Juno Velocity & Torque Control IC
Programming Reference

PERFORMANCE
MOTION DEVICES

Performance Motion Devices, Inc.
1 Technology Park Drive
Westford, MA 01886

Revision 1.1, May, 2020

NOTICE

This document contains proprietary and confidential information of Performance Motion Devices, Inc., and is protected by federal copyright law. The contents of this document may not be disclosed to third parties, translated, copied, or duplicated in any form, in whole or in part, without the express written permission of Performance Motion Devices, Inc.

The information contained in this document is subject to change without notice. No part of this document may be reproduced or transmitted in any form, by any means, electronic or mechanical, for any purpose, without the express written permission of Performance Motion Devices, Inc.

Juno, ATLAS, Magellan, ION, Prodigy, Pro-Motion, C-Motion, and VB-Motion are registered trademarks of Performance Motion Devices, Inc.

Warranty

Performance Motion Devices, Inc. warrants that its products shall substantially comply with the specifications applicable at the time of sale, provided that this warranty does not extend to any use of any Performance Motion Devices, Inc. product in an Unauthorized Application (as defined below). Except as specifically provided in this paragraph, each Performance Motion Devices, Inc. product is provided "as is" and without warranty of any type, including without limitation implied warranties of merchantability and fitness for any particular purpose.

Performance Motion Devices, Inc. reserves the right to modify its products, and to discontinue any product or service, without notice and advises customers to obtain the latest version of relevant information (including without limitation product specifications) before placing orders to verify the performance capabilities of the products being purchased. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement and limitation of liability.

Unauthorized Applications

Performance Motion Devices, Inc. products are not designed, approved or warranted for use in any application where failure of the Performance Motion Devices, Inc. product could result in death, personal injury or significant property or environmental damage (each, an "Unauthorized Application"). By way of example and not limitation, a life support system, an aircraft control system and a motor vehicle control system would all be considered "Unauthorized Applications" and use of a Performance Motion Devices, Inc. product in such a system would not be warranted or approved by Performance Motion Devices, Inc.

By using any Performance Motion Devices, Inc. product in connection with an Unauthorized Application, the customer agrees to defend, indemnify and hold harmless Performance Motion Devices, Inc., its officers, directors, employees and agents, from and against any and all claims, losses, liabilities, damages, costs and expenses, including without limitation reasonable attorneys' fees, (collectively, "Damages") arising out of or relating to such use, including without limitation any Damages arising out of the failure of the Performance Motion Devices, Inc. product to conform to specifications.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent procedural hazards.

Disclaimer

Performance Motion Devices, Inc. assumes no liability for applications assistance or customer product design. Performance Motion Devices, Inc. does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of Performance Motion Devices, Inc. covering or relating to any combination, machine, or process in which such products or services might be or are used. Performance Motion Devices, Inc.'s publication of information regarding any third party's products or services does not constitute Performance Motion Devices, Inc.'s approval, warranty or endorsement thereof.

Patents

Performance Motion Devices, Inc. may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights that relate to the presented subject matter. The furnishing of documents and other materials and information does not provide any license, express or implied, by estoppel or otherwise, to any such patents, trademarks, copyrights, or other intellectual property rights.

Patents and/or pending patent applications of Performance Motion Devices, Inc. are listed at https://www.pmdcorp.com/company/patents.

Related Documents

Juno Velocity & Torque Control IC User Guide

Complete description of all members of the Juno Velocity & Torque Control IC family including the MC71112, MC71112N, MC73112, MC73112N, MC74113, MC74113N, MC75113, MC75113N, MC71113, MC73113, and MC78113 ICs. Includes features and functions with detailed theory of operations.

MC78113 Electrical Specifications

Complete electrical specifications for MC78113 ICs containing physical and electrical characteristics, timing diagrams, pinouts, and pin descriptions.

DK78113 Developer Kit User Manual

How to install and configure the DK78113 developer kit. This developer kit supports all 64-pin TQFP Juno ICs including MC71112, MC73112, MC71113, MC73113, MC74113, MC75113, and MC78113.

Pro-Motion User Guide

User's guide to Pro-Motion, the easy-to-use motion system development tool and performance optimizer. Pro-Motion is a sophisticated, easy-to-use program which allows all motion parameters to be set and/or viewed, and allows all features to be exercised.

DK74113N Developer Kit User Manual

How to install and configure the DK74113N developer kit. This developer kit supports the two 56-pin VQFN Juno step motor control ICs; MC74113N and MC75113N.

DK73112N Developer Kit User Manual

How to install and configure the DK73112N Developer Kit. This developer kit supports the 56-pin VQFN Juno torque control ICs in cluding MC71112N and MC73112N.

PMD Resource Access Protocol Programming Reference

Describes the PMD Resource access Protocol (PRP) used for communication between the host and a PRP device, the software interfaces and binary protocols, the procedures and data types used for programs, software libraries and C-Motion library code.

Table of Contents

This page intentionally left blank.

1. The Juno MC78113 IC Family

In This Chapter

1.1 Introduction

This guide describes the programming interfaces to the MC78113, MC71113, MC73113, MC74113, MC74113N, MC75113, MC75113N, MC71112, MC71112N, MC73112, and MC73112N ICs from Performance Motion Devices, Inc. These devices comprise PMD's Juno Velocity & Torque Control IC family.

The Juno ICs provide high performance velocity and current control for Brushless DC, DC Brush, and step motors. They are ideal for a wide range of applications including precision liquid pumping, laboratory automation, scientific automation, flow rate control, pressure control, high speed spindle control, and many other robotic, scientific, and industrial applications.

Juno provides full four quadrant motor control and directly inputs quadrature encoder, index, and Hall sensor signals. It interfaces to external bridge-type switching amplifiers utilizing PMD's proprietary current and switch signal technology for ultra smooth, ultra quiet motor operation.

Juno ICs can be pre-configured via NVRAM for auto power-up initialization and standalone operation with SPI (Serial Peripheral Interface), direct analog input, or pulse & direction command input. Alternatively Juno can interface via SPI, point-to-point serial, multi-drop serial, or CANbus to a host microprocessor.

Internal profile generation provides acceleration and deceleration to a commanded velocity with 32-bit precision. Additional Juno features include performance trace, programmable event actions, FOC (field oriented control), microstep signal generation, and external shunt resistor control.

All Juno ICs are available in 64-pin TQFPs (Thin Quad Flat Packages) measuring 12.0 mm x 12.0 mm including leads. The MC74113 and MC75113 step motor control ICs and torque control ICs are also available in 56-pin VQFN (Very thin Quad Flat Non-leaded) packages measuring 7.2 mm x 7.2 mm. These VQFN parts are denoted via a "N" suffix in the part number;
MC74113N, MC75113N, MC71112, and MC73112N.

1.2 Family Overview

The following table summarizes the operating modes and control interfaces supported by the Juno IC family:

	MC74113 MC74113N MC75113 MC75113N MC78113	MC71112 MC71112N	MC71113 MC78113	MC73112 MC73112N	MC73113 MC78113
Motor Type & Control Mode					
Motor Type	Step motor	DC Brush	DC Brush	Brushless DC	Brushless DC
Velocity			✓		✓
Torque/current	✓	✓	✓	✓	✓
Position & outer loop			✓		✓
Host Interface					
Serial point-to-point	✓	✓	✓	✓	✓
Serial multi-drop			✓		✓
SPI			✓		✓
CANbus			✓		✓
Command Input					
Analog velocity or torque		✓	✓	✓	✓
SPI velocity or torque		✓	✓	✓	✓
Pulse & direction	✓		✓		✓
SPI position increment	✓				✓
Motion I/O					
Quadrature encoder input	✓ (MC74113 & MC74113N only)		✓	✓	✓
Hall sensor input				✓	✓
Tachometer input			✓		✓
AtRest input	✓				
FaultOut output	✓	✓	✓	✓	✓
HostInterrupt output	✓	✓	✓	✓	✓
Amplifier Control					
PWM High/Low	✓	✓	✓	✓	✓
PWM Sign/Magnitude	✓	✓	✓		
DC Bus & Safety					
Shunt		✓	✓	✓	✓
Overcurrent detect	✓	✓	✓	✓	✓
Over/undervoltage detect	✓	✓	✓	✓	✓
Temperature input	✓	✓	✓	✓	✓
Brake	✓	✓	✓	✓	✓

2. C-Motion

2.1 Introduction

C-Motion is a C source code library that contains all the code required for communicating with either Juno or Magellan Motion Control ICs.

C-Motion includes the following features:

- Axis virtualization.
- The ability to communicate to multiple Juno or Magellan Motion Control ICs.
- Can be easily linked to any C/C++ application.

C-Motion callable functions are broken into two groups, those callable functions that encapsulate motion control IC specific commands, and those callable functions that encapsulate product-specific capabilities.

The motion control IC specific commands are detailed in Chapter 7, *Instruction Reference*. They are the primary commands that you will use to control the major motion features including profile generation, servo loop closure, motor output PWM signal generation, fault handling, trace operations, and many other functions.

Each Juno Motion Control IC command has a C-Motion command of the identical name, but prefaced by the letters "PMD." For example, the Juno command **SetVelocity** is called **PMDSetVelocity**.

2.2 C-Motion Versions

To provide more efficient compiled code for the environments in which different C-Motion-based programs are likely to be used, two separate implementations of C-Motion are provided:

- The CME SDK, for host programs that use either Microsoft Visual Basic or Microsoft Visual C#. This version of C-Motion is also used to communicate with Magellan PRP devices, such as ION/CME digital drives and Prodigy/CME boards. This version is also used for programming the C-Motion Engine on PRP devices.
- The PMD SDK, for host programs written solely in C or C++. This version is simpler to port to non-Windows targets, such as microcontrollers. It supports only the Juno/Magellan command protocols, and does not support PRP.

Both of these C-Motion versions share the same calling sequences for all Magellan commands, however they may not be mixed in the same program. They do not share the same mechanisms for opening a connection to a Motion Control IC, as discussed for the PMD SDK in Section 2.4, "Using C-Motion (PMD SDK)," on page 10.

The CME SDK C-Motion supports both the Juno/Magellan protocols, which are used to communicate with Juno or Magellan attached Motion Control ICs, and also PRP, which is used to communicate with Prodigy/CME boards and ION/CME digital drives. The procedures of this library are exported in a DLL (dynamically linked library), which can be used in Visual Basic or C# program. The DLL is not a managed .NET DLL, it just exports C-Motion procedures.

For more information on using the CME SDK, see the *PMD Resource Access Protocol Programming Reference.*

2.3 Files

The following table lists the files that make up the C-Motion distribution in the PMD SDK.

C-Motion.h	Declarations for the PMD Juno and Magellan command set
C-Motion.c	Implementation of the PMD Juno and Magellan command set
PMDW32ser.h/PMDW32ser.c	Windows serial communication interface functions
PMDutil.h/PMDutil.c	General utility functions
PMDtrans.h/PMDtrans.c	Generic transport (interface) functions
PMDecode.h	Defines the PMD Magellan and C-Motion error codes
PMDocode.h	Defines the control codes for Magellan commands
PMDtypes.h	Defines the basic types required by C-Motion
PMDCAN.h/PMDCAN.c	CAN interface command/data transfer functions.
PMDIXXATCAN.h	CAN interface for IXXAT VCI (Virtual Can Interface) API
PMDIXXATCAN3.c	CAN interface for IXXAT VCI (Virtual Can Interface) API v3.x
PMDNISPI.h	SPI interface for National Instruments USB-8452
PMDNISPI.c	SPI interface for National Instruments USB-8452
PMDcommon.c	Miscellaneous procedures
PMDdevice.h	
PMDdiag.h/PMDdiag.c	Diagnostic functions
IXXAT*.*	IXXAT VCI v3.x (CAN) include and library files
NI*.*	National Instruments (SPI) include and library files

2.4 Using C-Motion (PMD SDK)

C-Motion can be linked to your application code by including the above C language source files in your application. Then, for any application source file that calls the C-Motion API, include “C-Motion.h.”

As distributed, C-Motion supports the National Instruments USB-8452 device for SPI communication, IXXAT devices using the v3.x VCI (Virtual CAN Interface for CANBus), and the native Windows interface for serial ports. By customizing a small number of base interface functions, C-Motion can be ported to almost any hardware interface.

C-Motion is a set of functions that encapsulate the motion control IC command set. Every command has as its first parameter an “axis handle.” The axis handle is a structure containing information about the interface to the motion control IC and the axis number that the handle represents. Before communicating to the motion control IC, the axis handle must be initialized using the following sequence of commands:

```
// the axis handles
PMDAxisHandle hAxis1;

// open interface to PMD Juno processor on COM1
PMDSetupAxisInterface_Serial(&hAxis1, PMDAxis1, 1);
```

The above is an example of initializing communication using the serial communication interface. Each interface .c source file contains an example of initializing the interface. Once the axis handle has been initialized, any of the motion control IC commands can be executed.

The header file “C-Motion.h” includes the function prototypes for all motion control IC commands as implemented in C-Motion. See this file for the required parameters for each command. For information about the operation and purpose of each command, see Chapter 4, *C# Interface*..

Many functions require additional parameters. Some standard values are defined by C-Motion and can be used with the appropriate functions. See “PMDtypes.h” for a complete list of defined types. An example of calling one of the C-Motion functions with the pre-defined types is shown below:

```
PMDSetEventAction(&hAxis1, PMDEventActionMotionError, PMDEventActionPassveBraking);
```

2.4.1 C-Motion Functions (PMD SDK)

The table below describes the functions that are provided by C-Motion in addition to the standard chip command set.

C-Motion functions	Arguments	Function description
PMDSerial_SetConfig	*axis_handle.transport_data* *baudrate* *parity*	Used to set serial port configuration after PMDSetupAxisInterface_Serial.
PMDSerial_SetProtocol	*axis_handle.transport_data* *mode*	Used to set serial port mode after PMDSetupAxisInterface_Serial, required for multi-drop communication.
PMDSerial_SetMultiDropAddress	*axis_handle.transport_data* *address*	Used to set multi-drop address after PMDSetupAxisInterface_Serial, required for multi-drop communication.
PMDCreateMultiDropHandle	*dest_axis_handle* *src_axis_handle* *axis_number* *nodeID*	Used to open an axis interface to a CAN or multi-drop serial axis using an existing handle on the same bus. Must be used for connections after the first.
PMDSetupAxisInterface_Serial	*axis_handle,* *axis_number* *port_number*	Used to setup an axis interface connection for communicating over a RS232 or RS485 serial bus.
PMDSetupAxisInterface_CAN	*axis_handle,* *axis_number* *board_number*	Used to setup an axis interface connection for communicating over a CAN bus.
PMDSetupAxisInterface_SPI	*axis_handle* *axis_number* *device*	Used to setup an axis interface connection for communicating over an SPI bus.
PMDCloseAxisInterface	*axis_handle*	Should be called to terminate an interface connection.
PMDGetErrorMessage	*ErrorCode*	Returns a character string representation of the corresponding PMD chip or C-Motion error code.
GetCMotionVersion	*MajorVersion* *MinorVersion*	Returns the major and minor version number of C-Motion.

This page intentionally left blank.

3. Visual Basic Interface

3.1 Introduction

The CME SDK provides a language binding to Microsoft Visual Basic .NET to the PMD C-Motion library for control of Juno and Magellan Motion Processors. It can be easily integrated with any .NET application. The library supports communication to Juno Developers Kit boards and Juno Motion Controllers via serial (point to point or multi-drop) and CAN (IXXAT). SPI communication is not supported.

There are two parts to the Visual Basic interface code:

1 "C-Motion.dll" is a dynamically loadable library of all documented procedures in the PMD host libraries, including all C-Motion procedures. A source project called "DLLBuild" and all files needed to build the dll are included in the SDK.

2 "PMDLibrary.vb" is Visual Basic source code containing definitions and declarations for DLL procedures, enumerated types, and data structures supporting the use of C-Motion.dll from Visual Basic. "PMDLibrary.vb" should be included in any Visual Basic project for PMD device control.

3 "PMDLibrary.dll" is a .NET library compiled from "PMDLibrary.vb" and can be used with both Visual Basic and C# projects. "PMDLibrary.dll" should be included in any C# project for PMD device control.

Both debug and release versions of "C-Motion.dll" and "PMDLibrary.dll" are provided in directories "CMESDK\HostCode\Debug" and "CMESDK\HostCode\Release," respectively. Both 32- and 64-bit versions are included. The library input file "C-Motion.lib" is also provided so that "C-Motion.dll" may be used with C/C++ language programs. When compiling C/C++ programs to be linked against the DLL the preprocessor symbol **PMD_IMPORTS** must be defined.

"C-Motion.dll" must be in the executable path when using it, either from a C or a Visual Basic program. Frequently the easiest and safest way of doing this is to put it in the same directory as the executable file.

"PMDLibrary.vb" is located in the directory "CMESDK\HostCode\DotNet."

3.2 Visual Basic Classes

The file "PMDLibrary.vb" defines a Visual Basic class for each of the opaque data types used in the PMD library:

PMDPeripheral, **PMDDevice**, **PMDAxis**, and **PMDMemory**. **PMDPeripheral** is inherited by a set of derived classes for each peripheral type: **PMDPeripheralCOM** and **PMDPeripheralCAN**. Each class takes care of allocating and freeing the memory used for the "handle" structures used in the C language interface. Please see the *PMD Resource Access Protocol Programming Reference* for more information.

The following example illustrates how to obtain a Juno axis object connected to a serial port.

```
Public Class Examples
    Public Sub Example2()
        Dim periph As PMDPeripheral
        Dim Juno As PMDDevice
        Dim axis1 As PMDAxis

        ' Open the connection on COM1, using appropriate serial port parameters
        periph = New PMDPeripheralCOM(1, PMDSerialBaud.Baud57600, _
        PMDSerialParity.None, PMDSerialStopBits.Bits1)

        ' Obtain a Juno device object using the peripheral.
        Juno = New PMDDevice(periph, PMDDeviceType.MotionProcessor)

        ' Finally instantiate an axis object for axis number 1.
        axis1 = New PMDAxis(Magellan, PMDAxisNumber.Axis1)

        ' Example operation: Get the event status
        Dim status As UInt16
        status = axis1.EventStatus
    End Sub
End Class
```

4. C# Interface

4.1 Introduction

The CME SDK provides a language binding to Microsoft Visual C# .NET to the PMD C-Motion library for control of Juno and Magellan Motion Processors. It can be easily integrated with any .NET application. The library supports communication to Juno Developers Kit boards and Juno Motion Controllers via serial (point to point or multi-drop) and CAN (IXXAT). SPI communication is not supported.

There are three parts to the Visual Basic interface code:

1. "C-Motion.dll" is a dynamically loadable library of all documented procedures in the PMD host libraries, including all C-Motion procedures.
2. "PMDLibrary.vb" is Visual Basic source code containing definitions and declarations for DLL procedures, enumerated types, and data structures supporting the use of "C-Motion.dll" from .NET applications. The PMDLibrary project should be included in any Visual C# project for PMD device control.
3. "PMDLibrary.dll" is a .NET library compiled from "PMDLibrary.vb" and can be used with both Visual Basic and C# projects. "PMDLibrary.dll" should be included in any C# project for PMD device control. Both debug and release versions of "C-Motion.dll" and "PMDLibrary.dll" are provided in directories "CMESDK\Host-Code\Debug and CMESDK\HostCode\Release," respectively. The library input file "C-Motion.lib" is also provided so that "C-Motion.dll" may be used with C/C++ language programs . When compiling C/C++ programs to be linked against the DLL the preprocessor symbol **PMD_IMPORTS** must be defined.

"C-Motion.dll" and "PMDLibrary.dll" must be in the executable path when using them, either from a C or a Visual Basic program. Frequently the easiest and safest way of doing this is to put it in the same directory as the executable file. "PMDLibrary.vb" is located in the directory "CMESDK\HostCode\DotNet."

4.2 Visual C# Classes

The file "PMDLibrary.dll" defines a class for each of the opaque data types used in the PMD library:

PMDPeripheral, PMDDevice, PMDAxis, and PMDMemory.

PMDPeripheral is inherited by a set of derived classes for each peripheral type: **PMDPeripheralCOM** and **PMDPeripheralCAN**. Each class takes care of allocating and freeing the memory used for the "handle" structures used in the C language interface.

The following example illustrates how to obtain a Juno axis object connected to a serial port.

```
using PMDLibrary;
class Example
   {
      PMD.PMDPeripheral periph;
      PMD.PMDDevice device;
      PMD.PMDMemory memory;
      PMD.PMDAxis axis;

      public void Run()
```

```
    {
      try
      {
        // connect to Juno product over the COM1 serial interface.
        periph = new PMD.PMDPeripheralCOM(0, 57600, PMD.PMDSerialParity.None, PMD.PMDSerialStopBits.SerialStopBits1);
        device = new PMD.PMDDevice(periph, PMD.PMDDeviceType.MotionProcessor);

        // Set up the axis handle
        PMD.PMDAxis axis = new PMD.PMDAxis(device, PMD.PMDAxisNumber.Axis1);

        Int32 pos;
        // C-Motion procedures returning a single value become class properties, and may be
        // retrieved or set by using an assignment.  The "Get" or "Set" part of the name is dropped.
        pos = axis.ActualPosition;

        // Close the connection
        axis.Close();
        device.Close();
        periph.Close();
      }

     catch (Exception e)
     {
       Console.WriteLine(e.Message);
     }
   }
}
```

5. Script Interface

5.1 Introduction

The Juno command interface can be expressed in a simple script language used by the Pro-Motion setup and tuning application. This interface may be used in an interactive command window used to communicate with a Juno or Magellan device. It is also used to specify initialization command sequences to be written by Pro-Motion to NVRAM.

Pro-Motion script files consist of ASCII text, with one statement on each line. An example script is shown in Figure 5-1. Each Juno command is a statement, and there are a small number of other directives. There are no control flow or conditional statements, all commands are executed in order.

```
#ScriptVersion 1
:DESC "Motor 2 settings"
:CVER 1.3
SetDrivePWM 1 561
SetDrivePWM 2 0x80ff
SetDrivePWM 4 8
SetDrivePWM 5 2013
SetDrivePWM 6 2013
SetOutputMode 7
SetMotorCommand 0
SetSignalSense0x0001
SetPhaseParameter 0 0
SetCurrentControlMode 1
SetFOC 512 680
ETC...
```

Figure 5-1: Sample Pro-Motion Script File

The initial script version statement is included to allow some flexibility in upgrading the script language. As of this writing the current script version is 1.

Statements beginning with a colon indicate PSF (PMD Structured Data Format) data. PSF is used to store both NVRAM initialization sequences and data about them, or about the Juno configuration, for example text descriptions, version information, measurement scaling factors and so forth. The :DESC statement contains a description of the Juno configuration, the :CVER statement contains a user version number.

User-specified, labeled data, either as strings or numbers, may be added to NVRAM and later read by a host processor. PSF is described in Chapter 6, *Non-Volatile (NVRAM) Storage*.

Any line beginning with an apostrophe ' is a comment, and will not affect script processing.

Lines beginning with alphabetic characters are command statements.

The first word of a command is the mnemonic name, followed by zero to three arguments. Each argument is one or two 16 bit words. Currently all command arguments are literal numbers, decimal by default or hexadecimal if prefixed by "0x".

In a few cases multiple command arguments are encoded as bitfields in a single word, and must be combined by the user. The arithmetic needed to do so, and an example, will be included in the "Script API" section of the command description.

6. Non-Volatile (NVRAM) Storage

6.1 Introduction

A primary purpose of the NVRAM is to allow Juno initialization information to be stored so that upon power up it can be automatically loaded rather than requiring an external controller to perform this function. In addition however the NVRAM can be used for other functions such as labeling the stored initialization sequence, or for general purpose user-defined storage.

All data stored in the Juno NVRAM utlizes a data format known as PMD Structured data Juno Storage Format (PSF). Users who rely only on PMD's Pro-Motion software package to communicate with Atlas and store and retrieve initialization parameters may not need to concern themselves with the details of PSF. Users who want to address the NVRAM from their own software, or who want to create their own user-defined storage on the Juno NVRAM will utilize the PSF format details provided in the subsequent sections.

PSF is also used as the NVRAM format for Atlas Digital Amplifiers, although the command set and command encoding are different. For more information see the *Atlas Digital Amplifier Complete Technical Reference.*

6.1.1 PMD Structured Data Format

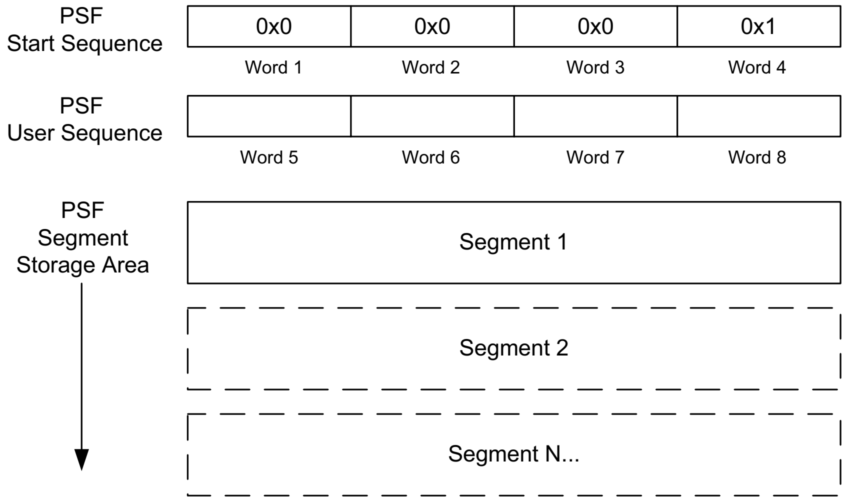


Figure 6-1: High-Level Format of a PSF (PMD Structured Data Format) Memory Space

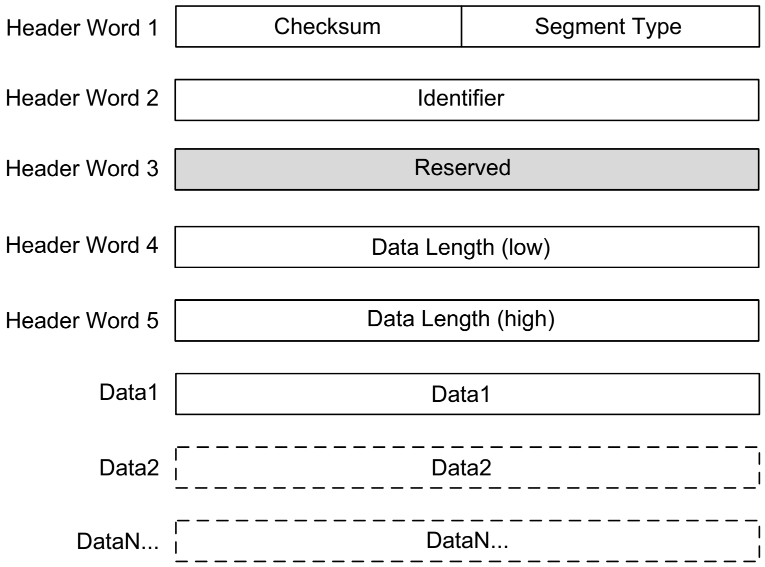
PSF (PMD Structured data Format) is a general purpose data storage format designed for use with non-volatile storage memory such as provided by Juno IC. PSF provides a method to store and label initialization information used by Juno during startup, as well as to allow user-defined storage in NVRAM.

Figure 6-1 shows the overall format of a PSF-managed memory area. The PSF memory space begins with a 4-word start sequence and a 4-word user programmable sequence. Each word is 16 bits in size, as are future references to words in the following sections unless otherwise noted. The start sequence must contain, in order, the values 0x0, 0x0, 0x0, and 0x1. The user sequence can be specified by the user and may contain any values. The user sequence can be used for any purpose but is often used to identify the type of information stored in the PSF memory space.

Following the eight words of sequence words are one or more data storage blocks known as segments, which are themselves structured memory blocks which must follow a specific format.

6.1.2 PSF Data Segments

Figure 6-2: PSF Data Segment Format



The central mechanism which PSF provides to store data is called a data segment. PSF data segments come with their own headers which allow structuring and data integrity checks of the PSF memory space. Figure 6-2 shows the format of a PSF data segment. The following section details each of the elements in this data structure.

Checksum - is the ones complement of an 8-bit ones complement checksum with a seed of 0xAA. It is computed over the entire segment space including the header. If the checksum field is computed correctly then the checksum will be 255 (0xff). The size of this field is one byte.

Segment type - specifies the formatting of the data stored in the segment. This 8 bit field encodes the values 0 through 255. Users may assign segment type values 192-255 for segment types of their own design while all other values are reserved. The size of this field is one byte.

Data length low word & high word - contains a 32 bit value encoding the number of 16-bit words of data (data0, data1, etc…) included with this segment. Data segments can be defined such that a variable number of data words is expected or a fixed number of words is expected. Whether the number of data words varies or not, the data length word must always be specified correctly for the number of data words actually contained in the segment.

Identifier - contains an unformatted 16-bit value that may be used for any purpose but is generally used to identify separate instances of multiply stored segments of the same segment type. For example if there was an array of stored segments, each of the same segment type, the identifier field might be used to identify a specific element within of the overall array of segments.

Data0, Data1, etc… is the data that is being stored in this segment. The exact format of this data is determined by the segment type.

6.1.3 Pre-Defined Segment Types

There are two pre-defined Juno PSF storage segment types. The *Initialization Commands* storage type defines the segment that holds configuration information used during power-up while the *Parameter List* segment holds information that is useful to label the contents of the *Initialization Commands* segment.

During power up Juno scans the NVRAM space for a properly formatted segment with type '*Initialization Commands*,' and if found it initializes itself using the information provided. The *Initialization Commands* segment type is defined in detail in Section 6.1.4, "Initialization Commands Segment Type," on page 21.

A segment of type *Parameter List*, when preceding another segment and when containing certain specific values in the data, stores identification information associated with that segment. For example a human-readable name for the segment can be assigned along with information such as when the segment data was stored. This segment-identifying data is not utilized directly by Juno but rather by software programs such as Pro-Motion. The *Parameter List* segment type is discussed in detail in Section 6.1.5, "Parameter List Segment Type," on page 23.

6.1.4 Initialization Commands Segment Type

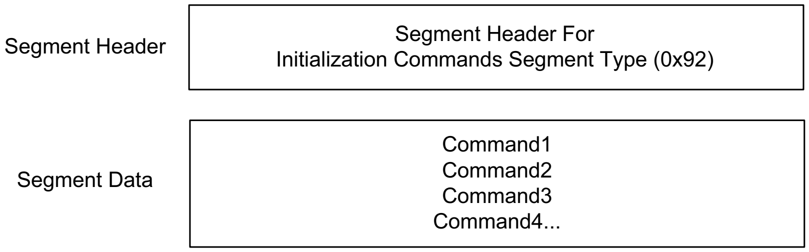


Figure 6-3: Initialization Commands Segment Format

The *Initialization Commands* segment type selects a segment format that holds the PMD commands that are processed during powerup. The segment type value for the *Initialization Commands* segment type is 0x92. The overall format of this segment type is shown in Figure 6-3.

Juno commands stored into the segment data portion of the Initialization Commands segment is formatted similarly to SPI host commands, see *Juno Velocity and Torque Control IC User Guide*, section 13.4, SPI (Serial Peripheral Interface) Communications for more information. The one difference is the order of the two first words, in the SPI format the opcode and axis is sent first, but in the NVRAM format the checksum is first, and the axis and opcode second.

Figure ? and the following table show the details of the command format.

The table below shows a portion of an example initialization command sequence. These example commands enable automatic event recovery mode, delay for 256 cycles so that other system components may initialize themselves, and enable motor output and current control.

