pot modes, requires the use of the external On/Off switch connected to the D0 screw terminal. In R/C Servo Mode the R/C receiver outputs for channels 1 and 2 are connected to the EX0 and EX1 screw terminal inputs and the R/C pulse width measurements are mapped linearly to the desired motor position.

### 13.4 Pot Servo Mapping

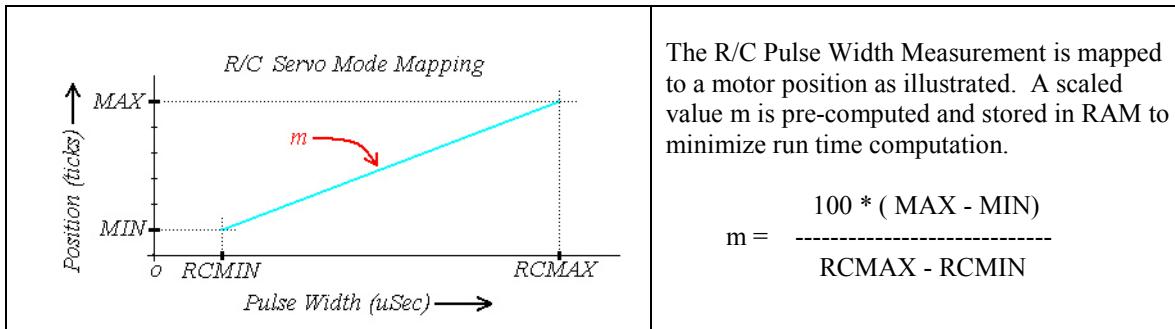


### 13.5 Pot Servo Parameters

Besides the previously mentioned **PMSP** (the default will probably be fine for you here), the **MAX** and **MIN** parameters are obviously important and should be set to meet the needs of your servo application. Because the servo modes utilize PID, the **KP**, **KI**, and **KD** parameters are important.

If you will be using analog feedback (instead of incremental encoders) for your motor position, the value of **DMAX** or **AD\_GAP** may be important to you but probably the defaults will be fine. See the section of this document describing analog feedback for details.

### 13.6 R/C Servo Mapping



### 13.7 R/C Tuning

The process of adjusting the transmitter or alternatively changing the default values of RCMIN and RCMAX to match the receiver outputs for your transmitter switch endpoints is identical to that described in

the earlier R/C section. This step is important to achieve optimum R/C servo performance. If your receiver outputs don't closely match your stored values for RC<sub>MAX</sub> and RC<sub>MIN</sub> the above mapping won't be accurate and your position limits won't be achieved.

### **13.8 R/C Servo Mode Parameters**

Besides the previously mentioned **RCSP** (the default will probably be fine for you here), the **MAX**, **MIN**, **RC<sub>MAX</sub>**, and **RC<sub>MIN</sub>** parameters are obviously important and should be set to meet the needs of your servo application. Because the servo modes utilize PID, the **K<sub>P</sub>**, **K<sub>I</sub>**, and **K<sub>D</sub>** parameters are important.

If you will be using analog feedback (instead of incremental encoders) for your motor position, the value of **D<sub>MAX</sub>** or **AD\_GAP** may be important to you but probably the defaults will be fine. See the section of this document describing analog feedback for details.

## 14 Trapezoidal Trajectory Generator

### 14.1 Trajectory Generator Overview

The Trajectory Generator is enabled whenever a closed loop motor control command is issued (\*). The role of the Trajectory Generator during the course of a motor move command is to periodically generate the desired position (“**waypoint**”) to be compared with the actual (encoder) position by the PID Control. The Trajectory Generator by itself does nothing to alter motor behavior – it just produces data. It is the job of the PID Controller (next section) to make the motor follow the velocity profile produced by the Trajectory Generator. It is useful to understand some of the details of the Trajectory Generator.

Consider the typical case where the motor is stopped and you want the motor to move to a particular target destination (represented by a specific encoder count). The role of the Trajectory Generator is to generate a periodic sequence of waypoints (desired encoder positions) representing smooth velocity transitions to the target. The idea is that if the PID controller can force the motor to follow the path generated by the Trajectory Generator, with the appropriate timing, it will make for a reasonably smooth motor journey. No sudden jerks on start or stop, but instead a smooth ramped acceleration to a midcourse constant velocity that would be maintained until a smooth ramped deceleration to a stop. The “Trapezoidal” portion of the name describes the typical appearance of the velocity profile (velocity vs. time graph) which is the model for the math that generates the “waypoint” data.

There are cases in which controlled motor movements cannot achieve the desired trapezoidal velocity profile. One example is the case of a motor starting position and ending position that are sufficiently close that there isn’t sufficient time to achieve the specified ramp-up, steady velocity, ramp-down to target position given the input parameters for acceleration and midcourse velocity. In fact, the firmware treats 4 different velocity profile cases and does not require the motor to be initially stopped. If the motor is currently moving in a direction opposite to that which is required to achieve the target, it will be smoothly ramped to a stop before the trajectory driven move is started.

#### A note about the PID and Trajectory Generator Math:

The PID motor control and the Trajectory Generator use dedicated floating point computations. By dedicated, I mean that they don’t use the Microchip Floating Point Library routines supplied with the compiler. Instead they use completely separate routines and additionally do not share data storage for parameters and results with the compiler library routines.

(\*) - The Trajectory Generator is -NOT- active during execution of the “PID Tuning” command. Instead, the PID Tuning uses a single, static, step value as the target destination.

### 14.2 Trajectory Generator Details

#### Units

The Trapezoidal Generator and the PID Controller could work in any consistent units. While SI units might be appropriate, they are not computationally efficient. Instead, encoder “ticks” are used as the positional unit, “ticks/vsp” as the velocity unit, and of course, ticks/vsp<sup>2</sup> as the acceleration unit. Given the value of VSP, and the TPR of your encoder (TPR = 4\*CPR), it is easy to convert position, velocity and acceleration values to SI units.

### 14.2.1 Trajectory Generator Timing

The Trajectory Generator Period (the time between generation of successive waypoints) is the Velocity Sampling Period (**VSP**) parameter in the Parameter Block. The value of VSP is an important parameter for closed loop motor control, and should be adjusted to fit your motor/encoder characteristics. An inappropriate value of VSP can cause poor closed loop motor control and/or sluggish system response.

#### How to determine VSP?

In general, for PID control, it is desirable to have the smallest practical sampling period. In practice, there are limitations to making VSP too small. First, there is some serious floating point math that gets executed for each waypoint by both the Trajectory Generator and the PID Controller. If the value of VSP is too small, you will lug down the system with the frequent math. If the value of VSP is too large, there will be too long a period of time between motor update commands during PID controlled operations. This can lead to motor instability or undesirable oscillations. For many applications, the default value of VSP=5 msec will be fine, but if not, how do you choose a value of VSP?

First it is necessary to understand the relationship between VSP and the value of V (ticks/vsp) reported by the “Get Motor Velocity” command. Try this experiment: Use the command interface to apply constant power to your unloaded motor (Cmd\_X). Use the “Get Velocity” (Cmd\_V) command to record the value of V (ticks/vsp). Now use the command interface to double the value for VSP in ERAM and repeat the experiment. You won’t see any change in the actual motor speed of course, but what will change is the value reported for V. If you double the sampling period (VSP) with the motor running at the same speed, then twice as many ticks will occur in the new sampling period, hence V (ticks/vsp) will also double. **For constant motor velocity, the value of V (ticks/vsp) is directly proportional to VSP.** What is important about this relationship is that the value of V can be used to assess a suitable value for the VSP parameter.

Example: Assume that your motor is running at minimal speed and the command interface reports that  $V=6$  (ticks/vsp). If PID were active, at this velocity, new motor commands will be issued every 6 ticks of the encoder (once every VSP). Another way to say the same thing is that the motor will have had a chance to advance 6 ticks since the last PID motor correction. For very slow motor speeds (eg; near the trajectory start and end points), 6 ticks is probably too large. If possible, decrease VSP until  $V \leq 2$  (ticks/vsp) when running at your minimal motor speed.

#### VSP Tips:

- Use the largest value of VSP that provides acceptable closed loop performance.
- If you haven’t already purchased encoders, choose the encoder CPR value to meet your resolution needs, but don’t get finer encoder resolution than you really need.
- Using the CMD Interface, adjust VSP until you get a value of  $V \leq 2$  while running at the lowest power level that gives consistent motor speed. I don’t recommend  $VSP < 3\text{ms}$ , but with some very fast motor/encoder combinations, this may be necessary to achieve acceptable closed loop control.

In order to understand the Trajectory Generator timing, it is helpful to know a few of the details about the closed loop control system provided by the board and firmware. The following sections provide a bit of an “under-the-hood” discussion of what takes place in the Trajectory Generator firmware during closed loop operations.

### 14.2.2 Trajectory Setup

When a closed loop motor control command is issued, the **Trajectory Setup Routine** is called. The setup routine is called only once for each motor command and is responsible for sanity checks on the input parameters, capture of the initial encoder position and motor velocity, conversion of integer arguments to floating point format, determination of the Trapezoidal Case, computation of timing interval endpoints, and computation of the coefficients for the 2<sup>nd</sup> order polynomial  $f(t) = c_0 + c_1*t + c_2*t^2$  which gives the desired distance from the encoder start position at time t. There are as many as 3 timing intervals ( $[0,t_1]$ ,  $[t_1,t_2]$ ,  $[t_2,t_3]$ ) depending on the Trapezoidal Case, and a different polynomial is used for each interval. The timing state variable is set initially to ( $t=0$ ), and flags are set that will cause the Trajectory Routine and PID Routine to be called periodically from the main loop every VSP msec.

### 14.2.3 Trajectory Routine

Every VSP msec the **Trajectory Routine** (not the setup routine) is called to generate a waypoint. The waypoint at time t is a 24 bit 2’s complement value  $x(t)$  which gives the desired motor encoder position at that moment. The Trajectory Routine is implemented as a state machine, but during the execution of any given move command only a subset (depending on the “Trapezoidal Case”) of the states will be traversed.

Every VSP msec, the main loop will first call the Trajectory Routine, followed immediately by a call to the **PID Routine**. The Trajectory Routine increments the timing variable  $t \leftarrow t+1$  (“t” is in units of VSP msec), and evaluates  $d = f(t)$ , the desired distance from the initial encoder position at time t. This is then converted into a 24 bit, 2’s complement format, waypoint  $x(t)$  which represents the desired motor encoder position at time t. If t exceeds, the right endpoint of the current timing interval (a boundary crossing representing a transition in the trapezoidal velocity profile), a transition to a new state occurs requiring the use of new coefficients for  $f(t)$ .

The PID Filter Routine accepts the waypoint  $x(t)$ , the current motor position ( $encode(t)$ ), and the various PID control parameters and outputs a PWM control value that will drive the motor to a position that reduces the error. See additional detail on the PID routine in the next section. I have explained these details here because I want to discuss timing a bit. I have done timing measurements with some typical examples and here is what I have observed.

#### Timing:

The PID and Trajectory Setup Routines are pretty predictable with respect to timing. Generally, the initial call to Trajectory Setup takes the most time by far. I have listed in the table below some typical execution times. Timing for the Trajectory Routine varies depending on the Trajectory Case and the value of the state variable (which timing interval is active). A typical trajectory path (depending on distance to the target and other factors) can often have hundreds or more waypoints. Keeping in mind that an instruction cycle period is  $tOSC/4 = 0.1$  microseconds, here is a table for timing guidance.

Routine	Cycles	Time
Trajectory Setup	9,100	910 uSec
Trajectory	680	68 uSec
PID	1,250	125 uSec

So, assuming a single motor is moving because of a closed loop motor control command you can expect about  $68 + 125 = 193$  uSec to be devoted to the trajectory and PID stuff within every VSP interval. This

figure is an approximation of course, but it is probably accurate to within +/- 15%. Keep in mind that, if you have 2 motors moving at once, these figures are doubled. As an example, if VSP is 10 msec, and both motors are executing a closed loop move then roughly 400 microseconds of every 10 msec (about 4% of the CPU usage) is being used for just the motor movement. Of course there is lots of other stuff going on as well! Your encoders are causing interrupts to increment/decrement the position counters, the AtoD module is generating interrupts to sample the AtoD inputs, timer interrupts are occurring, and of course you want to leave some room for communication interrupts and other processing.

### Trapezoidal Generator Inputs

The Trapezoidal Generator accepts 4 inputs (X, Vm, a, V0). Three of these inputs are provided thru the command line interface when you request a PID Controlled move and the fourth comes from periodic sampling done by the firmware.

- **X: Target Position (ticks)** – This is where you want the motor to go. If you specify a 24 bit value of `0xFFFF23` (-221) as the destination, the current encoder position will determine whether or not the motor moves forward or reverse to get to this destination. The target position can also be specified in relative terms.
- **Vm: Midcourse Velocity (ticks/vsp)** – This value is how fast you want the motor to go during the midcourse (or constant velocity) portion of the journey.
- **a: Acceleration (ticks/vsp^2)** – This value governs how fast the motor ramps the velocity to Vm during the start portion of the path, and how fast it ramps down the velocity to 0 at the target.
- **V0: Initial Velocity (ticks/vsp)** – This value is not entered as part of the motor movement commands, but is an input that affects the trajectory calculations. Its value is the velocity at the time the trajectory path is initiated. **The value of V0 need not be zero, but if the motor is currently going in the wrong direction, the trajectory request is rescheduled after the firmware ramps the motor to a stop (using AMINP for the acceleration control).**

The Trajectory Generator computes the **distance to the target d** as  $d = |X - \text{Encoder}|$  (ticks) and uses this (together) with the other inputs to characterize the velocity profile.

### 14.3 Trajectory Generator Cases

The Trapezoidal Generator is named for the typical velocity profile which looks like a trapezoid. However there are really 4 cases handled by the firmware. Only the most common of these (Case 3) has a velocity profile that actually resembles a trapezoid. Here is a brief outline assuming initial velocity V0 and distance to the final target destination is d:

First compute  $d_{min}$  = minimum stopping distance.

**CASE 1:  $V0 > 0$ ,  $d < d_{min}$**       **[Ramp: Very Close to Target; Ramp down fast to target]**

If not case 1, compute  $V_p$ =maximum velocity before requiring ramp down

**CASE2:  $V0 < V_p < V_m$ ,  $d \geq d_{min}$**       **[Triangle: Ramp up to  $V_p$ , Ramp down to target]**

**CASE3:  $V0 \leq V_m \leq V_p$ ,  $d \geq d_{min}$**       **[Trapezoid. Ramp up to  $V_m$ , Hold, Ramp down to target]**

**CASE4:  $V0 > V_m$ ,  $d \geq d_{min}$**       **[Polygon: Ramp down to  $V_m$ , Hold, Ramp down to target]**

Each of the velocity profiles associated with the 4 cases, can be described by 3 time intervals  $[0,t_1]$ ,  $[t_1,t_2]$ ,  $[t_2,t_3]$ , during which the velocity or acceleration is constant. The computation of the interval endpoints  $t_1, t_2$ , and  $t_3$  takes place during the Trajectory Setup phase (before waypoint generation and the PID controller are actually started).