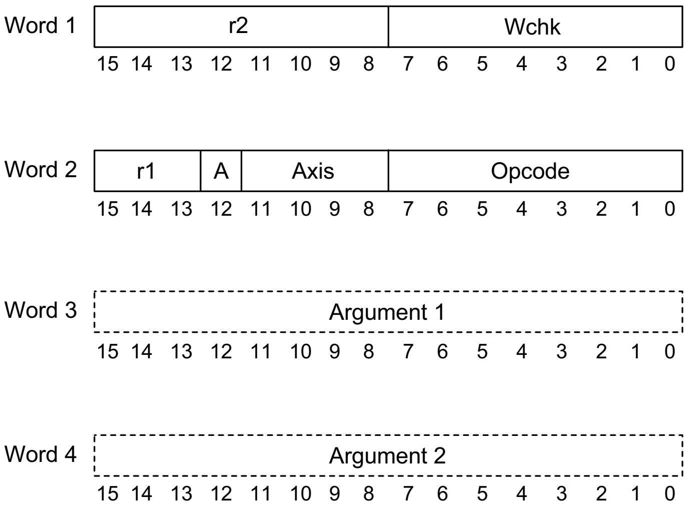
Segment Data Address	Mnemonic	Stored Code (in hex)	Comments
Data1	SetDriveFaultParameter 2 1	0x00EF	Checksum
Data2		0x0062	Axis (0) and opcode
Data3		0x0002	Argument 1: event handling mode
Data4		0x0001	Argument 2: automatic event recovery
Data5	ExecutionControl 0 256	0x001F	Checksum
Data6		0x0035	Axis (0) and opcode
Data7		0x0000	Argument 1: time delay
Data8		0x0000	Argument 2: delay, high word

Segment Data Address	Mnemonic	Stored Code (in hex)	Comments
Data9		0x0100	Argument 2: delay, low word
Data10	SetOperatingMode 7	0x00E8	Checksum
Data11		0x0065	Axis (0) and opcode
Data12		0x0007	Argument 1: Enable output, current loop

See Section 6.1.4, “Initialization Commands Segment Type,” on page 21 for an example of a complete PSF memory image including an initialization command sequence.

See *Juno Velocity and Torque Control IC User Guide*, section 12, Power-Up, Configuration Storage, & NVRAM. for more information on initialization command processing during power up.

Figure 6-4: NVRAM Command Format



This is shown in Figure 6-4 which shows the overall sequence and format for NVRAM commands. The following table details the content of these words:

Field	Bit	Name	Description
Wchk	0-7	Write checksum	Contains the logical negation of an 8-bit ones complement checksum computed over all bits in the command except for the checksum field, and a seed of 0xAA. If the checksum computed by Juno is incorrect (does not equal 0xff), the command will not be executed,NVRAM processing will halt, motor output will be disbaled, and an Instruction Error event signaled.
r2	8-15	*Reserved*	*Reserved, must contain 0.*
Opcode	0-7	Opcode	Contains the 8 bit command opcode
r1	8-15	*Reserved*	*Reserved, must contain 0.*

The additional word writes argument1, argument2, shown in Figure 6-4 contain data (if any) associated with the command. For example for the command **SetMotorCommand**, then a single 16-bit data word, consisting of the programmed command value is stored in argument1. Only the required number of argument data words should be present.

6.1.5 Parameter List Segment Type

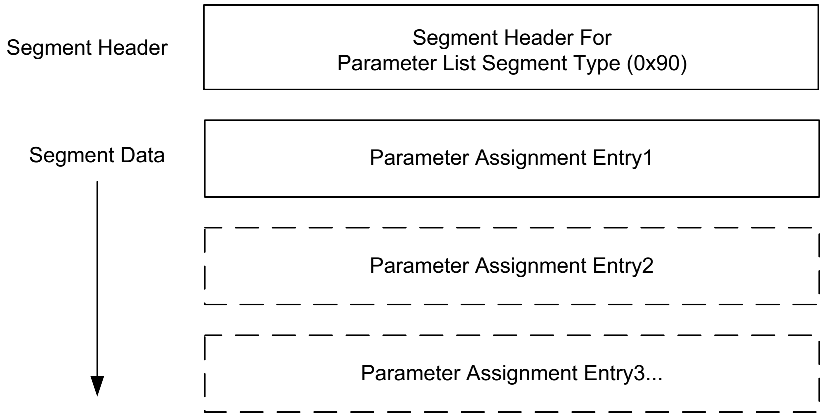

Segment Header
Segment Header For
Parameter List Segment Type (0x90)
Segment Data
Parameter Assignment Entry1
Parameter Assignment Entry2
Parameter Assignment Entry3...

Figure 6-5: Parameter List Segment Format

The *Parameter List* segment type provides a general purpose mechanism for the assignment of values to parameters. A major use of the *Parameter List* segment type is to allow human-readable identification information to be recorded and read back, thereby assisting with the identification of PSF-stored data. See Section 6.1.5.2, "Using the ID Segment Mechanism," on page 24 for information on how this segment ID mechanism is used within the PSF system. The segment type value for the *Parameter List* segment type is 0x90. The overall format of this segment type is shown in Figure 6-5.

The parameter list segment type contains one or more assignments of the general form:

Parameter = Assigned Value

Parameter specifies the name of the parameter being assigned. *Assigned Value* contains the value to assign to the parameter. *Assigned Value* may be formatted as a character string, an integer, a floating point number, or other formats depending on the *Parameter* being assigned.

The data structure that is used to encode each such assignment in the *Parameter List* segment data area is called a parameter assignment entry. The following section details the format of this data structure.

6.1.5.1 Parameter Assignment Entry

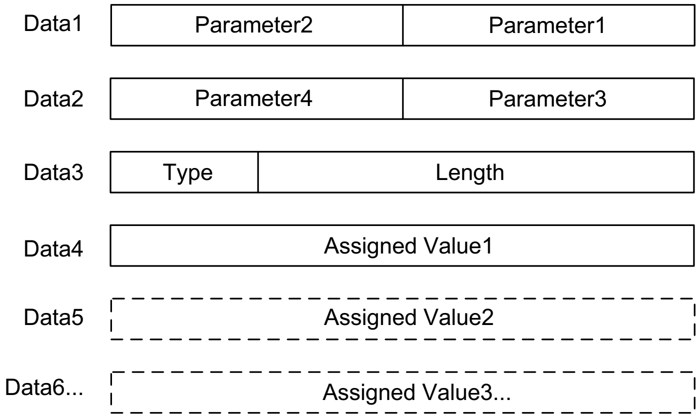


Figure 6-6: Format of Parameter Assignment Entry

Figure 6-6 shows the encoding of the data words for a parameter assignment entry.

The *Parameter* field is specified as four byte-length ASCII characters.

The *Type* determines the encoding of the *Assigned Value* data. This field has a length of four bits.

The *Length* field determines the number of words contained in the *Assigned Value*. This field has a length of 12 bits.

Assigned Value1, *Assigned Value2*, etc… hold the data words comprising the *Assigned Value*.

Six specific parameters can be assigned for the purpose of segment identification. Note that not all of these parameters need to be recorded. If not found, Pro-Motion will simply not display the contents for those specific segment ID-related parameters. The following table provides details on the six available segment-ID related parameters

Parameter Field Encoding	Data Encoding Length & Type	Description
C, N, [0], [0]	The Assigned Value fields contain a UTF-16 uni-code character string of a variable length set via the length field. The type code for a UTF-16 encoded string is 0.	The CN parameter specifies a general purpose name identifier for the segment to follow. An example name might be "X axis motor init. cmds." Note that the two unused parameter field words after "CN" are filled with zeroes.
C,V,E,R	See above	The CVER parameter specifies a version identifier for the segment to follow. An example version might be "version12.3."
D,E,S,C	See above	The DESC parameter specifies a general purpose comment for the segment to follow. An example comment might be "These gain factors were determined using the prototype unit in the engineering lab."
F,N,[0],[0]	See above	The FN parameter specifies the script file name used to store or retrieve the data in the segment to follow. An example file name might be "xaxis.txt." Note that the two unused parameter field bytes after "FN" are filled with ASCII nuls.
F,D,[0],[0]	See above	The FD parameter specifies the modification time of the script file used to store the data in the segment to follow. Times should be recorded in ISO-8601 format "YYYY-MM-DDThh:mm:ss", with hh recorded in 24 hour format. If desired only the year, month and day need be specified. The time portion of this assigned value is optional. An example assigned value might be "2017-01-25T17:13:00" to store a date and time of January 25, 2017 at 5:13pm. Note that the two unused parameter field bytes after "FD" are filled with ASCII nuls.
W,D,[0],[0]	See above	The WD parameter specifies the time that data in the segment to follow was written to NVRAM. See "FD" description for encoding and usage example. Note that the two unused parameter field bytes after "WD" are filled with ASCII nuls.

6.1.5.2 Using the ID Segment Mechanism

Collectively the six parameters from the table above are known as an ID segment. ID Segments specify information for the data segment that immediately follows it in the NVRAM PSF memory space.

When used to provide segment identifying information Pro-Motion, or a similar software program, takes ID information provided by the user and stores it in the correct format into the *Parameter List* segment. The same software program can later search the PSF memory space for segments of type *Parameter List* which hold the correct parameters to retrieve these assigned values for display to the user.

For example if the segment name (see Section 6.1.5.1, "Parameter Assignment Entry," on page 23 for the various types of ID information that can be stored) was specified and saved to NVRAM as "Axis 1 motor gains" by the user during development, Pro-Motion would read from a Juno IC with unknown contents and retrieve this same string for display to the user.

> Other than checking the segment checksum the Juno IC does not read or otherwise process the ID segment. ID segment information is recorded and retrieved by programs such as Pro-Motion for the convenience and utility of the user. Inclusion of an ID-containing segment is therefore optional.

6.1.6 User Defined Segment Types

PSF is a highly flexible data storage system that allows the user to store and if desired, label via the ID segment mechanism structured data into the Juno NVRAM.

Other than ensuring that the overall NVRAM memory size is not exceeded and that the segment header format is followed there are no restrictions placed on what can be stored in the PSF memory space.

Although not required, PMD recommends that each user-defined segment be preceded with an ID segment that identifies the contents as detailed in Section 6.1.5, "Parameter List Segment Type," on page 23. Doing so will assist in keeping track of what data was stored, when, etc… It will also allow the user to develop software tools that can scan the content of the PSF NVRAM space and display a summary of what is stored there, or to utilize Pro-Motion to provide this function.

6.1.7 Complete Example PSF Memory Space

Figure 6-7 provides a word-by-word example of an NVRAM image used to store PSF-formatted initialization commands along with associated segment ID content.

Figure 6-7: Example PSF Memory Space Image

Addr	Word	Contents	Comments
0	0x0000	0	PSF Start Sequence
1	0x0000	0	
2	0x0000	0	
3	0x0001	1	
4	0x0005	5	PSF User Sequence
5	0x0006	6	
6	0x0007	7	
7	0x0008	8	
8	0x2D90	Chksm, seg. type	Parameter List
9	0x0000	identifier	Segment
10	0x0000	reserved	
11	0x002D	length low	
12	0x0000	length high	
13	0x4E43	‘C’, 'N'	Assign CN = "Init1"
14	0x0000	nul, nul	
15	0x0005	type, length	
16	0x0049	"I"	
17	0x006E	"n"	
18	0x0069	"i"	
19	0x0074	"t"	
20	0x0031	"1"	
21	0x5643	‘C’, ‘V’	Assign CVER="1.2"
22	0x5245	'E', 'R'	
23	0x0003	type, length	
24	0x0031	"1"	
25	0x002E	"."	
26	0x0032	"2"	
27	0x4544	‘D’, ‘E’	Assign DESC = "test"
28	0x4353	‘S', 'C'	
29	0x0004	type, length	
30	0x0074	"t"	
31	0x0065	"e"	
32	0x0073	"s"	
33	0x0074	"t"	
34	0x4E46	‘F’, ‘N’	Assign FN = "file.txt"
35	0x0000	nul, nul	
36	0x0008	type, length	
37	0x0066	"f"	
38	0x0069	"i"	
39	0x006c	"l"	
40	0x0065	"e"	
41	0x002E	"."	
42	0x0074	"t"	
43	0X0078	"x"	
44	0x0074	"t"	
45	0x4457	‘W’, ‘D’	Assign WD =
46	0x0000	nul, nul	“2017-01-25”
47	0x000A	type, length	
48	0x0032	"2"	
49	0x0030	"0"	
50	0x0031	"1"	
51	0x0037	"7"	
52	0x002D	"-"	
53	0x0030	"0"	
54	0x0031	"1"	
55	0x002D	"-"	
56	0x0032	"2"	
57	0x0035	"5"	
58	0xB692	chksum, seg. type	Initialization Comments
59	0x0000	identifier	Segment
60	0x0000	reserved	
61	0x000C	length low	
62	0x0000	length high	
63	0x00EF		SetDriveFault
64	0x0062		Parameter 2 1
65	0x0002		
66	0x0001		
67	0x001F		ExecutionControl 0 256
68	0x0035		
69	0x0000		
70	0x0000		
71	0x0100		
72	0x00E8		SetOperatingMode 7
73	0x0065		
74	0x0007		

The Juno command interface can be expressed in a simple script language used by the Pro-Motion setup and tuning application. This interface may be used in an interactive command window used to communicate with a Juno or Magellan device. It is also used to specify initialization command sequences to be written by Pro-Motion to NVRAM.

See Chapter 5, *Script Interface*, for the script file format.

Figure 6-8: Sample Pro-Motion Script File

```
#ScriptVersion 1
:DESC "Motor 2 settings"
:CVER 1.3
SetDrivePWM 1 561
SetDrivePWM 2 0x80ff
SetDrivePWM 4 8
SetDrivePWM 5 2013
SetDrivePWM 6 2013
SetOutputMode 7
SetMotorCommand 0
SetSignalSense0x0001
SetPhaseParameter 0 0
SetCurrentControlMode 1
SetFOC 512 680
ETC...
```

The initial script version statement is included to allow some flexibility in upgrading the script language. As of this writing the current script version is 1.

Statements beginning with a colon indicate PSF (PMD Structured Data Format) data. PSF is used to store both NVRAM initialization sequences and data about them, or about the Juno configuration, for example text descriptions, version information, measurement scaling factors and so forth. The :DESC statement contains a description of the Juno configuration, the :CVER statement contains a user version number.

User-specified, labeled data, either as strings or numbers, may be added to NVRAM and later read by a host processor. PSF is described in Section 6.1.1, “PMD Structured Data Format,” on page 19

Any line beginning with an apostrophe ' is a comment, and will ont affect script processing.

Lines beginning with alphabetic characters are command statements.

The first word of a command is the mnemonic name, followed by zero to three arguments. Each argument is one or two 16 bit words. Currently all command arguments are literal numbers, decimal by default or

hexadecimal if prefixed by "0x".

In a few cases multiple command arguments are encoded as bitfields in a single word, and must be combined by the user. The arithmetic needed to do so, and an example, will be included in the "Script API" section of the command description.

This page intentionally left blank.

7. Instruction Reference

7.1 How to Use This Reference

The instructions are arranged alphabetically, except that all "Set/Get" pairs (for example, **SetVelocity** and **GetVelocity**) are described together. Each description begins on a new page and most occupy no more than a single page. Each page is organized as follows:

Name	The instruction mnemonic is shown at the left, its hexadecimal code at the right.
Motor Types	The motor types to which this command applies. Supported motor types are printed in black; unsupported motor types for the command are greyed out.
Arguments	There are two types of arguments: encoded-field and numeric. Encoded-field arguments are packed into a single 16-bit data word, except for axis, which occupies bits 8–9 of the instruction word. The name of the argument (in italic) is that shown in the generic syntax. Instance (in italic) is the mnemonic used to represent the data value. Encoding is the value assigned to the field for that instance. For numeric arguments, the parameter value, the type (signed or unsigned integer), and the range of acceptable values are given. Numeric arguments may require one or two data words. For 32-bit arguments, the high-order part is transmitted first.
Packet Structure	This is a graphic representation of the 16-bit words transmitted in the packet: the instruction, which is identified by its name, followed by 1, 2, or 3 data words. Bit numbers are shown directly below each word. For each field in a word, only the high and low bits are shown. For 32-bit numeric data, the high-order bits are numbered from 16 to 31, the low-order bits from 0 to 15. The hex code of the instruction is shown in boldface. Argument names are shown in their respective words or fields. For data words, the direction of transfer—read or write—is shown at the left of the word's diagram. Unused bits are shaded. All unused bits must be 0 in data words and instructions sent (written) to the motion control IC.
Description	Describes what the instruction does and any special information relating to the instruction.
Restrictions	Describes the circumstances in which the instruction is not valid, that is, when it should not be issued. For example, velocity, acceleration, deceleration, and jerk parameters may not be issued while an S-curve profile is being executed.
Errors	Lists the error codes that may be returned by the instruction and what they mean in the context of the instruction.
C-Motion API	The syntax of the C function call in the PMD C-Motion library that implements this motion control IC command.
Script API	The syntax for the command in Pro-Motion scripts used for programming NVRAM.
C# API	The syntax for the function in the C# binding for C-Motion. The type of each argument is included as in a declaration, in the actual call syntax the type names would not be included.
Visual Basic API	The Visual syntax for the function in the Visual Basic binding for C-Motion. The type of each argument is included as in a declaration, In the actual call syntax the type names would not be included.
see	Refers to related instructions.

AdjustActualPosition F5h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding		
axis	*Axis1*	0		
	Type	**Range**	**Scaling**	**Units**
position	signed 32 bits	-2^{31} to $2^{31}-1$	unity	counts microsteps

Packet Structure

AdjustActualPosition

0	*axis*	**F5h**
15 … 12	11 … 8	7 … 0

write

position (high-order part)
31 … 16

write

position (low-order part)
15 … 0

Description

The *position* specified as the parameter to **AdjustActualPosition** is summed with the actual position register (encoder position) for the specified *axis*. This has the effect of adding or subtracting an offset to the current actual position. At the same time, the commanded position is replaced by the new actual position value minus the position error. This prevents a servo “bump” when the new axis position is established. In effect, this command establishes a new reference position from which subsequent positions can be calculated. It is commonly used to set a known reference position after a homing procedure.

Errors

None

C-Motion API

```
PMDresult PMDAdjustActualPosition(PMDAxisInterface axis_intf,
                                  PMDint32 position);
```

Script API

```
AdjustActualPosition position
```

C# API

```
PMDAxis.AdjustActualPosition(Int32 position);
```

Visual Basic API

```
PMDAxis.AdjustActualPosition(ByVal position As Int32)
```

see

GetPositionError (p. 60), **GetActualVelocity** (p. 41), **Set/GetActualPositionUnits** (p. 87), **Set/GetActualPosition** (p. 86)

CalibrateAnalog 6Fh

Motor Types

DC Brush	Brushless DC	Microstepping	

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
option	*leg currents*	0
	analog command	1
	tachometer	2

Returned data

None

Packet Structure

	15–12	11–8	7–0
	0	*axis*	**6F**h

	15–0
write	*option*

Description

The **CalibrateAnalog** command is used to adjust the adjustable offsets for some analog input channels. The leg current option calibrates only the leg current sensors used for the current motor type. The analog command and tachometer options calibrate a single input. The option argument controls the set of analog channels calibrated, currently the only choice is to calibrate the four leg current inputs for a Juno motion control IC.

The calibration process assumes that the actual input to the analog channels will be zero. For the leg current sensors it is generally sufficient to set the motor command to zero and ensure that the motor is not moving. Whether motor output should be enabled or not depends on external circuitry.

Calibration is accomplished by averaging a number of readings; 100 ms after sending the command the process may be assumed to be complete. When the calibration process starts the Calibrated bit in the Drive Status register will be cleared, when the process is completed it will be set. The Drive Status register may be polled in order to determine when calibration is complete.

The calibration offsets computed by the **CalibrateAnalog** command are stored in volatile RAM, they may be read using the **GetAnalogCalibration** command. Calibration offsets are preserved across calls to the **SetMotorType** command, but are lost during a reset. It is possible to store calibration offsets in NVRAM using the NVRAM command, see Chapter 6 "*Non-Volatile (NVRAM) Storage*" for more information. It is also possible to call the **CalibrateAnalog** command from NVRAM, in which case the **ExecutionControl** command should be used afterwards to wait for the Activity Status Calibrated bit to be set.

Errors

Invalid Parameter: Unrecognized option.

C-Motion API

```
PMDresult PMDCalibrateAnalog(PMDAxisInterface axis_intf,
                             PMDuint16 option);
```

Script API

```
CalibrateAnalog option
```

C# API

```
PMDAxis.CalibrateAnalog(Int16 option);
```

Visual Basic API

```
PMDAxis.CalibrateAnalog(ByVal option As Int16)
```

see

GetDriveStatus (p. 48), **Set/GetAnalogCalibration** (p. 88), **ReadAnalog** (p. 75), **NVRAM** (p. 72), **ExecutionControl** (p. 35)

ClearDriveFaultStatus 6Ch

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Packet Structure

ClearDriveFaultStatus

15 – 12	11 – 8	7 – 0
0	*axis*	**6C**h

Description

ClearDriveFaultStatus clears all bits in the Drive Fault Status register. A bit is cleared only if it has been read by **GetDriveFaultStatus** since the last detection of the fault condition, so that information on faults detected between **GetDriveFaultStatus** and **ClearDriveFaultStatus** is not lost.

Errors

None

C-Motion API

```
PMDresult PMDClearDriveFaultStatus (PMDAxisInterface axis_intf);
```

Script API

```
ClearDriveFaultStatus
```

C# API

```
PMDAxis.ClearDriveFaultStatus();
```

Visual Basic API

```
PMDAxis.ClearDriveFaultStatus()
```

see

GetDriveFaultStatus (p. 46)

ClearInterrupt ACh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Packet Structure

ClearInterrupt

0	*axis*	**4Ch**
15 12	11 8	7 0

Description

ClearInterrupt resets the /HostInterrupt signal to its inactive state. If interrupts are still pending, the /HostInterrupt line will return to its active state within one chip cycle. See **Set/GetSampleTime** (p. 151) for information on chip cycle timing. This command is used after an interrupt has been recognized and processed by the host; it does not affect the Event Status register. The **ResetEventStatus** command should be issued prior to the **ClearInterrupt** command to clear the condition that generated the interrupt. The **ClearInterrupt** command has no effect if it is executed when no interrupts are pending.

When communicating using CAN, this command resets the interrupt message sent flag. When an interrupt is triggered on an *axis*, a single interrupt message is sent and no further messages will be sent by that *axis* until this command is issued.

When serial or parallel communication is used, the axis number is not used.

Errors

None

C-Motion API

```
PMDresult PMDClearInterrupt (PMDAxisInterface axis_intf);
```

Script API

```
ClearInterrupt
```

C# API

```
PMDAxis.ClearInterrupt();
```

Visual Basic API

```
PMDAxis.ClearInterrupt()
```

see

Set/GetInterruptMask (p. 132), **ResetEventStatus** (p. 82).

ClearPositionError 47h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Packet Structure

ClearPositionError

15 … 12	11 … 8	7 … 0
0	*axis*	**47**h

Description

ClearPositionError sets the profile's commanded position equal to the actual position (encoder input), thereby clearing the position error for the specified ***axis***. This command can be used when the axis is at rest, or when it is moving.

Errors

None

C-Motion API

```
PMDresult PMDClearPositionError (PMDAxisInterface axis_intf);
```

Script API

```
ClearPositionError
```

C# API

```
PMDAxis.ClearPositionError();
```

Visual Basic API

```
PMDAxis.ClearPositionError()
```

see

GetPositionError (p. 60)

ExecutionControl 35h

Motor Types

Brush DC	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding	
axis	Axis1	0	
condition	delay	0	
	— (Reserved)	1-7	
	event status	8	
	activity status	9	
	signal status	10	
	drive status	11	
	— (Reserved)	12-255	
timeScale	multiply by 2	0	
	multiply by 256 (2^8)	1	
	multiply by 32768 (2^{15})	2	
	multiply by 4194034 (2^{22})	3	
timeValue	unsigned 6 bit	0-63	51.2 µs
value	unsigned 32bit	see below	

Packet Structure

ExecutionControl

0	*axis*	1h
15 12	11 8	7 0

write	*timeScale*	*timeValue*	*condition*
	15 14	13 8	7 0

write	*value* (high-order part)
	15 0

write	*value* (low-order part)
	15 0

Description

ExecutionControl is used to delay execution during NVRAM initialization, usually so that some hardware external to the Juno IC may become ready. In all cases the timeout value is measured in units of the 51.2 µs commutation time.

If the condition is ***delay***, then a pure delay for a fixed time. In this case the ***value*** argument is an unsigned count of commutation cycles to wait. The exit status in this case is always zero, or no error. In this case the ***timeScale*** and ***timeValue*** arguments must both be zero.

If the condition is ***event status, activity status, signal status***, or ***drive status***, then execution will be delayed until either a specified condition becomes true for the specified register, or a timeout expires. The condition is defined by the supplied ***value*** – the high order part is a selection mask for the register value, and the low order part is a sense mask. The wait will end successfully when the register value, logically ANDed with the selection mask is equal to the sense mask.

For example, to wait for phase initialization to complete, the condition should be *activity status*, because bit 0 of the activity status register is defined as *Phasing Initialized*. The selection mask in this case would be 0001h, and the sense mask also 00001h.

Description (cont.)

As another example, to wait until the ~*Enable* signal is low (active), one should wait until bit 13 of the Signal Status register is clear. The condition should be *signal status*, the selection mask 2000h, and the sense mask 0000h.

When waiting conditionally on a register value, the ***timeScale*** and ***timeValue*** arguments specify a timeout period in commutation cycles. If the timeout period elapses before the condition becomes true then the command will exit with an error status of *Wait Timed Out*, NVRAM command processing will stop, and motor output will be disabled. The *Instruction Error* bit of the event status register will be set, and the **GetInstructionError** command may be used to read the error status.

A ***timeValue*** of zero means “wait forever”; a timeout will never occur.

timeValue is multiplied by ***timeScale***, to give a wider range. The minimum timeout is 2 commutation cycles, the maximum value is 63 x 2^{22} = 264,241,152, or approximately 3.7 hours.

Juno does not normally accept host input on the serial, CAN, or SPI channels until NVRAM initialization has completed, however if an **ExecutionControl** wait is started then the host interfaces will be initialized and host commands accepted. In this situation it is possible for NVRAM commands to be executed after outside host commands, changing Juno state. In all cases only one command, from any source, is executed at a time.

The script interface combines the condition, ***timeValue*** and ***timeScale*** arguments into a single option argument as shown below. For example, if the condition is event status (8), and the desired timeout value is 768 commutation cycles, then the ***timeScale*** x256 (1) and the ***timeValue*** is 3. The option argument should be 8 + 256*3 + 16384*1 = 17160

Restrictions

Valid only when executed from NVRAM.

Errors

Invalid Parameter: Condition is not a supported value, tvalue or tscale nonzero for pure delay.

Initialization Only: Command was sent using serial, CAN, or SPI host channel.

Wait Timed Out: Timeout elapsed before condition became true.

C-Motion API

```
PMDresult PMDExecutionControl(PMDAxisInterface axis_intf, PMDuint8
                              condition, PMDuint8 timeScale, PMDuint16
                              timeValue, PMDint32 value);
```

Script API

```
ExecutionControl option value
where option = condition + 256*timeValue + 16384*timeScale
```

C# API

```
PMDAxis.ExecutionControl(Int16 condition, Int16 timeValue,
                         Int16 timeScale, Int32 value);
```

Visual Basic API

```
PMDAxis.ExecutionControl(ByVal condition As Int16, ByVal timeValue
                         As Int16,ByVal timeScale As Int16,ByVal
                         value as Int32)
```

see

NVRAM (p. 72), **GetEventStatus** (p. 52), **GetActivityStatus** (p. 40), **GetDriveStatus** (p. 48), **GetSignalStatus** (p. 64), **GetInstructionError** (p. 56)

GetActiveMotorCommand 3Ah

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned data

	Type	Range	Scaling	Units
command	signed 16 bits	-2^{15} *to* $2^{15}-1$	$100/2^{15}$	% output

Packet Structure

GetActiveMotorCommand

0	*axis*	**3Ah**
15 – 12	11 – 8	7 – 0

Data

	command
read	15 – 0

Description

GetActiveMotorCommand returns the value of the motor output command for the specified *axis*. This is the input to the commutation or FOC current control. Its source depends on the motor type, as well as the operating mode of the *axis*.

For brushless DC or DC brush motors: If the velocity loop is enabled, it is the output of the velocity servo filter, if the position/outer loop is enabled but the velocity loop is not, it is the output of the outer loop servo filter, divided by 65536. If the command source is enabled without either the position/outer loop nor the velocity loop then it is the command input divided by 65536.

For microstepping motors: It is the contents of the motor output command register, subject to holding current reduction.

Errors

None

C-Motion API

```
PMDresult PMDGetActiveMotorCommand (PMDAxisInterface axis_intf,
                                    PMDint16* command);
```

Script API

```
GetActiveMotorCommand
```

C# API

```
Int16 command = PMDAxis.ActiveMotorCommand;
```

Visual Basic API

```
Int16 command = PMDAxis.ActiveMotorCommand
```

see

Set/GetMotorCommand (p. 138), **Set/GetOperatingMode** (p. 144), **GetActiveOperatingMode** (p. 38)

GetActiveOperatingMode 57h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned Data

	Type	
mode	unsigned 16 bits	bit field

Packet Structure

GetActiveOperatingMode

0	*axis*	**57**h
15 12	11 8	7 0

read	*mode*
	15 0

Description

GetActiveOperatingMode gets the actual operating mode that the *axis* is currently in. This may or may not be the same as the static operating mode, as safety responses or programmable conditions may change the **Active Operating Mode**. When this occurs, the **Active Operating Mode** can be changed to the programmed static operating mode using the **RestoreOperatingMode** command. The bit definitions of the operating mode are given below.

Name	Bit	Description
—	0	Reserved
Motor Output Enabled	1	0: *axis* motor outputs disabled. 1: *axis* motor outputs enabled.
Current Control Enabled	2	0: *axis* current control bypassed. 1: *axis* current control active.
Velocity Loop	3	0: velocity loop bypassed, 1: velocity loop active
Position Loop Enabled	4	0: *axis* position loop bypassed. 1: *axis* position loop active.
Command Source Enabled	5	0: command source disabled. 1: command source enabled.
—	6-7	Reserved
Braking	8	PWM output is set for passive braking.
Smooth Stop	9	A smooth stop is in progress.
—	10–15	Reserved

When the axis is disabled, no processing will be done on the axis, and the axis outputs will be at their reset states. When the axis motor output is disabled, the axis will function normally, but its motor outputs will be in their disabled state. When a loop is disabled (position or current loop), it operates by passing its input directly to its output, and clearing all internal state variables (such as integrator sums, etc.). When the command source is disabled, if either the position/outer or velocity loops are active then the command is set to zero, otherwise if motor output is enabled it is set to the value of the motor command register.

The braking and smooth stop bits may not be set directly by using **SetOperatingMode**, they are only set as a part of event processing. The braking bit means that passive braking has been triggered, and, as a result, normal PWM output is suppressed. When braking the motor output, command source, and all control loops will be disabled. After clearing the responsible event bits the operating mode may be set or restored to re-enable PWM output.

Description (cont.)

The smooth stop bit means that a smooth stop has been triggered as a part of event processing while the command source was something other than the internal profile. In this case a smooth stop is arranged by switching the command source to the internal profile, starting with the commanded velocity from the previous command source, and using the value of the maximum deceleration register for deceleration. If the maximum deceleration value is zero then the value of the maximum acceleration value will be used instead. If the maximum acceleration value is also zero then an abrupt stop will be done by simply disabling the command source.

After a smooth stop restoring the operating mode will automatically restore the command source to its commanded value, typically the one it had before the smooth stop began.

Restrictions

The possible modes of an axis are product specific, and in some cases axis specific. See the product user guide for a description of what modes are supported on each axis.

Errors

None

C-Motion API

```
PMDresult PMDGetActiveOperatingMode(PMDAxisInterface axis_intf,
                                    PMDuint16* mode);
```

Script API

```
GetActiveOperatingMode
```

C# API

```
UInt16 mode = PMDAxis.ActiveOperatingMode;
```

Visual Basic API

```
UInt16 mode = PMDAxis.ActiveOperatingMode
```

see

GetOperatingMode (p. 144), **RestoreOperatingMode** (p. 83), **Set/GetEventAction** (p. 125), **SetDeceleration** (p. 113), **SetAcceleration** (p. 84)

GetActivityStatus A6h

Motor Types

DC Brush	Brushless DC	Microstepping

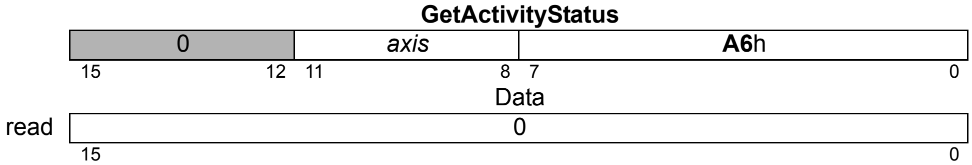
Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned Data

	Type	
status	unsigned 16 bits	see below

Packet Structure



Description

GetActivityStatus reads the 16-bit Activity Status register for the specified ***axis***. Each of the bits in this register continuously indicate the state of the motion control IC without any action on the part of the host. There is no direct way to set or clear the state of these bits, since they are controlled by the motion control IC.

The following table shows the encoding of the data returned by this command.