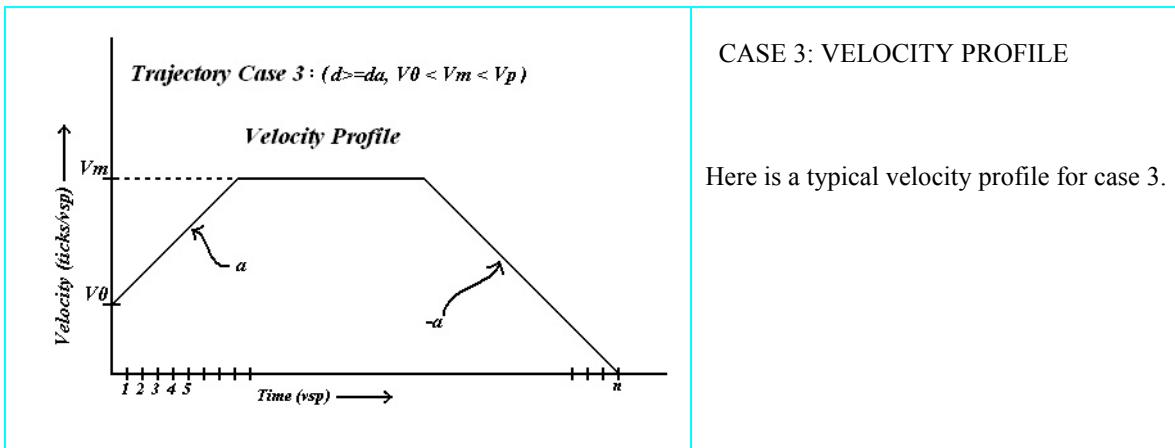
The current time interval and the trapezoidal case determine a “state” in the Trajectory Generator Code. For each t, the math generates the desired distance f(t) from the starting point using a 2<sup>nd</sup> order polynomial whose coefficients depend on the Trajectory State. The value of f(t) and the initial encoder position is used to generate a 24 bit, 2’s complement waypoint x(t) representing the desired motor position at time t.

$$f(t) = c_0 + c_1*t + c_2*t^2 \quad (\text{float32})$$

$$x(t) = \text{waypoint} \quad (24 \text{ bit, 2's complement})$$

(The coefficients (c0,c1,c2) depend on the Trajectory State)

The coefficients for all intervals are computed during the Trajectory Setup phase. State transitions to subsequent states (interval boundary crossings) reload new coefficients from RAM before evaluation of f(t). The format of x(t) is consistent with the output from the encoder. Each motor has its’ own single 24 bit storage location x1(t) and x2(t) which is overwritten as each new waypoint is generated.



#### 14.4 Integral Windup

The term “Integral Windup” refers to an issue with the PID err summation term (See the section “PID (Closed-Loop) Motor Control”). The only thing to mention here is that when the Trajectory Generator finishes (has generated the last waypoint) it enables error summations allowing PID to achieve a zero steady state error, but at the same time avoiding the Integral Windup issue.

## 15 PID (Closed-Loop) Motor Control

There are lots of tutorials, product descriptions, design guidelines, pitfalls, etc. on the web on the subject of motor controllers and PID in particular. Just type PID into google to get instantly overwhelmed. People write books on this subject, and I have even read a good diploma thesis that discussed PID for the control of a mobile robot. I won't try to compete with any of that here. What I will give you is a concise description of the computations used by this firmware, including scaling. This should prove useful in explaining observable motor behavior and modifying your motor control parameters in order to best suit your application. The discussion here is limited to motor position, but obviously, PID used as a control filter can be applied to many different applications. I will also give you some guidelines on "manual tuning" of the PID control parameters and discuss what each of the terms in the PID equation(s) does for you. Hopefully this will help you make the adjustments needed to optimize PID performance for your application.

### 15.1 Proportional-Integral-Derivative (PID) Equations

During trajectory setup (prior to PID start), Err(0)=ErrSum(0)=PrevErr=0, t=0.

The PID controller is periodically called every VSP msec. The timing variable 't' is incremented with every call, hence t is measured in units of VSP msec. The Trajectory Generator terminates after t>t3 (see Trajectory Generator Section of this document), but the PID controller will remain active with a static target. In practice, and somewhat depending on parameter VMIN, this means that a small voltage may still be applied across the motor terminals after target acquisition.

For each call to PID Controller (every VSP msec), the desired target position x(t) has been made available by a prior call to the Trajectory Generator. Encode(t) is the position as determined by the motors position encoder. Here is a summary of the PID Controller Actions:

```
t ← t+1
Err(t) ← x(t) – Encode(t)
ErrDiff(t) ← Err(t) – Err(t-1)

If (sumho=0)      // Err Sum Hold-Off flag not set
{
    ErrSum ← ErrSum + Err;
    If( |ErrSum| > MAXSUM) ErrSum ← Sgn(ErrSum)*MAXSUM; // clip to MAXSUM
}

If (|Err| >= MAXERR) {stop motors; disable PID and Traj Gen;}      //fatal error condition

Else
{
    PIDErr = KP*Err + KI*ErrSum/256 + KD*ErrDiff                  // Basic PID Equation
    SPID = |PIDErr|/1024 + VMIN                                     // Scaled |PIDErr|
    PID = INT( MIN(SPID, VMAX) )                                     // SPID clipped to VMAX
}
```

The value of PID [0-100%] is written to the duty cycle register to control PWM,  
The sign of PIDErr determines the motor direction ("+" = forward).

**That is pretty much it! This activity occurs every VSP msec while PID is active.**

### **PID Equation Notes:**

- Scaling ErrSum by 256 allows larger values (finer resolution) in the value used for KI.
- Scaling by 1024 allows non-integral values for KP, KI, and KD (finer adjustment resolution).
- The PWM duty cycle can never be driven to zero (unless VMIN=0).
- VMAX should never exceed 100 = 0x64.

## **15.2 PID Motor Control Tuning**

Here is a list of some of the parameters that you should consider customizing to tune your motor for closed loop control. All of these Parameter Block parameters are discussed in Appendix D. ERAM based copies can be found starting at offset 0x100 for runtime only changes. It is a good idea to stop the motor before altering any of the ERAM versions of these variables. This section will concentrate on the {KP, KI, and KD} parameters assuming that you have already established reasonable values for the others (see the Trajectory Generator section for a detailed discussion of VSP).

KP .....	PID proportional constant	(16 bits)
KI .....	PID integral constant	(16 bits)
KD .....	PID derivative constant	(16 bits)
fPWM .....	PWM frequency control index	(8 bits)
TPR .....	Ticks Per Revolution (4*CPR)	(16 bits)
VSP .....	Velocity sampling period	(8 bits; 0-255 msec)
VMIN .....	PID Min PWM (deadband)	(% [0-100])
VMAX .....	Max PWM	(% [0-100])
MAXERR ...	PID position error limit	(16 bits)
MAXSUM ...	PID integral limit	(16 bits)
MODE1.....	Motor Control Flags 1	(8 bits)
VMID .....	Midcourse velocity default	(16 bits)
ACC .....	Traj Gen acceleration default	(16 bits)

Be sure that you have the MODE1 bits set properly (in particular the “aleadsb” bit). The defaults for the other bits in MODE1 will be fine for now (I recommend leaving the sumho flag set initially and maybe forever!).

There are analytical methods for determining the setting for some of the PID Control parameters, but even if that makes sense for you, experimental methods still need to be employed to verify and test the results. You should be able to do a quite satisfactory job of tuning by just experimenting with your parameter selections while using the “Step Response” command and being observant.

There are lots of articles on the Internet concerning experimental or “Manual Tuning” of PID control parameters. All of the articles attempt to isolate the effect of the various terms in order to optimize one constant at a time. None of the articles attempts to determine KI first - in fact, I believe that all of them treat KI last, which makes some sense when you understand its role in PID. Unfortunately, the variables are not completely independent (changes in one variable suggest tweaks to another variable that you thought was perfect). I will present a tuning procedure that should work fine for you, but first read this short discussion of the PID equation and what the various terms accomplish. This will lay a framework for your tuning observations.

### **The PID terms:**

Refer back to the section that provides the PID equations. The equation for PIDErr contains three terms; the **P-term**, the **I-term**, and the **D-term**. The P-term alone (Proportional or Err Term) is capable of driving your motor toward the target and in fact does most of the work. However, without the assist of the other 2 terms, you may (depending on the magnitude of KP) get combinations of any of the following: undershoot, overshoot, oscillations about the target, or a non-zero steady state error. The D-term is used to provide

damping - mitigating effects such as overshoot, undershoot, and oscillations about the target. The I-term (Integral Term) is useful to obtain a zero steady state error, but some systems won't require it and there are some issues with its use. If you have heard the saying: "too much of anything is bad", it is especially true when talking about the scaling constants KP, KI, and KD, so start by experimenting with small values.

#### The role of the KP Term:

The KP Term controls the **system gain**. From the definition, Err is the position error and its sign is such to drive the motor toward the target. KP controls how rapidly the system will try to acquire the target. If you consider the step response graph, KP governs the slope of the early portion of the graph.

#### The role of the KD Term:

The KD Term controls **system damping**. Its role is to "dampen" the control system response. If you consider the step response graph, KD will slow down the rate of target acquisition and flatten oscillations that occur near target acquisition. To better understand this, expand the ErrDiff term of the PIDErr equation:

$$\begin{aligned}\text{ErrDiff} &= (\text{VSP} * [\text{x}(t) - \text{x}(t-1)] - [\text{encode}(t) - \text{encode}(t-1)]) / \text{VSP} \\ &= \text{VSP} * \{\text{Desired Velocity} - \text{Actual Velocity}\}\end{aligned}$$

This shows that **ErrDiff is proportional to the Velocity Error**. A positive value for ErrDiff indicates that the motor is going slower than desired. A negative value indicates that the motor is going too fast. So, the KD, or derivative term, has the role of reducing velocity errors.

Note that during execution of the PID Tuning Command, the target position  $x(t)$  is constant, so that in this special case, the ErrDiff term reduces to:

$$\text{ErrDiff} = -\text{VSP} * (\text{Actual Velocity}) \quad [\text{during PID Step Response}]$$

The KD term, has the role of reducing velocity to zero (a damping effect on the step response).

#### The role of the KI Term:

The KI Term controls the **system steady state error**. Even with well chosen KP and KD parameters, the system can still have a non-zero steady state error. The continual accumulation of errors in the ErrSum term is sufficient to overcome the steady state error, driving the motor to the target. There is a technical problem with the use of the KI Term known as "**Integral Windup**". The issue is that errors that have accumulated in the early portion of PID operation can cause the integral term to overwhelm the other terms when near target acquisition (when fine control is needed). The PID Control System used on the Dalf Board deals with this by providing the MAXSUM parameter and the sumho bit setting. I recommend using both the MAXSUM variables and the sumho flag to restrain the integral contribution in normal operation

If you are ready to begin PID Tuning for your application, let me encourage you at this point to become familiar with some helpful commands:

- Cmd\_P: Set/Get PID Parameters
- Cmd\_Q: PID Step Response
- Cmd\_R: Read Memory Byte
- Cmd\_W: Write Memory Byte
- Cmd\_Z: Upload To EEPROM

These commands are your PID motor tuning friends. When using Cmd\_P to change the PID constants, you will be making temporary changes to ERAM variables. Be sure to record permanent changes or work that you wish to keep for the next session in the Parameter Block.

## 16 Firmware Library Routines

These functions, written in PIC Assembler, are provided in the library file <dalf.lib>. They are compatible with the runtime environment (eg; stack, parameter passing, etc.) of the Microchip mcc18.exe compiler. If you are customizing or extending the board functionality and wish to use any of the functions described in this section you will need to link the file included on the CD into your build. Use of the functionality in the <dalf.lib> file is limited by the license to use on this board only. The <dalf.lib> library was created from files compiled using the {Large code, Large Data, Multibank Stack} model settings.

Function	Description
WriteIOExp1	Write byte to IO Expander #1 Register (Microchip MCP23017)
WriteIOExp2	Write byte to IO Expander #2 Register
ReadIOExp1	Read byte from IO Expander #1 Register
ReadIOExp2	Read byte from IO Expander #2 Register
GetTime	Get elapsed time since power up (resolution: 1 tick = 30.518 uS)
SetDelay	Establish delay = future time (Now + Delay)
TimeOut	Has delay expired? (set delay with SetDelay)
WriteExtEE_BYTE	Write byte to external eeprom (24LC512; 64K Byte EEPROM)
ReadExtEE_BYTE	Read byte from external eeprom
WriteExtEE_BLOCK	Write block of bytes to external eeprom
ReadExtEE_BLOCK	Read block of bytes from external eeprom
WriteIntEE_BYTE	Write byte to internal eeprom (PIC18F6722; 1K Byte EEPROM)
ReadIntEE_BYTE	Read byte from internal eeprom
WriteIntEE_BLOCK	Write block of bytes to internal eeprom
ReadIntEE_BLOCK	Read block of bytes from internal eeprom
WritePot1	Write byte to Pot1 Register (MAXIM5478; 50K; dual digital pot)
WritePot2	Write byte to Pot2 Register

You will get more than the functions listed above, and documented in the sections that follow, from <dalf.lib>. There are numerous other functions used in <main.c> with fairly transparent usage that do not appear in the above list. As one example, this <main.c> function uses the “TeCmdDispatch()” function in <dalf.lib> to effect a closed loop motor move, exercising the same code that is used by the serial interfaces for Cmd\_Y. This means that you will be able to perform motor control operations with your own code linked with <dalf.lib> and <main.c>.

```
//-----
void MoveMtrClosedLoop(BYTE mtr, short long tgt, WORD v, WORD a)
{
    CMD = 'Y';
    ARG[0] = mtr;
    ARG[1] = (tgt & 0xFF0000)>>16;
    ARG[2] = (tgt & 0xFF00)>>8;
    ARG[3] = tgt & 0xFF;
    ARG[4] = (BYTE) (v>>8);
    ARG[5] = (BYTE) (v & 0xFF);
    ARG[6] = (BYTE) (a>>8);
    ARG[7] = (BYTE) (a & 0xFF);
    ARGcount = 0x08;
    TeCmdDispatch();
}
//-----
```

## 16.1 IOEXP Functions

The board has two I/O expanders (**Device: MCP23017**) on the primary serial I2C bus (400 kHz) with each part providing 16 GPIO's routed to a ribbon cable connector. None of these GPIO's are currently used by any of the board firmware (they are all available for custom use). The functions described in this section operate at the device level, so you do not need to know anything about the I2C bus to access the parts. If you need to know more about the device registers than what is presented below, refer to the device data sheet on the Microchip web site. For convenience, a copy of the data sheet is included on the CD that ships with the product.