Name	Bit(s)	Description
Phasing Initialized	0	Set to 1 if phasing is initialized (brushless DC axes only).
At Maximum Velocity	1	Set to 1 when the trajectory is at maximum velocity. This bit is determined by the trajectory generator, not the actual encoder velocity.
—	2-8	Reserved
Position Capture	9	Set to 1 when a value has been captured by the high speed position capture hardware but has not yet been read.
In-motion	10	Set to 1 when the trajectory generator is executing a profile.
—	11-15	Reserved

Errors

None

C-Motion API

```
PMDresult PMDGetActivityStatus(PMDAxisInterface axis_intf,
                               PMDuint16* status);
```

Script API

```
GetActivityStatus
```

C# API

```
UInt16 status = PMDAxis.ActivityStatus;
```

Visual Basic API

```
UInt16 status = PMDAxis.ActivityStatus
```

see

GetEventStatus (p. 52), **GetSignalStatus** (p. 64), **GetDriveStatus** (p. 48)

GetActualVelocity ADh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned Data

	Type	Range	Scaling	Units
actual velocity	signed 32 bits	-2^{31} *to* $2^{31}-1$	$1/2^{16}$	counts/cycle

Packet Structure

GetActualVelocity

0	*axis*	**AD**h
15 – 12	11 – 8	7 – 0

read	*actual velocity* (high-order part)
	31 – 16
read	*actual velocity* (low-order part)
	15 – 0

Description

GetActualVelocity reads the value of the *actual velocity* for the specified *axis*. The *actual velocity* is derived by subtracting the actual position during the previous chip cycle from the actual position for this chip cycle. The result of this subtraction will always be integer because position is always integer. As a result the value returned by **GetActualVelocity** will always be a multiple of 65,536 since this represents a value of one in the 16.16 number format. The low word is always zero (0). This value is the result of the last encoder input, so it will be accurate to within one cycle.

Scaling example: If a value of 1,703,936 is retrieved by the **GetActualVelocity** command (high word: 01Ah, low word: 0h), this corresponds to a velocity of 1,703,936/65,536 or 26 counts/cycle.

C-Motion API

```
PMDresult PMDGetActualVelocity(PMDAxisInterface axis_intf,
                               PMDint32* velocity);
```

Script API

```
GetActualVelocity
```

C# API

```
Int32 velocity = PMDAxis.ActualVelocity;
```

Visual Basic API

```
Int32 velocity = PMDAxis.ActualVelocity
```

see

GetCommandedVelocity (p. 45), **GetActualPosition** (p. 86)

GetCaptureValue 36h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned data

	Type	Range	Scaling	Units
position	signed 32 bits	-2^{31} *to* $2^{31}-1$	unity	counts microsteps

Packet Structure

GetCaptureValue

0	*axis*	**36**h
15 — 12	11 — 8	7 — 0

read	*position* (high-order part)
	31 — 16

read	*position* (low-order part)
	15 — 0

Description

GetCaptureValue returns the contents of the position capture register for the specified *axis*. This command also resets bit 9 of the Activity Status register, thus allowing another capture to occur.

If actual position units is set to steps, the returned position will be in units of steps.

Errors

None

C-Motion API

```
PMDresult PMDGetCaptureValue(PMDAxisInterface axis_intf,
                             PMDint32* position);
```

Script API

```
GetCaptureValue
```

C# API

```
Int32 position = PMDAxis.CaptureValue;
```

Visual Basic API

```
Int32 position = PMDAxis.CaptureValue
```

see

Set/GetActualPositionUnits (p. 87), **GetActivityStatus** (p. 40)

GetCommandedAcceleration **A7**h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned data

	Type	Range	Scaling	Units
acceleration	signed 32 bits	-2^{31} *to* $2^{31}-1$	$1/2^{24}$	counts/cycle^2 microsteps/cycle^2

Packet Structure

GetCommandedAcceleration

0	*axis*	**A7**h
15 12	11 8	7 0

read	*acceleration* (high-order part)
	31 16
read	*acceleration* (low-order part)
	15 0

Description

GetCommandedAcceleration returns the commanded *acceleration* value for the specified *axis.* Commanded acceleration is the instantaneous acceleration value output by the trajectory generator.

Scaling example: If a value of 11, 468,890 is retrieved using this command then this corresponds to 11,468,890/16,777,216 = 0.6836 counts/cycle^2 acceleration value.

Restrictions

Does not return a meaningful value unless command source is internal profile.

Errors

None

C-Motion API

```
PMDresult PMDGetCommandedAcceleration(PMDAxisInterface axis_intf,
                                      PMDint32* acceleration);
```

Script API

```
GetCommandedAcceleration
```

C# API

```
Int32 acceleration = PMDAxis.CommandedAcceleration;
```

Visual Basic API

```
Int32 acceleration = PMDAxis.CommandedAcceleration
```

see

GetCommandedPosition (p. 44), **GetCommandedVelocity** (p. 45), **Set/GetDriveCommandMode** (p. 114)

GetCommandedPosition 1Dh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned data

	Type	Range	Scaling	Units
position	signed 32 bits	-2^{31} *to* $2^{31}-1$	unity	counts microsteps

Packet Structure

GetCommandedPosition

0	*axis*	**1D**h
15 12	11 8	7 0

read	*position* (high-order part)
	31 16

read	*position* (low-order part)
	15 0

Description

GetCommandedPosition returns the commanded *position* for the specified *axis*. Commanded position is the instantaneous position value output by the trajectory generator.

This command functions in all drive command modes.

Errors

None

C-Motion API

```
PMDresult PMDGetCommandedPosition(PMDAxisInterface axis_intf,
                                  PMDint32* position);
```

Script API

```
GetCommandedPosition
```

C# API

```
Int32 position = PMDAxis.CommandedPosition;
```

Visual Basic API

```
Int32 position = PMDAxis.CommandedPosition
```

see

GetCommandedAcceleration (p. 43), **GetCommandedVelocity** (p. 45)

GetCommandedVelocity 1Eh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned data

	Type	Range	Scaling	Units
velocity	signed 32 bits	-2^{31} *to* $2^{31}-1$	$1/2^{16}$	counts/cycle microsteps/cycle

Packet Structure

GetCommandedVelocity

0	*axis*	**1E**h
15 … 12	11 … 8	7 … 0

read	*velocity* (high-order part)
	31 … 16

read	*velocity* (low-order part)
	15 … 0

Description

GetCommandedVelocity returns the commanded *velocity* value for the specified *axis*. Commanded velocity is the instantaneous velocity value output by the command source.

Scaling example: If a value of –1,234,567 is retrieved using this command (FFEDh in high word, 2979h in low word) then this corresponds to –1,234,567/65,536 = –18.8380 counts/cycle velocity value.

When the command source is internal profile the commanded velocity is taken directly from the profile output.

When the command source is analog or direct SPI for servo motors the commanded velocity is the command value divided by the velocity scalar to convert it to counts/cycle.

When the command source is pulse & direction or direct SPI for step motors the commanded velocity is the difference between two successive commanded positions, and may be quite noisy.

Errors

None

C-Motion API

```
PMDresult PMDGetCommandedVelocity(PMDAxisInterface axis_intf,
                                  PMDint32* velocity);
```

Script API

```
GetCommandedVelocity
```

C# API

```
Int32 velocity = PMDAxis.CommandedVelocity;
```

Visual Basic API

```
Int32 velocity = PMDAxis.CommandedVelocity
```

see

GetCommandedAcceleration (p. 43), **GetCommandedPosition** (p. 44), **Set/GetDriveCommandMode** (p. 114)

GetDriveFaultStatus 6Dh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned Data

	Type	
status	unsigned 16 bits	see below

Packet Structure

GetDriveFaultStatus

15 – 12	11 – 8	7 – 0
0	*axis*	**6D**h

	15 – 0
read	*Status*

Description

GetDriveFaultStatus reads the Drive Fault Status register, which is a bitmap of fault conditions.

Several of the faults recorded in the Drive Fault Status register are handled by raising a Drive Exception event. Reading the Drive Fault Status register is required after detecting a Drive Exception event, in order to determine what happened.

An Overcurrent fault occurs when either the bus supply current or the bus return current exceeds the limit that was set using **SetDriveFaultParameter**. The bus supply current is measured using an analog input signal. The bus return current is calculated from the measured leg currents and the PWM duty cycles.

When an Overcurrent fault is detected the Drive Exception event will be raised and an action specified by **SetEventAction** is performed. The default action is to disable all motor output.

An Undervoltage or Overvoltage fault occurs when the measured bus voltage falls below the minimum or rises above the maximum specified using **SetDriveFaultParameter**. When an Undervoltage or Overvoltage fault is detected a Bus Voltage Fault event will be raised and an action specified by **SetEventAction** is performed. The default action is to disable all motor output.

An Overtemperature fault occurs when the analog temperature signal exceeds the minimum value specified using **SetDriveFaultParameter**. When an Overtemperature fault is detected an Overtemperature event is raised, and an action specified by **SetEventAction** is performed. The default action is to disable all motor output.

A Brake Signal fault occurs when the *Brake* signal becomes active. When an active *Brake* signal is detected a Drive Exception event is raised, and an action specified by **SetEventAction** is performed. The default action is to begin passive braking.

An SPI Mode Change occurs when the SPI command mode is direct input, and a particular input sequence is sent in order to restore SPI host command input. See "GetSPIMode 0Bh" on page 65. When an SPI mode change request is detected a Drive Exception event will be raised, an action specified by **SetEventAction** is performed, the direct input bit in the SPI mode register is cleared, and host commands will read on the SPI bus and serviced.

All bits in the Drive Fault Status register are latched, and may be cleared by using the **ClearDriveFaultStatus** command, which unconditionally clears all bits that have been previously been read. The Drive Fault Status register should be cleared before attempting to handle any disabling condition, so the cause of subsequent failures may be determined.

Description (cont.)

The table below shows the bit definitions of the Drive Fault Status register.

Name	Bit
Overcurrent Fault	0
— (Reserved)	1-3
SPI Mode Change	4
Overvoltage Fault	5
Undervoltage Fault	6
— (Reserved)	7
Current Foldback	8
Overtemperature Fault	9
— (Reserved)	10
Watchdog Timeout	11
— (Reserved)	12
Brake signal	13
— (Reserved)	14-15

Restrictions

This command is not available in products without drive amplifier support.

C-Motion API

```
PMDresult PMDGetDriveFaultStatus(PMDAxisInterface axis_intf,
                                 PMDuint16* status);
```

Script API

```
GetDriveFaultStatus
```

C# API

```
Uint16 status = PMDAxis.DriveFaultStatus;
```

Visual Basic API

```
Uint16 status = PMDAxis.DriveFaultStatus
```

see

ClearDriveFaultStatus (p. 32), **SetMotorType** (p. 142)**, SetEventAction** (p. 125)**,** **Set/GetDriveFaultParameter** (p. 116), **GetSPIMode** (p. 65)

GetDriveStatus 0Eh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned data

	Type	
status	unsigned 16 bits	see below

Packet Structure

GetDriveStatus

15–12	11–8	7–0
0	*axis*	**0E**h

	15–0
read	*Status*

Description

GetDriveStatus reads the Drive Status register for the specified *axis*. All of the bits in this status word are set and cleared by the motion control IC. They are not settable or clearable by the host. The bits represent staes or conditions in the motion control IC that are of a transient nature.

Name	Bit(s)	Description
Calibrated	0	Set to 0 at the start of a calibration, set to 1 when complete.
In Foldback	1	Set to 1 when the unit is in the current foldback state– the output current is limited by the foldback limit.
Overtemperature	2	Set to 1 when the overtemperature condition is present.
Shunt active	3	The bus voltage limiting shunt PWM is active.
In Holding	4	Set to 1 when the unit is in the holding current state– the output current is limited by the holding current limit.
Overvoltage	5	Set to 1 when the overvoltage condition is present.
Undervoltage	6	Set to 1 when the undervoltage condition is present.
—	7	Reserved, may be 0 or 1.
—	8–11	Reserved; not used; may be 0 or 1.
Output Clipped	12	Drive output is limited because it has reached 100%, or the Drive PWM limit, or the current loop integrator limit.
—	13	Reserved; not used; may be 0 or 1.
Initializing	14	Set to 1 at the beginning of initialization from NVRAM, set to 0 when initialization is complete

The Calibrated bit is set by the **AnalogCalibration** command, and may be polled to determine that the calibration is complete.

The Initializing bit is set when the initialization command sequence in NVRAM is begun, and is cleared when it is complete, or has been aborted due to an error. NVRAM initialization is begun before enabling host communication, reading this bit set normally means that initialization is waiting for some condition using the **ExecutionControl** command. **GetBufferReadIndex** for buffer 1 may be used to determine the address of the NVRAM command currently being executed.

Restrictions

The bits available in this register depend upon the products. See the product user guide.

Errors

None

C-Motion API

```
PMDresult PMDGetDriveStatus(PMDAxisInterface axis_intf,
                            PMDuint16* status);
```

Script API

```
GetDriveStatus
```

C# API

```
Uint16 status = PMDAxis.DriveStatus;
```

Visual Basic API

```
Uint16 status = PMDAxis.DriveStatus
```

see **ExecutionControl** (p. 35)**, CalibrateAnalog** (p. 31), **GetBufferReadIndex** (p. 94)

GetDriveValue 70h

Motor Types

DC Brush	Brushless DC	Microstepping

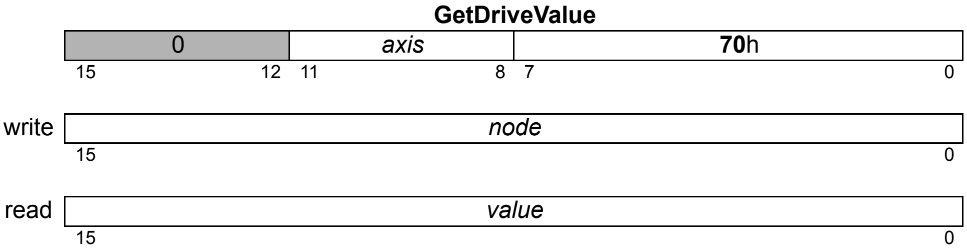
Arguments

Name	Instance	Encoding
axis	*Axis1*	0
node	*Bus Voltage*	0
	Temperature	1
	Bus Current Supply	2
	Bus Current Return	3

Returned data

	Type	Range/Scaling
value	signed or unsigned 16 bits	see below

Packet Structure



Description

GetDriveValue is used to read values associated with drive output or state, and enumerated by node.

The following nodes are supported:

Bus Voltage is the most recent bus voltage reading from the axis, returned as an unsigned 16 bit value. Zero corresponds to 0V (corrected for offset) at the analog input, 65535 to 3.3V.

Temperature is the most recent temperature reading from temperature sensor monitoring axis, returned as a signed 16 bit value. Zero corresponds to 0V (corrected for offset) at the analog input, 32767 to 3.3V. If the temperature limit set by **SetDriveFaultParameter** is negative then the sense of the temperature is inverted by subtracting the measured value from 32768.

Bus Current Supply is the most recent reading from the bus current supply sensor, returned as an unsigned 16 bit value. Zero corresponds to 0V (corrected for offset) at the analog input, 32767 to 3.3V.

Bus Current Return is the most recent current return reading computed from all leg current readings and PWM duty cycles, returned as a signed 16 bit number. The scaling is the same as the leg current scaling.

Restrictions

GetDriveValue is currently supported only by MC58113 series motion control ICs.

Errors

Invalid parameter: node is not a supported value.

C-Motion API

```
PMDresult PMDGetDriveValue(PMDAxisInterface axis_intf,
                  PMDuint8 node,
                  PMDuint16 * value);
```

Script API

```
GetDriveValue node
```

C# API

```
UInt16 value = PMDAxis.DriveValue(PMDDriveValue node);
```

Visual Basic API

```
UInt16 value = PMDAxisDriveValue(ByVal node As PMDDriveValue)
```

see

Set/GetAnalogCalibration (p. 88), **CalibrateAnalog** (p. 31)**, SetDriveFaultParameter** (p. 116)

GetEventStatus 31h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned data

	Type	
status	unsigned 16 bits	see below

Packet Structure

GetEventStatus

	0	*axis*	**31**h
	15 12	11 8	7 0

Data

read	*status*
	15 0

Description

GetEventStatus reads the Event Status register for the specified ***axis***. All of the bits in this status word are set by the motion control IC and cleared by the host. To clear these bits, use the **ResetEventStatus** command. The following table shows the encoding of the data returned by this command.

Name	Bit(s)	Description
—	0	Reserved, may be 0 or 1.
Wrap-around	1	Set to 1 when the actual (encoder) position has wrapped from maximum allowed position to minimum, or vice versa.
—	2	Reserved, may be 0 or 1.
Capture Received	3	Set to 1 when a position capture has occurred.
Motion Error	4	Set to 1 when a motion error has occurred.
—	5-6	Reserved, may be 0 or 1.
Instruction Error	7	Set to 1 when an instruction error has occurred.
Disabled	8	Set to 1 when a “disable” event due to user /Enable line has occurred.
Overtemperature Fault	9	Set to 1 when overtemperature condition has occurred.
Drive Exception	10	An drive event occurred causing output to be disabled. This bit is used on ION products to indicate a bus voltage fault, and with an attached Atlas amplifier to indicate any disabling drive event.
Commutation Error	11	Set to 1 when a commutation error has occurred.
Current Foldback	12	Set to 1 when current foldback has occurred.
Runtime Error	13	Set to 1 when a runtime error occurs. A runtime error is an error condition not directly caused by an erroneous command.
—	14	Set to 1 when breakpoint 2 has been triggered.
—	15	Reserved; not used; may be 0 or 1.

Errors

None

C-Motion API

```
PMDresult PMDGetEventStatus(PMDAxisInterface axis_intf,
                            PMDuint16* status);
```

Script API

```
GetEventStatus
```

C# API

```
UInt16 status = PMDAxis.EventStatus;
```

Visual Basic API

```
UInt16 status = PMDAxis.EventStatus
```

see

GetActivityStatus (p. 40), **GetRuntimeError** (p. 63), **GetSignalStatus** (p. 64), **GetDriveStatus** (p. 48), **GetDriveFaultStatus** (p. 46)

GetFOCValue 5Ah

Motor Types

	Brushless DC	Microstepping	

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
loop	*Direct (D)*	0
	Quadrature (Q)	1
node	*Reference (D,Q)*	0
	Feedback (D,Q)	1
	Error (D,Q)	2
	Integrator Sum (D,Q)	3
	— (Reserved)	4,5
	Output (D,Q)	6
	FOC Output (Alpha,Beta)	7
	Actual Current (A,B)	8
	I^2t Energy	10

Returned data

	Type	Range/Scaling
value	signed 32 bits	see below

Packet Structure

GetFOCValue

15–12	11–8	7–0
0	*axis*	**5A**h

write

15–12	11–8	7–0
0	*loop*	*node*

read

31–16
value (high-order part)

read

15–0
value (low-order part)

Description

GetFOCValue is used to read the value of a ***node*** of the FOC current control. See the product user guide for more information on the location of each ***node*** in the FOC current control algorithm.

Though the data returned is signed 32 bits regardless of the ***node***, the range and format vary depending on the ***node***, as follows:

Node	Range	Scaling	Units
Reference (D,Q)	-2^{15} to $2^{15}-1$	$100/2^{14}$	% max current
Feedback (D,Q)	-2^{15} to $2^{15}-1$	$100/2^{14}$	% max current
Error (D,Q)	-2^{15} to $2^{15}-1$	$100/2^{14}$	% max current
Integrator Contribution (D,Q)	-2^{31} to $2^{31}-1$	$100/2^{14}$	% PWM
Output (D,Q)	-2^{15} to $2^{15}-1$	$100/2^{14}$	% PWM
FOC Output (Alpha,Beta)	-2^{15} to $2^{15}-1$	$100/2^{14}$	% PWM
Actual Current (A,B)	-2^{15} to $2^{15}-1$	$100/2^{14}$	% max current
I^2t Energy	-2^{31} to $2^{31}-1$	$100/2^{30}$	% max energy

Description (cont.)

Most of the nodes have units of % maximum representable current, and most have a scaling of $100/2^{14}$. That is, a value of 2^{14} corresponds to 100% maximum representable current. The maximum representable current is greater than the maximum measureable current by a factor of 1.6.

Nodes labeled "(Alpha, Beta)" reference the non-rotating FOC frame; loop 0 means the alpha component, and loop 1 the beta component.

Nodes labeled "(A, B)" reference the actual motor phases. For one-phase motors the only phase is A, D, or alpha. For two-phase motors phase A is identical with the alpha phase, and phase B is identical with the beta phase. For three-phase motors loop 0 means phase A, and loop 1 means phase B. Phase C current may be computed by noting that the three phase currents must sum to zero.

The script interface combines the loop and node arguments in a single option argument as shown below. For example, if the loop is q (1), and the node is Output (6), then option = 1*256 + 6 = 262.

Errors

Invalid parameter: node is not a supported value.

C-Motion API

```
PMDresult PMDGetFOCValue (PMDAxisInterface axis_intf,
                          PMDuint8 loop,
                          PMDuint8 node,
                          PMDint32* value);
```

Script API

```
GetFOCValue option
where option = loop*256 + node
```

C# API

```
Int32 value = PMDAxis.FOCValue(PMDFOC loop, PMDFOCValueNode node);
```

Visual Basic API

```
Int32 value = PMDAxis.FOCValue(ByVal loop As PMDFOC, ByVal node As
                               PMDFOCValueNode)
```

see

Set/Get Current (p. 106), **Set/GetCurrentControlMode** (p. 108), **Set/GetFOC** (p. 130)

GetInstructionError A5h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

None

Returned data

	Type	Range
error	unsigned 16 bits	0 *to* 35

Packet Structure

GetInstructionError

15 – 8	7 – 0
0	**A5**h

Data

	15 – 8	7 – 0
read	*second error*	*first error*

Description

GetInstructionError returns the code for the first instruction error since the last read operation, and then resets the error to zero (0). Generally, this command is issued only after the instruction error bit in the Event Status register indicates there was an instruction error.

All Juno products will return both the first and second errors after the last read operation. This is especially helpful in debugging initialization commands executed at startup from non-volatile RAM, since the first error is always a Processor reset (1).The error codes are encoded as defined below:

Error Code	Encoding
No error	0
Processor reset	1
Invalid instruction	2
Invalid axis	3
Invalid parameter	4
Trace running	5
— (Reserved)	6
Buffer	7
Trace buffer zero (0)	8
Bad serial checksum	9
— (Reserved)	10
— (Reserved)	11-14
Command invalid in NVRAM mode	15
Invalid operating mode restore after event-triggered change	16
Invalid operating mode for command	17
Invalid register state for command	18
— (Reserved)	19-26
Read-only buffer	27
Command valid only for NVRAM	28
Incorrect data count for command	29
Move in error	30
Wait timed out	31
NVRAM buffer busy	32
Invalid clock signal	33
NVRAM initialization delayed	34
Invalid interface for command	35

Errors

None

C-Motion API

```
PMDresult PMDGetInstructionError (PMDAxisInterface axis_intf,
                                  PMDuint16* error);
```

Script API

```
GetInstructionError
```

C# API

```
UInt16 error = PMDAxis.InstructionError;
```

Visual Basic API

```
UInt16 error = PMDAxis.InstructionError
```

see

GetEventStatus (p. 52), **ResetEventStatus** (p. 82)

GetLoopValue 38h

Motor Types

DC Brush	Brushless DC	Microstepping

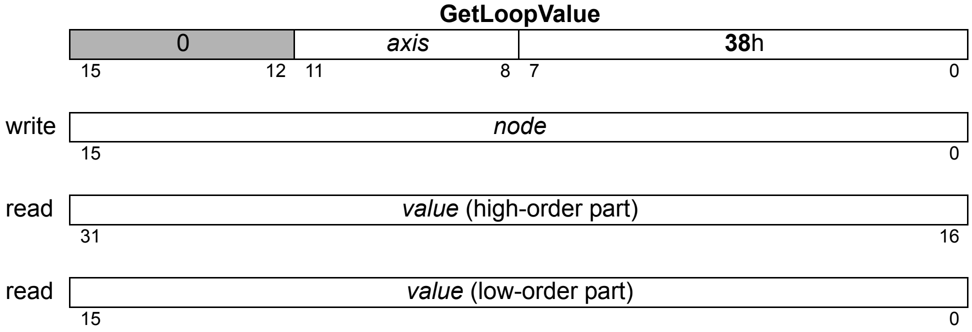
Argument

Name	Instance	Encoding
Node		
	Velocity Loop Reference	0
	Velocity Loop Feedback	1
	Velocity Loop Error	2
	Velocity Loop Integrator Sum	3
	— (Reserved)	4
	Velocity Loop Output	5
	Feedback Biquad Input	6
	Command Biquad Input	7
	— (Reserved)	8-255
	Position Loop Reference	256
	Position Loop Feedback	257
	Position Loop Error	258
	Position Loop Integrator Sum	259
	— (Reserved)	260
	Position Loop Output	261

Returned Data

	Type	Range	Scaling/Units
value	signed 32bits	-2^{31} *to* $2^{31}-1$	see below

Packet Structure



Description

GetLoopValue is used to find the value of a node in either the velocity loop or the position/outer loop. See the *Juno Velocity & Torque Control IC User Guide* for more information on the location of each node in the position loop processing. For the velocity loop, or for the outer loop (analog or SPI feedback to the position/outer loop), all quantities are 16.16 fixed point fractional values. For the position loop the reference and feedback values have units of encoder counts; consult the *Juno Velocity & Torque Control IC User Guide* for the scaling of other loop nodes.

Errors

Invalid parameter: node or loop is not a supported value.

C-Motion API

```
PMDresult PMDGetLoopValue (PMDAxisInterface axis_intf, PMDuint16 node,
                           PMDint32* value);
```

Script API

```
GetLoopValue node
```

C# API

```
Int32 value = PMDAxis.LoopValue(PMDLoop value node);
```

Visual Basic API

```
Int32 value = PMDAxis.LoopValue(ByVal node As PMDLoop value)
```

see

GetPositionError 99h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned data

	Type	Range	Scaling	Units
error	signed 32 bits	-2^{31} *to* $2^{31}-1$	unity	counts microsteps

Packet Structure

GetPositionError

0	*axis*	**99**h
15 12	11 8	7 0

read	*error* (high-order part)
	31 16
read	*error* (low-order part)
	15 0

Description

GetPositionError returns the position error of the specified *axis*. The error is the difference between the actual position (encoder position) and the commanded position (instantaneous output of the trajectory generator). When used with the motor type set to microstepping or pulse & direction, the error is defined as the difference between the encoder position (represented in microsteps or steps) and the commanded position (instantaneous output of the trajectory generator).

C-Motion API

```
PMDresult PMDGetPositionError(PMDAxisInterface axis_intf,
                              PMDint32* error);
```

Script API

```
GetPositionError
```

C# API

```
Int32 error = PMDAxis.PositionError;
```

Visual Basic API

```
Int32 error = PMDAxis.PositionError
```

see

SetLoop (p. 134)

GetProductInfo 1h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
index	*firmware state*	0
	version	1
	product class	2
	checksum	3
	— (Reserved)	4
	part number 3:0	5
	part number 7:4	6
	part number 11:8	7
	part number 15:12	8
	— (Reserved)	9-12
	RAM size	13
	NVRAM size	14
	— (Reserved)	15-256
	boot version	257
	boot product class	258
	boot checksum	259
	boot part number 3:0	261
	boot part number 7:4	262
	boot part number 11:8	263
	boot part number 15:12	264

Returned Data

	Type
value	unsigned 32 bits

Packet Structure

GetProductInfo

	0	*axis*	1h
	15 12	11 8	7 0
write	*index*		
	15		0
read	*value* (high-order part)		
	31		16
read	*value* (low-order part)		
	15		0

Description

GetProductInfo is used to retrieve fixed information about the Juno IC. All data is read in 32-bit units, most of the values are split into fields as explained below.

The *firmware state* is a an enumerated value, 0 means that the normal application firmware is running, and 1 indicates that the boot firmware, which is used for programming NVRAM, is running.

The *version,* and *boot version* consist of four 8-bit bytes, the least significant byte numbered zero. Byte 1 is the firmware major version, byte 0 is the minor version. Byte 2 is a custom code, zero for standard products. Byte 3 is reserved.

Description (cont.)

The *checksum* and *boot checksum* are 32 bit numbers that may be used to verify the identity of a product. The checksum values are documented in product release notes.

The *part number* and *boot part number* are 16 character strings indicating the IC and boot firmware part numbers . There is one ASCII character per 8-bit byte. The first character is stored in the least significant byte of *part number 3:0*, the second character in bits 15:8 of *part number 3:0*. The fourth character is stored in the least significant byte of *part number 7:4*, and so forth. Any unused characters at the end of the string are encoded as zero, ASCII null, but the string may not be null terminated.

The *RAM size* is the number of 32-bit words available for trace RAM.

The NVRAM size is the number of 16-bit words of non-volatile storage available.

GetProductInfo replaces and extends the Magellan commands **GetVersion** and **GetChecksum**. Juno supports GetVersion, but that command always returns zero.

A value of zero returned by **GetVersion** should be taken to mean that **GetProductInfo** is supported.

Errors

Invalid parameter: index is not a supported value.

C-Motion API

```
PMDresult PMDGetProductInfo (PMDAxisInterface axis_intf, PMDuint16 index, PMDuint32* value);
```

Script API

```
GetProductInfo index
```

C# API

```
Int32 value = PMDAxis.GetProductInfo(PMDProductInfo index);
```

Visual Basic API

```
Int32 value = PMDAxis.GetProductInfo(ByVal index As PMDProductInfo)
```

see

NVRAM (p. 72), SetBufferStart (p. 96), SetBufferLength (p. 92), ReadBuffer (p. 76), ReadBuffer16 (p. 77), GetVersion (p. 70)

GetRuntimeError 3Dh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned Data

Type	Range/scaling
unsigned 16 bits	see below

Packet Structure

GetRuntimeError

0	*axis*	**30**h
15 12	11 8	7 0

Data

read	error code
	15 0

Description

GetRuntimeError is used to retrieve an error code describing a runtime error condition, that is, an error not directly caused by an incorrect command. When a runtime error ocurs bit 13 of the event status register is set. This bit may be cleared by using **ResetEventStatus**, merely reading the error code does not clear the event bit.

Currently only two runtime error codes are used by Juno products, 0 means no error, and 5 means an overflow ocurred when multiplying actual or commanded velocity by the velocity scalar.

Errors

None

C-Motion API

```
PMDresult PMDGetRuntimeError (PMDAxisInterface axis_intf, PMDuint16* error);
```

Script API

```
GetRuntimeError
```

C# API

```
PMDRuntimeError error = PMDAxis.RuntimeError;
```

Visual Basic API

```
PMDRuntimeError error = PMDAxis.RuntimeError
```

see

GetEventStatus (p. 52), **ResetEventStatus** (p. 82)

GetSignalStatus A4h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned data

	Type
status	unsigned 16 bits

Packet Structure

GetSignalStatus

0	*axis*	**A4**h
15 … 12	11 … 8	7 … 0

Data

	status
read	15 … 0

Description

GetSignalStatus returns the contents of the Signal Status register for the specified ***axis***. The Signal Status register contains the value of the various hardware signals connected to each axis of the motion control IC. The value read is combined with the Signal Sense register (see **SetSignalSense** (p. 155)) and then returned to the user. For each bit in the Signal Sense register that is set to 1, the corresponding bit in the **GetSignalStatus** command will be inverted. Therefore, a low signal will be read as 1, and a high signal will be read as a 0. Conversely, for each bit in the Signal Sense register that is set to 0, the corresponding bit in the **GetSignalStatus** command is not inverted. Therefore, a low signal will be read as 0, and a high signal will be read as a 1.

All of the bits in the **GetSignalStatus** command are inputs, except FaultOut. The value read for these bits is equal to the value output by the FaultOut mechanism. See **SetFaultMask** (p. 128) for more information. The bit definitions are as follows:

Description	Bit Number
Encoder A	0
Encoder B	1
Encoder Index	2
— (Reserved)	3-6
Hall A	7
Hall B	8
Hall C	9

Description	Bit Number
— (Reserved)	10
Positive Input	11
— (Reserved)	12
/Enable	13
FaultOut	14
Direction Input	15

Errors

None

C-Motion API

```
PMDresult PMDGetSignalStatus(PMDAxisInterface axis_intf,
                             PMDuint16* status);
```

Script API

```
GetSignalStatus
```

C# API

```
UInt16 status = PMDAxis.SignalStatus;
```

Visual Basic API

```
UInt16 status = PMDAxis.SignalStatus
```

see

GetActivityStatus (p. 40), **GetEventStatus** (p. 52), **GetSignalSense** (p. 155)

GetSPIMode 0Bh

Motor Types

DC Brush	Brushless DC	Microstepping

Argument

None

Returned Data

Name	Instance	Encoding
mode	*Host Command*	0
	Direct	8000h

Packet Structure

GetSPIMode

	0		**0B**h
15	8	7	0

Data

read	*mode*	
	15	0

Description

GetSPIMode may be used to determine the mode of the SPI input port. If bit 15 is 0, then the port is in Host Command mode, and can be used for reading state or setting parameters using any of the commands in this section. If bit 15 is 1, then the port is in Direct Input mode, and cannot be used for normal host commands.

In Direct Input mode simple SPI data is written to set the current velocity, torque, or position command, or to set the current outer loop feedback value.

Direct Input mode may be entered by using **SetDriveCommandMode**, or by using **SetLoop** to set the outer loop feedback source.

If no communication channel other than SPI is available then direct input mode may be terminated, and host command mode resumed, by sending three specific 16-bit SPI words in the same packet, eg with only one falling edge and one rising edge of the ~SPIEnable signal. The three words are 55AAh, 33CCh, and 0FF0h. When this message is received, a Drive Exception event will be raised, and bit 4 of the Drive Fault status register set to 1 to indicate an SPI mode change. The action to take in this case is programmable, for example motor output could be disabled, or a smooth stop executed.

C-Motion API

```
PMDresult PMDGetSPIMode(PMDAxisInterface axis_intf, PMDuint16* mode);
```

Script API

```
GetSPIMode
```

C# API

```
PMDSPIMode mode = PMDAxis.SPIMode;
```

Visual Basic API

```
PMDSPIMode mode = PMDAxis.SPIMode
```

see

SetOutputMode (p. 146), **SetDriveCommandMode** (p. 114), **SetLoop** (p. 134), **GetEventStatus** (p. 52), **SetEventAction** (p. 125), **GetDriveFaultStatus** (p. 46)

GetTime 3Eh

Motor Types

DC Brush	Brushless DC	Microstepping

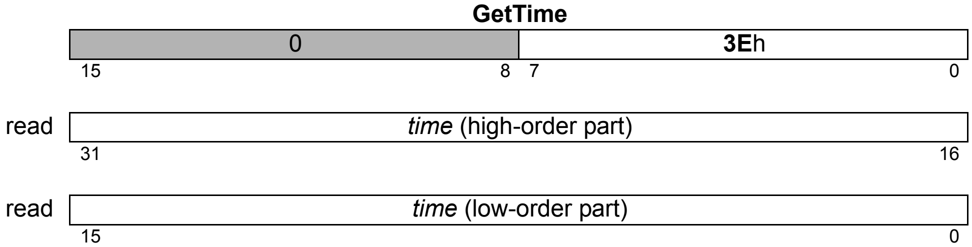
Arguments

None

Returned data

Name	Type	Range	Scaling	Units
time	unsigned 32 bits	0 *to* 2^{32}–1	unity	cycles

Packet Structure



Description

GetTime returns the number of cycles which have occurred since the motion control IC was last reset. The time per cycle is determined by **SetSampleTime**.

Errors

None

C-Motion API

```
PMDresult PMDGetTime(PMDAxisInterface axis_intf,
                     PMDuint32* time);
```

Script API

```
GetTime
```

C# API

```
UInt32 time = PMDAxis.Time;
```

Visual Basic API

```
UInt32 time = PMDAxis.Time
```

see

Set/GetSampleTime (p. 151)

GetTraceCount BBh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

None

Returned data

Name	Type	Range	Scaling	Units
count	unsigned 32 bits	0 *to* 2^{32}–1	unity	samples

Packet Structure

GetTraceCount

0	**BB**h
15 … 8	7 … 0

read	*count* (high-order part)
	31 … 16
read	*count* (low-order part)
	15 … 0

Description

GetTraceCount returns the number of points (variable values) stored in the trace buffer since the beginning of the trace. If the trace mode is rolling buffer than the trace count may include values that have been overwritten.

Errors

None

C-Motion API

```
PMDresult PMDGetTraceCount(PMDAxisInterface axis_intf,
                           PMDuint32* count);
```

Script API

```
GetTraceCount
```

C# API

```
UInt32 count = PMDAxis.TraceCount;
```

Visual Basic API

```
UInt32 count = PMDAxis.TraceCount
```

see

GetTraceStatus (p. 68), **ReadBuffer** (p. 76), **Set/GetBufferLength** (p. 92), **Set/GetTraceMode** (p. 157), **Set/GetTraceStart** (p. 159), **Set/GetTraceStop** (p. 162),

GetTraceStatus BAh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

None

Returned data

Name	Type
status	unsigned 16 bits

Packet Structure

GetTraceStatus

0	**BA**h
15 … 8	7 … 0

Data

read	*status*
	15 … 0

Description

GetTraceStatus returns the trace status. The definitions of the individual status bits are as follows:

Name	Bit Number	Description
Wrap Mode	0	Set to 0 when trace is in one-time mode, 1 when in rolling mode.
Activity	1	Set to 1 when trace is active (currently tracing), 0 if trace not active.
Data Wrap	2	Set to 1 when trace has wrapped, 0 if it has not wrapped. If 0, the buffer has not yet been filled, and all recorded data is intact. If 1, the trace has wrapped to the beginning of the buffer; any previous data may have been overwritten if not explicitly retrieved by the host using the **ReadBuffer** command while the trace is active.
Overrun	3	Set to 0 at trace start, set to 1 if values are overwritten before being read from buffer 1.
NotEmpty	4	Set to 1 only if some values have been written by trace but not yet read from buffer 1, 0 otherwise.
—	5-15	— (Reserved)

Restrictions

Trace Overrun and NotEmpty conditions make sense only if all trace reads are done using buffer 1, but another buffer could be set up to read trace data as well.

Errors

None

C-Motion API

```
PMDresult PMDGetTraceStatus(PMDAxisInterface axis_intf,
                            PMDuint16* status);
```

Script API

```
GetTraceStatus
```

C# API

```
UInt16 status = PMDAxis.TraceStatus;
```

Visual Basic API

```
UInt16 status = PMDAxis.TraceStatus
```

see

Set/GetTraceStart (p. 159), **Set/GetTraceMode** (p. 157)

GetTraceValue 28h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Type	Encoding
variableID	unsigned 8 bit	see below

Returned data

Value	Type	Range/Scaling
	32 bit	see below

Packet Structure

GetTraceValue

	15 … 8	7 … 0
	0	**28**h
write	0	variableID

	31 … 16
read	Value (high order part)

	15 … 0
read	Value (low order part)

Description

GetTraceValue returns a single sample of any trace variable, without using the trace mechanism. The variableID encoding is the same as for **SetTraceVariable**. The use of this command does not change or depend upon any of the trace parameters. The scaling depends on the variableID, and is the same as for trace.

Errors

Invalid parameter: variableID is not a supported value.

C-Motion API

```
PMDresult PMDGetTraceValue(PMDAxisInterface axis_intf,
                           PMDuint8 variable, PMDuint32 *value);
```

Script API

```
GetTraceValue variableID
```

C# API

```
Int32 value = PMDAxis.GetTraceValue(PMDTraceVariable variableID);
```

Visual Basic API

```
Int32 value = PMDAxis.GetTraceValue(ByRef variableID
                                    As PMDTraceVariable)
```

see

SetTraceVariable (p. 164)

GetVersion 8Fh

Motor Types

DC Brush	Brushless DC	Microstepping

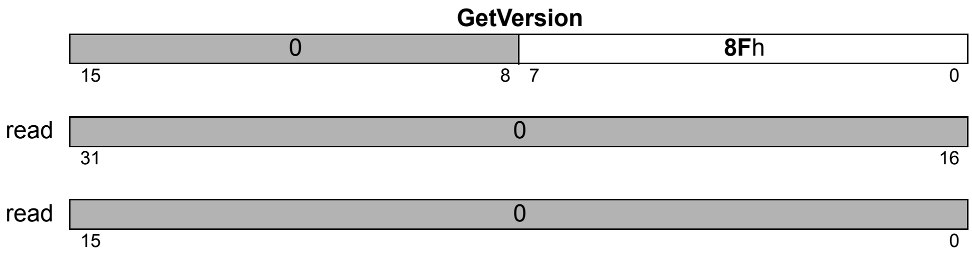
Arguments

None

Returned data

Name	Type
version	unsigned 32 bits

Packet Structure



Description

GetVersion is used in Magellan products to return product information. It is retained in Juno only for backwards compatibility, and always returns zero. The **GetProductInfo** command may be used to read product version and other information.

Errors

None

C-Motion API

```
PMDresult PMDGetVersion(PMDAxisInterface axis_intf,
                        PMDuint16* family,
                        PMDuint16* motorType,
                        PMDuint16* numberAxes,
                        PMDuint16* special_and_chip_count,
                        PMDuint16* custom,
                        PMDuint16* major,
                        PMDuint16* minor);
```

Script API

GetVersion

C# API

```
PMDAxis.GetVersion(ref UInt16 family,
                           Ref PMDMotorTypeVersion MotorType,
                           Ref UInt16 NumberAxes,
                           Ref UInt16 special_and_chip_count,
                           Ref UInt16 custom,
                           Ref UInt16 major,
                           Ref UInt16 minor);
```

Visual Basic API

```
PMDAxis.GetVersion(ByRef family As UInt16,
                           ByRef MotorType As PMDMotorTypeVersion,
                           ByRef NumberAxes As UInt16,
                           ByRef special_and_chip_count As UInt16,
                           ByRef custom As UInt16,
                           ByRef major As UInt16,
                           ByRef minor As UInt16)
```

see

GetProductInfo (p. 61)

InitializePhase 7Ah

Motor Types

	Brushless DC		

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Returned data

None

Packet Structure

InitializePhase

15–12	11–8	7–0
0	*axis*	**7A**h

Description

InitializePhase initializes the phase angle for the specified ***axis*** using the mode (Hall-based or pulse) specified by the **SetPhaseInitializationMode** command.

The Activity Status Phasing Initialized bit is cleared by the InitializePhase command, and set when the initialization process is complete. In the case of pulse phase initialization the Activity Status register may be polled to determine when initialization is complete. The Event Status Commutation Error bit will be set during phase initialization in case an error occurred that might have resulted in incorrect phasing.

In the case of Hall-based phase initialization the Phasing Initialized bit is not set until the motor has moved past a Hall sensor transition. The Commutation Error bit is set and the phase initialization process halted in case an incorrect (all high or all low) Hall state is detected.

Restrictions

Warning: If the phase initialization mode has been set to pulse, then, after this command is sent, the motor may suddenly move in an uncontrolled manner.

Errors

Invalid register state for command: Phase counts less than 4 or less than 4 times phase denominator.
Invalid operating mode for command: Motor output not enabled, or position loop, velocity loop, or command source enabled.

C-Motion API

```
PMDresult PMDInitializePhase(PMDAxisInterface axis_intf);
```

Script API

```
InitializePhase
```

C# API

```
PMDAxis.InitializePhase();
```

Visual Basic API

```
PMDAxis.InitializePhase()
```

see

GetActivityStatus (p. 40), **GetEventStatus** (p. 52), **Set/GetCommutationMode** (p. 102)

NVRAM 30h

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
option	*NVRAM mode*	256
	Erase NVRAM	1
	Write	2
	Block Write Begin	3
	Block Write End	4
	Skip	8

	Type	Range
value	unsigned 16 bit	see below

Packet Structure

NVRAM

15–12	11–8	7–0
0	*axis*	**30**h

	15–0
write	*option*
write	*value*

Description

The **NVRAM** command is used to write the non-volatile RAM (NVRAM) used for initialization. The **NVRAM** command is first used to put the processor to be programmed into NVRAM mode, which supports only the commands necessary for its purpose. Once the processor is in NVRAM mode more **NVRAM** commands are used to erase and re-program NVRAM. NVRAM mode is exited by using the reset command.

Changing to NVRAM mode, erasing, or writing NVRAM data may take more time than the other commands. When programming the MC78113 NVRAM the timeout period should be increased to at least 10 seconds; after each operation fully completes the return status may be read to confirm that the operation succeeded.

The option argument to **NVRAM** specifies the particular operation to perform:

NVRAM mode (256) will put an MC78113 series motion control IC into NVRAM mode. Motor output must be disabled.

The remaining operations will succeed only if either the Juno processor is in NVRAM mode, otherwise an Invalid register state for command error will be raised. The value argument should be zero for this command.

Erase NVRAM (1) will erase the entire non-volatile memory, meaining that all bits will be set. NVRAM must be completely erased before any words may be written. The value argument should be zero for this command.

Write (2) will write a single word of NVRAM, which is specified by the value argument. Words are written in sequence, from the beginning.

Skip (8) may be used to leave the number of words specified in the value argument unwritten, that is, with a value of 0xFFFF. Writing may resume afterwards. It is not necessary to use this command in the usual case.

Description (cont'd)

Block Write Begin (3) and Block Write End (4) may be used to speed up NVRAM operations that are limited by communication bandwidth; their use is not required.

A block write operation is begun by using the **BlockWriteBegin** command, with the number of words that will be sent as a block specified in the value argument. A block may be at most 32 words. No polling procedure is required after a Block Write Begin command.

The next step is to send the data words. These are sent without the usual Magellan command format, therefore no other commands may be sent until the entire block is transmitted.

If using serial communications the words are sent as is, high byte first.

If using CANBus, the words are sent without any additional formatting. At most four words may be sent per CAN packet.

If using SPI communications, the words are sent without any additional formatting. at most four words may be sent for each cycle of the *~HostSPIEnable* signal.

If using parallel communications the words are sent without any additional formatting, with the *~HostWrite* signal high, that is, as though they were command words. At most one word may be sent per *~HostWrite* cycle.

The block write operation is concluded by sending a **BlockWriteEnd** comamnd. The value argument to this command must be the 16-bit ones complement checksum of all words sent since the **BlockWriteBegin** command. If the checksum matches then the processor will write all words to NVRAM, in order. When programming MC58113 NVRAM a long wait may be required. When programming Atlas NVRAM the polling procedure described above for NVRAM writes should be followed.

Restrictions

Once put in NVRAM mode an Atlas amplifier or MC58113 series motion control IC will accept only a restricted set of commands. There is no way to enable motor output, and Atlas will not accept torque commands.

Errors

Invalid parameter: option not supported or value incorrect.
Invalid register state for command: Attempt to call **NVRAM** command from NVRAM.
Invalid register state for command: Attempt to write flash before erasing, or to write past sector end.

C-Motion API

```
PMDresult PMDNVRAM  (PMDAxisInterface axis_intf,
                                   PMDuint16 option,
                                   PMDuint16 value);
```

Script API

```
NVRAM option value
```

C# API

```
PMDAxis.NVRAM(PMDNVRAMOption option, UInt16 value);
```

Visual Basic API

```
PMDAxis.NVRAM(ByRef option As PMDNVRAMOption, ByRef value As UInt16)
```

see

GetDriveStatus (p. 48), **GetEventStatus** (p. 52), **GetInstructionError** (p. 56), **Reset** (p. 78)

NoOperation 00h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	Axis1	0

Returned data

None

Packet Structure

NoOperation

0	*axis*	**00**h
15 12	11 8	7 0

Description

The **NoOperation** command has no effect on the motion control IC. It may be used to verify communication.

Errors

None

C-Motion API

```
PMDresult PMDNoOperation(PMDAxisInterface axis_intf);
```

Script API

```
NoOperation
```

C# API

```
PMDAxis.NoOperation();
```

Visual Basic API

```
PMDAxis.NoOperation()
```

see

ReadAnalog EFh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	Axis1	0

Name	Type	Range	Scaling	Units
portID	unsigned 16 bits	0 *to* 10	unity	-

Returned data

	Type	Range	Scaling	Units
value	unsigned 16 bits	0 *to* 2^{16}-1	$100/2^{16}$	% input

Packet Structure

ReadAnalog

15–12	11–8	7–0
0	*axis*	**EF**h

write

15–4	3–0
0	*portID*

read

15–0
value

Description

ReadAnalog returns a 16-bit value representing the voltage presented to the specified analog input. See the *Juno Velocity & Torque Control IC User Guide* and *MC78113 Electrical Specifications* for more information on analog input and scaling.

Errors

Invalid parameter: portID not supported.

C-Motion API

```
PMDresult PMDReadAnalog(PMDAxisInterface axis_intf, PMDuint16 portID,
                        PMDuint16* value);
```

Script API

```
ReadAnalog portID
```

C# API

```
UInt16 value = PMDAxis.ReadAnalog(Int16 portID);
```

Visual Basic API

```
UInt16 value = PMDAxis.ReadAnalog(ByVal portID As Int16)
```

see

ReadBuffer C9h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Type	Range
bufferID	unsigned 16 bits	0 *to* 7

Returned data

	Type	Range
data	signed 32 bits	-2^{31} *to* $2^{31}-1$

Packet Structure

ReadBuffer

	15–8	7–0
	0	**C9**h

	15–5	4–0
write	0	*bufferID*

	31–16
read	*data* (high-order part)

	15–0
read	*data* (low-order part)

Description

ReadBuffer returns the 32-bit contents of the location pointed to by the read buffer index in the specified buffer. After the contents have been read, the read index is incremented by 1. If the result is equal to the buffer length (set by **SetBufferLength**), the index is reset to zero (0).

Two buffers are used for special purposes: Data is written automatically to Buffer 0 during trace, and the read index of buffer 1 is used to indicate the current NVRAM command executing during initialization. An error is signaled if an attempt is made to read from buffer 0 when trace is active, or to read from buffer 1 when NVRAM initialization is active.

Errors

Invalid parameter: bufferID out of range.
Block out of bounds: Attempt to read from a zero length buffer.
Trace running: Attempt to read buffer 0 when trace is running.
NVRAM buffer busy: Attempt to read buffer 1 when NVRAM initialization is running.
Invalid register state for command: 32 bit read from an NVRAM buffer when read index is odd.

C-Motion API

```
PMDresult PMDReadBuffer(PMDAxisInterface axis_intf, PMDuint16 bufferID,
                        PMDint32* data);
```

Script API

```
ReadBuffer bufferID
```

C# API

```
Int32 data = PMDAxis.ReadBuffer(Int16 BufferId);
```

Visual Basic API

```
Int32 data = PMDAxis.ReadBuffer(ByVal BufferId As Int16)
```

see

Set/GetBufferReadIndex (p. 94), **Set/GetBufferStart** (p. 96), **Set/GetBufferLength** (p. 92)

ReadBuffer16 CDh

Motor Types

DC Brush	Brushless DC	Microstepping

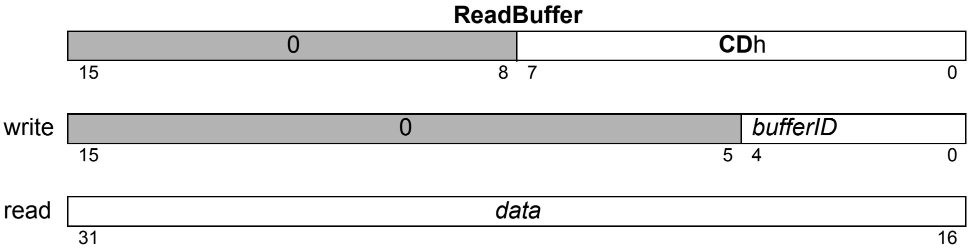
Arguments

Name	Type	Range
bufferID	unsigned 16 bits	0 *to* 7

Returned data

	Type	Range
data	signed 16 bits	-2^{15} *to* $2^{15}-1$

Packet Structure



Description

ReadBuffer16 returns the 16-bit contents of the location pointed to by the read buffer index in the specified buffer. After the contents have been read, the read index is incremented by 1. If the result is equal to the buffer length (set by **SetBufferLength**), the index is reset to zero (0). This command is intended to read from a buffer located in non-volatile RAM, which has a 16-bit word size. ReadBuffer should be used for all other buffers.

Restrictions

This command is only available on products that support non-volatile RAM.

Errors

Invalid parameter: bufferID out of range or attempt to read from a buffer in 32 bit RAM.
Block out of bounds: Attempt to read from a zero length buffer.
NVRAM buffer busy: Attempt to read buffer 1 when NVRAM initialization is running.

C-Motion API

```
PMDresult PMDReadBuffer16(PMDAxisInterface axis_intf,
                          PMDuint16 bufferID, PMDint32* data);
```

Script API

```
ReadBuffer16 bufferID
```

C# API

```
Int16 data = PMDAxis.ReadBuffer16(Int16 BufferId);
```

Visual Basic API

```
Int16 data = PMDAxis.ReadBuffer16(ByVal BufferId As Int16)
```

see

Set/GetBufferReadIndex (p. 94), **WriteBuffer** (p. 176), **Set/GetBufferStart** (p. 96), **Set/GetBufferLength** (p. 92)

Reset 39h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

None

Returned data

None

Packet Structure

Reset

15	8	7	0
0		39h	

Description

Reset restores the motion control IC to its initial condition, setting all motion control IC variables to their default values. Most variables are motor-type independent; however several default values depend upon the configured motor type of the axis. Some of the default values also depend on the state of Magellan pin OutputMode0 when power is applied, if this pin is grounded, Magellan will be in an “Atlas-compatible” state, if it is floating, “backwards-compatible.” MC58113 series products always behave in an Atlas-compatible way. The motor-type independent values are listed here.

	Default Value
Interrupts	
Interrupt Mask	0
Commutation	
Commutation Mode	motor dependent
Phase Angle	0
Phase Counts	motor dependent
Phase Denominator	1
Phase Offset	–1
Phase Initialize Mode	0
Phase Initialize Ramp Time	0
Phase Initialize Negative Pulse Time	0
Phase Initialize Positive Pulse Time	0
Phase Initialize Ramp Command	0
Phase Initialize Pulse Command	0
Phase Correction Mode	motor dependent
Current Control	
Currrent Control Mode	1
FOC Kp (both D and Q loops)	0
FOC Ki (both D and Q loops)	0
FOC Integrator Sum Limit	0
Holding Motor Limit	32767
Step Drive Current	0
Position/Outer Loop	
Position Error Limit	65535
Position Loop Kp	0
Position Loop Ki	0
Position Loop Kd	0
Position Loop Integrator Sum Limit	0
Position Loop Derivative Time	1
Position Loop Kout	65535
Current Limit	32767

Description (cont.)

Position/Outer Loop (cont.)	Default Value
Motor Command	0
Outer Loop Feedback Source	0
Outer Loop Period	1
Outer Loop Output Upper Limit	7FFFFFFFh
Outer Loop Output Lower Limit	-80000000h
Encoder	
Actual Position	0
Actual Position Units	motor dependent
Encoder Source	motor dependent
Encoder To Step Ratio	04000400h
Motor Output	
Operating Mode	0001h
Active Operating Mode	0001h
Output Mode	10
Motor Type	0
PWM Frequency	5000
PWM Limit	16384
PWM Dead Time	16879 *must be changed*
PWM Signal Sense	80FFh
PWM Refresh Period	1
PWM Refresh Time	32767 *must be changed*
PWM Current Sense Time	32767 *must be changed*
Position Servo Loop Control	
Sample Time	102
Profile Generation	
Acceleration	0
Deceleration	0
Profile Mode	1
Start Velocity	0
Velocity Loop	
Velocity Loop Kp	0
Velocity Loop Ki	0
Velocity Loop Integrator Sum Limit	1
Velocity Scalar	0
Velocity Error Limit	7FFFFFFFh
Velocity Feedback Source	0
Deadband Upper Limit	0
Deadband Lower Limit	0
RAM Buffer	
Buffer Length	buffer 0 3072 buffer 1 8192 others 0
Buffer Read Index	0
Buffer Start	buffer 1 20000000h others 0
Buffer Write Index	0

Description (cont.)

	Default Value
Safety	
Motion Error Event Action	4
Current Foldback Event Action	7
OvervoltageThreshold	65535
Undervoltage Threshold	0
OvertemperatureThreshold	32767
FaultOut Mask	0600h
Continuous Current Limit	32768
Energy Limit	32768
Status Registers and AxisOut Indicator	
Signal Sense	0
Traces	
Trace Mode	0
Trace Period	1
Trace Start	0
Trace Stop	0
Trace Variables	all are 0
Trace Trigger Values	all are 0
Miscellaneous	
CAN Mode	C000h (see Notes)
Serial Port Mode	0004h (see Notes)

The motor-type dependent default values are listed in the following tables.

Variable	DC Brush	Brushless DC (3 phase)
Actual Position Units	0	0
Commutation Mode	-	0
Encoder Source	0	0
Phase Correction Mode	-	1
Phase Counts	-	1

Variable	Microstepping (2 phase)
Actual Position Units	1
Commutation Mode	0
Encoder Source	2
Phase Correction Mode	-
Phase Counts	256

Notes

See **Set/GetSampleTime** (p. 151) for more information regarding SampleTime.

Restrictions

Not all of the listed variables are available on all products. See the product user guide.

Errors

No errors. **GetInstructionError** will indicate Parameter Reset error the first time it is called after reset.

C-Motion API	`PMDresult PMDReset(PMDAxisInterface axis_intf);`
Script API	`reset`
C# API	`PMDAxis.Reset ();`
Visual Basic API	`PMDAxis.Reset()`
see	

ResetEventStatus 34h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
mask	*Wrap-around*	FFFDh
	Capture Received	FFF7h
	Motion Error	FFEFh
	Instruction Error	FF7Fh
	Disable	FEFFh
	Overtemperature Fault	FDFFh
	Drive Exception	FBFFh
	Commutation Error	F7FFh
	Current Foldback	EFFFh
	Runtime Error	DFFFh

Returned data

None

Packet Structure

ResetEventStatus

0	*axis*	**34**h
15 12	11 8	7 0

Data

write	*mask*
	15 0

Description

ResetEventStatus clears (sets to 0), for the specified ***axis***, each bit in the Event Status register that has a value of 0 in the ***mask*** sent with this command. All other Event Status register bits (bits that have a mask value of 1) are unaffected.

Events that cause changes in operating mode or trajectory require, in general, that the corresponding bit in Event Status be cleared prior to returning to operation. That is, prior to restoring the operating mode (in cases where the event caused a change in it) or prior to performing another trajectory move (in cases where the event caused a trajectory stop). The one exception to this is *Motion Error*, which is not required to be cleared if the event action for it includes disabling of the position or velocity loops.

Restrictions

Not all bits in **ResetEventStatus** are supported in some products. See the product user guide.

Errors

None

C-Motion API

```
PMDresult PMDResetEventStatus(PMDAxisInterface axis_intf,
                              PMDuint16 status);
```

Script API

```
ResetEventStatus mask
```

C# API

```
PMDAxis.ResetEventStatus(UInt16 mask);
```

isual Basic API

```
PMDAxis.ResetEventStatus(ByVal mask As UInt16)
```

see

GetEventStatus (p. 52)

RestoreOperatingMode 2Eh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

Packet Structure

RestoreOperatingMode

0	*axis*	**2E**h
15 12	11 8	7 0

Description

RestoreOperatingMode is used to command the ***axis*** to return to its static operating mode. It should be used when the active operating mode has changed due to actions taken from safety events or other programmed events. Calling **RestoreOperatingMode** will re-enable all loops that were disabled as a result of events.

Restrictions

Before using **RestoreOperatingMode** to return to the static operating mode, the event status bits should all be cleared. If a bit in event status that caused a change in operating mode is not cleared, this command will return an error. An exception to this is Motion Error, which does not have to be cleared prior to restoring the operating mode.

Though **RestoreOperatingMode** will re-enable the profile generator (if it was disabled as a result of an event action), it will not resume a move. This must be done using **SetVelocity**.

If the current command source is analog or SPI instead of the trajectory generator then motion may resume immediately. The external command source may have to be managed to avoid any problems.

Errors

Invalid operating mode restore after event triggered change.

C-Motion API

```
PMDresult PMDRestoreOperatingMode(PMDAxisInterface axis_intf);
```

Script API

```
RestoreOperatingMode
```

C# API

```
PMDAxis.RestoreOperatingMode();
```

Visual Basic API

```
PMDAxis.RestoreOperatingMode()
```

see

GetActiveOperatingMode (p. 38), **Set/GetOperatingMode** (p. 144), **Set/GetEventAction** (p. 125)

SetAcceleration 90h
GetAcceleration 4Ch

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0

	Type	Range	Scaling	Units
acceleration	unsigned 32 bits	0 *to* 2^{31}–1	$1/2^{8}$	counts/cycle^2 microsteps/cycle^2

Packet Structure

SetAcceleration

0	*axis*	**90**h
15 — 12	11 — 8	7 — 0

write	*acceleration* (high-order part)
	31 — 16
write	*acceleration* (low-order part)
	15 — 0

GetAcceleration

0	*axis*	**4C**h
15 — 12	11 — 8	7 — 0

read	*acceleration* (high-order part)
	31 — 16
read	*acceleration* (low-order part)
	15 — 0

Description

SetAcceleration loads the maximum acceleration buffer register for the specified *axis*. This command is used with the internal profile generator.

SetAcceleration may also be used to specify the maximum acceleration used during a smooth stop when the command mode is analog or SPI.

GetAcceleration reads the maximum acceleration buffer register.

Scaling example: To load a value of 1.750 counts/cycle^2, multiply by 2^{24} (giving 29,360,128) and load the resultant number as a 32-bit number, giving 01C0h in the high word and 0200h in the low word. Values returned by **GetAcceleration** must correspondingly be divided by 2^{24} to convert to units of counts/cycle^2 or steps/cycle^2.

Errors

Invalid Parameter: A negative acceleration was supplied.

C-Motion API

```
PMDresult PMDSetAcceleration(PMDAxisInterface axis_intf,
                             PMDuint32 acceleration);
PMDresult PMDGetAcceleration(PMDAxisInterface axis_intf,
                             PMDuint32* acceleration);
```

SetAcceleration (cont.) 90h
GetAcceleration 4Ch

Script API

```
GetAcceleration
SetAcceleration acceleration
```

C# API

```
UInt32 acceleration = PMDAxis.Acceleration;
PMDAxis.Acceleration = acceleration;
```

Visual Basic API

```
UInt32 acceleration = PMDAxis.Acceleration
PMDAxis.Acceleration = acceleration
```

see

Set/GetDeceleration (p. 113), **Set/GetVelocity** (p. 174)

SetActualPosition 4Dh
GetActualPosition 37h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding		
axis	*Axis1*	0		
	Type	**Range**	**Scaling**	**Units**
position	signed 32 bits	-2^{31} *to* $2^{31}-1$	unity	counts microsteps

Packet Structure

SetActualPosition

	0	*axis*	**4D**h
	15 12	11 8	7 0
write	*position* (high-order part)		
	31 16		
write	*position* (low-order part)		
	15 0		

GetActualPosition

	0	*axis*	**37**h
	15 12	11 8	7 0
read	*position* (high-order part)		
	31 16		
read	*position* (low-order part)		
	15 0		

Description

SetActualPosition loads the position register (encoder position) for the specified ***axis***. At the same time, the commanded position is replaced by the loaded value minus the position error. This prevents a servo "bump" when the new axis position is established. In effect, this instruction establishes a new reference position from which subsequent positions can be calculated. It is commonly used to set a known reference position after a homing procedure.

Note: For axes configured as microstepping motor types, actual position units determines if the position is specified and returned in units of counts or steps.

GetActualPosition reads the contents of the encoder's actual position register. This value will be accurate to within one cycle (as determined by **Set/GetSampleTime**).

Errors

None

C-Motion API

```
PMDresult PMDSetActualPosition(PMDAxisInterface axis_intf,
                               PMDint32 position);
PMDresult PMDGetActualPosition(PMDAxisInterface axis_intf,
                               PMDint32* position);
```

Script API

```
GetActualPosition
SetActualPosition position
```

C# API

```
Int32 position = PMDAxis.ActualPosition;
PMDAxis.ActualPosition = position;
```

Visual Basic API

```
Int32 position = PMDAxis.ActualPosition
PMDAxis.ActualPosition = position
```

see

GetPositionError (p. 60), **Set/GetActualPositionUnits** (p. 87), **AdjustActualPosition** (p. 30)

SetActualPositionUnits BEh
GetActualPositionUnits BFh

Motor Types

▨	▨	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
mode	*Counts*	0
	Steps	1

Packet Structure

SetActualPositionUnits

0	*axis*	**BE**h
15 … 12	11 … 8	7 … 0

Data

write

0	*mode*
15 … 1	0

GetActualPositionUnits

0	*axis*	**BF**h
15 … 12	11 … 8	7 … 0

Data

read

0	*mode*
15 … 1	0

Description

SetActualPositionUnits determines the units used by the **Set/GetActualPosition**, **AdjustActualPosition** and **GetCaptureValue** for the specified *axis*. It also affects the trace variable Actual Position. When set to *Counts*, position units are in encoder counts. When set to *Steps*, position units are in microsteps. The step position is calculated using the ratio as set by the **SetEncoderToStepRatio** command.