Reg#	Name	Port A Registers
0x00	IODIRA	Data Direction ('0'=Output, '1'=Input)
0x02	IPOLA	Input Polarity ('0'=non-inverted, '1'=inverted)
0x04	GPINTENA	GPIO Interrupt Enable ('0'=disable, '1'=enable)
0x06	DEFVALA	Default compare value for interrupts
0x08	INTCONA	Interrupt Control ('0'=use pin compares, '1'=use DEFVALA compares)
0x0A	IOCON	I/O Configuration
0x0C	GPPUA	GPIO Pullup Enable ('0'=disable, '1'=enable)
0x0E	INTFA	Interrupt Flags ('1'=request pending) ( <b>Read Only</b> )
0x10	INTCAPA	Interrupt Capture ( <b>Read Only</b> )
0x12	GPIOA	GPIO Port
0x14	OLATA	Output Latch

Reg#	Name	Port B Registers
0x01	IODIRB	Data Direction ('0'=Output, '1'=Input)
0x03	IPOLB	Input Polarity ('0'=non-inverted, '1'=inverted)
0x05	GPINTENB	GPIO Interrupt Enable ('0'=disable, '1'=enable)
0x07	DEFVALB	Default compare value for interrupts
0x09	INTCONB	Interrupt Control ('0'=use pin compares, '1'=use DEFVALA compares)
0x0B	IOCON	I/O Configuration
0x0D	GPPUB	GPIO Pullup Enable ('0'=disable, '1'=enable)
0x0F	INTFB	Interrupt Flags ('1'=request pending) ( <b>Read Only</b> )
0x11	INTCAPB	Interrupt Capture ( <b>Read Only</b> )
0x13	GPIOB	GPIO Port
0x15	OLATB	Output Latch

The 16 GPIO's on each part are implemented as two 8-bit ports: GPIOA[7..0] and GPIOB[7..0]. Each of the 16 pins can source or sink up to 25 mA (but the part is limited to 200 mA total). The configuration registers on the part are organized as 22 byte wide registers (21 if you consider that IOCON is cloned for use by both ports). The part offers two options for a register numbering scheme, but only the one indicated in the above table is supported by the library (BANK=0; See the device data sheet for details). The tables below list the registers and a brief description. IOCON (at 0x0A and 0x0B) is actually a single register serving both PORTA and PORTB.

### Notes:

- Because the interrupt on change feature of the parts is not supported, you should be able to use the default power up configuration for many of these registers. There are a couple of reasons for not supporting the interrupt on change feature. First, its use requires resources (eg; interrupt lines to the micro). More importantly, clearing the interrupt flags in an interrupt service routine requires I2C device access which introduces interrupt latency issues.

- Depending on your usage, you may need to reconfigure any of **IODIR**, **IOPOL**, **IOCON**, and perhaps **GPPU**. It is important that **IOCON.7=’0’** (**BANK=0**) be maintained to preserve the “interleaved” register addressing scheme described in the tables.
- Routine I/O applications, after configuration, will access only the **GPIO** and perhaps the **OLAT** register.
- Function execution times were measured with fOSC=40 MHz, I2C Baud Rate=400 kHz.
- See the section describing the I/O Expanders for the connector pin outs.

### 16.1.1 WriteIOExp1, WriteIOExp2

Function: **Write I/O Expanders.**

Include: dalf.lib

Prototypes: **void** WriteIOExp1(**BYTE** reg, **BYTE** data)  
**void** WriteIOExp2(**BYTE** reg, **BYTE** data)

Arguments: reg: Destination for write (0x00 - 0x15)  
data: Byte to write.

Return Value: None

File Name: ioexp.asm

Execution Time: 91 microseconds

**Code Example:** **WriteIOExp1( 0x13, 0x7F);**

This writes 0x7F to the GPIOB register.

If, for example, the part was previously configured with GPIOB pins as all outputs, this example would result in driving GPIOB[6..0] outputs high and GPIOB[7] low. Note that a read of the same register reads the pin states and could produce something different than 0x7F. A write to the OLAT register will also drive the port output pins.

### 16.1.2 ReadIOExp1, ReadIOExp2

Function: **Read I/O Expanders.**

Include: dalf.lib

Prototypes: **BYTE** ReadIOExp1(**BYTE** reg)  
**BYTE** ReadIOExp2(**BYTE** reg)

Arguments: reg: Source for read (0x00 - 0x15)

Return Value: RtnVal: Byte value read from reg address.

File Name: ioexp.asm

Execution Time: 124 microseconds

**Code Example:** **val = ReadIOExp1( 0x12);**

This example returns val = GPIOA from IO Expander 1.

## 16.2 Delay and Timing Functions

TMR1 is a 16-bit counter clocked by an external watch crystal (32.768 kHz). Its main role is to maintain a RTC. The functions in this section also use the TMR1 resource to provide accurately timed delays without software loops or a special interrupt. Here is a description of how this works:

The **ticks** on TMR1 occur at a rate of  $1/32768 \text{ Hz} = 30.5 \text{ microseconds}$ . In order to maintain the RTC, TMR1 is preloaded with a value of 0x8000 to generate an interrupt every 1 second. It is the TMR1 interrupt handler firmware that maintains the RTC by updating the supporting HH/MM/SS registers. During the same interrupt, a separate DWORD (32-bit) counter **seconds** is also incremented. This register is unrelated to the RTC system and is cleared only on power-up. It continues to record elapsed seconds since the last power up. Working together, the elapsed ticks (TMR1 - 0x8000) from TMR1, and the value of the seconds register provide a “software watch” with a resolution of 30.5 uSecs and a range of  $2^{32} \times 1\text{sec} = 136 \text{ years!}$

The current value of the “software watch” can be recorded in a **TIME** data structure consisting of two fields: a DWORD for **seconds** and a WORD for **ticks**.

```
typedef struct
{
    ULONG secs;
    WORD ticks;
} TIME, *PTIME;
```

The “software watch”, and the TIME structure to record it, are the resources used to support the timed delays for the functions described in this section. Briefly, the function SetDelay() uses the function GetTime() to record the current value of the watch, and with a bit of arithmetic, determine a future value for the watch based on the desired delay. The function TimeOut() is used to periodically poll the current value of the watch to determine if the delay has passed. It does this by comparing the future value with the current value of the watch.

### 16.2.1 GetTime

Function: **Get Current Time**

Include: dalf.lib

Prototypes: **void** GetTime(**PTIME** pTime)

Arguments: pTime: Data structure of type PTIME for recording current time.

Return Value: TIME structure pointed to by pTime will now have the current time.

File Name: delay.asm

Execution Time: 5 microseconds

Remarks: You probably won't need to call GetTime() but it is required by the other timing functions and it is useful to know how it works.

**Code Example:**

```
TIME Now;  
GetTime(&Now);
```

### 16.2.2 SetDelay

Function: **Establish Future Time (Now + Delay).**

Include: dalf.lib

Prototypes: **void SetDelay(PTIME pTime, ULONG DelaySecs, WORD DelayTicks)**

Arguments: pTime: Data structure that will be filled with the target time (Now + Delay).  
DelaySecs: Component of desired delay in **seconds**.  
DelayTicks: Component of desired delay in **ticks**.

Return Value: pTime: Future time corresponding to the desired delay (referenced by TimeOut()).

File Name: delay.asm

Execution Time: 12 microseconds

Remarks: The example uses both SetDelay() and TimeOut() to generate a square wave output on Port D.4 with frequency 1.0 Hz and duty cycle 10%. TimeOut() is called continuously until the delay has expired. In this example, nothing is done while waiting for the timeout to occur. Less trivial usages would have the software do something useful while periodically polling TimeOut() to check if the delay has expired.

**Code Example:**

```
#define msec_100      3277 // ticks worth 100 mSec
#define msec_900      29491 // ticks worth 900 mSec
TIME Delay;
while(1) // Sq wave on PORT D.4 with 10% (= 0.1/1.0) duty cycle
{
    LATD |= 0x10;                      // D.4 = 1
    SetDelay(&Delay, 0, msec_100);       // tON: 0.100s
    while( !Timeout(&Delay) );
    LATD &= ~0x10;                     // D.4 = 0
    SetDelay(&Delay, 1, msec_900);       // tOFF: 0.900s
    while( !Timeout(&Delay) );
}
```

### 16.2.3 TimeOut

Function: **Determine if Specified Delay Has Expired.**

Include: dalf.lib

Prototypes: **int Timeout(PTIME pTime);**

Arguments: pTime: Data structure that was previously filled (by SetDelay()) with the target time.

Return Value: RnVal: FALSE=0, TRUE= 1 (TRUE if current time > pTime).

File Name: delay.asm

Execution Time: 13 microseconds

Remarks: The example uses TimeOut() to turn off a motor after 1 hour of use.

**Code Example:**

```
TIME MotorOnDelay;  
MotorOn();  
SetDelay(&MotorOnDelay, 3600,0);  
while(1)  
{  
    .. Stuff..  
    if (TimeOut(&MotorOnDelay))  MotorOff();  
    ..Stuff..  
}
```

### 16.3 External EEPROM Functions

The functions in this section provide access to the external, 64K byte, serial EEPROM device (**24LC512**) on the primary I2C bus (400KHz). The functions operate at the device level, so knowledge of the I2C bus is not required to use the device. There are functions which access a single EEPROM byte and functions that access a block of bytes. Accessing consecutive bytes as a block is much more efficient than accessing the same block of bytes individually. This is particularly true for write access!

The address range of the part is [0x0000 - 0xFFFF], and it is organized as 512 “pages” of 128 (0x80) bytes per page. A block write can be used to write up to one page of data with a single function call, but the data block that is written must not cross a page boundary on the part. For example; It is ok to write 0x43 bytes starting at address 0x2030, but it would be a mistake to attempt to write the same 0x43 bytes starting at address 0x2040 (the data would cross the page boundary at 0x2080). Block reads are limited to 128 bytes (by the function design in this case, not the device), but there are no issues with page boundaries. For both block reads and writes you should avoid crossing the end of the physical device.

#### 16.3.1 WriteExtEE\_Byte

Function: **Write Byte to External EEPROM**

Include: dalf.lib

Prototypes: **void** WriteExtEE\_Byte(**WORD** address, **BYTE** dat);

Arguments: address: (WORD) location in the EEPROM to write the data byte.  
dat: Data byte to record in EEPROM.

Return Value: None.

File Name: eeprom.asm

Execution Time: 120 microseconds + 5msec

Remarks: The part requires a 5 msec write cycle time at the end of the data transfer before it can be accessed again. To avoid reliability issues, this function does not return until the required delay has expired.

**Code Example:**

```
WriteExtEE_Byte(0x0017, 0x23);
```

### **16.3.2 ReadExtEE\_Byte**

Function: **Read Byte from External EEPROM**

Include: dalf.lib

Prototypes: **BYTE ReadExtEE\_BYTE(WORD address);**

Arguments: address: (WORD) location in EEPROM to read.

Return Value: RtnVal: Byte that was read from the EEPROM.

File Name: eeprom.asm

Execution Time: 150 microseconds

Remarks: ReadExtEE\_BYTE is much faster than WriteExtEE\_BYTE because the 5 msec write cycle time is not present.

**Code Example:**

```
val = ReadExtEE_BYTE(0x0017);
```

### **16.3.3 WriteExtEE\_Block**

Function: **Write Block to External EEPROM**

Include: dalf.lib

Prototypes: **void WriteExtEE\_Block(WORD address, BYTE \*buff, BYTE n);**

Arguments: address: (WORD) location in EEPROM for start of write.  
buff: RAM Buffer that is the source for the block transfer.  
n: Number of bytes in block (1 <= N <= 0x80).

Return Value: None.

File Name: eeprom.asm

Execution Time: n \* (35 microseconds) + 5 msec

Remarks: If n>128, only the first 128 bytes will be written. The 5 msec cycle time is still required for block writes, but there is only one delay for the entire block. The example copies the first 0x30 bytes from the RAM buffer mybuff[] into the eeprom at locations 0x0300 thru 0x032F. As mentioned previously, avoid page boundary crossings on the eeprom.

**Code Example:**

```
WriteExtEE_Block(0x0300, &mybuff, 0x30);
```

#### **16.3.4 ReadExtEE\_Block**

Function:     Read Block from External EEPROM

Include:     dalf.lib

Prototypes:   **void**    ReadExtEE\_Block(**WORD** address, **BYTE** \*buff, **BYTE** n);

Arguments:    address: (WORD) location in EEPROM for start of read.  
                buff: RAM Buffer that is the destination for the block transfer.  
                n: Number of bytes in block (1 <= N <= 0x80).

Return Value:    BUFF[] will have been filled with block from EEPROM.

File Name:     eprom.asm

Execution Time: n \*39 microseconds

Remarks:       If n>128, only the first 128 bytes will be read. The example copies 0x30 bytes from the EEPROM starting at address 0x0000 into the RAM buffer mybuff[].

**Code Example:**

```
ReadExtEE_Block(0x0000, &mybuff, 0x30);
```

### **16.4 Internal EEPROM Functions**

The 1k Byte internal EEPROM has a maximum write cycle delay of 4 msec after each byte write. There is no significant timing advantage for block writes (repeated byte writes). The firmware polls for write completion and a typical write cycle delay is 2.7 msec. Read operations do not have the delay and are quite fast.

#### **16.4.1 WriteIntEE\_Byte**

Function:     **Write Byte to Internal EEPROM**

Include:     dalf.lib

Prototypes:   **void**    WriteIntEE\_Byte(**WORD** address, **BYTE** dat);

Arguments:    address: (WORD) location in the EEPROM to write the data byte.  
                dat: Data byte to record in EEPROM.

Return Value:   None.

File Name:     eprom.asm

Execution Time: Typical 2.7 msec (Max 4.0 msec).

Remarks:       The part requires the write cycle delay at the end of each byte write before it can be accessed again. To avoid reliability issues, this function does not return until the required delay has expired.

**Code Example:**

```
WriteIntEE_Byte(0x0017, 0x23);
```

### **16.4.2 ReadIntEE\_Byte**

Function: **Read Byte from Internal EEPROM**

Include: dalf.lib

Prototypes: **BYTE ReadIntEE\_BYTE(WORD address);**

Arguments: address: (WORD) location in EEPROM to read.

Return Value: RtnVal: Byte that was read from the EEPROM.

File Name: eeprom.asm

Execution Time: 2.4 microseconds

Remarks: ReadIntEE\_BYTE is much faster than WriteIntEE\_BYTE because the write cycle time is not present.

**Code Example:**

```
val = ReadIntEE_BYTE(0x0017);
```

### **16.4.3 WriteIntEE\_Block**

Function: **Write Block to Internal EEPROM**

Include: dalf.lib

Prototypes: **void WriteIntEE\_Block(WORD address, BYTE \*buff, BYTE n);**

Arguments: address: (WORD) location in EEPROM for start of write.  
buff: RAM Buffer that is the destination for the block transfer.  
n: Number of bytes in block (1 <= N <= 0x80).

Return Value: None.

File Name: eeprom.asm

Execution Time: Typical: n \* (2.7 msec); Max: n \* (4.0 msec)

Remarks: If n>128, only the first 128 bytes will be written. Unlike the external EEPROM, there is no speed advantage for block writes. The internal device does not support block writes, so each byte that is written exhibits the write cycle timing delay. The example copies the first 0x30 bytes from the RAM buffer mybuff[] into the Internal EEPROM at locations 0x0300 thru 0x032F.

**Code Example:**

```
WriteIntEE_Block(0x0300, &mybuff, 0x30);
```

#### **16.4.4 ReadIntEE\_Block**

Function:     Read Block from Internal EEPROM

Include:      dalf.lib

Prototypes:   **void**    ReadIntEE\_Block(**WORD** address, **BYTE** \*buff, **BYTE** n);

Arguments:     address: (WORD) location in EEPROM for start of read.  
                buff: RAM Buffer that is the destination for the block transfer.  
                n: Number of bytes in block (1 <= N <= 0x80).

Return Value:   BUFF[] will have been filled with block from EEPROM.

File Name:     eprom.asm

Execution Time: n \* 2.4 microseconds

Remarks:       If n>128, only the first 128 bytes will be read. The example copies 0x30 bytes from the EEPROM starting at address 0x0000 into the RAM buffer mybuff[].

**Code Example:**

```
ReadIntEE_Block(0x0000, &mybuff, 0x30);
```

## 16.5 Digital Pot Functions

There are two dual digital pot parts (POT1 and POT2) on the board. Each part has two digitally controlled, 50k Ohm, pots (potA and potB) for a total of 4 pots. Each pot has 256 tap positions that can be assigned to the wiper. The board ties the end of all 4 pots to VDD and GND, with the wiper positions routed to the 4 test pads (V1H, V1L, V2H, V2L) - See schematic. The parts have both a volatile register (VREG) and a non-volatile register (NVREG) associated with each pot and the write operations described here can initiate copies between these two registers, or a write to either register. Any operation that affects the value of the volatile register VREG, also records that value as the tap position for the wiper (altering the output voltage at one of the 4 test pads). When the part receives power, the value of the NVREG associated with each pot is moved into the wiper, but the Dalf System Initialization code quickly replaces this using the values stored in the Parameter Block. The parts are write-only.

### 16.5.1 WritePot1

Function: **Write operation to Pot1**

Include: dalf.lib

Prototypes: **void WritePot1(BYTE cmd, BYTE data);**

Arguments: cmd: One of 12 arguments specifying desired operation.  
data: If required by operation, tap position for pot wiper.

Return Value: None.

File Name: digpot.asm

Execution Time: 91 microseconds

Remarks: Here is the list of valid commands

cmd	Operation	Description
<b>(Writes to volatile register ; update wiper)</b>		
0x11	VREGA	Write to VREGA and update wiperA
0x12	VRGB	Write to VRGB and update wiperB
0x13	VREG	Write to both VREGA and VRGB and update both wipers
<b>(Writes to non-volatile register)</b>		
0x21	NVREGA	Write to NVREGA.
0x22	NVRGB	Write to NVRGB.
0x23	NVREG	Write to both NVREGA and NVRGB.
<b>(Copy NVREG to VREG; update wiper)</b>		
0x61	NVA -to- VA	Copy NVREGA to VREGA; update wiper.
0x62	NVB -to- VB	Copy NVRGB to VRGB; update wiper.
0x63	NV -to- V	Copy: NVREGA to VREGA; NVRGB to VRGB; update wipers.
<b>(Copy VREG to NVREG)</b>		
0x51	VA -to- NVA	Copy VREGA to NVREGA.
0x52	VB -to- NVB	Copy VRGB to NVRGB.
0x53	V -to- NV	Copy: VREGA to NVREGA; VRGB to NVRGB.

**Code Example:**

```
WritePot1(0x11, 0xA0);  
WritePot1(0x12, 0x60);
```

The schematic shows that wiper for potA on the POT1 device is routed to the V1H test pad. Similarly the wiper for potB on that device is routed to the V1L test pad. This example alters the V1H and V1L test pad voltages as follows:

$$V1H = (0xA0/256)*5V = (160/256)*5V = 3.13V$$
$$V1L = (0x60/256)*5V = 96/256*5V = 1.88V$$

### 16.5.2 WritePot2

Function: **Write operation to Pot2**

(See “WritePot1”. All of the content for WritePot1 applies here as well, but applied to POT2 device)

## 17 Potential Future Enhancements

This list contains features that might be provided at a later date. They are listed in no particular order, and it is always possible that some, or all, will not be provided (by me). I consider the GUI to be the most important, and the most likely to appear as an enhancement at some time in the future.

- 0) **Windows GUI** using API. Currently under development with expected release of a beta quality version soon.
- 1) **Temperature Sensing:** Currently there is fan control, but no temperature firmware. Implement either NTC thermistor (using ADC) or I2C sensor (on primary I2C bus) and tie fan action or graceful shutdown to over-temperature.
- 2) **Home.** A homing feature using limit switches would be pretty easy.
- 3) **USART2** Interrupt Handler and verification.

## 18 Appendix A - Step Response Examples

This section has two graphs showing typical motor step responses obtained with the “PID Step Response” Command (CMD\_Q) when applied to an application using a 12V PM, geared, wheel chair motor. The first graph shows the response with no contribution from the sum or difference terms of the PID Equation ( $K_P=0x0A00$ ,  $K_I=0x0000$ ,  $K_D=0x0000$ ). The second shows the response from the same motor with a bit of damping added ( $K_P=0x0A00$ ,  $K_I=0x0000$ ,  $K_D=0x0C00$ ). The graphs were obtained by capturing the terminal data to a file (logging enabled) and then importing the file into an Excel Spreadsheet. When the Windows GUI becomes available the step response graph will be provided as part of the application.

The plots show “-Err” vs “Time”. Look at the actual data as you visualize the plots. Things to observe are the slope (the difference in the Err terms sample-to-sample) and the behavior as the Err terms get close to zero. With a bit of practice, you should be able to visualize the important aspects of the graph by looking at the data alone, and you will be able to skip the graphing step when using Cmd\_Q.

### What is the goal?

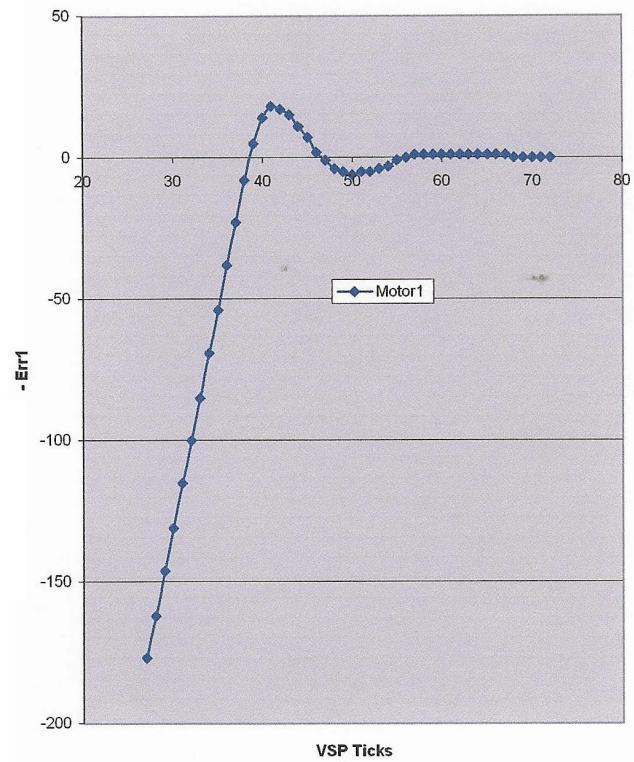
The goal is a set of PID parameters that will rapidly drive your motor to the step destination with little or no overshoot and with the error rapidly settling at or near zero without continuing oscillations.

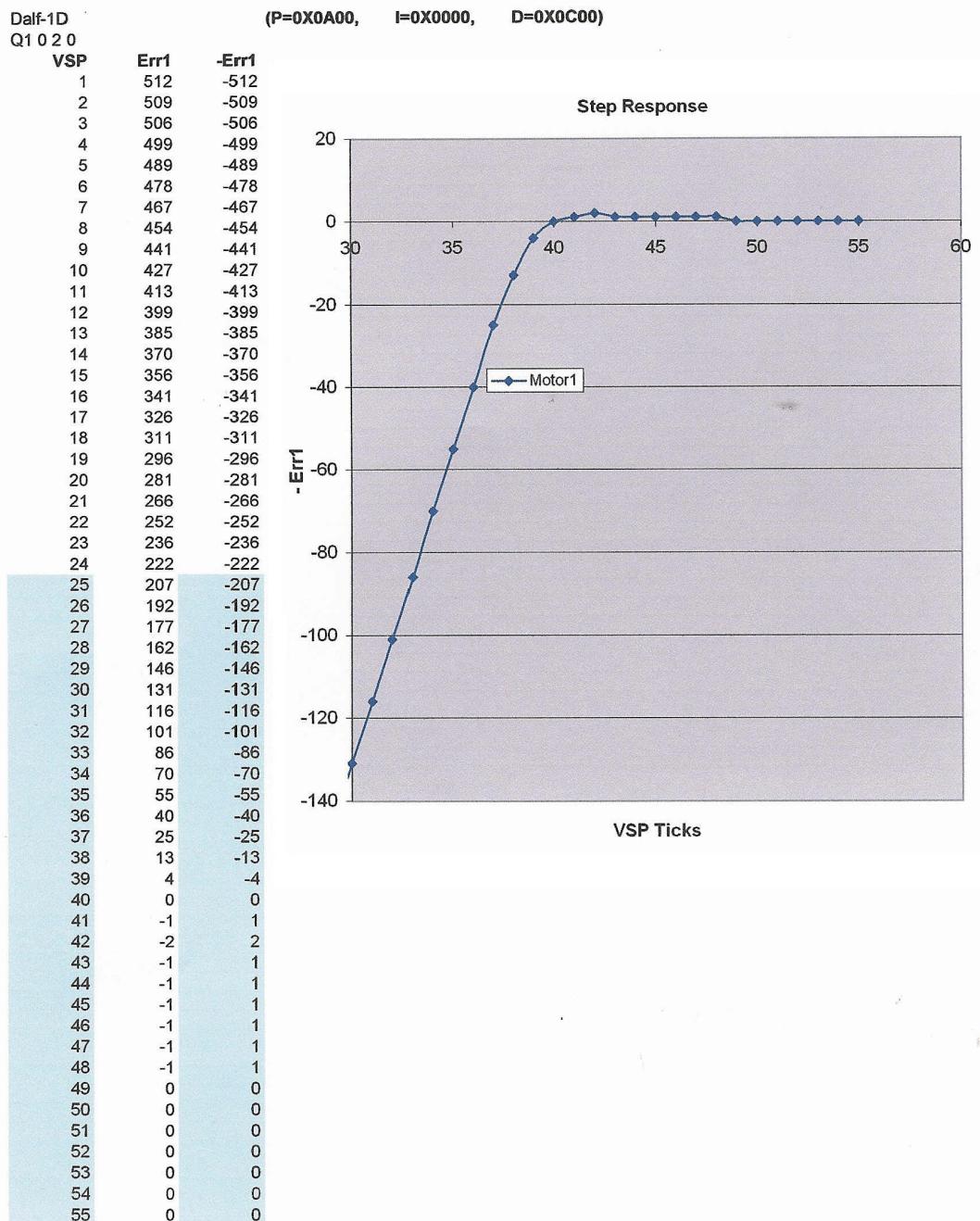
Obviously the second graph is much better than the first in meeting the goal. Can you tell this by looking at the data alone? If so, you are ready for Cmd\_Q!

Dalf-1D (P=0XA00, I=0X0000, D=0X0000)  
Q1 0 2 0

VSP	Err1	-Err1
1	512	-512
2	513	-513
3	509	-509
4	502	-502
5	493	-493
6	481	-481
7	469	-469
8	457	-457
9	443	-443
10	429	-429
11	415	-415
12	401	-401
13	387	-387
14	372	-372
15	357	-357
16	343	-343
17	328	-328
18	313	-313
19	298	-298
20	283	-283
21	268	-268
22	253	-253
23	237	-237
24	223	-223
25	207	-207
26	192	-192
27	177	-177
28	162	-162
29	146	-146
30	131	-131
31	115	-115
32	100	-100
33	85	-85
34	69	-69
35	54	-54
36	38	-38
37	23	-23
38	8	-8
39	-5	5
40	-14	14
41	-18	18
42	-17	17
43	-15	15
44	-11	11
45	-7	7
46	-2	2
47	1	-1
48	4	-4
49	5	-5
50	6	-6
51	5	-5
52	5	-5
53	4	-4
54	3	-3
55	1	-1
56	0	0
57	-1	1
58	-1	1
59	-1	1
60	-1	1
61	-1	1
62	-1	1
63	-1	1
64	-1	1
65	-1	1
66	-1	1
67	-1	1
68	0	0
69	0	0
70	0	0
71	0	0
72	0	0