GetActualPositionUnits returns the position units for the specified *axis*.

Restrictions

The trace variable, capture value, is not affected by this command. The value is always in counts.

Errors

Invalid Parameters: mode other than 0 or 1.

C-Motion API

```
PMDresult PMDSetActualPositionUnits(PMDAxisInterface axis_intf,
                                    PMDuint16 mode);
PMDresult PMDGetActualPositionUnits(PMDAxisInterface axis_intf,
                                    PMDuint16* mode);
```

Script API

```
GetActualPositionUnits
SetActualPositionUnits mode
```

C# API

```
PMDActualPositionUnits mode = PMDAxis.ActualPositionUnits;
PMDAxis.ActualPositionUnits = mode;
```

Visual Basic API

```
PMDActualPositionUnits mode = PMDAxis.ActualPositionUnits
PMDAxis.ActualPositionUnits = mode
```

see

Set/GetActualPosition (p. 86), **Set/GetEncoderToStepRatio** (p. 123), **AdjustActualPosition** (p. 30), **GetCaptureValue** (p. 42), **Set/GetTraceVariable** (p. 164)

SetAnalogCalibration 29h
GetAnalogCalibration 2Ah

Motor Types

DC Brush	Brushless DC	Microstepping	▨

Arguments

Name	Instance	Encoding		
axis	*Axis1*	0		
channel	*current leg A offset*	0		
	current leg B *offset*	1		
	current leg C offset	2		
	current leg D offset	3		
	Analog command offset	7		
	Tachometer offset	8		
	Analog command gain	0x207		
	Type	**Range**	**Scaling**	**Units**
offset	signed 16 bits	-2^{15} *to* 2^{15}-1	$100/28^{16}$	% input
gain	unsigned 15 bits	0 *to* 32767	$1/2^{15}$	dimensionless

Packet Structure

SetAnalogCalibration

▨	*axis*	**29**h
15 12	11 8	7 0

write	*channel*
	15 0

write	*offset or gain*
	15 0

GetAnalogCalibration

▨	*axis*	**2A**h
15 12	11 8	7 0

write	*channel*
	15 0

read	*offset or gain*
	15 0

Description

The **SetAnalogCalibration** command sets the offset applied to the specified analog input channel, to compensate for the vagaries of external amplification circuitry. The offset is subtracted from the raw analog reading, as returned by the **ReadAnalog** command, before any scaling is applied.

It is frequently more convenient to use the **CalibrateAnalog** command than to compute the apropriate offsets.

SetAnalogCalibration may also be used to set the gain associated with the analog command channel. The gain is applied to the analog command signal after the offset, and may be used to scale the command appropriately for an application. By default the the analog command gain is 50% (16384), which is frequently reasonable for velocity control.

GetAnalogCalibration retrieves the values set by **SetAnalogCalibration**.

Errors

C-Motion API

```
PMDresult PMDSetAnalogCalibration(PMDAxisInterface axis_intf,
                     PMDuint16 channel,
                     PMDint16 offset);
PMDresult PMDGetAnalogCalibration(PMDAxisInterface axis_intf,
                     PMDuint16 channel,
                     PMDint16 *offset);
```

Script API

```
GetAnalogCalibration channel
SetAnalogCalibration channel offset
```

C# API

```
Int16 offset = PMDAxis.GetAnalogCalibration(UInt16 channel);
PMDAxis.SetAnalogCalibration(UInt16 channel, Int16 offset);
```

Visual Basic API

```
Int16 offset = PMDAxis.SetAnalogCalibration(UInt16 channel)
PMDAxis.SetAnalogCalibration(Uint16 channel, Int16 offset)
```

see

ReadAnalog (p. 75), **CalibrateAnalog** (p. 31)

SetTraceTriggerValue D6h
GetTraceTriggerValue D7h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
ID	*start*	256
	stop	257
	stop delay	258

value (see below)

Packet Structure

SetTraceTrigger

0	*axis*	**D6**h
15 12	11 8	7 0

write	*ID*
	15 0

write	*value* (high-order part)
	31 16

write	*value* (low-order part)
	15 0

GetTraceTrigger

0	*axis*	**D7**h
15 12	11 8	7 0

write	*ID*
	15 0

read	*value* (high-order part)
	31 16

read	*value* (low-order part)
	15 0

Description

SetTraceTriggerValue sets the comparison trigger value for some trace start or stop conditions.

Not all trace start/stop conditions require a value.

The *value* parameter is interpreted according to the trigger condition for trace start or stop; see **SetTraceStart**. The data format for each trigger condition is as follows**:**

The *value* parameter is interpreted according to the trigger condition for the selected ID; see **SetTraceStart** (p. 159). The data format for each trigger condition is as follows:

Trace Trigger	Value Type	Range	Units
Signed greater than trace value	signed 32-bit	-2^{31} to $2^{31}-1$	same as trace value
Signed less than trace value	signed 32-bit	-2^{31} to $2^{31}-1$	same as trace value
Unsigned higher than trace value	unsigned 32-bit	0 to $2^{32}-1$	same as trace value
Unsigned lower than trace value	unsigned 32-bit	0 to $2^{32}-1$	same as trace value
Bitwise match for trace value	2 word mask	-	boolean status values

Description (cont.)

For the bitwise match condition, the high order part of value is the selection mask, and the low-order part is the sense mask. The condition will trigger when the bitwise logical AND of the selection mask with the lower 16 bits of the trace value is equal to the sense mask.

For example, to trigger a trace start when both the Hall A and Hall B signals are high and the Hall C signal is low, set the trace start value to 03800180h, set the first trace variable to the signal status register (14), then set the trace start condition to bitwise match (11).

SetTraceTriggerValue is also used to set the number of trace periods between the time the trace stop condition is satisfied and the time trace actually stops. This delay allows collecting trace data after the point of interest identified by the trace stop condition. The maximum delay is 65536. The delay register is set to zero during a trace stop; the delay value must be set each time.

GetTraceTriggerValue returns any of the values set by **SetTraceTriggerValue**. Each value will be used for only one trigger, the value must be set again before the condition will trigger.

Restrictions

Always load the breakpoint comparison value (**SetTraceTriggerValue** command) before setting a new breakpoint condition (**SetTraceStart**, **SetTraceStop** command). Failure to do so will likely result in unexpected behavior.

Errors

Invalid Parameter: ID not supported.

C-Motion API

```
PMDresult PMDSetTraceTrigger(PMDAxisInterface axis_intf,
                                 PMDuint16 breakpointID,
                                 PMDint32 value);
PMDresult PMDGetTraceTrigger(PMDAxisInterface axis_intf,
                                 PMDuint16 breakpointID,
                                 PMDint32* value);
```

Script API

```
GetTraceTriggerValue ID
SetTraceTriggerValue ID value
```

C# API

```
Int32 value = PMDAxis.GetTraceTriggerValue(PMDTraceTriggerID ID);
PMDAxis.SetTraceTriggerValue(PMDTraceTriggerID ID, Int32 value);
```

Visual Basic API

```
Int32 value = PMDAxis.GetTraceTriggerValue(PMDTraceTriggerID ID)
PMDAxis.SetTraceTriggerValue(ByVal ID As PMDTraceTriggerID, ByVal value As
Int32)
```

see

SetBufferLength C2h
GetBufferLength C3h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Type	Range
bufferID	unsigned 16 bits	0 *to* 7
length	unsigned 32 bits	1 *to* $2^{30} - 1$

Packet Structure

SetBufferLength

	0	**C2**h
	15 8	7 0
write	0	*bufferID*
	15 5	4 0
write	*length* (high-order part)	
	31 16	
write	*length* (low-order part)	
	15 0	

GetBufferLength

	0	**C3**h
	15 8	7 0
write	0	*bufferID*
	15 5	4 0
read	*length* (high-order part)	
	31 16	
read	*length* (low-order part)	
	15 0	

Description

SetBufferLength sets the *length*, in numbers of 32-bit elements, of the buffer in the memory block identified by *bufferID*. For buffers pointing to non-volatile RAM, the length should be specified in 16-bit words.

Note: The **SetBufferLength** command resets the buffers read and write indexes to 0.

The **GetBufferLength** command returns the *length* of the specified buffer.

Restrictions

The buffer length plus the buffer start address cannot exceed the memory size of the product. See the product user guide.

Errors

Invalid Parameter: bufferID not supported, or length out of range.
Trace Running: Attempt to set length of buffer 0 when trace is running.
NVRAM buffer busy: Attempt to set length of buffer 1 before NVRAM initialization is complete.

C-Motion API

```
PMDresult PMDSetBufferLength(PMDAxisInterface axis_intf,
                             PMDuint16 bufferID, PMDuint32 length);
PMDresult PMDGetBufferLength(PMDAxisInterface axis_intf,
                             PMDuint16 bufferID, PMDuint32* length);
```

SetBufferLength (cont.) C2h
GetBufferLength C3h

Script API

```
GetBufferLength bufferID
SetBufferLength bufferID length
```

C# API

```
Int32 length = PMDAxis.GetBufferLength(Int16 BufferId);
PMDAxis.SetBufferLength(Int16 BufferId, Int32 length);
```

Visual Basic API

```
Int32 length = PMDAxis.GetBufferLength(ByVal BufferId As Int16)
PMDAxis.SetBufferLength(ByVal BufferId As Int16, ByVal length As Int32)
```

see

Set/GetBufferReadIndex (p. 94), **Set/GetBufferStart** (p. 96), **Set/GetBufferWriteIndex** (p. 98)

SetBufferReadIndex C6h
GetBufferReadIndex C7h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Type	Range	Scaling	Units
bufferID	unsigned 16 bits	0 *to* 7	unity	-
index	unsigned 32 bits	0 *to* buffer length - 1	unity	double words

Packet Structure

SetBufferReadIndex

	0	**C6**h
	15 8	7 0
write	0	*bufferID*
	15 5	4 0
write	*index* (high-order part)	
	31 16	
write	*index* (low-order part)	
	15 0	

GetBufferReadIndex

	0	**C7**h
	15 8	7 0
write	0	*bufferID*
	15 5	4 0
read	*index* (high-order part)	
	31 16	
read	*index* (low-order part)	
	15 0	

Description

SetBufferReadIndex sets the address of the read *index* for the specified **bufferID**. For buffers pointing to non-volatile RAM, the read index should be specified in 16-bit words.

GetBufferReadIndex returns the current read *index* for the specified **bufferID**.

Restrictions

If the read index is set to an address beyond the length of the buffer, the command will not be executed and will return host I/O error code 7, buffer bound exceeded.

Errors

Invalid Parameter: bufferID not supported.
Block out of bounds: index greater than or equal to buffer length.
Trace Running: Attempt to set read index of buffer 0 when trace is running.
NVRAM buffer busy: Attempt to set read index of buffer 1 before NVRAM initialization is complete.

C-Motion API

```
PMDresult PMDSetBufferReadIndex(PMDAxisInterface axis_intf,
                                PMDuint16 bufferID,
                                PMDuint32 index);
PMDresult PMDGetBufferReadIndex(PMDAxisInterface axis_intf,
                                PMDuint16 bufferID,
                                PMDuint32* index);
```

Script API

```
GetBufferReadIndex bufferID
SetBufferReadIndex bufferID index
```

C# API

```
Int32 index = PMDAxis.GetBufferReadIndex(Int16 BufferId);
PMDAxis.SetBufferReadIndex(Int16 BufferId, Int32 index length);
```

Visual Basic API

```
Int32 index = PMDAxis.GetBufferReadIndex(ByVal BufferId As Int16)
PMDAxis.SetBufferReadIndex(ByVal BufferId As Int16, ByVal index As Int32)
```

see

Set/GetBufferLength (p. 92), **Set/GetBufferStart** (p. 96), **Set/GetBufferWriteIndex** (p. 98)

SetBufferStart C0h
GetBufferStart C1h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Type	Range	Units
bufferID	unsigned 16 bits	0 *to* 7	-
address	unsigned 32 bits	0 *to* $2^{31} - 1$	double words

Packet Structure

SetBufferStart

	0	**C0**h
	15 8	7 0
write	0	*bufferID*
	15 5	4 0
write	*address* (high-order part)	
	31 16	
write	*address* (low-order part)	
	15 0	

GetBufferStart

	0	**C1**h
	15 8	7 0
write	0	*bufferID*
	15 5	4 0
read	*address* (high-order part)	
	31 16	
read	*address* (low-order part)	
	15 0	

Description

SetBufferStart sets the starting *address* for the specified buffer, in double-words, of the buffer in the memory block identified by ***bufferID.*** In products with non-volatile RAM (NVRAM), the address range beginning at 20000000h is used for NVRAM. Buffers pointing to NVRAM use a word size of 16 bits, unlike buffers pointing to DRAM, which use a word size of 32 bits. For NVRAM buffers the start should be specified in 16-bit words pluse 20000000h.

Note: The **SetBufferStart** command resets the buffers read and write indexes to 0.

The **GetBufferStart** command returns the starting *address* for the specified *bufferID*.

Restrictions

The buffer start address plus the buffer length cannot exceed the memory size of the product. See the product user guide.

Errors

Invalid Parameter: bufferID not supported, start address not in RAM or NVRAM, or start address plus length out of bounds.
Trace Running: Attempt to set starting address of buffer 0 when trace is running.
NVRAM buffer busy: Attempt to set starting address of buffer 1 before NVRAM initialization is complete.

SetBufferStart (cont.) C0h
GetBufferStart C1h

C-Motion API

```
PMDresult PMDSetBufferStart(PMDAxisInterface axis_intf,
                            PMDuint16 bufferID, PMDuint32 address);
PMDresult PMDGetBufferStart(PMDAxisInterface axis_intf,
                            PMDuint16 bufferID, PMDuint32* address);
```

Script API

```
GetBufferStart bufferID
SetBufferStart bufferID address
```

C# API

```
Int32 address = PMDAxis.GetBufferStart(Int16 BufferId);
PMDAxis.SetBufferStart(Int16 BufferId, Int32 address);
```

Visual Basic API

```
Int32 address = PMDAxis.GetBufferStart(ByVal BufferId As Int16)
PMDAxis.SetBufferStart(ByVal BufferId As Int16, ByVal address As Int32)
```

see

Set/GetBufferLength (p. 92), **Set/GetBufferReadIndex** (p. 94), **Set/GetBufferWriteIndex** (p. 98)

SetBufferWriteIndex C4h
GetBufferWriteIndex C5h

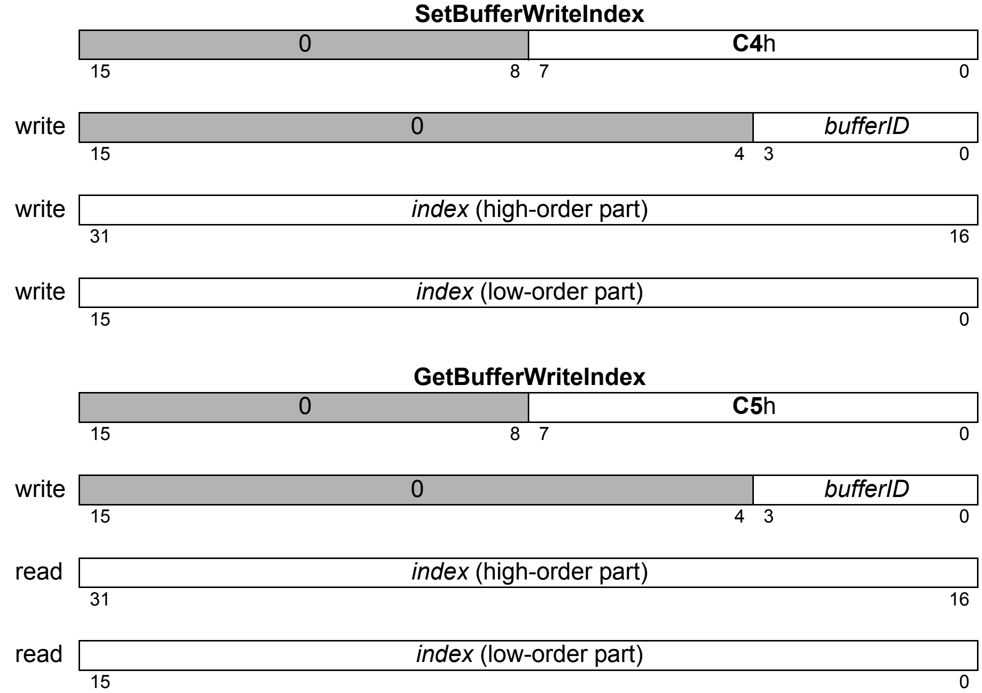
Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Type	Range	Scaling	Units
bufferID	unsigned 16 bits	0 *to* 7	unity	-
index	unsigned 32 bits	0 *to* buffer length - 1	unity	double words

Packet Structure



Description

SetBufferWriteIndex sets the write *index* for the specified *bufferID*. For buffers pointing to non-volatile RAM, the write index should be specified in 16-bit words.

GetBufferWriteIndex returns the write *index* for the specified *bufferID*.

Errors

Invalid Parameter: bufferID not supported.
Block out of bounds: index greater than or equal to buffer length.
Trace Running: Attempt to set write index of buffer 0 when trace is running.

C-Motion API

```
PMDresult PMDSetBufferWriteIndex(PMDAxisInterface axis_intf,
                                 PMDuint16 bufferID, PMDuint32 index);
PMDresult PMDGetBufferWriteIndex(PMDAxisInterface axis_intf,
                                 PMDuint16 bufferID, PMDuint32* index);
```

Script API

```
GetBufferWriteIndex bufferID
SetBufferWriteIndex bufferID index
```

SetBufferWriteIndex (cont.) C4h
GetBufferWriteIndex C5h

C# API

```
Int32 index = PMDAxis.GetBufferWriteIndex(Int16 BufferId);
PMDAxis.SetBufferWriteIndex(Int16 BufferId, Int32 index length);
```

Visual Basic API

```
Int32 index = PMDAxis.GetBufferWriteIndex(ByVal BufferId As Int16)
PMDAxis.SetBufferWriteIndex(ByVal BufferId As Int16,
                            ByVal index As Int32)
```

see

Set/GetBufferLength (p. 92), **Set/GetBufferReadIndex** (p. 94), **Set/GetBufferStart** (p. 96)

SetCANMode 12h
GetCANMode 15h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Type	Encoding
mode	unsigned 16 bits	see below

Packet Structure

SetCANMode

0	**12**h	
15 8	7 0	

Data

write	*transmission rate*	0	*nodeID*
	15 13	12 7	6 0

GetCANMode

0	**15**h
15 8	7 0

Data

read	*transmission rate*	0	*nodeID*
	15 13	12 7	6 0

Description

SetCANMode sets the CAN 2.0B communication parameters for the motion control IC. After completion of this command, the motion control IC will respond to a CAN receive message addressed to 600h + nodeID. CAN responses are sent to 580h + nodeID. The CAN transmission rate will be as specified in the ***transmission rate*** parameter. Note that when this command is used to change to a new nodeID, the command response (for this command) will be sent to the new nodeID. The following table shows the encoding of the data used by this command.

The script interface combines the nodeID and transmission rate arguments into a single mode argument as shown below. For example, if the nodeID is 3, and the transmission rate is 500,000 baud (2), then option = 2*8192 + 3 = 16387.

Bits	Name	Instance	Encoding
0–6	CAN NodeID	*Address 0*	0
		Address 1	1
		...	...
		Address 127	127
7–12	— (Reserved)		
13–15	Transmission Rate	*1,000,000 baud*	0
		Reserved	1
		500,000	2
		250,000	3
		125,000	4
		50,000	5
		20,000	6
		10,000	7

Errors

Invalid Parameter: Transmission rate code not supported.

C-Motion API

```
PMDresult PMDSetCANMode(PMDAxisHandle axis_handle, PMDuint8 nodeID,
                        PMDuint8 transmission_rate);
PMDresult PMDGetCANMode(PMDAxisHandle axis_handle, PMDuint8* nodeID,
                        PMDuint8* transmission_rate);
```

Script API

```
GetCANMode
SetCANMode mode
where mode = transmissionRate*8192 + nodeID
```

C# API

```
PMDAxis.GetCANMode(ref byte NodeId, ref PMDCANBaud TransmissionRate);
PMDAxis.SetCANMode(byte NodeId, PMDCANBaud TransmissionRate);
```

Visual Basic API

```
PMDAxis.GetCANMode(ByRef NodeId As Byte,
                   ByRef TransmissionRate As PMDCANBaud)
PMDAxis.SetCANMode(ByVal NodeId As Byte,
                   ByVal TransmissionRate As PMDCANBaud)
```

see

SetCommutationMode E2h
GetCommutationMode E3h

Motor Types

	Brushless DC		

Arguments

Name	**Instance**	**Encoding**
axis	*Axis1*	0
mode	*Sinusoidal*	0
	Hall-based	1

Packet Structure

SetCommutationMode

0	*axis*	**E2**h
15 12	11 8	7 0

Data

write	0	*mode*
	31 1	0

GetCommutationMode

0	*axis*	**E3**h
15 12	11 8	7 0

Data

read	0	*mode*
	31 1	0

Description

SetCommutationMode sets the phase commutation *mode* for the specified *axis.*

When set to *Sinusoidal*, as the motor turns, the encoder input signals are used to calculate the phase angle. This angle is in turn used to generate sinusoidally varying outputs to each motor winding.

When set to *Hall-based,* the Hall effect sensor inputs are used to commutate the motor windings using a "six-step" or "trapezoidal" waveform method.

When using FOC current control, this command is used to define the method used for motor phase determination.

GetCommutationMode returns the value of the commutation mode.

Errors

Invalid Parameter: Mode code not supported.

C-Motion API

```
PMDresult PMDSetCommutationMode(PMDAxisInterface axis_intf,
                                PMDuint16 mode);
PMDresult PMDGetCommutationMode(PMDAxisInterface axis_intf,
                                PMDuint16* mode);
```

Script API

```
GetCommutationMode
SetCommutationMode mode
```

C# API

```
PMDCommutationMode mode = PMDAxis.CommutationMode;
PMDAxis.CommutationMode = mode;
```

Visual Basic API

```
PMDCommutationMode mode = PMDAxis.CommutationMode
PMDAxis.CommutationMode = mode
```

see

SetCommutationParameter 63h
GetCommutationParameter 64h

Motor Types

▨	**Brushless DC**	**Microstepping**

Arguments

Name	**Instance**	**Encoding**	
axis	Axis1	0	
parameter	phase counts	0	
	phase angle	1	
	phase offset	2	
	phase denominator	3	
	Type	**Range**	**Scaling/Units**
value	unsigned 32-bits	0 to 2^{31}-1	counts

Packet Structure

SetCommutationParameter

0	*axis*	33h
15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0

write	*parameter*
	15 0
write	*value* (high-order part)
	15 0
write	*value* (low-order part)
	15 0

GetCommutationParameter

0	*axis*	64h
15 12	11 8	7 0

write	*parameter*
	15 0
read	*value* (high-order part)
	15 0
read	*value* (low-order part)
	15 0

Description

SetCommutationParameter is used to set several 32-bit quantities used for motor commutation or microstep generation.

For brushless DC motors, the PhaseCounts and PhaseDenominator registers specify the number of encoder counts per electrical revolution. If this number is an integer, PhaseDenominator may be left at its default value of 1, and PhaseCounts set to the counts per electrical revolution. Alternatively, PhaseDenominator may be set to the number of motor pole pairs, and PhaseCounts to the number of encoder counts per mechanical revolution.

For example, for a six pole motor using an encoder with 1024 counts per revolution there are 341 1/3 encoder counts per electrical revolution, PhaseCounts may be set to 1024, and PhaseDenominator to 3.

PhaseAngle and PhaseOffset are both values that may be set by command but are normally altered by the commutation process. PhaseAngle gives the current position in the electrical cycle; to convert to degrees divide PhaseAngle by PhaseCounts and multiply by 360. For example, for the motor in the example above, a PhaseAngle of 256 corresponds to an angle of (256/1024)*360 = 90 degrees.

PhaseOffset is the non-negative offset from the index mark to the internal zero phase angle. Setting PhaseOffset has no immediate effect, but, if phase correction is enabled, sets the phase angle when an index pulse is detected. The default value of PhaseOffset is -1, which means that at the first index pulse the PhaseOffset should be set equal to the current phase angle. If phase initialization is correctly set up it is normally not necessary to set PhaseOffset.PhaseOffset may be read to determine whether an index pulse has been detected since phase initialization.

Setting the PhaseAngle has the side-effect of setting PhaseOffset to the default value of -1.

The maximum value for PhaseOffset is 2^{31}- 1, any value with bit 31 set is interpreted as negative, and equivalent to -1. If set by command PhaseOffset should be less than PhaseCounts, but that condition is not checked.

For microstep motors PhaseCounts sets the number of microsteps per electrical revolution, and PhaseAngle the current position in the electrical cycle. Each electrical revolution is four full steps. The maximum supported value is 1024 microsteps per electrical revolution, or 256 microsteps per full step. The PhaseDenominator parameter is ignored for microstep motors.

For microstep motors PhaseOffset, which is zero by default, specifies an offset to be added to PhaseAngle to produce the current electrical phase angle. 08000h corresponds to 360 degrees for PhaseOffset.

To obtain traditional full-stepping both phases are always driven at full output, either positive or negative, set PhaseCounts to 4, and set Offset to 01000h or 45 degrees.

The minimum value for PhaseCounts, for either step or BLDC motors, is 4. The minimum value for PhaseDenominator is 1, and the maximum possible value is 32767. For proper commutation PhaseCounts must be greater than PhaseDenominator, although that condition is not checked.

Errors

Invalid Parameter: Unrecognized parameter or value out of bounds.

C-Motion API

```
PMDresult PMDGetCommutationParameter (PMDAxisInterface axis_intf,
                                      PMDuint16 parameter,
                                      PMDint32* value);
PMDresult PMDSetCommutationParameter (PMDAxisInterface axis_intf,
                                      PMDuint16 parameter,
                                      PMDint32 value);
```

Script API

```
GetCommutationParameter parameter
SetCommutationParameter paramter value
```

C# API

```
Int32 value = PMDAxis.GetCommutationParameter(PMDCommutationParameter
                                               parameter);
PMDAxis.SetCommutationParameter(PMDCommutationParameter parameter,
                                Int32 value);
```

Visual Basic API

```
Int32 value = PMDAxis.GetCommutationParameter(ByVal parameter
                                               As PMDCommutationParameter)
PMDAxis.SetCommutationParameter(ByVal parameter
                                As PMDCommutationParameter,
                                ByVal value As Int32)
```

see **Set/GetPhaseCorrectionMode**

SetCurrent 5Eh
GetCurrent 5Fh

Motor Types

		Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
parameter	*Holding Motor Limit*	0
	— (Reserved)	1
	Drive Current	2

	Type	Range/Scaling
value	unsigned 16-bit	see below

Packet Structure

SetCurrent

	0	*axis*	**5E**h
	15 12	11 8	7 0
write	*parameter*		
	15 0		
write	*value*		
	15 0		

GetCurrent

	0	*axis*	**5F**h
	15 12	11 8	7 0
write	*parameter*		
	15 0		
read	*value*		
	15 0		

Description

SetCurrent configures the operation of the holding current. The Holding Motor Limit is applied whenever the AtRest signal is active.

The *Holding Motor Limit* is in units of % maximum current, with scaling of $100/2^{15}$. Its range is 0 to $2^{15}–1$. It defines the value to which the current will be limited when in the holding state. This limit is applied as an additional limit to the current limit, so the lower of the two will affect the true limit.

The Drive Current is in units of % maximum current, with a scaling of $100/2^{15}$. Its range is 0 to 2^{15}- 1. It defines the value used for the active motor command when driving a step motor, that is, when not in a holding state.

GetCurrent gets the indicated holding current parameter.

Errors

Invalid Parameter: Unrecognized parameter code or parameter out of bounds.

C-Motion API

```
PMDresult PMDSetCurrent       (PMDAxisInterface axis_intf,
                               PMDuint16 parameter,
                               PMDuint16 value);
PMDresult PMDGetCurrent       (PMDAxisInterface axis_intf,
                               PMDuint16 parameter,
                               PMDuint16* value);
```

SetCurrent (cont.) 5Eh
GetCurrent 5Fh

Script API

```
GetCurrent parameter
SetCurrent parameter value
```

C# API

```
UInt16 value = GetCurrent(PMDCurrent parameter);
SetCurrent(PMDCurrent parameter, UInt16 value);
```

Visual Basic API

```
UInt16 value = GetCurrent(ByVal parameter As PMDCurrent)
SetCurrent(ByVal parameter As PMDCurrent, ByVal value As UInt16)
```

see

GetDriveStatus (p. 48), **Set/GetSampleTime** (p. 151), **SetMotorCommand** (p. 138)

SetCurrentControlMode 43h
GetCurrentControlMode 44h

Motor Types

	Brushless DC	**Microstepping**	

Arguments

Name	**Instance**	**Encoding**
axis	*Axis1*	0
mode	*reserved*	0
	FOC	1
	Third leg floating	2

Packet Structure

SetCurrentControlMode

0	*axis*	**43**h
15 12	11 8	7 0

write	*mode*
	15 0

GetCurrentControlMode

0	*axis*	**44**h
15 12	11 8	7 0

read	*mode*
	15 0

Description

SetCurrentControlMode configures an axis controlling a three phase BLDC motor to use either the default field oriented control (FOC) method, or the third leg floating method, in which only two of the three motor terminals is actively driven at any time, the remaining terminal being left floating (both high- and low-side switches off). The third leg floating method may be appropriate for motors intended for commutation by Hall effect sensors.

In third leg floating mode there is only one current control loop, to control the current between the two active terminals. This current loop uses the q-phase parameters.

For two phase motors FOC is the only supported current control scheme.

For single phase DC motors there is only one phase current to control; it uses the q-phase parameters.

Errors

Invalid Parameter: Unsupported mode.

C-Motion API

```
PMDresult PMDSetCurrentControlMode(PMDAxisInterface axis_intf,
                                   PMDuint16 mode);
PMDresult PMDGetCurrentControlMode(PMDAxisInterface axis_intf,
                                   PMDuint16* mode);
```

Script API

```
GetCurrentControlMode
SetCurrentControlMode mode
```

C# API

```
PMDCurrentControlMode mode = PMDAxis.CurrentControlMode;
PMDAxis.CurrentControlMode = mode;
```

Visual Basic API

```
PMDCurrentControlMode mode = PMDAxis.CurrentControlMode
PMDAxis.CurrentControlMode = mode
```

see

GetFOCValue (p. 54), **Get/SetFOC** (p. 130)

SetCurrentFoldback 41h
GetCurrentFoldback 42h

Motor Types

DC Brush	Brushless DC	Microstepping	▨

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
parameter	*Continuous Current Limit*	0
	Energy Limit	1

	Type	Range/Scaling
value	unsigned 16-bit	see below

Packet Structure

SetCurrentFoldback

	0	axis	**41**h
	15 12	11 8	7 0

write	*parameter*
	15 0

write	*value*
	15 0

GetCurrentFoldback

	0	axis	**42**h
	15 12	11 8	7 0

write	*parameter*
	15 0

read	*value*
	15 0

Description

SetCurrentFoldback is used to set various I^2t foldback-related parameters. Two parameters can be set, the ***Continuous Current Limit***, and the ***Energy Limit***. The range is from 0% to the factory default continuous current limit setting. The scaling for the continuous current limit is exactly the same as for the leg current sensors.

The units of ***Energy Limit*** are convertible to A^2s. The scaling factor is $2^{-31}/51.2e\text{-}6$ µs / $(A/count)^2$, where A/count is the current scaling factor and 51.2e-6 µs is the current loop cycle time.
The ***Continuous Current Limit*** is used by the current foldback algorithm. When the current output of the drive exceeds this setting, accumulation of the I^2 energy above this setting begins. Once the accumulated excess I^2 energy exceeds the value specified by the ***Energy Limit*** parameter, a current foldback condition

exists and the commanded current will be limited to the specified ***Continuous Current Limit***. When this occurs, the Current Foldback bit in the Event Status and Drive Status registers will be set. When the accumulated I^2 energy above the ***Continuous Current Limit*** drops to zero (0), the limit is removed, and the Current Foldback bit in the Drive Status register is cleared.

SetCurrentFoldback (cont.) 41h
GetCurrentFoldback 42h

Description (cont.)

SetEventAction can be used to configure a change in operating mode when current foldback occurs. Doing this does not interfere with the basic operation of Current Foldback described above. If this is done, the Current Foldback bit in the Event Status register must be cleared prior to restoring the operating mode, regardless of whether the system is in current foldback or not.