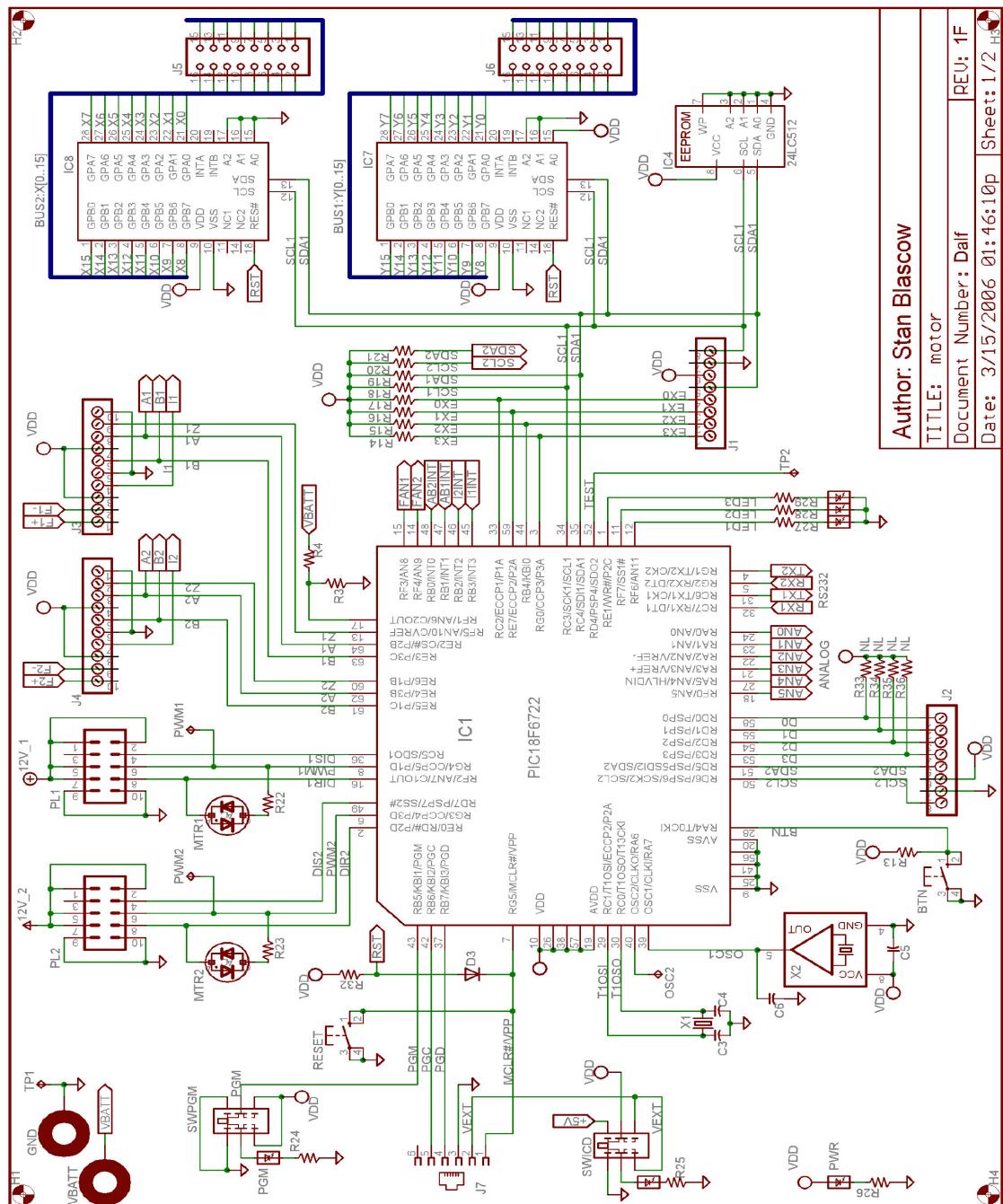
Step Response

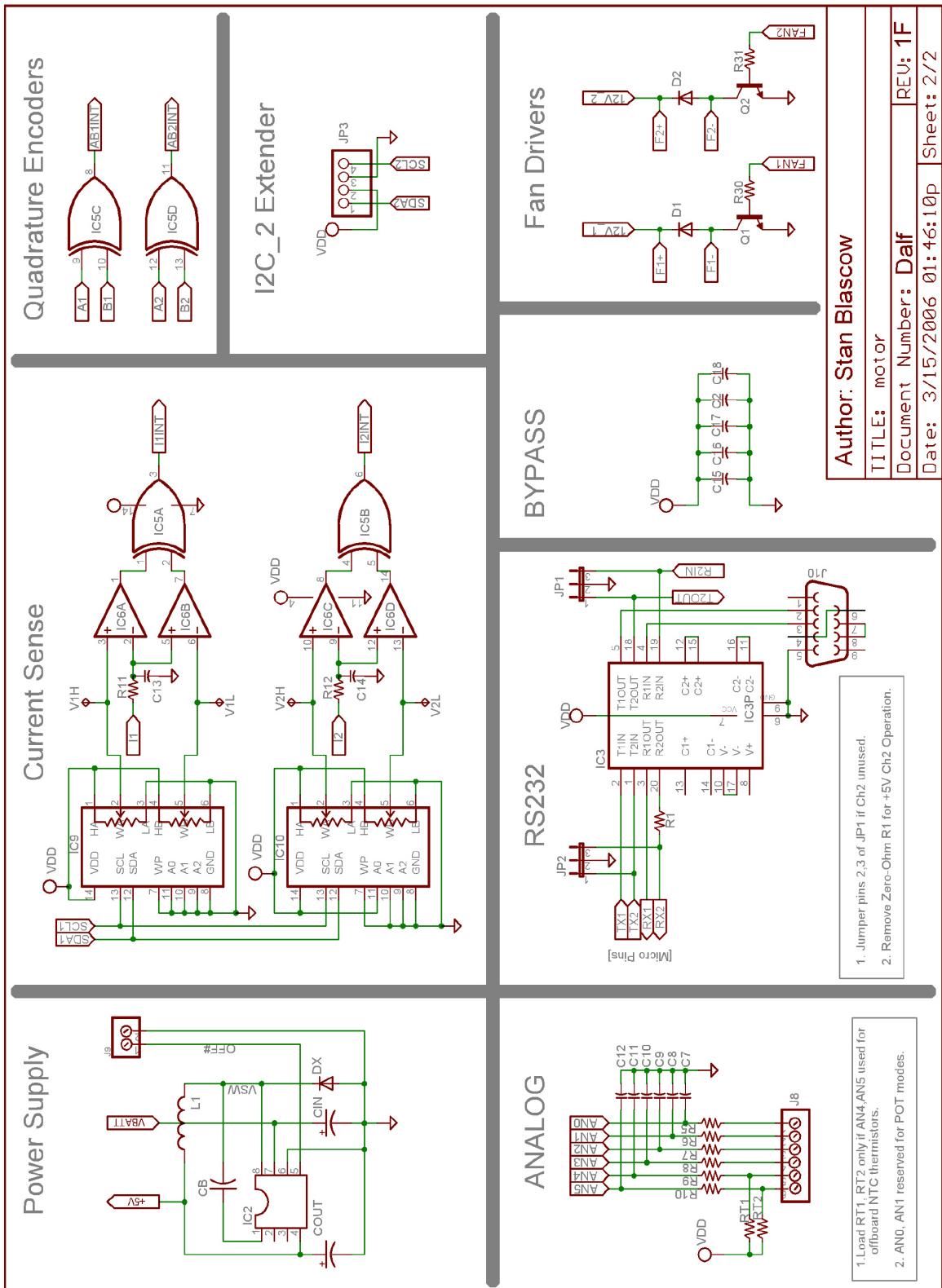


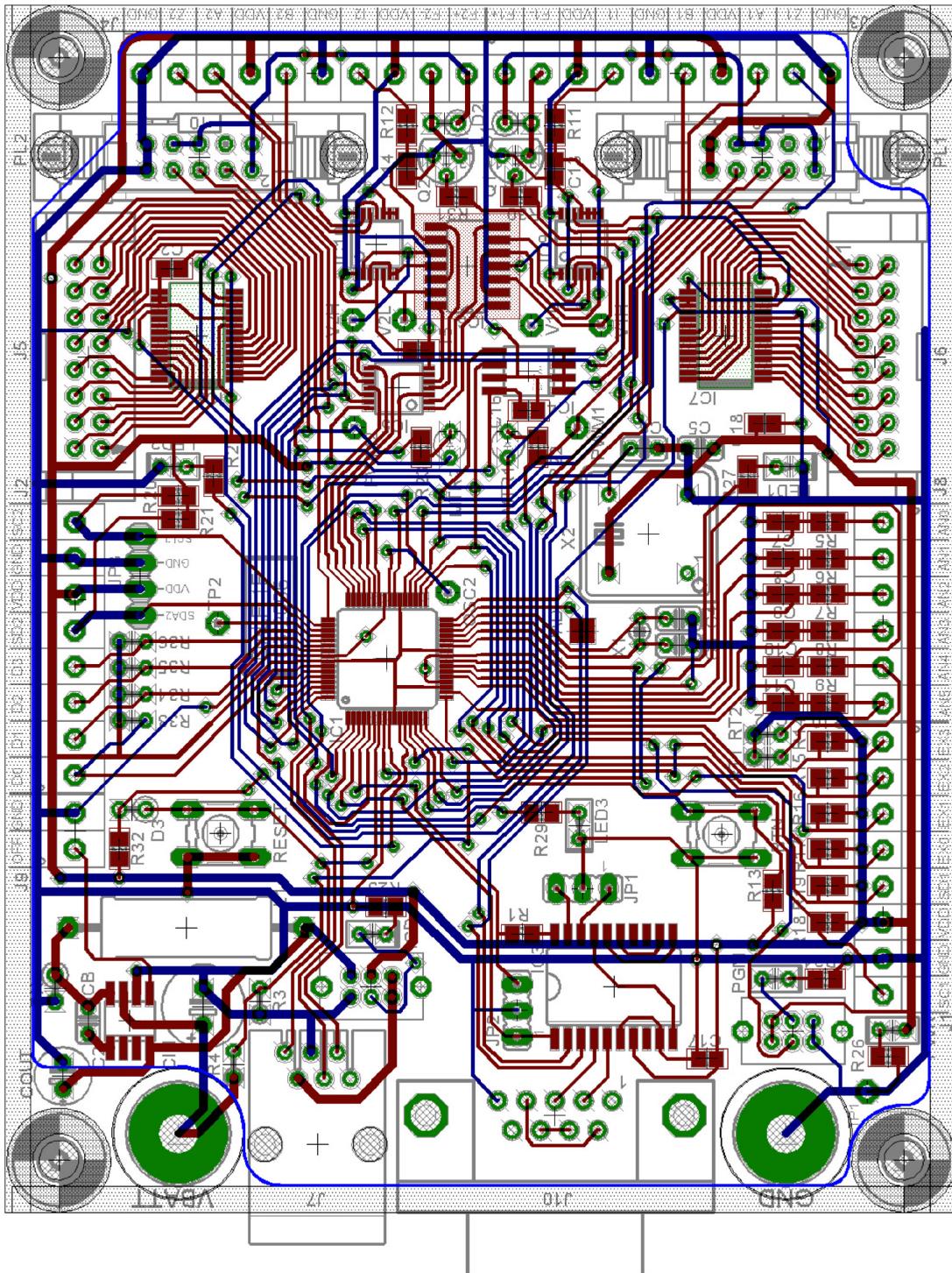


## **19 Appendix B - Board Schematic/Layout**

This section contains schematics and board layout.







## 20 Appendix C - Board Parts List

#	Part Name	Manuf	Manuf#	Description	Supplier	Part#
1	X1	Citizen America Corp	CFS206-KDZF-UB ECS-2200BX-400	Crystal, 32.768kHz, Cyl, 12.5pF, 2x6mm	Digi-Key	300-8303-ND
1	X2	ECS Inc		Osc, ECS-2200BX-400, 40MHz, ½ Size	Digi-Key	XC1191-ND
2	D1,D2	Diodes Inc	UF1002-T	Diode Ultra Fst Sw 100V 1A D0-41	Digi-Key	UF1002DICT-ND
2	D3,DX	International Rectifier	11DQ06	Diode, Schottky, 60V, 1.1A, D0-41	Digi-Key	11DQ06-ND
1	LED3	Lumex	SSL-LX2573ID	LED – Radial, 2x5mm, rect, RED	Digi-Key	67-1047-ND
3	LED1,LED2, PWR	Lumex	SSL-LX2573GD	LED – Radial, 2x5mm, rect, GRN	Digi-Key	67-1046-ND
2	PGM, ICD	Lumex	SSL-LX2573YD	LED - Radial, 2x5mm, rect, YEL	Digi-Key	67-1050-ND
2	MTR1, MTR2	LITE-ON Inc	LTL-14CHJ	LED – Radial, 3mm, 2.2V,BiColor Red/Grn	Digi-Key	160-1058-ND
1	IC1	Microchip	PIC18F6722-I/PT	IC Microcontroller – TQFP64 ; 10x10 mm	Digi-Key	PIC18F6722-I/PT-ND
1	IC2	National Semi	LM2672M-5.0	IC Switching Regulator, SO_08M; 5V,1A	Digi-Key	LM2672M-5.0-ND
1	IC3	Maxim	MAX233ACWP	IC RS232 Dual 5V Dvr/Rcvr, 20-SOIC	Digi-Key	MAX233ACWP-ND
1	IC4	Microchip	24LC512T-I/SM	IC Serial EEPROM, 64KByte, 8-SOIC	Microchip	24LC512-I/SM
1	IC5	TI	SN74LAHCT86PWR	IC Quad 2-In, XOR; 14-TSSOP	Digi-Key	296-4778-1-ND
1	IC6	Maxim	MAX944CSD	IC Quad Comparator, 14-SOIC	Digi-Key	MAX944CSD-ND
2	IC7, IC8	Microchip	MCP23017	I2C Serial I/O Expander, 28 pin SSOP	Digi-Key	MCP23017-E/SS-ND
2	IC9, IC10	Maxim	MAX5478EUD	Dual digital pot, 50K, I2C, NV, 14 TSSOP	Maxim	MAX5478EUD
2	Q1,Q2	On Semi	P2N2222A	2N2222A Transistors in T092 package	Digi-Key	P2N2222AOS-ND
2	PGM, ICD	E-Switch	EG2207	Switch; Slide DPDT, 0.2A, L=4mm	Digi-Key	EG1940-ND
2	BTN, RESET	Omron	B3F-1052	Switch Tact, 6mm, Mom, 150gf	Digi-Key	SW405-ND
2	J3,J4	On Shore Tech	ED555/10DS	Term Blk –10 Pin,3.5mm, 28#-16#,125V,6A	Digi-Key	ED1522-ND
2	J1,J2	On Shore Tech	ED555/8DS	Term Blk –8 Pin,3.5mm, 28#-16#,125V,6A	Digi-Key	ED1520-ND
1	J9	On Shore Tech	ED555/2DS	Term Blk –2 Pin, 3.5mm, 28#-16#,125V,6A	Digi-Key	ED1514-ND
1	J8	On Shore Tech	ED555/6DS	Term Blk - 6 Pin, 3.5mm, 28#-16#, 125V,6A	Digi-Key	ED1518-ND
2	J5, J6	Amp/Tyco	104313-3	Conn, LoPro Header, 16pos(2x8) + latches	Digi-Key	A26318-ND
2	J5,J6	Amp/Tyco	746285-3	Conn, IDC Socket, 16 pos(2x8), with polarity	Digi-Key	AKC16G-ND
1	J7	Amp/Tyco	5520470-3	Conn Modular Jack PCB 6 Pin 50 Au, IDE	Digi-Key	A31417-ND
1	J10	Amp/Tyco	7478444-4	Conn D-SUB Recept, Right Angle 9Pos, PCB	Allied Elec	512-0476
2	PL1,PL2	3M/ESD	3793-6002	Conn Protect Header, 10 pin Gold, Vert	Digi-Key	MHS10K-ND
2	PL1,PL2	3M/ESD	3473-6600	Conn IDC Socket, 10 Pos, w/pol, gold.	Digi-Key	MKC10K-ND
2	PL1,PL2	3M/ESD	3505-2	Conn Ejector Latches - header (short) - 2/pkg	Digi-Key	MLKS01-ND
1	JP1	Amp/Tyco	4-102976-0	Conn Header, 3 pin Vert, 0.1, [Use: 3 of 40]	Digi-Key	A26518-ND
1	JP1	Sullins Elec	SSC02SYAN	Conn - 2 Pin Jumper, Shorting, Gold	Digi-Key	S9002-ND
1	JP2 (NL)	Amp/Tyco	4-102976-0	Conn Header, 3 pin Vert, 0.1, [Use: 3 of 40]	Digi-Key	A26518-ND

1	R3	Yageo	CFR-12JB-330R	Resistor - 1/6 watt, 330 Ohms, 0204V	Digi-Key	330EBK-ND
1	R4	Yageo	CFR-12JB-2K2	Resistor - 1/6 watt, 2.2K Ohms, 0204V	Digi-Key	2.2KEBK-ND
2	RT1, RT2 (NL)	Yageo	CFR-12JB-10K	Resistor - 1/6 watt, 10K Ohms, 0204V	Digi-Key	10KEBK-ND
4	R33-R36 (NL)	Yageo	CFR-12JB-10K	Resistor - 1/6 watt, 10K Ohms, 0204V	Digi-Key	10KEBK-ND
1	R1	Yageo	9C08052A0R00JLHFT	Resistor - 0 Ohms, 0805, 1/8W, SMD, 5%	Digi-Key	311-0.0ACT-ND
8	R22 - R29	Yageo	9C08052A3300JLHFT	Resistor - 330 Ohms, 0805, 1/8W, SMD, 5%	Digi-Key	311-330ACT-ND
2	R30, R31	Yageo	9C08052A5600JLHFT	Resistor - 560 Ohms, 0805, 1/8W, SMD, 5%	Digi-Key	311-560ACT-ND
1	R13	Yageo	9C08052A1001JLHFT	Resistor - 1K Ohms, 0805, 1/8W, SMD, 5%	Digi-Key	311-1.0KACT-ND
4	R18 - R21	Yageo	9C08052A2701JLHFT	Resistor - 2.7K Ohms, 0805, 1/8W, SMD, 5%	Digi-Key	311-2.7KACT-ND
8	R5 - R12	Yageo	9C08052A4701JLHFT	Resistor - 4.7K Ohms, 0805, 1/8W, SMD, 5%	Digi-Key	311-4.7KACT-ND
5	R14 - R17,R32	Yageo	9C08052A1002JLHFT	Resistor - 10K Ohms, 0805, 1/8W, SMD, 5%	Digi-Key	311-10KACT-ND
2	Cb, C5	Kemet	C315C103K5R5CA	Cap, Ceramic - 0.01uF, 50V, 10%, Radial	Digi-Key	399-2004-ND
2	C3,C4	EPCOS Inc	B37979N1120J000	Cap, Ceramic - 12pF, 100V, 5%, Radial	Digi-Key	P4838-ND
1	C6	EPCOS Inc	B37979N1150J000	Cap, Ceramic - 15pF, 100V, 5%, Radial	Digi-Key	P4839-ND
13	C7 - C18, C2	Yageo	CC0805KRX7R9BB104	Cap, Ceramic - 0.1uF, 0805	Digi-Key	311-1140-1-ND
1	COut	Panasonic	EEU-FC1A221	Cap - 220uF, 10V, Electrolytic, Radial, FC	Digi-Key	P11183-ND
1	CIn	Panasonic	EEU-FC1J680	Cap - 68uF, 63V, Electrolytic, Radial, FC	Digi-Key	P10341-ND
1	L1	API Delavan	2474-21L	Inductor, 47uH, Power, Axial	Digi-Key	DN7421-ND

## 21 Appendix D - Parameter Block Detail

The first 128 bytes of the 64K Byte External EEPROM part is the **Parameter Block**. Multi-byte values are stored in Little Endian Format. The Parameter Block is copied into RAM during power up configuration. The RAM copy is referred to as the **Runtime Environment** or alternately **ERAM** (EEPROM RAM Copy). ERAM begins at offset 0x100 in RAM (The fPWM parameter is at address 0x0000 in the EEPROM but its RAM copy is at address 0x0100). The parameters and their usage are described in the section that follows the table. **Unless you make changes to one of these areas after power up, the content of ERAM and that of the Parameter Block will always be identical. A reasonable way to think of these memory areas is that ERAM is the current working copy of the Parameter Block.**

Some values in ERAM are used only to initialize the hardware during power up configuration. An example is the fPWM parameter. If this value is changed subsequent to power up by simply writing a new value into ERAM, it will not effect a change in the PWM frequency. Other variables are used by the firmware after power-up and changes to the RAM based copy will have an immediate affect on run time behavior (an example would be the PID Variables). Parameter Block locations (0x0078 - 0x007F) and the remainder of the EEPROM address space (0x0080 – 0xFFFF) are currently unused. For future firmware updates I expect that the general structure, size and location of the Parameter Block in EEPROM to remain unchanged. The location of the RAM copy of the table (ERAM) at offset 0x100 is also unlikely to change. **The contents of the Parameter Block and usage of additional EEPROM are subject to change in future firmware updates.**

### 21.1 Parameter Block Table

Address	Value	Size	Name	Description
0x0000	0x0D	BYTE	fPWM	Selects PWM Frequency (default: 20,000 Hz)
0x0001	0x01	BYTE	AD_ACQ	ADC Acquisition Timing Control (default: 20 uSec)
0x0002	0x02	BYTE	AD_CNV	ADC Conversion Timing Control (default: 35 uSec)
0x0003	0x09	BYTE	AD_GAP	ADC Channel Gap Timing Control (default: 500 uSec)
0x0004	0x00	BYTE	SYSMODE	System Control Flags
				---- I/O Expanders ----
0x0005	0xFF	BYTE	X1_IODIRA	I/O Direction, Port A (All Inputs)
0x0006	0x00	BYTE	X1_IPOLA	Input Polarity, Port A (All non-inverted)
0x0007	0x00	BYTE	X1_GPINTENA	Interrupt Enable, Port A (All disabled)
0x0008	0x00	BYTE	X1_DEFVALA	Default Compare Value, Port A (All low)
0x0009	0x00	BYTE	X1_INTCONA	Interrupt Control , Port A (Compare basis: pins)
0x000A	0x00	BYTE	X1_IOCNA	I/O Configure, Port A (Port addresses interleaved)
0x000B	0x00	BYTE	X1_GPPUA	Pull up Enable, Port A (All pull ups disabled)
0x000C	0x00	BYTE	X1_GPIOA	GPIO Register, Port A
0x000D	0x00	BYTE	X1_OLATA	Output Latch, Port A
0x000E	0xFF	BYTE	X1_IODIRB	I/O Direction, Port B (All Inputs)
0x000F	0x00	BYTE	X1_IPOLB	Input Polarity, Port B (All non-inverted)
0x0010	0x00	BYTE	X1_GPINTENB	Interrupt Enable, Port B (All disabled)
0x0011	0x00	BYTE	X1_DEFVALB	Default Compare Value, Port B (All low)
0x0012	0x00	BYTE	X1_INTCONB	Interrupt Control , Port B (Compare basis: pins)
0x0013	0x00	BYTE	X1_GPPUB	Pull up Enable, Port B (All pull ups disabled)
0x0014	0x00	BYTE	X1_GPIOB	GPIO Register, Port B
0x0015	0x00	BYTE	X1_OLATB	Output Latch, Port B

### PARAMETER BLOCK (Cont'd)

Address	Value	Size	Name	Description
0x0016	0xFF	BYTE	X2_IODIRA	I/O Direction, Port A (All Inputs)
0x0017	0x00	BYTE	X2_IPOLA	Input Polarity, Port A (All non-inverted)
0x0018	0x00	BYTE	X2_GPINTENA	Interrupt Enable, Port A (All disabled)
0x0019	0x00	BYTE	X2_DEFVALA	Default Compare Value, Port A (All low)
0x001A	0x00	BYTE	X2_INTCONA	Interrupt Control , Port A (Compare basis: pins)
0x001B	0x00	BYTE	X2_IOCON	I/O Configure, Port A (Port address interleaved)
0x001C	0x00	BYTE	X2_GPPUA	Pull up Enable, Port A (All pull ups disabled)
0x001D	0x00	BYTE	X2_GPIOA	GPIO Register, Port A
0x001E	0x00	BYTE	X2_OLATA	Output Latch, Port A
0x001F	0xFF	BYTE	X2_IODIRB	I/O Direction, Port B (All Inputs)
0x0020	0x00	BYTE	X2_IPOLB	Input Polarity, Port B (All non-inverted)
0x0021	0x00	BYTE	X2_GPINTENB	Interrupt Enable, Port B (All disabled)
0x0022	0x00	BYTE	X2_DEFVALB	Default Compare Value, Port B (All low)
0x0023	0x00	BYTE	X2_INTCONB	Interrupt Control , Port B (Compare basis: pins)
0x0024	0x00	BYTE	X2_GPPUB	Pull up Enable, Port B (All pull ups disabled)
0x0025	0x00	BYTE	X2_GPIOB	GPIO Register, Port B
0x0026	0x00	BYTE	X2_OLATB	Output Latch, Port B
				---- VBATT ----
0x0027	0x07AA	WORD	VBCAL	VBATT calibration parameter
0x0029	0x3E80	WORD	VBWARN	VBATT low battery warning level (0x3E80 = 16,000 mV)
				---- MTR COMMON ----
0x002B	0x0A	BYTE	AMINP	Acceleration Minimum Period (10 ms)
0x002C	0x2000	WORD	MAXERR	PID Err Limit (Ticks)
0x002E	0x0100	WORD	MAXSUM	PID ErrSum Limit (Ticks)
				---- MTR1----
0x0030	0x12	BYTE	MTR1_MODE1	Mtr1 - Mode1 Control
0x0031	0x00	BYTE	MTR1_MODE2	Mtr1 - Mode2 Control
0x0032	0x29	BYTE	MTR1_MODE3	Mtr1 - Mode3 Control
0x0033	0x0003	WORD	ACC1	Acc default (Ticks/VSP^2); Closed Loop
0x0035	0x0400	WORD	VMID1	Velocity Midcourse, Motor1 (Ticks/VSP)
0x0037	0x05	BYTE	VSP1	Velocity Sampling Period, Motor1 (ms)
0x0038	0x0260	WORD	KP1	PID Proportional Constant; Motor1
0x003A	0x0000	WORD	KI1	PID Integral Constant; Motor1
0x003C	0x0600	WORD	KD1	PID Derivative Constant; Motor1
0x003E	0x02	BYTE	VMIN1	PID Minimum PWM Duty Cycle (%)
0x003F	0x64	BYTE	VMAX1	Maximum PWM Duty Cycle (%)
0x0040	0x0100	WORD	TPR1	Encoder Motor1 Ticks-Per-Revolution (=4*CPR)
0x0042	0x0000	WORD	MIN1	Servo1 MIN limit
0x0044	0x00FF	WORD	MAX1	Servo1 MAX limit

**PARAMETER BLOCK (Cont'd)**

Address	Value	Size	Name	Description		
			<b>---- MTR2 ----</b>			
0x0046	0x10	BYTE	MTR2_MODE1	Mtr2 - Mode1 Control		
0x0047	0x00	BYTE	MTR2_MODE2	Mtr2 - Mode2 Control		
0x0048	0x29	BYTE	MTR2_MODE3	Mtr2 - Mode3 Control		
0x0049	0x0003	WORD	ACC2	Acceleration, Motor2 (Ticks/VSP^2)		
0x004B	0x0400	WORD	VMID2	Velocity Midcourse, Motor2 (Ticks/VSP)		
0x004D	0x05	BYTE	VSP2	Velocity Sampling Period, Motor2; (ms)		
0x004E	0x0260	WORD	KP2	PID Proportional Constant; Motor2		
0x0050	0x0000	WORD	KI2	PID Integral Constant; Motor2		
0x0052	0x0600	WORD	KD2	PID Derivative Constant; Motor2		
0x0054	0x02	BYTE	VMIN2	PID Minimum PWM Duty Cycle (%)		
0x0055	0x64	BYTE	VMAX2	Maximum PWM Duty Cycle (%)		
0x0056	0x0100	WORD	TPR2	Encoder Motor2 Ticks-Per-Revolution (=4*CPR)		
0x0058	0x0000	WORD	MIN1	Servo1 MIN limit		
0x005A	0x00FF	WORD	MAX1	Servo1 MAX limit		
			<b>---- R/C ----</b>			
0x005C	0x03E8	WORD	RC1MIN	RC Channel 1; Pulse Width Minimum (uS)		
0x005E	0x07D0	WORD	RC1MAX	RC Channel 1; Pulse Width Maximum (uS)		
0x0060	0x03E8	WORD	RC2MIN	RC Channel 2; Pulse Width Minimum (uS)		
0x0062	0x07D0	WORD	RC2MAX	RC Channel 2; Pulse Width Maximum (uS)		
0x0064	0x03E8	WORD	RC3MIN	RC Channel 3; Pulse Width Minimum (uS)		
0x0066	0x07D0	WORD	RC3MAX	RC Channel 3; Pulse Width Maximum (uS)		
0x0068	0x1E	BYTE	RCD	RC DeadBand (uS)		
			<b>---- DIGITAL POTS ---</b>			
0x0069	0x90	BYTE	POT1A	Pot 1, Wiper A tap (V1H)		
0x006A	0x70	BYTE	POT1B	Pot 1, Wiper B tap (V1L)		
0x006B	0x90	BYTE	POT2A	Pot 2, Wiper A tap (V2H)		
0x006C	0x70	BYTE	POT2B	Pot 2, Wiper B tap (V2L)		
			<b>--- COMMUNICATION ---</b>			
0x006D	0x04	BYTE	nBR	USART1 Baud Rate Index (19,200)		
0x006E	0x01	BYTE	NID	Network ID of this board (1-254)		
0x006F	0x64	BYTE	RX1TO	Rx1 Time Out (ms) [counter reset value]		
0x0070	0x0100	WORD	NPID	PID Tuning; 256 Err outputs when tuning motor		
0x0072	0x60	BYTE	DALFA	Dalf slave address on secondary I2C bus		
			<b>--- MISC ---</b>			
0x0073	0x32	BYTE	RCSP	RC Sampling Period (ms)		
0x0074	0x32	BYTE	PSP	Pot Mode Sampling Period (ms)		
0x0075	0x10	BYTE	DMAX	Max  Diff  =  LastA - A  (analog feedback usage)		
			<b>--- FILTER ---</b>			
0x0076	0x40	BYTE	FENBL	Filter Enable Bits {AN6 ... AN0}		
0x0077	0x88	BYTE	DECAY	Filter decay constant: d=DECAY/256=0.53		
0x0078	0x14	BYTE	CMDSP	Cmd Interface Sampling Period for Timeout (msec)		
0x0079	0xFF	BYTE	CMDTIME	Cmd Interface Timeout (CMDSP msec units)		
0x007A	0xFF	BYTE	///UNUSED///	//////////UNUSED/////////		
.....	.....	.....	.....	<b>--- UNUSED ---</b>		
0x007F	0xFF	BYTE	///UNUSED///	//////////UNUSED/////////		

## **21.2 Parameter Block Recovery**

Changes made to the Parameter Block could potentially prevent successful power-up. If you have this problem, simply hold down the push-button switch (BTN) during power-up until you see the greeting in the terminal emulator window. This will restore all of the factory default values to the Parameter Block (losing any customizations). It is a good idea to keep a hardcopy of any changes that you make to the Parameter Block. In the unlikely event that you have to use this feature to restore the factory default values, you will be able to restore your changes using the hardcopy as a guide.