When current control is not active, a current foldback event always causes a change to the disabled state (all loops and motor output are disabled), regardless of the programmed Event Action. Changing the operating mode from disabled requires clearing of the Current Foldback bit in Event Status.

GetCurrentFoldback gets the maximum continuous current setting.

Errors

Invalid Parameter: Unrecognized parameter code, or value greater than 32768.

C-Motion API

```
PMDresult PMDSetCurrentFoldback(PMDAxisInterface axis_intf,
                                PMDuint16 parameter,
                                PMDuint16 value);
PMDresult PMDGetCurrentFoldback(PMDAxisInterface axis_intf,
                                PMDuint16 parameter,
                                PMDuint16* value);
```

Script API

```
GetCurrentFoldback parameter
SetCurrentFoldback parameter value
```

C# API

```
UInt16 value = PMDAxis.GetCurrentFoldback(PMDCurrentFoldback parameter);
PMDAxis.SetCurrentFoldback(PMDCurrentFoldback parameter, UInt16 value);
```

Visual Basic API

```
UInt16 value = PMDAxis.GetCurrentFoldback(ByVal parameter
                                          As PMDCurrentFoldback)
PMDAxis.SetCurrentFoldback(ByVal parameter As PMDCurrentFoldback,
                           ByVal value As UInt16)
```

see

GetEventStatus (p. 52), **ResetEventStatus** (p. 82), **GetDriveStatus** (p. 48), **RestoreOperatingMode** (p. 83), **GetActiveOperatingMode** (p. 38)

SetCurrentLimit 06h
GetCurrentLimit 07h

Motor Types

DC Brush	Brushless DC	▨	▨

Arguments

Name	Instance	Encoding		
axis	*Axis1*	0		
	Type	**Range**	**Scaling**	**Units**
limit	unsigned 16 bits	0 *to* 2^{14}–1	$100/2^{15}$	% representable current

Packet Structure

SetMotorLimit

	0	*axis*	**06**h
	15 12	11 8	7 0
		Data	
write		*limit*	
	15		0

GetMotorLimit

	0	*axis*	**07**h
	15 12	11 8	7 0
		Data	
read		*limit*	
	15		0

Description

SetCurrentLimit sets the maximum value for the commanded current allowed by the digital servo filter of the specified ***axis***. Current command values beyond this value will be clipped to the specified current command limit. For example if the current limit was set to 1,000 and the servo filter determined that the current command value should be 1,100, the actual command value would be 1,000. Conversely, if the output value was –1,100, then it would be clipped to –1,000. This command is useful for protecting amplifiers, motors, or system mechanisms when it is known that a current exceeding a certain value will cause damage.

GetCurrentLimit reads the motor limit value.

Scaling example: If it is desired that a current limit of 25% of full scale be established, then this register should be loaded with a value of 25.0 * 32,768/100 = 8,192 (decimal). This corresponds to a hexadecimal value of 02000h.

Restrictions

This command only affects the motor output when the current loop is enabled. When the motion control IC is in open loop mode, this command has no effect.

Errors

Invalid Parameter: Limit out of range.
Invalid Register State for Command: Microstep motor type.

C-Motion API

```
PMDresult PMDSetMotorLimit(PMDAxisInterface axis_intf,
                            PMDuint16 limit);
PMDresult PMDGetMotorLimit(PMDAxisInterface axis_intf,
                            PMDuint16* limit);
```

Script API

```
GetMotorLimit
SetMotorLimit limit
```

C# API

```
Int16 limit = PMDAxis.MotorLimit;
PMDAxis.MotorLimit = limit;
```

SetCurrentLimit (cont.) 06h
GetCurrentLimit 07h

Visual Basic API

```
Int16 limit = PMDAxis.MotorLimit
PMDAxis.MotorLimit = limit
```

see **Set/GetMotorCommand** (p. 138), **Set/GetOperatingMode** (p. 144)

SetDeceleration 91h
GetDeceleration 92h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding		
axis	*Axis1*	0		
	Type	**Range**	**Scaling**	**Units**
deceleration	unsigned 32 bits	0 *to* 2^{31}–1	$1/2^8$	counts/cycle^2 microsteps/cycle^2

Packet Structure

SetDeceleration

	15–12	11–8	7–0
	0	*axis*	**91**h
write	*deceleration* (high-order part) 31–16		
write	*deceleration* (low-order part) 15–0		

GetDeceleration

	15–12	11–8	7–0
	0	*axis*	**92**h
read	*deceleration* (high-order part) 31–16		
read	*deceleration* (low-order part) 15–0		

Description

SetDeceleration loads the maximum deceleration register for the specified *axis*.

GetDeceleration returns the value of the maximum deceleration.

Scaling example: To load a value of 1.750 counts/cycle^2 multiply by 65,536 (giving 114,688) and load the resultant number as a 32-bit number, giving 0001 in the high word and C000h in the low word. Retrieved numbers (**GetDeceleration**) must correspondingly be divided by 65,536 to convert to units of counts/cycle^2 or steps/cycle^2

Note: If *deceleration* is set to zero (0), then the value specified for acceleration (**SetAcceleration**) will automatically be used to set the magnitude of deceleration.

Errors

Invalid Parameter: negative deceleration value.

C-Motion API

```
PMDresult PMDSetDeceleration(PMDAxisInterface axis_intf,
                                 PMDuint32 deceleration);
PMDresult PMDGetDeceleration(PMDAxisInterface axis_intf,
                                 PMDuint32* deceleration);
```

Script API

```
GetDeceleration
SetDeceleration deceleration
```

C# API

```
UInt32 deceleration = PMDAxis.Deceleration;
PMDAxis.Deceleration = deceleration;
```

Visual Basic API

```
UInt32 deceleration = PMDAxis.Deceleration
PMDAxis.Deceleration = deceleration
```

see

Set/GetAcceleration (p. 84), **Set/GetVelocity** (p. 174)

SetDriveCommandMode 7Eh
GetDriveCommandMode 7Fh

Motor Type

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
mode	— (Reserved)	0-31
	Analog command	32
	SPI twos complement	33
	Internal Profile	34
	Pulse and Direction	35

Packet Structure

SetDriveCommandMode

	15 … 8	7 … 0
	0	**7**Eh
write	*mode* (15 … 0)	

GetDriveCommandMode

	15 … 8	7 … 0
	0	**7**Fh
read	*mode* (15 … 0)	

Description

SetDriveCommandMode is used to change the source or format of the external command that drives Juno output. The default value is 34 for all motor types, meaning use the internal profile generator.

Analog command means use the AnalogCmd input. This mode is supported only for servo (BLDC or brush DC) motors. If the velocity and position/outer loops are disabled then the command input is used to control either motor voltage or, if the current loop is enabled, current. In the case of current control the analog reading as a 16-bit signed number is divided by two to obtain the commanded current.

If the velocity loop is enabled, but the position/outer loop is not, then the analog reading is multiplied by 2^{16} to obtain the scaled commanded velocity.

If the position/outer loop is enabled, and is in position mode, that is, the outer loop feedback source is encoder, then the commanded position will be obtained by integrating the scaled commanded velocity.

If the position/outer loop is enabled, and is in outer loop mode, that is, the outer loop feedback source is either analog or SPI, then analog reading is multiplied by 2^{16} and used as the outer loop command. In the case of analog command and analog feedback the outer loop, if properly tuned, will act so as to make the two analog signals the same.

SPI twos complement means to expect a stream commands, interpreted as 16-bit twos complement numbers, on the SPI port. In this mode SPI host commands are not possible. For servo motors the signed SPI input reading is used in the same way as the analog reading, except that the SPI port may not be used as the outer loop feedback source.

For microstep motors SPI command input is interpreted as an increment in the commanded position, in microsteps.

SetDriveCommandMode (cont.) 7Eh
GetDriveCommandMode 7Fh

Description (cont.)

Internal Profile means to use the internal profile generator to compute the commanded voltage, current, or velocity from the commanded acceleration, deceleration, and velocity limits. The output of the profile generator is multiplied by the velocity scalar to produce the scaled commanded velocity, which is used as the command input to the velocity loop.

In the case the velocity and position/outer loops are disabled the scaled commanded velocity is divided by 2^{16} to produce the motor command, which is divided by 2 to produce the commanded current if the current loops are enabled.

When the position/outer loop is in outer loop mode, that is, the feedback source is analog or SPI, then the scaled commanded velocity is used as the outer loop command.

Pulse and Direction means to use external pulse and direction signals to set the commanded position. SPI host commands are not possible in this mode, because the pulse and direction signals are shared with SPIClock and SPIRcv. For step motors the commanded position is computed in microsteps. For servo motors the commanded position is necessarily in encoder counts, but the raw command is multiplied by the encoder counts/microstep ratio specified by the **SetEncoderToStepRatio** command.

It is not recommended to use pulse and direction input for servo motors with only current or voltage control enabled, or with the position/outer loop in outer loop mode.

Errors

Invalid Parameter: Unrecognized mode.
Invalid register state for command: Command source temporarily changed to internal profile while performing a smooth stop (operating mode must be restored). Or, outer loop feedback source is already SPI.

C-Motion API

```
PMDresult PMDSetDriveCommandMode(PMDAxisInterface axis_intf,
                                 PMDuint16 mode);
PMDresult PMDGetDriveCommandMode(PMDAxisInterface axis_intf,
                                 PMDuint16* mode);
```

Script API

```
GetDriveCommandMode
SetDriveCommandMode mode
```

C# API

```
PMDDriveCommandMode mode = PMDAxis.DriveCommandMode;
PMDAxis.DriveCommandMode = mode;
```

Visual Basic API

```
PMDDriveCommandMode mode = PMDAxis.DriveCommandMode
PMDAxis.DriveCommandMode = mode
```

see

SetAcceleration (p. 84), **SetDeceleration** (p. 113), **SetLoop** (p. 134), **SetVelocity** (p. 174)

SetDriveFaultParameter 62h
GetDriveFaultParameter 60h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
parameter	*Overvoltage Limit*	0
	Undervoltage Limit	1
	Event Recovery Mode	2
	Watchdog Limit	3
	Temperature Limit	4
	Temperature Hysteresis	5
	— (Reserved)	6,7
	Shunt voltage limit	8
	Shunt duty	9
	Bus current supply limit	10
	Bus current return limit	11

	Type	Range	Scaling
value	unsigned 16 bits	see below	see below

Packet Structure

SetDriveFaultParameter

0	*axis*	**62**h
15 12	11 8	7 0

write	*parameter*
	15 0

write	*value*
	15 0

GetDriveFaultParameter

0	*axis*	**60**h
15 12	11 8	7 0

write	*parameter*
	15 0

read	*value*
	15 0

Description

SetDriveFaultParameter sets various drive operation limits. The particular limit set depends on the parameter argument. When an operation limit is exceeded, motor output will be disabled and either a Drive Exception or Overtemperature event will be raised, and a bit set in the Drive Fault Status register to indicate the fault.

Not all products support all limits, consult product-specific documentation for more detail.

GetDriveFaultParameter returns the limits set by **SetDriveFaultParameter**.

Description (cont'd)

The Overvoltage and Undervoltage limit parameters set the thresholds for determination of overvoltage and undervoltage conditions. If the bus voltage exceeds the Overvoltage Limit value, an overvoltage condition occurs. If the bus voltage is less than the Undervoltage Limit value, an undervoltage condition occurs. Both the Overvoltage Limit and Undervoltage Limit have ranges of 0 to 2^{16} - 1; the scaling is product-dependent.

For example, to set the overvoltage threshold to 30V, ***Overvoltage Limit*** should be set to 30V/1.3612mv = 22039.

GetDriveFaultParameter reads the indicated limit.

The Event Recovery Mode is used to enable or disable automatic event recovery. The default mode is disabled, meaning that in order to return to normal operation after output is disabled by a fault host commands must be used to clear event status bits and to restore the active operating mode. Automatic event recovery mode is typically used when the system controlling Juno is not capable of sending host commands. Only two digital signals, FaultOut and ~Enable, are used to control Juno state.

When using automatic event recovery the FaultOut signal should be configured using **SetFaultOutMask** so that any event resulting in output being disabled will also result in FaultOut asserted. When FaultOut becomes active the external controller should wait for at least 150 µs, de-assert the ~Enable signal, wait again for at least 150 µs, and re-assert ~Enable. After ~Enable is re-asserted Juno will continue to attempt to clear all event status bits and re-enable the operating mode, until it succeeds in re-establishing output.

A parameter code of 0 means automatic event recover is disabled, 1 means enabled.

A side-effect of enabling automatic event recovery is that the behavior of **SetOperatingMode** is changed. When using automatic event recovery, if an event condition prevents enabling the specified operating mode then **SetOperatingMode** will not raise an error, but will set the commanded operating mode only. This feature allows the desired operating mode to be set even while, for example, Juno is disabled by the ~Disable signal.

The Watchdog Limit is used to disable output in case of an apparent failure of an external command processor. The default value of zero disables the watchdog, nonzero values specify the number of 51.2 µs commutation periods to allow between commands before signaling a Drive Exception event. The value is scaled by a factor of 8, for example a value of 2 means 16 * 51.2 = 819 µs.

The meaning of "command" depends on the Drive Command Mode:

1. For analog or pulse and direction command modes, the watchdog timer will never elapse.

2 For SPI command mode, the watchdog timer will be reset whenever an SPI velocity or step command is received.

3 For internal profile mode, the watchdog timer will be reset whenever any host command on any non-NVRAM interface is received. In order to reset the watchdog a command must have the correct checksum, a valid opcode, and the correct number of arguments, but need not actually succeed without error.

The action taken when the watchdog timer elapses is programmable, using **SetEventAction**. The default is to disable motor output.

SetDriveFaultParameter (cont.) 62h
GetDriveFaultParameter 60h

Description (cont'd)

Temperature Limit and Temperature Hysteresis are used either with an attached Atlas amplifier or with a motion control IC with a temperature input. In the case of the motion control IC the temperature scaling depends on external hardware. Because the input thermistor voltage may either rise or fall with actual temperature the sign of the temperature limit is used to indicate the sign of the gain: With a positive sign the internal temperature reading is just the input voltage. With a negative sign, the internal temperature reading is the input voltage subtracted from 3.3V, and the limit applied to that reading is the absolute value of the argument. In both cases 08000h corresponds to 3.3V.

Shunt voltage limit and Shunt duty are used with motion control ICs that support a shunt PWM output to control bus voltage rise due to regeneration. As long as the bus voltage remains below the shunt voltage limit the shunt PWM will remain inactive, when bus voltage rises above the limit, the shunt PWM will become active, with a duty cycle specified by Shunt duty. Shunt duty is scaled so that 08000h corresponds to 100%. The shunt PWM will remain active until bus voltage falls below the shunt voltage limit by a fixed hysteresis of 2.5%.

The bus current supply and bus current return limits are limits on the measured bus current supply and the computed bus current return values. When either current exceeds the specified limit motor output will be disabled, a DriveException event raised, and the Overcurrent Fault bit set in the Drive Fault status register.

Errors

Invalid Parameter: Unrecognized parameter code, or value out of bounds.

C-Motion API

```
PMDresult PMDSetDriveFaultParameter(PMDAxisInterface axis_intf,
                                     PMDuint16 parameter,
                                     PMDuint16 value);
PMDresult PMDGetDriveFaultParameter(PMDAxisInterface axis_intf,
                                     PMDuint16 parameter,
                                     PMDuint16* value);
```

Script API

```
GetDriveFaultParameter parameter
SetDriveFaultParameter parameter value
```

C# API

```
UInt16 value = PMDAxis.GetDriveFaultParameter(PMDDriveFaultParameter
                                               parameter);
PMDAxis.SetDriveFaultParameter(PMDDriveFaultParameter parameter,
                               UInt16 value);
```

Visual Basic API

```
UInt16 value = PMDAxis.GetDriveFaultParameter(ByVal parameter
                                              As PMDDriveFaultParameter)
PMDAxis.SetDriveFaultParameter(ByVal parameter
                               As PMDDriveFaultParameter,
                               ByVal value As UInt16)
```

see

Set/GetFaultOutMask (p. 128), **GetDriveFaultStatus** (p. 46), **ClearDriveFaultStatus** (p. 32), **GetEventStatus** (p. 52), **ResetEventStatus** (p. 82), **SetEventAction** (p. 125)

SetDrivePWM 23h
GetDrivePWM 24h

Motor Type

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
parameter	Limit	0
	Dead Time	1
	Signal Sense	2
	Frequency	3
	Refresh Period	4
	Refresh Time	5
	Minimum Current Read Time	6

	Type	Range/Scaling
value	16-bit unsigned	see below

Packet Structure

SetDrivePWM

	0		**23**h	
	15	8	7	0
write	0		*parameter*	
	15	8	7	0
write	*value*			
	15			0

GetDrivePWM

	0		**24**h	
	15	8	7	0
write	0		*parameter*	
	15	8	7	0
read	*value*			
	15			0

Description

SetDrivePWM sets parameters used for controlling amplifier PWM output. The PWM Limit register limits the maximum PWM duty cycle, and hence the effective output voltage. The range is from 0 to 2^{14}, 2^{14} corresponding to 100% PWM modulation.

The PWM Dead Time option controls the dead time added for High/Low PWM output between turning off the high side switch and turning on the low side, or vice versa. It has units of ns.

The PWM Frequency option controls the frequency for all PWM signals, the value is approximately the actual frequency, in Hz, scaled by 1/4. The available options are shown in the table below. Not all products support all frequencies.

Approximate Frequency	PWM Resolution	Actual Frequency	SetPWMFrequency Value
20 kHz	1:1536	19.531 kHz	5,000
40 kHz	1:708	39.062 kHz	10,000
80 kHz	1:384	78.124 kHz	20,000
120 kHz	1:256	117.187 kHz	30,000

SetDrivePWM (cont.) 23h
GetDrivePWM 24h

Description (cont.)

The PWM Signal Sense register controls whether an individual PWM signal is active high, encoded by a set bit, or active low, encoded by a clear bit. The PWM signal sense is not applied in the case of the sign signal for sign/magnitude PWM. The register layout is shown below:

Signal	Bit
PWM A High/PWM A Mag	0
PWM A Low	1
PWM B High/PWM B Mag	2
PWM B Low	3
PWM C High/PWM C Mag	4
PWM C Low	5
PWM D High/PWM D Mag	6
PWM D Low	7
— (Reserved)	8-14
PWM shunt	15

The PWM Refresh Period and PWM Refresh Time options are used to specify a minimum amount of off time when in High/Low PWM output mode. This may be required in order to allow charge pump capacitors to recharge. The Refresh Time is specified in ns, and the Refresh Period in commutation cycles. The low side of each PWM channel will be guaranteed to be on for at least the Refresh Time for every Refresh Period cycles.

The PWM Minimum Current Read Time option is used to specify a minimum amount of off time for two out of the three PWM output channels for three phase output in PWM High/Low output mode. For motion control ICs supporting leg current sensing this may be required in order to get accurate current measurement. It has units of ns.

GetDrivePWM returns the parameters set by **SetDrivePWM**.

Errors

Invalid Parameter: Unrecognized parameter code, parameter out of range.
Invalid operating mode for command: Attempt to change PWM parameter other than limit, with motor output enabled.

C-Motion API

```
PMDresult PMDSetDrivePWM(PMDAxisInterface axis_intf,
                         PMDuint16 option,
                         PMDuint16 value);
PMDresult PMDGetDrivePWM(PMDAxisInterface axis_intf,
                         PMDuint16 option,
                         PMDuint16* value);
```

Script API

```
GetDrivePWM parameter
SetDrivePWM parameter value
```

C# API

```
UInt16 value = PMDAxis.GetDrivePWM(PMDDrivePWM parameter);
PMDAxis.SetDrivePWM(PMDDrivePWM parameter, UInt16 value);
```

Visual Basic API

```
UInt16 value = PMDAxis.GetDrivePWM(ByVal parameter As PMDDrivePWM)
                                   PMDAxis.SetDrivePWM(ByVal
                                   parameter As PMDDrivePWM,
                                   ByVal value As UInt16)
```

SetEncoderSource DAh
GetEncoderSource DBh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
source	*Incremental*	0
	— (Reserved)	1
	None	2
	— (Reserved)	3,4
	Hall Sensors	5

Packet Structure

SetEncoderSource

	0	*axis*	**DA**h
	15 12	11 8	7 0

Data

	0	*source*
write	15 3	2 0

GetEncoderSource

	0	*axis*	**DB**h
	15 12	11 8	7 0

Data

	0	*source*
read	15 3	2 0

Description

SetEncoderSource sets the type of encoder feedback for the specified *axis*. When incremental quadrature is selected the motion control IC expects A and B quadrature signals to be input at the QuadA and QuadB axis inputs.

GetEncoderSource returns the code for the current type of feedback.

When Hall Sensors is selected the three signals HallA, HallB, and HallC are used to determine the actual position, with one count change per Hall state (six counts per electrical revolution). Three Hall sensors are frequently used for brushless motor commutation, see the *Juno Velocity and Torque Control IC User Guide* for more information.

An encoder source of none means that there is no way to measure actual position. This mode is used for microstep motors without position error control, and also for servo motors used in torque mode.

Errors

Invalid Parameter: Unsupported source code.

C-Motion API

```
PMDresult PMDSetEncoderSource(PMDAxisInterface axis_intf, PMDuint16 source);
PMDresult PMDGetEncoderSource(PMDAxisInterface axis_intf, PMDuint16* source);
```

Script API

```
GetEncoderSource
SetEncoderSource source
```

SetEncoderSource (cont.) DAh
GetEncoderSource DBh

C# API

```
PMDEncoderSource source = PMDAxis.EncoderSource;
PMDAxis.EncoderSource = source;
```

Visual Basic API

```
PMDEncoderSource source = PMDAxis.EncoderSource
PMDAxis.EncoderSource = source
```

see

SetEncoderToStepRatio DEh
GetEncoderToStepRatio DFh

Motor Types

		Microstepping

Arguments

Name	**Instance**	**Encoding**		
axis	*Axis1*	0		
	Type	**Range**	**Scaling**	**Units**
counts	unsigned 16 bits	1 *to* 2^{15}–1	unity	counts
steps	unsigned 16 bits	1 *to* 2^{15}–1	unity	microsteps

Packet Structure

SetEncoderToStepRatio

	15–12	11–8	7–0
	0	*axis*	**DE**h
write	*counts* (15–0)		
write	*steps* (15–0)		

GetEncoderToStepRatio

	15–12	11–8	7–0
	0	*axis*	**DF**h
read	*counts* (15–0)		
read	*steps* (15–0)		

Description

SetEncoderToStepRatio sets the ratio of the number of encoder counts to the number of output microsteps per motor rotation used by the motion control IC to convert encoder counts into steps. ***Counts*** is the number of encoder counts per full rotation of the motor. ***Steps*** is the number of steps output by the motion control IC per full rotation of the motor. Since this command sets a ratio, the parameters do not have to be for a full rotation as long as they correctly represent the encoder count to step ratio. **GetEncoderToStepRatio** returns the ratio of the number of encoder counts to the number of output steps per motor rotation.

The encoder to step ratio is also used for servo motors commanded by pulse and direction input to specify the ratio between input pulses and commanded position in encoder counts. The steps argument specifies the number of pulses per revolution, and the counts argument the number of encoder counts per revolution. The encoder to step ratio allows some extra flexibility in servo applications, but in many cases the default ratio of 1:1 is as good as any.

Errors

Invalid Parameter: One or both of the arguments is not positive

C-Motion API

```
PMDresult PMDSetEncoderToStepRatio(PMDAxisInterface axis_intf,
                                   PMDuint16 counts, PMDuint16 steps);
PMDresult PMDGetEncoderToStepRatio(PMDAxisInterface axis_intf,
                                   PMDuint16* counts, PMDuint16* steps);
```

SetEncoderToStepRatio (cont.) DEh
GetEncoderToStepRatio DFh

Script API

```
GetEncoderToStepRatio
SetEncoderToStepRatio ratio
where ratio = counts*65536 + steps
```

C# API

```
PMDAxis.GetEncoderToStepRatio(ref UInt16 counts, ref UInt16 steps);
PMDAxis.SetEncoderToStepRatio(UInt16 counts, UInt16 steps);
```

Visual Basic API

```
PMDAxis.GetEncoderToStepRatio(ByRef counts As UInt16,
                              ByRef steps As UInt16)
PMDAxis.SetEncoderToStepRatio(ByVal counts As UInt16,
                              ByVal steps As UInt16)
```

see

Set/GetActualPositionUnits (p. 87)

SetEventAction 48h
GetEventAction 49h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
event	*Immediate*	0
	— (Reserved)	1,2
	Motion Error	3
	Current Foldback	4
	Capture Received	5
	Overtemperature	6
	Disabled (Enable Signal)	7
	Commutation Error	8
	Overcurrent	9
	Overvoltage	10
	Undervoltage	11
	Watchdog Timeout	12
	Brake Signal	13
	SPI Direct Mode Change	14
action	*None*	0
	— (Reserved)	1
	Abrupt Stop	2
	Smooth Stop	3
	Disable Velocity Loop and Higher Modules	4
	Disable Position Loop & Higher Modules	5
	Disable Current Loop & Higher Modules	6
	Disable Motor Output & Higher Modules	7
	— (Reserved)	8, 9
	Passive Braking	10

Packet Structure

SetEventAction

	15–12	11–8	7–0
	0	*axis*	**48**h

	15–0
write	*event*
write	*action*

GetEventAction

	15–12	11–8	7–0
	0	*axis*	**49**h

	15–0
write	*event*
read	*action*

SetEventAction (cont.) 48h
GetEventAction 49h

Description (cont.)

SetEventAction configures what actions will be taken by the *axis* in response to a given *event*. The *action* can be either to modify the operating mode by disabling some or all of the loops, or, in the case of all loops remaining on, to perform an abrupt or smooth stop.

When, through **SetEventAction**, one of the *events* causes an *action*, the event bit in the Event Status register must be cleared prior to returning to operation. For internal profile stops, this means that the bit must be cleared prior to performing another trajectory move. For changes in operating mode, this means that the bit must be cleared prior to restoring the operating mode, either by **RestoreOperatingMode** or **SetOperatingMode**.

An exception is the Motion Error event, which only needs to be cleared in Event Status if its *action* is *Abrubt Stop* or *Smooth Stop*. If it causes changes in operating mode, the operating mode can be restored without clearing the bit in Event Status first.

A smooth or abrupt stop may be initiated even when the command source is not internal profile. For abrupt stop this is done by disabling the command source bit in the active operating mode. For smooth stop, in addition, bit 9, smooth stop, will be set in the active operating mode to indicate that the commanded torque, velocity, or position is temporarily obtained from the internal profile. In order to recover from either of these conditions it is necessary to set or restore the operating mode.

When using outer loop mode, that is, when the outer loop feedback source is not the encoder, then bit 4 (position/outer loop) of the active operating mode will be cleared as part of an abrupt or smooth stop. In order to recover from this condition it is necessary to set or restore the operating mode.

The Passive Braking action is possible only when using high/low PWM output. It disables normal PWM generation, and instead turns on all of the low side switches, causing the kinetic energy of the moving motor to be dissipated by resistance in the motor coils. When passive braking all active operating mode bits will be clear except for bit 0 (axis enabled), bit 1 (output enabled) and bit 8 (braking). In order to recover from this condition it is necessary to set or restore the operating mode.

The Immediate event simply means that the action should be performed immediately, without any special condition detected. This is the only way to command passive braking, or a smooth stop when using some command source other than the internal profile.

GetEventAction gets the action that is currently programmed for the given event with the exception of the *Immediate* event, which cannot be read back.

Restrictions

- The Disabled event must either disable motor output or brake.
- The Commutation Error event must either have no action, disable motor output, or brake.
- The Overcurrent event must either disable motor output or brake.
- The Brake Signal event must either disable motor output or brake.
- When changing the Brake Signal or Overcurrent event actions motor output must be disabled.

Errors

Invalid Parameter: Unrecognized event or action code, or invalid action for event, or action not supported for current motor type.
Invalid Operating Mode for Command: Attempt to set Brake Signal or Overcurrent action with motor output enabled.

SetEventAction (cont.) 48h
GetEventAction 49h

C-Motion API

```
PMDresult PMDSetEventAction (PMDAxisInterface axis_intf,
                             PMDuint16 event,
                             PMDuint16 action);
PMDresult PMDGetEventAction (PMDAxisInterface axis_intf,
                             PMDuint16 event,
                             PMDuint16* action);
```

Script API

```
GetEventAction event
SetEventAction event action
```

C# API

```
PMDEventAction action = PMDAxis.GetEventAction(PMDEventActionEvent
                                               ActionEvent);
PMDAxis.SetEventAction(PMDEventActionEvent ActionEvent, PMDEventAction Ac-
tion);
```

Visual Basic API

```
PMDEventAction action = PMDAxis.GetEventAction(ByVal ActionEvent
                                               As PMDEventActionEvent)
PMDAxis.SetEventAction(ByVal ActionEvent As PMDEventActionEvent,
                       ByVal Action As PMDEventAction)
```

see

GetActiveOperatingMode (p. 38), **RestoreOperatingMode** (p. 83), **Set/GetOperatingMode** (p. 144)

SetFaultOutMask FBh
GetFaultOutMask FCh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
mask	see below	bitmask

Packet Structure

SetFaultOutMask

0	*axis*	**FB**h
15 12	11 8	7 0

write	*mask*
	15 0

GetFaultOutMask

0	*axis*	**FC**h
15 12	11 8	7 0

read	*mask*
	15 0

Description

SetFaultOutMask configures the mask on Event Status register bits that will be ORed together on the FaultOut pin. The FaultOut pin is active high, as are the bits in Event Status. Thus, FaultOut will go high when any of the enabled bits in Event Status are set (1). The *mask* parameter is used to determine what bits in the Event Status register can cause FaultOut high, as follows:

Name	Bit
Motion Complete	0
Wrap-around	1
— (Reserved)	2
Position Capture	3
Motion Error	4
— (Reserved)	5, 6
Instruction Error	7
Disable	8
Overtemperature Fault	9
Drive Exception	10
Commutation Error	11
Current Foldback	12
Runtime Error	13
— (Reserved)	14, 15

For example, a *mask* setting of hexadecimal 0610h will configure the FaultOut pin to go high upon a motion error, Overtemperature Fault, or Drive Exception Fault. The FaultOut pin stays high until all Fault enabled bits in Event Status are cleared. The default value for the FaultOut *mask* is 0600h – Overtemperature Fault and Drive Exception enabled.

GetFaultOutMask gets the current *mask* for the indicated *axis*.

SetFaultOutMask (cont.) FBh
GetFaultOutMask FCh

C-Motion API

```
PMDresult PMDSetFaultOutMask (PMDAxisInterface axis_intf,
                              PMDuint16 mask);
PMDresult PMDGetFaultOutMask (PMDAxisInterface axis_intf,
                              PMDuint16* mask);
```

Script API

```
GetFaultOutMask
SetFaultOutMask mask
```

C# API

```
UInt16 mask = PMDAxis.FaultOutMask;
PMDAxis.FaultOutMask = mask;
```

Visual Basic API

```
UInt16 mask = PMDAxis.FaultOutMask
PMDAxis.FaultOutMask = mask
```

see

Set/GetInterruptMask (p. 132)

SetFOC F6h
GetFOC F7h

Motor Types

▨	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
loop	*Direct(D)*	0
	Quadrature(Q)	1
	Both(D and Q)	2
parameter	*Proportional Gain (KpDQ)*	0
	Integrator Gain (KiDQ)	1
	Integrator Sum Limit (ILimitDQ)	2

	Type	Range/Scaling
value	unsigned 16 bits	see below

Packet Structure

SetFOC

	0	*axis*	**F6**h
	15 12	11 8	7 0
write	0	*loop*	*parameter*
	15 12	11 8	7 0

write	*value*	
	15	0

GetFOC

	0	*axis*	**F7**h
	15 12	11 8	7 0
write	0	*loop*	*parameter*
	15 12	11 8	7 0

read	*value*	
	15	0

Description

Set/GetFOC is used to configure the operating parameters of the FOC-Current control. See the product user guide for more information on how each ***parameter*** is used in the current loop processing. The ***value*** written/read is always an unsigned 16-bit value, with the parameter-specific scaling shown below:

Parameter	Range	Scaling	Units
Proportional Gain (KpDQ)	0 to $2^{15}-1$	1/64	%output/%current
Integrator Gain (KiDQ)	0 to $2^{15}-1$	1/256	%output/%current/cycles
Integrator Sum Limit (ILimitDQ)	0 to $2^{15}-1$	2/100	%output

A setting of 64 for ***KpDQ*** corresponds to a gain of 1. That is, an error signal of 30% maximum current will cause the proportional contribution of the current loop output to be 30% of maximum output.