## **21.3 Parameter Descriptions**

### **0x0000: fPWM**

This parameter is used as an index into a frequency table to initialize the hardware registers that control PWM frequency. See the “PWM Freq Control” command for details.

### **0x0001 - 0x0003: AD\_ACQ, AD\_CNV, AD\_GAP**

These three parameters are used as indices into a lookup table to determine timing for automatic acquisition and storage of the seven analog input (including VBATT) signals. See the section “TMR2 - ADC State Machine” for details. Here is the timing table showing the index values and the corresponding Delays.

ADC TIMING TABLE			
Index	Delay (uSec)	T2CON	PR2
0x00	15	0x04	150
0x01	20	0x04	200
0x02	35	0x0C	175
0x03	50	0x0C	250
0x04	75	0x14	250
0x05	100	0x24	200
0x06	200	0x3C	250
0x07	300	0x5C	250
0x08	400	0x7C	250
0x09	500	0x25	250
0x0A	750	0x3D	234
0x0B	1,000	0x4D	250
0x0C	2,000	0x27	250
0x0D	3,000	0x3F	234
0x0E	4,000	0x4F	250
0x0F	5,000	0x67	240
0x10	6,000	0x77	250
0x11	6,500	0x7F	254

The value in the Index column is used to access values shown in the last two columns. When the Timer2 control registers (T2CON, PR2) are initialized as shown, the desired interrupt timing (Delay) is achieved before the next TMR2 Interrupt occurs.

#### **0x0004: SYSMODE**

The bits in this byte control general system, non motor dependent, features.

SYSMODE								
Idle	cmdto	////////	////////	////////	////////	////////	////////	////////
7	6	5	4	3	2	1	0	

**idle:** This bit enables the low power IDLE Mode. When this feature is enabled, and the board is not busy, the code will enter a low power mode in which the peripherals (timers, ADC, etc) remain clocked, but the CPU is halted (waiting for activity). Testing with this feature enabled using VBATT=+12V demonstrated a power savings of roughly 13% (Dalf Board only) without noticeable system degradation. Generally in a motor control application, the power needs of the drivers and the motors will make this savings insignificant.

**cmdto:** This bit when set enables the Serial Cmd Interface Timeout feature. See the section in this manual and the I2C2 and API documents for details.

#### **Expanded I/O Control Parameters**

##### **0x0005 - 0x0015: X1 Controls**

These parameters control initial power up configuration of the writeable registers on the IO Expander #1 part. See the section “IOEXP Functions” for details.

##### **0x0016 -0x0026: X2 Controls**

These parameters control initial power up configuration of the writeable registers on the IO Expander #2 part. See the section “IOEXP Functions” for details.

#### **Battery Related Parameters**

##### **0x0027 - 0x0028: VBCAL** (VBATT Measurement calibration parameter; scaled by 256)

VBCAL is the battery voltage calibration parameter to deal with resistor tolerances on the VBATT voltage divider. In the event that your measurement reads a bit low or a bit high, you can adjust the measurement by changing this parameter.

Let  $r = R3/(R3+R4) = 330/(330+2200) = 0.130435$  be the VBATT voltage divider constant. Let  $V6$  denote the voltage applied to the AN6 input pin on the microcontroller from this divider. **It is important that  $V6 \leq +5V$  to prevent damage to the ADC Module on the microcontroller (see the note below).**

The relationship between VBATT,  $V6$  and the calibration constant VBCAL is given by:

$$\boxed{\text{VBATT} = (1/r) * V6 = (\text{VBCAL}/256) * V6}$$

The VBCAL parameter represents the scaled nominal value of  $(1/r)$ . It is useful to compensate for possibly inaccurate (low precision) resistors R3 and R4. The equation illustrates how VBCAL, and the measured voltage  $V6$  are used to report the VBATT voltage. The default for VBCAL is  $(1/r)*256 = 7.666 * 256 = 1962 = 0x07AA$ . If the reported voltage is too low, adjust the VBCAL parameter upwards. If the reported voltage is too high, adjust the parameter downward.

Recommended Adjustment Procedure:

1. Apply a regulated power supply adjusted to provide your nominal battery voltage to the VBATT and GND terminals. Measure VBATT with a reliable meter.
2. Using Cmd\_C with a terminal emulator, obtain the board measured value for VBATT.

3. Change the ERAM copy of VBCAL as required, and repeat step 2. When you feel that your value of VBCAL is good enough, record the new VBCAL into the Parameter Block, reset the board, and retest.

**NOTE: R3 and R4**

The choice made for the values for R3 and R4 was designed to protect the ADC module in the worst case of VBATT=+40V, but this limits the measurement range (hence the resolution) of the ADC in monitoring your power supply if you will always be using VBATT<40V. **If you are absolutely certain that you will always be using a smaller input voltage than +40V, and you are handy with a soldering iron, you can increase the resolution of the battery measurement system by replacing the thru hole parts R3 and R4.** For example; If you can be certain that VBATT will never exceed +20V, values for R3 and R4 that would yield  $r \leq 0.25$  (eg; R3=470, R4=1500) would be appropriate. This choice would roughly double the resolution that you could obtain with the original R3 and R4 parts. Of course if you do replace R3 and R4 you will also want to change the value for VBCAL to match your new resistor selections.

**0x0029 - 0x002A: VBWARN (Low Batt Warning Level (mV))**

VBWARN is a low battery warning voltage threshold level used by the system to monitor VBATT. In normal operation, once every second, the VBATT value (see VBCAL for discussion of how this is computed) is compared with VBWARN. If VBATT<VBWARN, the low battery warning condition is shown on the error LED. For a nominal 18V battery, a reasonable VBWARN value might be 16000mV (=0x3E80). Set this parameter according to your nominal battery voltage and your desired warning level. Note that the firmware supplies hysteresis for LED turn off to avoid flickering for VBATT levels near the VBWARN threshold.

**Parameters Common to Both Motors**

**0x002B: AMINP (Acceleration Minimum Period (ms))**

This parameter is designed to protect against rapid changes in motor direction and/or abrupt stops which might result in drive train damage. This value is used by the firmware for all open loop motor control commands in a fashion similar to that of a governor on the throttle of car. In this case, the application is to limit the acceleration (slew rate) and not the velocity. The units of AMINP are milliseconds and represent **the minimal timing for a 1% change in the PWM duty cycle** (velocity control). A one-percent change in this context means 1% of the full speed range, not a 1% change to the current speed. For example, a value of 0x0A here limits the velocity change to 1% every 10 msec's even if your motor command requested an immediate stop. . Some open loop commands use AMINP as the default value for the slew rate input if it is not supplied in the command.

It is helpful to understand how AMINP is used. Every AMINP msec, the duty cycle for the motor is either; incremented (speedup), left alone (no change), or decremented (slowdown), until the desired target speed is achieved. For example, if the motor is going at full speed (100% duty cycle), and AMINP=0x0A, a full stop would take a minimum of  $100 * 10 \text{ msec} = 1 \text{ sec}$ . The AMINP constant applies to both motors. If you set AMINP to 0, and you request an immediate stop (drive train ouch!), that is exactly what you will get. AMINP is a factor in all open loop motor operations, and for any error condition which justifies stopping the motor.

**Note: During typical PID controlled movements, all changes to the PWM motor speed control are immediate (not restricted by AMINP). It is the role of the Trajectory Generator to provide smooth motor transitions during closed loop operations.**

**Note:** If you will be using open loop pot (or R/C) control methods be sure that AMINP < PSP (or AMINP < RCSP). Consider pot motor control operation with AMINP=50, and PSP=40. In this case, motor speed control commands are being issued every 40 msec based on the pot settings, but the value used

for AMINP limits responses to a 1% change every 50 ms! In this case, changes are being requested much faster than the acceleration governor will allow - the result: no changes occur.

#### **0x002C - 0x002D: MAXERR**

This 16 bit integer is used by the PID control firmware to detect an error condition (for example a stalled motor). If the position error during any sampling period of a PID controlled movement exceeds this value the motor will be ramped to a stop and PID will be disabled. Note that this constant applies to both motors.

#### **0x002E - 0x002F: MAXSUM**

This 16 bit integer is used by the PID control firmware to bound (by clipping to +/- MAXSUM) the integral sum of position errors, which contributes, as one term, to the PID control computation for motor PWM. Note that this is not considered an error condition (values for the integral term whose magnitude exceed MAXSUM will not trigger a motor stop). See the PID and Trajectory Generator sections of this document for additional details.

## Motor1 Parameters

The individual bits in the MODE bytes control numerous board features. There are 3 bytes for each motor.

### **0x0030: MTR1\_MODE1**

The MODE1 parameter contains bits that control PID, Trajectory Generator, Encoder signal decoding, DIS signal driver polarity, and Fan operation.

MTR1_MODE1							
	fan	analogfb	sumho	rel	trig	aleadsb	dis_activelo
7	6	5	4	3	2	1	0

**fan:** This bit controls the fan1 on/off status on power-up. 0=Off, 1=On.

**analogfb:** This bit should be set only if an analog encoder is being used for the motor. See the section devoted to analog feedback for details. See also the analogdir bit.

**sumho:** PID position error summation hold-off. If set, this bit will cause the integral portion of the PID term during motor movements to be held to zero until the trajectory generator has generated the final waypoint at the destination (the motor position should be close to the destination at this point). This is a mechanism to deal with a common problem with PID control called “Integral Windup” in which early accumulation of errors lead to instability. See the discussion in the PID and Trajectory Generator sections of this document.

**rel:** The state of this flag governs how the argument specifying the move destination in the closed loop motor control commands is treated. If this flag is set, then the input destination is treated as an offset relative to the current motor position. For example, if the argument is [000100] and the current encoder position is [0x00297F], the command interprets the argument as a request to move the motor to position [0x002A7F]. This mode is useful in situations where you might want the motor to go forward or reverse a given incremental distance regardless of current location. If this flag is clear, the same command would request a move to the position [0x000100].

**trig:** This flag is the **Closed Loop Trigger Control Mode Flag**. When set, this bit will cause closed loop motor control commands to be delayed, waiting for a trigger that is issued with the “CLOSED LOOP TRIGGER” command. The purpose of this flag is to allow synchronized movement of the motors. When in this mode, you can issue “MOVE MOTOR” commands to each motor and then simultaneously start both motor actions by using the TRIGGER command.

**aleadsb:** This bit should be set if the encoder signal A leads signal B (by 90 degrees) for forward motion on the motor. If instead, B leads A, the bit should be cleared (\*).

**dis\_activelo:** This bit controls the polarity of the DIS motor control signal. If your motor driver expects the DIS signal to be asserted LO (that is; motor disabled whenever DIS = 0 Volts), this bit should be set. Otherwise (DIS asserted HI; motor disabled whenever DIS = +5V), it should be clear.

(\*) Your encoder spec will provide information on this but it can be confusing. The reason is that the correct setting also depends on encoder orientation (left or right shaft mounting). Here is a simple procedure to make the correct setting: After power up, rotate the motor manually in the forward direction while monitoring motor position using the serial port (Cmd\_E). If the count advances, you have it correct. Otherwise, change the “aleadsb” setting and retest. **Repeat for both motors!**

## **0x0031: MTR1\_MODE2**

The MODE2 parameter contains bits that affect operating and interface control modes.

MTR1_MODE2							
potc	potf	rcnrm	rcmix	rcsvo	potsvo	potmix	/////////
7	6	5	4	3	2	1	0

- potc:** This bit selects the “**Pot Center Off**” motor control mode for the motor. In this mode, an external POT and an On/Off switch control motor operation. The center position of the pot corresponds to motor off. Excursions to either side of center increase motor speed in either the reverse or forward direction (#). The pot control is disabled if the On/Off switch is in the off position. See the section “**POT Mode Interface**” for details.
- potf:** This bit selects the “**Pot Full Scale**” motor control mode for the motor. In this mode, an external pot and two switches control motor operation. The full scale range of the pot controls motor speed and a separate reversing switch controls direction. The pot control is disabled if the On/Off switch is in the off position. See the section labeled “**POT Mode Interface**” for details.
- rcnrm:** This bit selects the **R/C Normal** (sometimes referred to as “tank mode”) motor control mode. If this bit is set in the Mtr1 parameter, R/C channel 1 (EX0) controls motor 1. If the bit is set in the Mtr2 parameter, R/C channel 2 (EX1) controls motor 2. See the section “**R/C MODE INTERFACE**” for details.
- rcmix:** This bit selects the **R/C Mix** motor control mode. In this mode both R/C channels 1 and 2 affect both motors thru mixing. See the section “**R/C MODE INTERFACE**” for details. To utilize the R/C Mix feature, this bit will need to be set for both motor 1 and motor 2.
- rcsvo:** This bit, when set, enables servo control of the motor using the associated R/C channel as the position control input. See the section “**Servo Modes**” for details.
- potsvo:** This bit, when set, enables servo control of the motor using the associated external pot as the position control input. See the section “**Servo Modes**” for details.
- potmix:** This bit, when set for both motors, enables “**POT Mix**” speed control of the motors using the associated external pot (generally a 2-axis analog joystick) as the control input. See the section “**POT Mode Interface**” for details.

```
#####
## NOTE: AT MOST ONE OF THE OPERATING MODE BITS (“potc”, “potf”, “rcnrm”,    ##
## “rcmix”, “rcsvo”, “potsvo”, and “potmix”) SHOULD BE SET.                      ##
#####
```

## 0x0032: MTR1\_MODE3

The MODE3 parameter contains bits that affect motor driver setup, over current detection and response, and output to the terminal emulator window.

MTR1_MODE3							
	analogdir	osmc	ocie	oc_fastoff	vel_d	pos_d	verbose
7	6	5	4	3	2	1	0

**analogdir:** This bit should only be set if an analog encoder is being used for the motor and the encoder mounting orientation is such that the sensor output indicates reverse movement when the motor movement is forward. See the section devoted to analog feedback for details. See also the analogfb bit.

**osmc:** This bit is the **Open Source Motor Control** bit. When set, will result in the firmware complementing the PWM duty cycle for reverse motor operation. This is consistent with expectations of the HIP4081A chip on the **OSMC** Speed Control Boards utilizing Sign Magnitude PWM for control of a brushed DC motor.

**Non OSMC motor drivers may require this bit to be “0” (the default for the bit is “1”).**

**ocie:** This bit is the **Over Current Interrupt Enable** bit for the motor. When set, it will cause the firmware to enable the interrupt used to detect transitions out of the adjustable voltage window controlled by the on-board digital pots. It should be set to ‘0’ unless you are using an off-board current sensor connected to the current sense screw terminal connectors.

**oc\_fastoff:** This bit, when set, will cause immediate power removal and the disabling of the motor control interface when an over current condition is detected. **The setting of this bit is only important if the ocie bit is set to ‘1’ (over current detection enabled).** The alternative response (when the bit is clear) is SlowRun. See the section “Current Sensor Feed Back” for the details.

**vel\_d:** This bit, when set, will cause motor encoder velocity outputs to be shown in decimal instead of the default hex. This feature applies to the TE command mode only.

**pos\_d:** This bit, when set, will cause motor encoder position outputs to be shown in decimal instead of the default hex. This feature applies to the TE command mode only.

**verbose:** This bit, when set, enables outputs of the PID Err term on the COMM port during execution of the “PID Step Response” Command. This can be very useful for initial PID Tuning of motor parameters. See the cmd “PID Step Response” for details.

**0x0033 - 0x0034: ACC1** (Acceleration default, Motor 1 (Ticks/VSP<sup>2</sup>))

This parameter is used as the acceleration default for Motor 1 Closed Loop commands that don't specify the desired acceleration/deceleration. Notice that the units are in terms of the Velocity Sampling Period for Motor 1. This parameter is also scaled by a factor of 256. See cmd "**Motor1 Move (Closed-Loop Control)**" for additional detail.

**0x0035 - 0x0036: VMID1** (Midcourse Velocity default, Motor 1 (Ticks/VSP))

This parameter is used as the velocity for Motor 1 Closed Loop commands that don't specify the desired midcourse velocity. Notice that the units are in terms of the Velocity Sampling Period for Motor 1. This parameter is also scaled by a factor of 256. See cmd "**Motor1 Move (Closed-Loop Control)**" for additional detail.

**0x0037: VSP1** (Velocity Sampling Period, Motor 1 (msec))

This value determines the frequency at which the motor is sampled for velocity computations. Use the CPR value for your encoder and the speed range at which you plan to drive the motor to select an appropriate value for VSP. Don't just automatically set this to the smallest value (1ms). If the value selected is too small, you will not get sufficient resolution in the number of ticks per sampling period at slow speeds. For example, if the "**Get Motor Velocity**" command returns 1 tick per sampling period (which could happen for even reasonable velocities if the sampling period is too small) then there could easily be a 1 tick error (50% error!) for reported velocity. Even more important here is that small values for VSP represent increased overhead on the firmware. The range on the VSP value is 1-255 msec.

**Note:** VSP1 is an important parameter for PID control of Motor 1. See the PID and Trajectory Generator section of this document for details.

**0x0038 - 0x0039: KP1** (PID Proportional Constant, Motor 1)

This value is the Proportional Constant for the "**Err**" term in the PID computation for Motor 1. See the PID and Trajectory Generator sections of this document for details.

**0x003A - 0x003B: KI1** (PID Integral Constant, Motor1)

This value is the Integral Constant for the "**Err**" term in the PID computation for Motor 1. See the PID and Trajectory Generator sections of this document for details.

**0x003C - 0x003D: KD1** (PID Derivative Constant, Motor 1)

This value is the Derivative Constant for the "**Err**" term in the PID computation for Motor 1. See the PID and Trajectory Generator sections of this document for details.

**0x003E: VMIN1** (Minimum PWM, Motor1 (duty cycle %))

This value is used in the PID Control Method for Motor 1 as the lower limit for the computed control value (duty cycle %). Even after target acquisition, there will be at least this much power applied to the motor 1. It therefore functions as both a brake and also helps to overcome some of the initial friction on motor startup. This parameter does not apply to open loop motor commands.

**0x003F: VMAX1** (Maximum PWM, Motor1 (duty cycle %))

This value is used by all motor control commands (both open and closed loop) to limit the motor power. It essentially provides a speed governor on the motor. **It should never be larger than 100% (0x64)**, but in some cases there may be the need to restrict full power to motor 1. An example might be a case where you are driving 12V motors from a 36V source and you want to limit the duty cycle of the 36V PWM pulse train to avoid possible motor damage. Another application of its use might be the case of a very fast motor with a high CPR encoder. By limiting the motor speed, you might be able to better control the motor.

**0x0040 - 0x0041: TPR1** (Ticks Per Revolution, Motor1)

This is a characteristic of the encoder attached to Motor 1. Typically, incremental encoders are specified in terms of “Counts Per Revolution” (CPR). The value of TPR1 reflects the quadrature counts from the encoder and should be made equal to 4\*CPR. Example; Using a US Digital Encoder with CPR=64 attached to Mtr1, the value of TPR1 should be set to 256 = 0x0100. If you are using an analog encoder, the value should be set to 256 = 0x100 to reflect the 8-bit resolution of the stored ADC samples.

**0x0042 - 0x0043: MIN1** (Servo Minimum Limit Position)

This parameter is only important if you will be using the motor in one of the servo modes. See the section “**Servo Modes**” for details.

**0x0044 - 0x0045: MAX1** (Servo Maximum Limit Position)

This parameter is only important if you will be using the motor in one of the servo modes. See the section “**Servo Modes**” for details.

**Motor2 Parameters**

<b>0x0046:</b>	<b>MTR2_MODE1</b> (Motor control flags)	--See MTR1_MODE1
<b>0x0047:</b>	<b>MTR2_MODE2</b> (Motor control flags)	--See MTR1_MODE2
<b>0x0048:</b>	<b>MTR2_MODE3</b> (Motor control flags)	--See MTR1_MODE3
<b>0x0049-0x004A:</b>	<b>ACC2</b> (Acceleration default, Motor 2)	--See ACC1
<b>0x004B-0x004C:</b>	<b>VMID2</b> (Midcourse Velocity default, Motor 2)	--See VMID1
<b>0x004D:</b>	<b>VSP2</b> (Velocity Sampling Period, Motor 2)	--See VSP1
<b>0x004E-0x004F:</b>	<b>KP2</b> (PID Proportional Constant, Motor 2)	--See KP1
<b>0x0050-0x0051:</b>	<b>KI2</b> (PID Integral Constant, Motor 2)	--See KI1
<b>0x0052-0x0053:</b>	<b>KD2</b> (PID Derivative Constant, Motor 2)	--See KD1
<b>0x0054:</b>	<b>VMIN2</b> (Minimum PWM, Motor 2 (duty cycle %))	--See VMIN1
<b>0x0055:</b>	<b>VMAX2</b> (Maximum PWM, Motor 2 (duty cycle %))	--See VMAX1
<b>0x0056-0x0057:</b>	<b>TPR2</b> (Ticks Per Revolution, Motor 2)	--See TPR1
<b>0x0058-0x0059:</b>	<b>MIN2</b> (Minimum Servo Limit)	--See MIN1
<b>0x005A-0x005B:</b>	<b>MAX2</b> (Maximum Servo Limit)	--See MAX1

**Open Loop R/C Parameters****0x005C - 0x005D: RC1MIN** (Minimum R/C Pulse Width, Ch 1, (uSec))

This 16-bit parameter should correspond to the output of the R/C Receiver when the R/C Transmitter Switch is at minimum. It is used to map the R/C pulse width on Channel 1 into a PWM response to drive the motor(s). See the section “**R/C Mode Interface**” for details.

**0x005E - 0x005F: RC1MAX** (Maximum R/C Pulse Width, Ch 1, (uSec))

This 16-bit parameter should correspond to the output of the R/C Receiver when the R/C Transmitter Switch is at maximum. It is used to map the R/C pulse width on Channel 1 into a PWM response to drive the motor(s). See the section “**R/C Mode Interface**” for details.

**0x0060 - 0x0061: RC2MIN** (Minimum R/C Pulse Width, Ch 2, (uSec))

This 16-bit parameter should correspond to the output of the R/C Receiver when the R/C Transmitter Switch is at minimum. It is used to map the R/C pulse width on Channel 2 into a PWM response to drive the motor(s). See the section “**R/C Mode Interface**” for details.

**0x0062 - 0x0063: RC2MAX** (Maximum R/C Pulse Width, Ch 2, (uSec))

This 16-bit parameter should correspond to the output of the R/C Receiver when the R/C Transmitter Switch is at maximum. It is used to map the R/C pulse width on Channel 2 into a PWM response to drive the motor(s). See the section “**R/C Mode Interface**” for details.

**0x0064 - 0x0065: RC3MIN** (Minimum R/C Pulse Width, Ch 3, (uSec))

This 16-bit parameter should correspond to the output of the R/C Receiver when the R/C Transmitter Switch is at minimum. Channel 3 is currently unused.

**0x0066 - 0x0067: RC3MAX** (Maximum R/C Pulse Width, Ch 3, (uSec))

This parameter should correspond to the output of the R/C Receiver when the R/C Transmitter Switch is at maximum. Channel 3 is currently unused.

**0x0068: RCD** (R/C DeadBand (uSec))

Assuming your switches are tuned to the RCMIN and RCMAX parameters the central position of your switch corresponds to a receiver output of RCmid = (RCMIN + RCMAX)/2. If d=RCD/2, then the firmware uses a PulseWidth-to-PWM mapping that maps all pulse widths in the interval [RCmid-d, RCmid+d] to a zero PWM. This provides an adjustable deadband around the central position of the R/C switches on your transmitter.

#### Current Limit Parameters

**0x0069 : POT1A**

Pot1; Wiper A tap position [0..255]. Voltage appears at V1H test point on the board.

**0x006A: POT1B**

Pot1; Wiper B tap position [0..255]. Voltage appears at V1L test point on the board.

**0x006B : POT2A**

Pot2; Wiper A tap position [0..255]. Voltage appears at V2H test point on the board.

**0x006C: POT2B**

Pot2; Wiper B tap position [0..255]. Voltage appears at V2L test point on the board.

#### Communication Parameters

**0x006D: nBR** (USART1 Baud Rate Index)

This parameter controls the baud rate used by the command interface.

Value	Baud Rate
0x00	300
0x01	1,200
0x02	2,400
0x03	9,600
0x04	19,200
0x05	57,600
0x06	115,200

**0x006E: NID**

Network ID of board [1..255]

**0x006F: RX1TO**

Rx1 serial port timeout (ms). This value is used to reset the timeout counter during serial port communications.

**0x0070 - 0x0071: NPID**

Number of PID Err outputs during use PID Tuning Command [0..65,535].

**0x0072: DALFA**

Dalf board address on the secondary I2C bus.

**Misc Other Parameters****0x0073: RCSP (R/C Sampling Period (msec))**

This parameter governs how frequently the firmware examines the latest captured receiver pulse widths for the purpose of issuing new motor control commands. It should probably be named the R/C command rate (R/C sampling actually takes place continuously and is interrupt driven). See also AMINP.

**0x0074: PSP (Pot Sampling Period (msec))**

This parameter governs how frequently the firmware examines the voltage measurements from the off board pots for the purpose of issuing new motor control commands. It should probably be named the POT command rate (A/D sampling is actually driven by a state machine with timing controlled by TMR2). See also AMINP.

**0x0075: DMAX (Max |Diff| = |LastA - A| [analog feedback algorithm] )**

See the section in this manual that discusses use of analog position encoders.

**Filter Parameters****0x0076: FENBL (Filter Enable For Analog Inputs)**

FENBL							
	AN6	AN5	AN4	AN3	AN2	AN1	AN0
7	6	5	4	3	2	1	0

The FENBL parameter contains bits that affect whether or not the digital recursive filter is applied to the analog input channels. If the bit is set, the corresponding analog input is filtered before being stored in memory. If the bit is clear, the digital filter is not applied and the raw reading is stored in memory. All channels that are enabled will use the same decay constant (see below).

**0x0077: DECAY (Filter Decay constant: d = DECAY/256)**

The DECAY parameter defines the characteristic of the low pass digital filter which can be employed with the use of the bits in FENBL. See the section in this manual that discusses the digital, recursive, filter for details.

**Serial Command Interface Timeout Parameters****0x0078: CMDSP (Command Interface Sampling Period)**

When enabled (See SYSMODE), this parameter governs how frequently the main loop checks the serial command interface for a possible timeout. The actual timeout is CMDSP \* CMDTIME (msec). See the section in this manual and the I2C2 and API documents for details.

**0x0079: CMDTIME (Initialization value for countdown timer for Cmd Interface Timeout Feature)**

When a valid serial command is received, the countdown timer is initialized with this value. The countdown timer decrements at the CMDSP rate. See the section in this manual and the I2C2 and API documents for details.

**0x007A - 0x007F: //Unused//**

## 22 Appendix E - Fixed Address RAM Variables

In general, don't expect any future code upgrades to respect variable locations, or even the continued usage of a variable, in RAM. There are a few exceptions to this. I intend to support fixed RAM locations for the following variables in all future revisions of the code. This allows a PC Application like a Windows GUI to access this data without requiring changes to the GUI. If it becomes absolutely necessary, even these locations could change, but at present I don't foresee any circumstance that would require that.

**Fixed RAM Table**

Address	Size	Name	Description (HEX unless otherwise indicated)
0x060	BYTE	ErrCode	Error Code
0x061	BYTE	LedErr	LED Error Code
0x062	BYTE	BOARD_ID	Board Version (ASCII Char)
0x063	BYTE	PIC_DEVID1	[ddd rrrrr]; rrrrr = PIC Chip Revision. ddd = D2 .. D0 bits of PIC Chip ID
0x064	BYTE	PIC_DEVID2	[ddddddd]; D10..D3 bits of PIC Chip ID
0x065	BYTE	MAJOR_ID	Major Software ID Ver#
0x066	BYTE	MINOR_ID	Minor Software ID Ver#
0x067-0x68	WORD	USER_ID	Board Serial#
0x069	BYTE	SCFG	Serial Port Config (0=TE, 1=API)

The PIC\_DEVID and USER\_ID variables are read from the PIC Configuration space and copied into RAM during system initialization. The Board ID and Software ID are sourced from #defines in the code and loaded into RAM during system initialization. The primary usage for these variables is to produce a greeting using the serial link to a PC Application. In the case of a Terminal Emulator link, the greeting will be in the terminal emulator window. A smart PC Application, knowing the fixed addresses, can read this data using the API Interface and do whatever it wants with it.

"ErrCode" may be useful to a PC Application using the API in order to determine the source of any COMM errors. "LedErr" indicates the source of an error that is shown on the red LED.