Description (cont.)

Similarly, setting *KiDQ* to 256 gives it a gain of 1; the value of the integrator sum would become the integrator contribution to the output.

ILimitDQ is used to limit the contribution of the integrator sum at the output. For example, setting *ILimitDQ* to 8192 results in a maximum integral contribution to the output of 2*8192 = 16384 = 50%.

The *loop* argument allows individual configuration of the parameters for the D and Q current loops. Alternately, a *loop* of 2 can be used with **SetFOC** to set the D and Q loops with a single API command. A *loop* of 2 is not valid for **GetFOC**.

The q component gains apply to brush DC motor current control, and to current control in third leg floating mode for three phase brushless DC motors.

The script interface combines the loop and parameter arguments into a single option argument as shown below. For example, if the loop is q (1) and the parameter is integrator gain (1), option = 1*256 + 1 = 257.

Restrictions

Loop code 2 (both) cannot be used with **GetFOC**.

Errors

Invalid Parameter: Unrecognized loop or parameter.

C-Motion API

```
PMDresult PMDSetFOC (PMDAxisInterface axis_intf,
                     PMDuint8 loop,
                     PMDuint8 parameter,
                     PMDuint16 value);
PMDresult PMDGetFOC (PMDAxisInterface axis_intf,
                     PMDuint8 loop,
                     PMDuint8 parameter,
                     PMDuint16* value);
```

Script API

```
GetFOC option
SetFOC option value
where option = loop*256 + parameter
```

C# API

```
UInt16 value = PMDAxis.GetFOC(PMDFOC ControlLoop,
                              PMDFOCParameter parameter);
PMDAxis.SetFOC(PMDFOC ControlLoop, PMDFOCParameter parameter,
               UInt16 value);
```

Visual Basic API

```
UInt16 value = PMDAxis.GetFOC(ByVal ControlLoop As PMDFOC,
                              ByVal parameter As PMDFOCParameter)
PMDAxis.SetFOC(ByVal ControlLoop As PMDFOC, ByVal parameter
               As PMDFOCParameter, ByVal value As UInt16)
```

see

GetFOCValue (p. 54), **Set/GetCurrentControlMode** (p. 108)

SetInterruptMask 2Fh
GetInterruptMask 56h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
mask	*Wrap-around*	0002h
	Capture Received	0008h
	Motion Error	0010h
	Instruction Error	0080h
	Disabled	0100h
	Overtemperature Fault	0200h
	Drive Exception	0400h
	Commutation Error	0800h
	Current Foldback	1000h
	Runtime Error	2000h

Packet Structure

SetInterruptMask

0	*axis*	**2F**h
15 12	11 8	7 0

Data

	mask
write	
	15 0

GetInterruptMask

0	*axis*	**56**h
15 12	11 8	7 0

Data

	mask
read	
	15 0

Description

SetInterruptMask determines which bits in the Event Status register of the specified *axis* will cause a host interrupt. For each interrupt *mask* bit that is set to 1, the corresponding Event Status register bit will cause an interrupt when that status register bit goes active (is set to 1). Interrupt mask bits set to 0 will not generate interrupts.

GetInterruptMask returns the *mask* for the specified *axis*.

SetInterruptMask also controls CAN event notification when using the motion control IC's CAN 2.0B interface. Whenever a host interrupt is activated, a CAN message is generated using message ID 180h + nodeID, notifying interested CAN nodes of the change in the Event Status register.

Example: The interrupt *mask* value 18h will generate an interrupt when either the Motion Error bit or the Capture Received bit of the Event Status register goes active (set to 1).

Errors

None

C-Motion API

```
PMDresult PMDSetInterruptMask(PMDAxisInterface axis_intf,
                              PMDuint16 mask);
PMDresult PMDGetInterruptMask(PMDAxisInterface axis_intf,
                              PMDuint16* mask);
```

Script API

```
GetInterruptMask
SetInterruptMask mask
```

C# API

```
UInt16 mask = PMDAxis.InterruptMask;
PMDAxis.InterruptMask = mask;
```

Visual Basic API

```
UInt16 mask = PMDAxis.InterruptMask
PMDAxis.InterruptMask = mask
```

see

ClearInterrupt (p. 33), **GetEventStatus** (p. 52), **Set/GetFaultOutMask** (p. 128)

SetLoop 78h
GetLoop 79h

Motor Types

DC Brush	Brushless DC	

Arguments

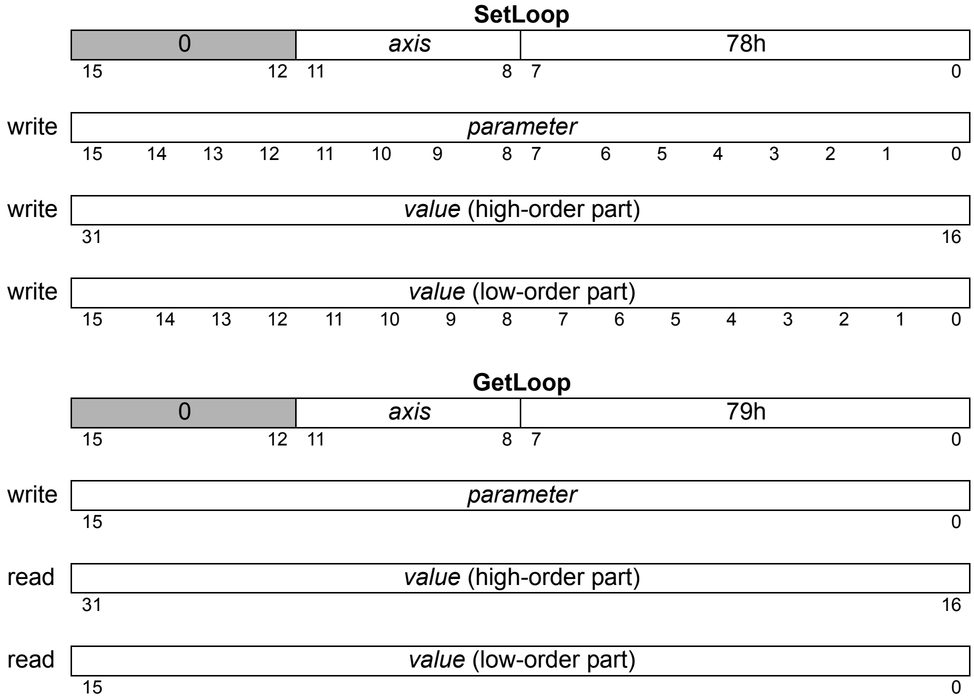
Name	Instance	Encoding
axis	Axis1	0
parameter		
	velocity Kp	0
	velocity Ki	1
	velocity ilimit	2
	— (Reserved)	3,4
	velocity Kout	5
	— (Reserved)	6
	velocity error limit	7
	velocity biquad enable	8
	velocity biquad b0	9
	velocity biquad b1	10
	velocity biquad b2	11
	velocity biquad a1	12
	velocity biquad a2	13
	command biquad enable	16
	command biquad b0	17
	command biquad b1	18
	command biquad b2	19
	command biquad a1	20
	command biquad a2	21
	— (Reserved)	22-63
	velocity feedback source	64
	velocity scalar Kvel	65
	outer loop feedback source	66
	velocity lower limit	67
	velocity upper limit	68
	outer/position loop Kp	256
	outer/position loop Ki	257
	outer/position loop ilimit	258
	outer/position loop Kd	259
	outer/position loop dtime	260
	outer/position loop Kout	261
	outer/position loop period	262
	position error limit	263
	outer loop deadband low	264
	outer loop deadband high	265

Returned Data

	Type	Range/Scaling
value	signed 32 bits	see below

SetLoop (cont.) 78h
GetLoop 79h

Packet Structure



Description

The **SetLoop** command is used to set the operating parameters of the velocity and position/outer loops. For more information on how these loops work and how the parameters are scaled see the *Juno Velocity & Torque Control IC User Guide.* All values are supplied as 32 bits, but in many cases the range is restricted.

The velocity loop Kp and Ki, and the position/outer loop Kp, Ki, and Kd parameters are limited to unsigned 16-bit values, that is, less than 2^{16}.

The velocity loop and position/outer loop ilimit parameters limit the maximum absolute value of the control loop integrated error, they are limited to non-negative signed 32-bit values, that is, less than 2^{31}. Setting an ilimit parameter to zero, the default value, disables integral action. Both the velocity and position/outer loops use an anti-windup algorithm, so choosing ilimit small is not normally necessary.

The velocity loop Kout is an unsigned 16-bit number scaled by 256, that is, an 8.8 fixed point fraction. The default value is 256, or 1.0 as a fraction.

The velocity scalar, Kvel, is an unsigned 32-bit number scaled by 65536, that is, a 16.16 fixed point fraction. Kvel is a conversion factor between velocity in encoder counts per sample period and the scaled velocity used by the velocity loop, see the *Juno Velocity & Torque Control IC User Guide* for more information.

The position/outer loop Kout is a signed 16-bit number scaled by 32768, that is, an 1.15 fixed point fraction. The default value is 32767, or approximately 1.0. A negative value for Kout may be used to invert the output of the position/outer loop.

The velocity biquad enable parameter is an enumerated value, 0 means disabled, 1, the default, means enabled. The velocity biquad filter is used to smooth feedback to the velocity loop.

SetLoop (cont.) 78h
GetLoop 79h

Description (cont.)

The command biquald enable parameter is an enumerated value, 0 means disabled, 1, the default, means enabled. The command biquad filter is used to smooth the analog command signal.

Biquad parameters b0, b1, b2, a0, and a1 are signed 32-bit numbers scaled by 65536, that is, 16.16 fixed point fractions. For a description of the biquad operation see the *Juno Velocity & Torque Control IC User Guide.*

The velocity and position/outer loop feedback sources are enumerated values, with the encoding shown below:The default feedback source for both loops is the encoder, which may be either a quadrature encoder or 3-phase Hall sensors, as set by **SetEncoderSource**. With encoder feedback the outer loop functions as a position loop: the feedback is the 32-bit actual position, and the reference is the integrated velocity command. With encoder feedback the reference of the velocity loop is the commanded velocity, and the feedback is an estimate of actual velocity made by filtering the difference in encoder position.

When the outer/position loop feedback is set to anything other than encoder, the loop is said to be in outer loop mode. In outer loop mode the loop reference is the scaled velocity command, rather than the commanded position obtained by integrating the unscaled commanded velocity.

Analog tachometer feedback may be used for either loop, but not for both simultaneously. The analog tachometer signal is biased by 1.65V and scaled to a signed 16-bit number, 0V corresponding to -32768 and 3.3V to 32767 3.3V. This value is then shifted left by 16 bits to produce either the commanded velocity or the outer loop reference.

Analog tachometer feedback inverted is the same as analog tachometer feedback, except that the sign is inverted, that is, 0V corresponds to 32767, and 3.3V to -32768.

SPI 2s complement feedback is supported only for the outer loop. In this mode Juno is an SPI slave, and the SPI master periodically sends a signed 16-bit 2s complement feedback value, which is shifted left by 16 bits and used as the outer loop feedback.

The position and velocity error limits define the minimum absolute position or velocity error that will result in a MotionError event. Only one limit is used at any time: If the position/outer loop is enabled then only the position error limit is used, otherwise, if the velocity loop is enabled then the velocity error limit is used.

When a motion error occurs the MotionError bit in the event status register will be set, and an action that may be programmed using **SetEventAction** will be performed.

The upper and lower velocity limits are limits on the outer/position loop output only, and may be used to constrain the outer loop output. For example, setting the lower velocity limit to zero with the outer and velocity loops enabled will prevent a negative velocity command. The upper velocity limit must be greater than or equal to zero, and the lower velocity limit must be less than or equal to zero.

The outer loop period is an integer between 1 and 32767, meaning the sample time of the outer loop, as a multiple of the sample time set by **SetSampleTime**. If the internal profile is used as the command source then the outer loop period will control it's rate as well.

Description (cont.)

The outer loop deadband feature is controlled by a low limit and a high limit. Both parameters are zero by default. This setting disables the deadband, and is normally used for position control. For outer loop pressure, level, or flow control the deadband feature may be useful to reduce "hunting" around the zero point. During outer loop operation the deadband has two states:

- If the output was previously nonzero then the absolute value of the output computed by the PID filter is compared to the deadband lower limit. If computed output is absolutely smaller, then the actual output is zero, otherwise it is the PID output.
- If the output was previously zero, then the absolute value of the output computed by the PID filter is compared to the deadband upper limit. If the computed output is absolutely smaller, then the actual output is zero, otherwise it is the PID output.

The upper limit must be set greater than or equal to the lower limit for correct operation, although this is not checked. An upper limit strictly greater than the lower limit provides hysteresis.

Errors

invalid parameter: argument is not a supported value, value is not within limits for the parameter.

invalid register state: Motor type is step – loops not supported.

C-Motion API

```
PMDresult PMDGetLoop (PMDAxisInterface axis_intf,
                      PMDuint16 parameter, PMDint32* value);
PMDresult PMDSetLoop (PMDAxisInterface axis_intf,
                      PMDuint16 parameter, PMDint32 value);
```

Script API

```
GetLoop parameter
SetLoop parameter value
```

C# API

```
Int32 value = PMDAxis.GetLoop(PMDLoop parameter);
PMDAxis.SetLoop(PMDLoop parameter, Int32 value);
```

Visual Basic API

```
Int32 value = PMDAxis.GetLoop(ByVal parameter As PMDLoop)
PMDAxis.SetLoop(ByVal parameter As PMDLoop, ByVal value As Int32)
```

see

SetEncoderSource (p. 121), **SetDriveCommandMode** (p. 114), **SetSampleTime** (p. 151), **GetEventStatus** (p. 52), **SetEventAction** (p. 125)

SetMotorCommand 77h
GetMotorCommand 69h

Motor Types

DC Brush	Brushless DC	Microstepping	

Arguments

Name	Instance	Encoding		
axis	*Axis1*	0		
	Type	**Range**	**Scaling**	**Units**
command	signed 16 bits	-2^{15} *to* $2^{15}-1$	$100/2^{15}$	% output

Packet Structure

SetMotorCommand

	0	*axis*	**77**h
	15 12	11 8	7 0
		Data	
write	*command*		
	15		0

GetMotorCommand

	0	*axis*	**69**h
	15 12	11 8	7 0
		Data	
read	*command*		
	15		0

Description

SetMotorCommand loads the Motor Command register of the specified ***axis***. For DC brush and brushless DC motors, this command directly sets the Motor Output register when the Position Loop, and Velocity Loop, and Command modules are disabled in the operating mode.

GetMotorCommand reads the contents of the motor command buffer register.

The **SetCurrent** command is used to control the output magnitude when driving a microstep motor.

Scaling example: If it is desired that a Motor Command value of 13.7% of full scale be output to the motor, then this register should be loaded with a value of 13.7 * 32,768/100 = 4,489 (decimal). This corresponds to a hexadecimal value of 1189h.

Note that if current control is enabled the q-phase commanded current will be half of the motor command, or 6.85% of the maximum representable current.

Restrictions

SetMotorCommand is a buffered command. The value set using this command will not take effect until the next **Update** or **MultiUpdate** command, with the Position Loop Update bit set in the update mask.

Errors

Invalid Opcode: Motor type is microstep.

C-Motion API

```
PMDresult PMDSetMotorCommand(PMDAxisInterface axis_intf,
                             PMDint16 command);
PMDresult PMDGetMotorCommand(PMDAxisInterface axis_intf,
                             PMDint16* command);
```

Script API

```
GetMotorCommand
SetMotorCommand command
```

C# API

```
Int16 command = PMDAxis.MotorCommand;
PMDAxis.MotorCommand = command;
```

Visual Basic API

```
Int16 command = PMDAxis.MotorCommand
PMDAxis.MotorCommand = command
```

see **SetCurrent** (p. 106), **Set/GetCurrentLimit** (p. 140), **Set/GetOperatingMode** (p. 144)

SetCurrentLimit 06h
GetCurrentLimit 07h

Motor Types

DC Brush	Brushless DC	▨	▨

Arguments

Name	Instance	Encoding		
axis	*Axis1*	0		
	Type	**Range**	**Scaling**	**Units**
limit	unsigned 16 bits	0 *to* 2^{14}–1	$100/2^{15}$	% representable current

Packet Structure

SetMotorLimit

	15–12	11–8	7–0
	0	*axis*	**06**h

Data

	15–0
write	*limit*

GetMotorLimit

	15–12	11–8	7–0
	0	*axis*	**07**h

Data

	15–0
read	*limit*

Description

SetCurrentLimit sets the maximum value for the commanded current allowed by the digital servo filter of the specified *axis*. Current command values beyond this value will be clipped to the specified current command limit. For example if the current limit was set to 1,000 and the servo filter determined that the current command value should be 1,100, the actual command value would be 1,000. Conversely, if the output value was –1,100, then it would be clipped to –1,000. This command is useful for protecting amplifiers, motors, or system mechanisms when it is known that a current exceeding a certain value will cause damage.

GetCurrentLimit reads the motor limit value.

Scaling example: If it is desired that a current limit of 25% of full scale be established, then this register should be loaded with a value of 25.0 * 32,768/100 = 8,192 (decimal). This corresponds to a hexadecimal value of 02000h.

Restrictions

This command only affects the motor output when the current loop is enabled. When the motion control IC is in open loop mode, this command has no effect.

Errors

Invalid Parameter: Limit out of range.
Invalid Register State for Command: Microstep motor type.

C-Motion API

```
PMDresult PMDSetMotorLimit(PMDAxisInterface axis_intf,
                           PMDuint16 limit);
PMDresult PMDGetMotorLimit(PMDAxisInterface axis_intf,
                           PMDuint16* limit);
```

Script API

```
GetMotorLimit
SetMotorLimit limit
```

C# API

```
Int16 limit = PMDAxis.MotorLimit;
PMDAxis.MotorLimit = limit;
```

Visual Basic API

```
Int16 limit = PMDAxis.MotorLimit
PMDAxis.MotorLimit = limit
```

see **Set/GetMotorCommand** (p. 138), **Set/GetOperatingMode** (p. 144)

SetMotorType 02h
GetMotorType 03h

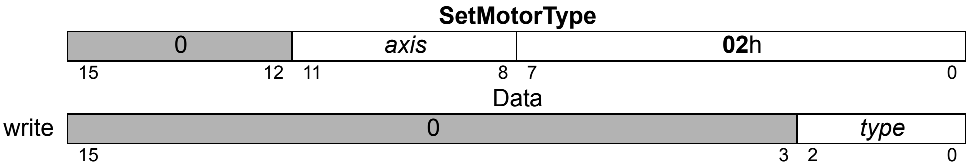
Motor Types

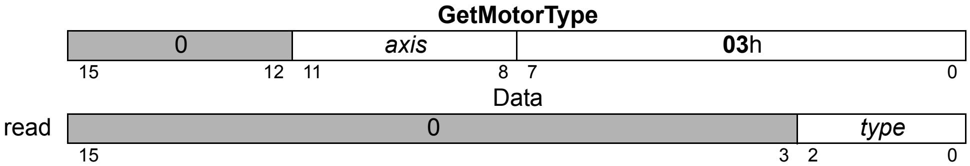
DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
type	*Brushless DC (3 phase)*	0
	— (Reserved)	1,2
	Microstepping (2 phase)	3
	— (Reserved)	4-6
	DC Brush	7

Packet Structure





Description

SetMotorType sets type of motor being driven by the selected ***axis***. This operation sets the number of phases for commutation on the axis, as well as internally configuring the motion control IC for the motor type.

The following table describes each motor type, and the number of phases to be commutated.

Motor type	Commutation
Brushless DC (3 phase)	3 phase
Microstepping (2 phase)	2 phase
DC Brush	None

GetMotorType returns the configured motor type for the selected ***axis***.

Restrictions

The motor type should only be set once immediately after reset using **SetMotorType**. Once it has been set, it should not be changed. Executing **SetMotorType** will reset many variables to their motor type specific default values.

Not all motor types are available on all products. See the product user guide.

Errors

Invalid Parameter: Unrecognized motor type code.
Invalid Operating Mode for Command: Motor output is enabled.

C-Motion API

```
PMDresult PMDSetMotorType (PMDAxisInterface axis_intf, PMDuint8 type);
PMDresult PMDGetMotorType (PMDAxisInterface axis_intf, PMDuint8* type);
```

Script API

```
GetMotorType
SetMotorType type
```

SetMotorType (cont.) 02h
GetMotorType 03h

C# API

```
PMDMotorType type = PMDAxis.MotorType;
PMDAxis.MotorType = type;
```

Visual Basic API

```
PMDMotorType type = PMDAxis.MotorType
PMDAxis.MotorType = type
```

see

Reset (p. 78)

SetOperatingMode 65h
GetOperatingMode 66h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
	Type	**Range/Scaling**
mode	unsigned 16-bit	see below

Packet Structure

SetOperatingMode

15–12	11–8	7–0
0	*axis*	**65**h

write

15–0
mode

GetOperatingMode

15–12	11–8	7–0
0	*axis*	**66**h

read

15–0
mode

Description

SetOperatingMode configures the operating mode of the *axis*. Each bit of the *mode* configures whether a feature/loop of the *axis* is active or disabled, as follows:

Name	Bit	Description
Axis Enabled	0	0: No *axis* processing, *axis* outputs in reset state. 1: *axis* active.
Motor Output Enabled	1	0: *axis* motor outputs disabled. 1: *axis* motor outputs enabled.
Current Control Enabled	2	0: *axis* current control bypassed. 1: *axis* current control active.
Velocity Loop Enabled	3	0:axis velocity loop bypassed 1:axis velocity loop active.
Position Loop Enabled	4	0: *axis* position loop bypassed. 1: *axis* position loop active.
Command Source	5	0: disabled. 1: enabled.
—	6–7	Reserved
	8	0:not braking 1:currently passive braking.
	9	0:normal operation 1:command source temporarily internal profile for smooth stop.
—	10–15	Reserved

When the axis motor output is disabled, the axis will function normally, but its motor outputs will be in their disabled state. When a loop is disabled (position, velocity, or current loop), it operates by passing its input directly to its output, and clearing all internal state variables (such as integrator sums, etc.). When the command source is disabled, it operates by commanding 0 velocity.

SetOperatingMode (cont.) 65h
GetOperatingMode 66h

Description (cont.)

For example, to configure an axis for Torque mode, (trajectory, valocity, and position loop disabled) the operating mode would be set to hexadecimal 0007h.

This command should be used to configure the static operating mode of the *axis*. The actual current operating mode may be changed by the axis in response to safety events, or user-programmable events. In this case, the present operating mode is available using **GetActiveOperatingMode**. **GetOperatingMode** will always return the static operating mode set using **SetOperatingMode**. Executing the **SetOperatingMode** command sets both the static operating mode and the active operating mode to the desired state.

The **SetOperatingMode** command attempts to determine whether an event has ocurred that will immediately result in disabling the new operating mode. In this case, by default, error 16, Invalid Operating Mode Restore, will be signaled. However, if automatic event recovery mode has been set using **SetDriveFaultParameter**, then the static operating mode will be set without altering the active operating mode, and the command will succeed.

The Braking and Smooth Stop operating mode bits indicate that the operating mode has been changed as a result of event handling.

Braking means that normal PWM high/low output has been disabled, and PWM output configured for passive braking. Smooth Stop means that the configured external command source (analog, pulse and direction, SPI) has been temporarily changed in order to allow a controlled smooth stop.

Neither the Braking nor Smooth Stop bits may be set by command, only cleared.

GetOperatingMode gets the operating mode of the *axis*.

Restrictions

The possible operating modes of an axis is product specific. See the product user guide for a description of which operating modes are supported on each axis.

Errors

Invalid Parameter: Unsupported bits set in argument.
Invalid Register State for Command: Operating mode not supported for current motor type or output mode.
Invalid Operating Mode Restore: Operating mode not permitted with current event status.

C-Motion API

```
PMDresult PMDSetOperatingMode(PMDAxisInterface axis_intf,
                              PMDuint16 mode);
PMDresult PMDGetOperatingMode(PMDAxisInterface axis_intf,
                              PMDuint16* mode);
```

Script API

```
GetOperatingMode
SetOperatingMode mode
```

C# API

```
UInt16 mode = PMDAxis.OperatingMode;
PMDAxis.OperatingMode = mode;
```

Visual Basic API

```
UInt16 mode = PMDAxis.OperatingMode
PMDAxis.OperatingMode = mode
```

see

GetActiveOperatingMode (p. 38), **GetEventStatus** (p. 52), **ResetEventStatus** (p. 82)
RestoreOperatingMode (p. 83), **SetDriveFaultParameter** (p. 116)

SetOutputMode E0h
GetOutputMode 6Eh

Motor Types

DC Brush	Brushless DC	Microstepping	▨

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
mode	*PWM Sign Magnitude*	1
	— (Reserved)	2-6
	PWM High/Low	7
	— (Reserved)	8,9
	None	10

Packet Structure

SetOutputMode

0	*axis*	**E0**h
15 12	11 8	7 0

Data

write

0	*mode*
15 4	3 0

GetOutputMode

0	*axis*	**6E**h
15 12	11 8	7 0

Data

read

0	*mode*
15 4	3 0

Description

SetOutputMode sets the form of the motor output signal of the specified *axis*. The default output mode is none; in this mode all the PWM outputs are high impedance

GetOutputMode returns the value for the motor output mode.

Restrictions

Not all output modes are available on all products. See the product user guide. The output mode cannot be changed when motor output is enabled in the active operating mode.

Errors

Invalid Parameter: Output mode unrecognized, or not supporte for the current motor type.
Invalid Operating Mode for Command: Motor output is enabled.

C-Motion API

```
PMDresult PMDSetOutputMode(PMDAxisInterface axis_intf, PMDuint16 mode);
PMDresult PMDGetOutputMode(PMDAxisInterface axis_intf, PMDuint16*
mode);
```

Script API

```
GetOutputMode
SetOutputMode mode
```

C# API

```
PMDOutputMode mode = PMDAxis.OutputMode;
PMDAxis.OutputMode = mode;
```

Visual Basic API

```
PMDOutputMode mode = PMDAxis.OutputMode
PMDAxis.OutputMode = mode
```

see

SetOperatingMode (p. 144)

SetPhaseCorrectionMode E8h
GetPhaseCorrectionMode E9h

Motor Types

	Brushless DC		

Arguments

Name	**Instance**	**Encoding**
axis	*Axis1*	0
mode	*Disabled*	0
	Index	1
	Hall	2

Packet Structure

SetPhaseCorrectionMode

0	*axis*	**E8**h
15 12	11 8	7 0

Data

write	0	*mode*
	15	2 0

GetPhaseCorrectionMode

0	*axis*	**E9**h
15 12	11 8	7 0

Data

read	0	*mode*
	15	2 0

Description

SetPhaseCorrectionMode controls the method used for phase correction on the specified axis. Phase correction is optional, and may be disabled by using mode 0. In mode 1 (Index) the encoder ***Index*** signal is used to update the commutation phase angle once per mechanical revolution. In mode 2 (Hall) a particular Hall sensor transition is used to update the commutation phase angle once every twelve electrical revolutions.

Phase correction ensures that the commutation angle will remain correct even if some encoder counts are lost due to electrical noise, or due to the number of encoder counts per electrical revolution not being an integer. Because Hall sensors normally have significant hysteresis index based correction is preferred if an index signal is available.

GetPhaseCorrectionMode returns the phase correction mode.

Errors

Invalid Parameter: Unrecognized mode.

C-Motion API

```
PMDresult PMDSetPhaseCorrectionMode(PMDAxisInterface axis_intf,
                                    PMDuint16 mode);
PMDresult PMDGetPhaseCorrectionMode(PMDAxisInterface axis_intf,
                                    PMDuint16* mode);
```

Script API

```
GetPhaseCorrectionMode
SetPhaseCorrectionMode mode
```

C# API

```
PMDPhaseCorrectionMode mode = PMDAxis.PhaseCorrectionMode;
PMDAxis.PhaseCorrectionMode = mode;
```

Visual Basic API

```
PMDPhaseCorrectionMode mode = PMDAxis.PhaseCorrectionMode
PMDAxis.PhaseCorrectionMode = mode
```

see

InitializePhase (p. 71)

SetPhaseInitializeMode E4h
GetPhaseInitializeMode E5h

Motor Types

	Brushless DC		

Arguments

Name	Instance	Encoding
axis	*Axis1*	0
mode	— (Reserved)	0
	Hall-based	1
	Pulse	*2*

Packet Structure

SetPhaseInitializeMode

0	*axis*	**E4**h
15 12	11 8	7 0

Data

write	0	*mode*
	15 2	0

GetPhaseInitializeMode

0	*axis*	**E5**h
15 12	11 8	7 0

Data

read	0	*mode*
	15 2	0

Description

SetPhaseInitializeMode establishes the mode in which the specified ***axis*** is to be initialized for commutation. The options are *Pulse* and *Hall-based*. In pulse mode the motion control IC briefly stimulates the motor windings and sets the initial phasing based on the observed motor response. In Hall-based initialization mode, the three Hall sensor signals are used to determine the motor phasing.

GetPhaseInitializeMode returns the value of the initialization mode.

Restrictions

Pulse mode should only be selected if it is known that the axis is free to move in both directions, and that a brief uncontrolled move can be tolerated by the motor, mechanism, and load.

Errors

Invalid Parameter: Unrecognized mode.

C-Motion API

```
PMDresult PMDSetPhaseInitializeMode(PMDAxisInterface axis_intf,
                                    PMDuint16 mode);
PMDresult PMDGetPhaseInitializeMode(PMDAxisInterface axis_intf,
                                    PMDuint16* mode);
```

Script API

```
GetPhaseInitializeMode
SetPhaseInitializeMode mode
```

C# API

```
PhaseInitializeMode mode = PMDAxis.PhaseInitializeMode;
PMDAxis.PhaseInitializeMode = mode;
```

Visual Basic API

```
PhaseInitializeMode mode = PMDAxis.PhaseInitializeMode
PMDAxis.PhaseInitializeMode = mode
```

see

InitializePhase (p. 71), **SetPhaseParameter** (p. 149)

SetPhaseParameter 85h
GetPhaseParameter 86h

Motor Types

	Brushless DC	

Arguments

Name	**Instance**	**Encoding**
axis	*Axis1*	0
parameter	ramp time	0
	positive pulse time	1
	negative pulse time	2
	pulse command	3
	— (Reserved)	4
	ramp command	5

	Type	**Range**	**Scaling/Units**
value	unsigned 16bits	0 *to* 2^{15}–1	counts

Packet Structure

SetPhaseParameter

	15–12	11–8	7–0
	0	*axis*	**85**h
write	*parameter* (15–0)		
write	*value* (15–0)		

GetPhaseParameter

	15–12	11–8	7–0
	0	*axis*	**85**h
write	*parameter* (15–0)		
read	*value* (15–0)		

Description

SetPhaseParameter is used to set parameters required for brushless DC motor pulse phase initialization. Phase initialization is required for commutation using an incremental encoder; the method used is set by **SetPhaseInitializeMode**.

The positive pulse time is a non-negative count of sample periods giving the duration of the first, positive pulse. The default sample period is 102 µs, but it can be changed by **SetSampleTime**.

The negative pulse time is a non-negative count of sample periods giving the duration of the second, negative pulse. Each negative pulse follows immediately after a positive pulse. The time between successive pulse pairs is given by three times the positive pulse time.

The pulse command is a non-negative value that is used as the motor command during both the positive and negative pulses.

The ramp time is a non-negative count of sample periods giving the duration of the pull-in ramp part of pulse phase initialization. It is possible, though not recommended, to set this to zero.

SetPhaseParameter (cont.) 85h
GetPhaseParameter 86h

Description (cont.)

The ramp command is a non-negative value that is used as the motor command during the pull-in ramp.

By default all phase parameters are zero, however phase initialization cannot possibly work in that state.

The process of pulse phase initialization and how to set the various parameters is discussed in the *Juno Velocity and Torque IC User Guide*.

GetPhaseParameter is used to read the values set by SetPhaseParameter.

Errors

Unrecognized parameter code, or value out of range.

C-Motion API

```
PMDresult PMDGetPhaseParameter (PMDAxisInterface axis_intf,
                                PMDuint16 parameter, PMDint16* value);
PMDresult PMDSetPhaseParameter (PMDAxisInterface axis_intf,
                                PMDuint16 parameter, PMDint16 value);
```

Script API

```
GetPhaseParameter parameter
SetPhaseParameter parameter value
```

C# API

```
Int32 value = PMDAxis.GetPhaseParameter(PMDPhaseParameter parameter);
PMDAxis.SetPhaseParameter(PMDPhaseParameter parameter, Int32 value);
```

Visual Basic API

```
Int32 value = PMDAxis.GetPhaseParameter(ByVal parameter
                                        As PMDPhaseParameter)
PMDAxis.SetPhaseParameter(ByVal parameter As PMDPhaseParameter,
                          ByVal value As Int32)
```

see

InitializePhase (p. 71), **SetPhaseInitializeMode** (p. 148)

SetSampleTime 3Bh
GetSampleTime 3Ch

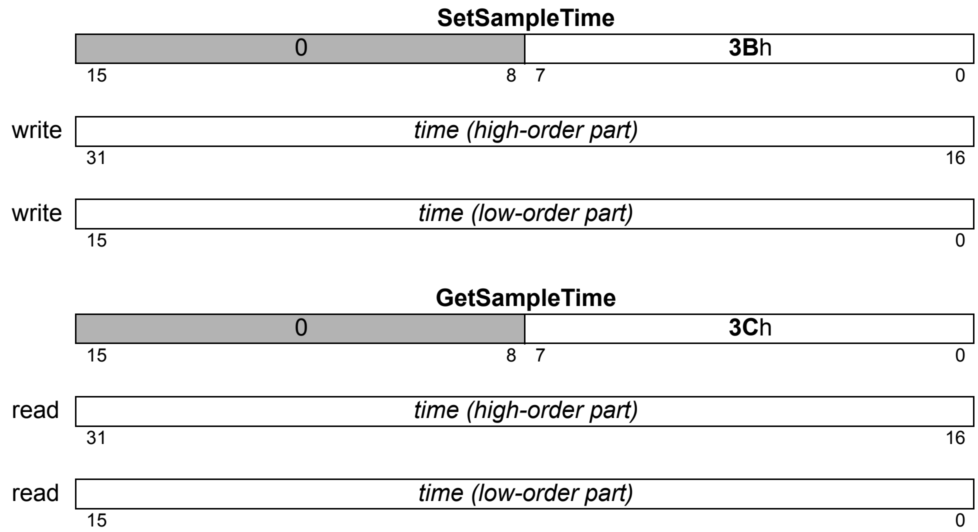
Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Type	Range	Units
time	unsigned 32 bits	51 *to* 2^{20}	microseconds

Packet Structure



Description

SetSampleTime sets the time basis for the motion control IC. This time basis determines the trajectory update rate for all motor types as well as the servo loop calculation rate for DC brush and brushless DC motors. It does not, however, determine the commutation rate of the brushless DC motor types, nor the PWM or current loop rates for any motor type.

The *time* value is expressed in microseconds. The motion control IC hardware can adjust the cycle time only in increments of 51.2 microseconds; the *time* value passed to this command will be rounded to the nearest increment of this base value.

Minimum cycle time depends on the product and number of enabled axes as follows:

# Enabled Axes	Minimum Cycle Time	Cycle Time w/ Trace Capture	Time per Axis	Maximum Cycle Frequency
1 (Juno)	102.4 μs	102.4 μs	102.4 μs	9.76 kHz

GetSampleTime returns the value of the sample time.

SetSampleTime (cont.) 3Bh
GetSampleTime 3Ch

Restrictions

This command cannot be used to set a sample time lower than the required minimum cycle time for the current configuration. Attempting to do so will set the sample time to the required minimum cycle time as specified in the previous table.

Errors

Invalid Parameter: Argument out of range.

C-Motion API

```
PMDresult PMDSetSampleTime(PMDAxisInterface axis_intf,
                           PMDuint32 time);
PMDresult PMDGetSampleTime(PMDAxisInterface axis_intf,
                           PMDuint32* time);
```

Script API

```
GetSampleTime
SetSampleTime time
```

C# API

```
UINT32 time = PMDAxis.SampleTime;
PMDAxis.SampleTime = time;
```

Visual Basic API

```
UINT32 time = PMDAxis.SampleTime
PMDAxis.SampleTime = time
```

see

SetSerialPortMode 8Bh
GetSerialPortMode 8Ch

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Type	Encoding
mode	unsigned 16 bits	see below

Packet Structure

SetSerialPortMode

0	*axis*	**8B**h
15	8	7 0

Data

	multi-drop address	0	protocol	stop bits	parity	transmission rate
write	15 11	10 9	8 7	6	5 4	3 0

GetSerialPortMode

0	*axis*	**8C**h
15	8	7 0

Data

	multi-drop address	0	protocol	stop bits	parity	transmission rate
read	15 11	10 9	8 7	6	5 4	3 0

Description

SetSerialPortMode sets the configuration for the asynchronous serial port. It configures the timing and framing of the serial port on the unit, regardless of whether RS-232 or RS-485 voltage levels are being used. The response to this command will use the serial port settings in effect before the command is executed, for example, transmission rate and parity. The new serial port settings must be used for the next command.

GetSerialPortMode returns the configuration for the asynchronous serial port, regardless of whether RS-232 or RS-485 voltage levels are being used.

The following table shows the encoding of the data used by this command.

Bit Number	Name	Instance	Encoding
0–3	Transmission Rate	1200 baud	0
		2400 baud	1
		9600 baud	2
		19200 baud	3
		57600 baud	4
		115200 baud	5
		230400 baud	6
		460800 baud	7
4–5	Parity	none	0
		odd	1
		even	2
6	Stop Bits	1	0
		2	1
7–8	Protocol	Point-to-point	0
		Multi-drop using idle-line detection	1
		— (Reserved)	2
		— (Reserved)	3
11–15	Multi-Drop Address	Address 0	0
		Address 1	1
			...
		Address 31	31

The script interface combines all argments into a single mode argument, as shown below. For example, for point-to-point (0) operation at 57600 baud (4) with no parity (0) and 2 stop bits (1), option = 0*2048 + 0*128 + 1*64 + 0*16 + 4 = 68.

SetSerialPortMode (cont.) 8Bh
GetSerialPortMode 8Ch

Restrictions

Multi-drop serial communication is not supported by all products, see the product user guide.

Errors

Invalid Parameter: Requested multi-drop protocol not supported.

C-Motion API

```
PMDresult PMDSetSerialPortMode(PMDAxisInterface axis_intf,
                               PMDuint8 baud,
                               PMDuint8 parity,
                               PMDuint8 stopBits,
                               PMDuint8 protocol,
                               PMDuint8 multiDropID);
PMDresult PMDGetSerialPortMode(PMDAxisInterface axis_intf,
                               PMDuint8* baud,
                               PMDuint8* parity,
                               PMDuint8* stopBits,
                               PMDuint8* protocol,
                               PMDuint8* multiDropID);
```

Script API

```
GetSerialPortMode
SetSerialPortMode mode
where mode = MultiDropId*2048 + protocol*128 + StopBits*64 + parity*16 +
             baud
```

C# API

```
PMDAxis.GetSerialPortMode(ref PMDSerialBaud baud,
                          ref PMDSerialParity parity,
                          ref PMDSerialStopBits StopBits,
                          ref PMDSerialProtocol protocol,
                          ref Byte MultiDropId);
PMDAxis.SetSerialPortMode( PMDSerialBaud baud,
                          PMDSerialParity parity,
                          PMDSerialStopBits StopBits,
                          PMDSerialProtocol protocol,
                          Byte MultiDropId);
```

Visual Basic API

```
PMDAxis.GetSerialPortMode(ByRef baud As PMDSerialBaud,
                          ByRef parity As PMDSerialParity,
                          ByRef StopBits As PMDSerialStopBits,
                          ByRef protocol As PMDSerialProtocol,
                          ByRef MultiDropId As Byte)
PMDAxis.SetSerialPortMode(ByVal baud As PMDSerialBaud,
                          ByVal parity As PMDSerialParity,
                          ByVal StopBits As PMDSerialStopBits,
                          ByVal protocol As PMDSerialProtocol,
                          ByVal MultiDropId As Byte)
```

see

SetSignalSense A2h
GetSignalSense A3h

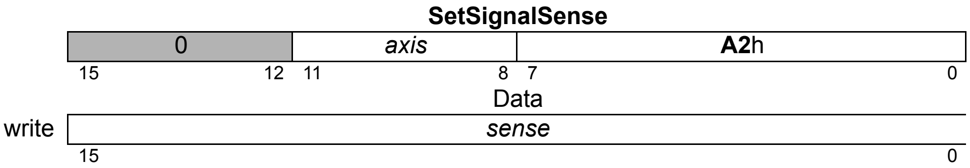
Motor Types

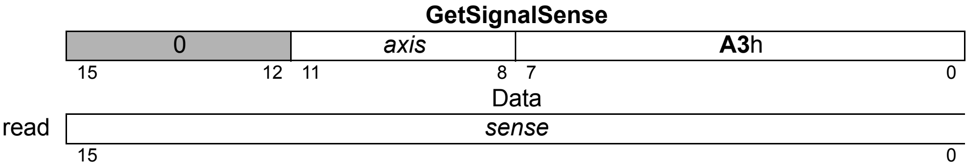
DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding	
axis	*Axis1*	0	
	Indicator	**Encoding**	**Bit Number**
sense	*EncoderA*	0001h	0
	EncoderB	0002h	1
	Encoder Index	0004h	2
	— (Reserved)		3-6
	HallA	0080h	7
	HallB	0100h	8
	HallC	0200h	9
	— (Reserved)		10
	Pulse Input	0800h	11
	Motor Direction	1000h	12
	— (Reserved)		13,14
	Direction Input	8000h	15

Packet Structure





Description

SetSignalSense establishes the sense of the corresponding bits of the Signal Status register, with the addition of ***Step Output*** and ***Motor Direction***, for the specified ***axis***.

For ***Encoder Index***, if the sense bit is 1, an index will be recognized for use in index-based phase correction or position capture if the index has a low to high transition.

For the ***Capture Input***, if the sense bit is 1, a capture will occur on a low-to-high signal transition. Otherwise, a capture will occur on a high-to-low transition.

SetSignalSense (cont.) A2h
GetSignalSense A3h

Description (cont.)

The Pulse Input and Direction Input bits are used when the command source is pulse and direction. If the Pulse Input bit is 0 then a pulse will be recorded when the signal transitions from a high state to a low state. If the Direction Input bit is 0 then a high level is interpreted as a move in the positive direction, and a low level as a move in the negative direction.

The Motor Direction bit may be used to invert the direction of positive torque. For brushless DC motors using encoder commutation the encoder direction (using one of EncoderA or EncoderB sense bits) must be inverted at the same time as Motor Direction. Phase initialization must be repeated whenever motor direction is changed.

GetSignalSense returns the value of the Signal Sense mask.

Restrictions

FaultOut and /Enable exist in the Signal Status register, but their sense is not controllable.

Not all bits are implemented for all products. See the product user guide.

Errors

None

C-Motion API

```
PMDresult PMDSetSignalSense(PMDAxisInterface axis_intf,
                            PMDuint16 sense);
PMDresult PMDGetSignalSense(PMDAxisInterface axis_intf,
                            PMDuint16* sense);
```

Script API

```
GetSignalSense
SetSignalSense sense
```

C# API

```
UInt16 sense = PMDAxis.SignalSense;
PMDAxis.SignalSense = sense;
```

Visual Basic API

```
UInt16 sense = PMDAxis.SignalSense
PMDAxis.SignalSense = sense
```

see

GetSignalStatus (p. 64)

SetTraceMode B0h
GetTraceMode B1h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
mode	*16-bit unsigned*	see below

Packet Structure

SetTraceMode

	0		**B0**h
15	8	7	0

Data

write	*mode*	
	15	0

GetTraceMode

	0		**B1**h
15	8	7	0

Data

read	*mode*	
	15	0

Description

SetTraceMode sets the behavior for the next trace. Mode is a bitmask, as shown below:

Name	Bit
Wrap Mode	0
— (Reserved)	1-15

Wrap mode may be either One Time (zero), or Rolling Buffer (one). In One Time mode, the trace continues until the trace buffer is filled, then stops. In Rolling Buffer mode, the trace continues from the beginning of the trace buffer after the end is reached. When in rolling mode, values stored at the beginning of the trace buffer are lost if they are not read before being overwritten by the wrapped data.

GetTraceMode returns the value for the trace mode.

Errors

Invalid Parameter: Reserved bit nonzero.

C-Motion API

```
PMDresult PMDSetTraceMode(PMDAxisInterface axis_intf, PMDuint16 mode);
PMDresult PMDGetTraceMode(PMDAxisInterface axis_intf, PMDuint16* mode);
```

Script API

```
GetTraceMode
SetTraceMode mode
```

C# API

```
PMDTraceMode mode = PMDAxis.TraceMode;
PMDAxis.TraceMode = mode;
```

Visual Basic API

```
PMDTraceMode mode = PMDAxis.TraceMode
PMDAxis.TraceMode = mode
```

see

GetTraceStatus (p. 68)

SetTracePeriod B8h
GetTracePeriod B9h

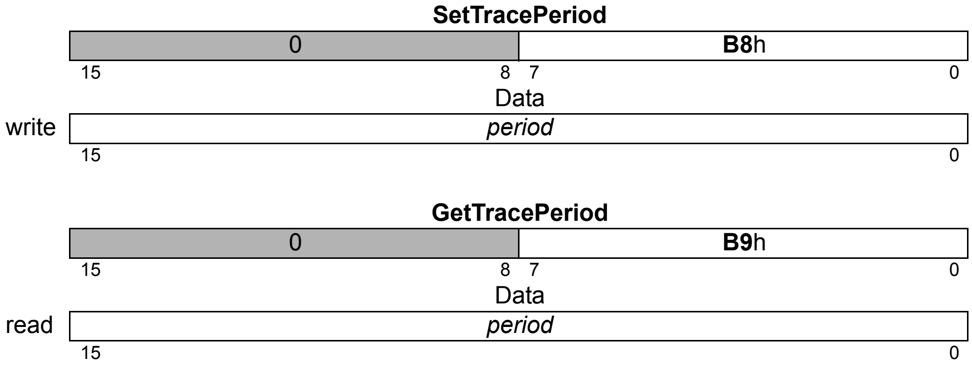
Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Type	Range	Scaling	Units
period	unsigned 16 bits	1 *to* 2^{16}–1	unity	cycles

Packet Structure



Description

SetTracePeriod sets the interval between contiguous trace captures. For example, if the trace period is set to one, trace data will be captured at the end of every chip cycle. If the trace period is set to two, trace data will be captured at the end of every second chip cycle, and so on.

GetTracePeriod returns the value for the trace period.

Errors

Invalid Parameter: Zero Period

C-Motion API

```
PMDresult PMDSetTracePeriod(PMDAxisInterface axis_intf,
                            PMDuint16 period);
PMDresult PMDGetTracePeriod(PMDAxisInterface axis_intf,
                            PMDuint16* period);
```

Script API

```
GetTracePeriod
SetTracePeriod period
```

C# API

```
UInt16 period = PMDAxis.TracePeriod;
PMDAxis.TracePeriod = period;
```

Visual Basic API

```
UInt16 period = PMDAxis.TracePeriod
PMDAxis.TracePeriod = period
```

see

Set/GetSampleTime (p. 151), **Set/GetTraceStart** (p. 159), **Set/GetTraceStop** (p. 162)

SetTraceStart B2h
GetTraceStart B3h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
triggerAxis	*Axis1*	0
condition	*Immediate*	0
	— (Reserved)	1
	Event Status	2
	Activity Status	3
	Signal Status	4
	Drive Status	5
	— (Reserved)	6
	Signed trace value greater than	7
	Signed trace value less than	8
	Unsigned trace value higher than	9
	Unsigned trace value lower than	10
	Trace value bitmask	11
triggerBit	*Status Register Bit*	0 *to* 15
triggerState (tS)	*Triggering State of the Bit*	0 (value = 0) 1 (value = 1)

Packet Structure

SetTraceStart

0	**B2**h
15 8	7 0

Data

write	0	*tS*	*triggerBit*	*condition*	*triggerAxis*
	15 13	12	11 8	7 4	3 0

GetTraceStart

0	**B3**h
15 8	7 0

Data

read	0	*tS*	*triggerBit*	*condition*	*triggerAxis*
	15 13	12	11 8	7 4	3 0

Description

SetTraceStart sets the condition for starting the trace. The *Immediate* condition requires no axis to be specified and the trace will begin upon execution of this instruction. The next four conditions require an axis to be specified, and when the condition for that axis is attained, the trace will begin.

When a status register bit is the trigger, the bit number and state must be included in the argument. The trace is started when the indicated bit reaches the specified state (0 or 1).

The last five conditions compare the value of the first trace variable configured with the value set using the **SetTraceTriggerValue** command. This value is always computed, whether trace is active or not. Unsigned comparisons should be used for a first trace variable with an unsigned result, conversely signed comparisons used for a first trace variable with signed results.

Once a trace has started, the trace-start trigger is reset to zero (0).

SetTraceStart (cont.) B2h
GetTraceStart B3h

Description (cont.)

The trace value bitmask condition is suitable for testing multiple bits from the 16-bit status registers. In this case the high order word of the comparison value is a selection mask; In order to trigger the bitwise logical AND of this mask with the first trace value must equal the low order word of the comparison value (the sense mask).

For all conditions the triggerState bit negates the sense of the condition, for example, if the triggerState bit is 1 then condition 7 is a signed less than or equal test, instead of greater than.

In the case of the immediate condition the triggerState bit must be 0 for the command to have any effect, otherwise the effective condition is Never.

GetTraceStart returns the value of the trace-start trigger.

The following table shows the corresponding value for combinations of ***triggerBit*** and ***register***0.

TriggerBit	Event Status Register	Activity Status Register	Signal Status Register	Drive Status Register
0		Phasing Initialized	Encoder A	Calibrated
1	Wrap-around	At Maximum Velocity	Encoder B	In Foldback
2			Encoder Index	Overtemperature
3	Position Capture			Shunt Active
4	Motion Error			In Holding
5				Overvoltage
6				Undervoltage
7	Instruction Error		Hall Sensor A	
8	Disable		Hall Sensor B	
9	Overtemperature Fault	Position Capture	Hall Sensor C	
0Ah	Drive Exception	In Motion		
0Bh	Commutation Error			
0Ch	Current Foldback			Clipping
0Dh	Runtime Error		/Enable Input	
0Eh			FaultOut	Initializing
0Fh				

The script interface combines all arguments into a single start argument, as shown below.

Examples:

If it is desired that the trace begin immediately, then the condition is zero, and all other arguments are not used, and can be set to zero. The start argument, and the actual word sent to the Juno processor is zero.

If it is desired that the trace begin when bit 7 of the Activity Status register for axis 1 goes to 0, then the trace start is loaded as follows: A 0 is loaded for axis number, a 3 is loaded for condition, a 7 is loaded for bit number, and a 0 is loaded for state. The start argument and the actual data word sent to the motor processor is 0730h.If it is desired that the trace begin when the raw bus voltage is less than 20,000.

First set the comparison value of 20,000 using **SetTraceTriggerValue** 0x100 20000
Next set the first trace variable to bus voltage (54, 036h) using **SetTraceVariable** 0 0x3600
Finally set the start condition to less than (8) using **SetTraceStart** 0x0080

Errors

Invalid Parameter: Parameter out of range.
Trace Buffer Zero: Immediate start with trace buffer length of zero.

Restrictions

Not all trace start conditions are available in all products. See the product user guide.

C-Motion API

```
PMDresult PMDSetTraceStart(PMDAxisInterface axis_intf,
                           PMDAxis traceAxis,
                           PMDuint8 condition,
                           PMDuint8 triggerBit,
                           PMDuint8 triggerState);
PMDresult PMDGetTraceStart(PMDAxisInterface axis_intf,
                           PMDAxis* traceAxis,
                           PMDuint8* condition,
                           PMDuint8* triggerBit,
                           PMDuint8* triggerState);
```

Script API

```
GetTraceStart
SetTraceStart start
where start = triggerState*2048 + triggerBit*256 + condition*16 +
             triggerAxis
```

C# API

```
PMDAxis.GetTraceStart(ref PMDAxisNumber triggerAxis,
                           ref PMDTraceCondition condition,
                           ref Byte bit,
                           ref PMDTraceTriggerState state);
PMDAxis.SetTraceStart(PMDAxisNumber triggerAxis,
                           PMDTraceCondition condition,
                           Byte bit,
                           PMDTraceTriggerState state);
```

Visual Basic API

```
PMDAxis.GetTraceStart(ByRef triggerAxis As PMDAxisNumber,
                           ByRef condition As PMDTraceCondition,
                           ByRef bit As Byte,
                           ByRef state As PMDTraceTriggerState)
PMDAxis.SetTraceStart(ByVal triggerAxis As PMDAxisNumber,
                           ByVal condition As PMDTraceCondition,
                           ByVal bit As Byte,
                           ByVal state As PMDTraceTriggerState)
```

see

Set/GetBufferLength (p. 92), **GetTraceCount** (p. 67), **Set/GetTraceMode** (p. 157), **Set/GetTracePeriod** (p. 158), **Set/GetTraceStop** (p. 162),**Set/GetTraceTriggerValue** (p. 90)

SetTraceStop B4h
GetTraceStop B5h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
triggerAxis	*Axis1*	0
condition	*Immediate*	0
	Next Update	1
	Event Status	2
	Activity Status	3
	Signal Status	4
	Drive Status	5
	— (Reserved)	6
	Signed trace value greater than	7
	Signed trace value less than	8
	Unsigned trace value higher than	9
	Unsigned trace value lower than	10
	Trace value bitmask	11
triggerBit	*Status Register Bit*	0 *to* 15
triggerState (tS)	*Triggering State of the Bit*	0 (value = 0) 1 (value = 1)

Packet Structure

SetTraceStop

15–8	7–0
0	**B4**h

Data

	15–13	12	11–8	7–4	3–0
write	0	*tS*	*triggerBit*	*condition*	*triggerAxis*

GetTraceStop

15–8	7–0
0	**B5**h

Data

	15–13	12	11–8	7–4	3–0
read	0	*tS*	*triggerBit*	*condition*	*triggerAxis*

Description

SetTraceStop sets the condition for stopping the trace. The *Immediate* condition requires no axis to be specified and the trace will stop upon execution of this instruction. All of the other conditions are identical to those for SetTraceStart, see the description for that command.

GetTraceStop returns the value of the trace-stop trigger.

Once a trace has stopped, the trace-stop trigger is reset to zero (0).

The script interface combines all arguments into a single stop argument, as shown below.

For examples of use, see "SetTraceStart B2h GetTraceStart B3h" on page 159, which uses the same argument encoding.

Restrictions

Not all trace stop conditions are available in all products. See the product user guide.

Errors

Invalid Parameter: Parameter out of range.

SetTraceStop (cont.) B4h
GetTraceStop B5h

C-Motion API

```
PMDresult PMDSetTraceStop(PMDAxisInterface axis_intf,
                          PMDAxis traceAxis,
                          PMDuint8 condition,
                          PMDuint8 triggerBit,
                          PMDuint8 triggerState);
PMDresult PMDGetTraceStop(PMDAxisInterface axis_intf,
                          PMDAxis* traceAxis,
                          PMDuint8* condition,
                          PMDuint8* triggerBit,
                          PMDuint8* triggerState);
```

Script API

```
GetTraceStop
SetTraceStop stop
where stop = triggerState*2048 + triggerBit*256 + condition*16 + triggerAxis
```

C# API

```
PMDAxis.GetTraceStop(ref PMDAxisNumber triggerAxis,
                           ref PMDTraceCondition condition,
                           ref Byte bit,
                           ref PMDTraceTriggerState state);
PMDAxis.SetTraceStop(PMDAxisNumber triggerAxis,
                           PMDTraceCondition condition,
                           Byte bit,
                           PMDTraceTriggerState state);
```

Visual Basic API

```
PMDAxis.GetTraceStop(ByRef triggerAxis As PMDAxisNumber,
                           ByRef condition As PMDTraceCondition,
                           ByRef bit As Byte,
                           ByRef state As PMDTraceTriggerState)
PMDAxis.SetTraceStop(ByVal triggerAxis As PMDAxisNumber,
                           ByVal condition As PMDTraceCondition,
                           ByVal bit As Byte,
                           ByVal state As PMDTraceTriggerState)
```

see

GetTraceCount (p. 67), **Set/GetTraceStart** (p. 159), **GetTraceStatus** (p. 68)

SetTraceVariable B6h
GetTraceVariable B7h

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding
variableNumber	*Variable1*	0
	Variable2	1
	Variable3	2
	Variable4	3
traceAxis	*Axis1*	0
variableID		
	None	0
	Position Error	1
	Commanded Position	2
	Commanded Velocity	3
	Commanded Acceleration	4
	Actual Position	5
	Actual Velocity	6
	Active Motor Command	7
	Motion Processor Time	8
	Capture Value	9
	Position Loop Integrator Sum	10
	Position/Outer Loop Derivative Term	11
	Event Status	12
	Activity Status	13
	Signal Status	14
	Phase Angle	15
	Phase Offset	16
	— (Reserved)	17-19
	Analog Raw Channel 0	20
	Analog Raw Channel 1	21
	Analog Raw Channel 2	22
	Analog Raw Channel 3	23
	Analog Raw Channel 4	24
	Analog Raw Channel 5	25
	Analog Raw Channel 6	26
	Analog Raw Channel 7	27
	— (Reserved)	28
	Phase Angle Scaled	29
	— (Reserved)	30
	Phase A Actual Current	31
	— (Reserved)	32-35
	Phase B Actual Current	36
	— (Reserved)	37-39
	d Component Reference	40
	d Component Error	41
	d Component Actual Current	42
	— (Reserved)	43
	d Component Integral Term	44
	d Component Output	45

Arguments (cont.)

variableID (cont.)

q Component Reference	46
q Component Error	47
q Component Actual Current	48
— (Reserved)	49
q Component Integral Term	50
q Component Output	51
Alpha Component Output	52
Beta Component Output	53
Bus Voltage	54
Temperature	55
Drive Status	56
Position/Outer Loop Integral Term	57
— (Reserved)	58-67
Foldback Energy	68
Leg A Current	69
Leg B Current	70
Leg C Current	71
Leg DCurrent	72
Alpha Component Current	73
Beta Component Current	74
PWM A Output	75
PWM B Output	76
PWM C Output	77
— (Reserved)	78
Drive Fault Status	79
— (Reserved)	80-82
Actual Velocity	83
Raw Encoder Reading	84
— (Reserved)	85
Bus Current Supply	86
Bus Current Return	87
— (Reserved)	88
Commutation Error	89
— (Reserved)	90-94
Estimated Velocity	95
Commanded Velocity	96
Velocity Error	97
Velocity Loop Integral Term	98
Velocity Loop Output	99
Velocity Biquad Input	100
Analog Command Biquad Input	101
Tachometer	102
Analog Command	103
Position/Outer loop Output	104
SPI Direct Input	105
— (Reserved)	106,107
Internal Profile Position	108
Internal Profile Velocity	109
Active Operating Mode	110
Analog Raw Channel 8	111
Analog Raw Channel 9	112
— (Reserved)	113-116

SetTraceVariable (cont.) B6h
GetTraceVariable B7h

Arguments (cont.)

variableID (cont.)

	Outer Loop Reference	117
	Outer Loop Feedback	118
	Commutation Error Cause	119
Trajectory Generator	*Commanded Position*	2
	Commanded Velocity	3
	Commanded Acceleration	4
Encoder	*Actual Position*	5
	Actual Velocity	6
	Position Capture Register	9
	Phase Angle	15
	Phase Offset	16
Position Loop	*Position Error*	1
	Position Loop Integrator Sum	10
	Position Loop Integrator Contribution	57
	Position Loop Derivative	11
	Biquad1 Input	64
	Biquad2 Input	65
Status Registers	*Event Status Register*	12
	Activity Status Register	13
	Signal Status Register	14
	Drive Status Register	56
	Drive Fault Status Register	79
Commutation/Phasing	*Active Motor Command*	7
	Phase A Command	17
	Phase B Command	18
	Phase C Command	19
	Phase Angle Scaled	29
Current Loops	*Phase A Reference*	66
	Phase A Error	30
	Phase A Actual Current	31
	Phase A Integrator Sum	32
	Phase A Integrator Contribution	33
	Current Loop A Output	34
	Phase B Reference	67
	Phase B Error	35
	Phase B Actual Current	36
	Phase B Integrator Sum	37
	Phase B Integrator Contribution	38
	Current Loop B Output	39
	D Feedback	40
	Q Feedback	48
	Leg A Current	69
	Leg B Current	70
	Leg C Current	71
	Leg D Current	72
Field Oriented Control	*D Reference*	40
	D Error	41
	D Feedback	42
	D Integrator Sum	43
	D Integrator Contribution	44
	D Output	45

SetTraceVariable (cont.) B6h
GetTraceVariable B7h

Arguments (cont.)

Field Oriented Control (cont.)		
	Q Reference	46
	Q Error	47
	Q Feedback	48
	Q Integrator Sum	49
	Q Integrator Contribution	50
	Q Output	51
	FOC Alpha Output	52
	FOC Beta Output	53
	Phase Alpha Actual Current	73
	Phase Beta Actual Current	74
Motor Output	*Bus Voltage*	54
	Temperature	55
	Foldback Energy	68
	Bus Current Supply	86
	Bus Current Return	87
	PWM Output A	75
	PWM Output B	76
	PWM Output C	77
Analog Inputs	*Analog Input0*	20
	Analog Input1	21
	Analog Input2	22
	Analog Input3	23
	Analog Input4	24
	Analog Input5	25
	Analog Input6	26
	Analog Input7	27
Miscellaneous	*None* (disable variable)	0
	Motion Control IC Time	8

Packet Structure

SetTraceVariable

	0	**B6**h
	15 8	7 0

write	0	*variableNumber*
	15 2	1 0

write	*variableID*	0	*traceAxis*
	15 8	7 4	3 0

GetTraceVariable

	0	**B7**h
	15 8	7 0

write	0	*variableNumber*
	15 2	1 0

read	*variableID*	0	*traceAxis*
	15 8	7 4	3 0

Description

SetTraceVariable assigns the given variable to the specified *variableNumber* location in the trace buffer. Up to four variables may be traced at one time.

All variable assignments must be contiguous starting with *variableNumber* = 0.

GetTraceVariable returns the variable and axis of the specified *variableNumber*.

Example: To set up a three variable trace capturing the commanded acceleration for axis 1, the actual position for axis 1, and the event status word for axis 2, the following sequence of commands would be used. First, a **SetTraceVariable** command with *variableNumber* of 0, *axis* of 0, and *variableID* of 4 would be sent. Then, a **SetTraceVariable** command with *variableNumber* of 1, *axis* of 0, and *variableID* of 5 would be sent. Finally, a **SetTraceVariable** command with a *variableNumber* of 3, *axis* of 0 and *variableID* of 0h would be sent.

The table below summarizes the data type and scaling factor for the trace variables supported by Juno. Note that all values are actually stored in the trace buffer or returned by **GetTraceValue** as 32 bit quantities. If the data type is "16 bit signed" then the data will be sign-extended to 32 bits. If the data type is "16 bit unsigned" then the high word will be zero.

Variable	Encoding	Type	Scaling	Units/Notes
Command Source				
Commanded Position	2	signed 32 bit	unity	counts or microsteps
Commanded Velocity	3	signed 32 bit	$1/2^{16}$	counts/cycle or microsteps/cycle
Commanded Acceleration	4	signed 32 bit	$1/2^{24}$	counts/cycle^2 or microsteps/cycle^2
Analog Command Biquad Input	101	signed 32 bit	$100/2^{30}$	% max analog command input
Analog Command	103	signed 16 bit	$100/2^{14}$	% max analog command input
SPI Direct Input	105	signed 16 bit	$100/2^{15}$	% max SPI input
Internal Profile Position	108	signed 32 bit	unity	counts or microsteps
Internal Profile Velocity	109	signed 32 bit	$1/2^{16}$	counts/cycle or microsteps/cycle
Encoder				
Actual Position	5	signed 32 bit	unity	counts or microsteps
Capture Value	9	signed 32 bit	unity	counts or microsteps
Actual Velocity (not smoothed)	83	signed 32 bit	unity	counts/cycle or microsteps/cycle
Raw Encoder Reading	84	signed 32 bit	unity	counts

Description (cont.)

Variable	Encoding	Type	Scaling	Units/Notes
Position/Outer Loop				
Position Error	1	signed 32 bit	unity	counts or microsteps
Position/Outer Loop Integrator Sum	10	signed 32 bit	$100K_{out}/2^{38}$	% output
Position/Outer Loop Derivative Term	11	signed 32 bit	$100K_{out}/2^{36}$	% output
Position/Outer Loop Integral Term	57	signed 32 bit	$100K_{out}/2^{30}$	% output (eg scaled velocity)
Position/Outer Loop Output	104	signed 32 bit	$100/2^{31}$	% output
Outer Loop Reference	117	signed 32 bit	$100/2^{31}$	% max input
Outer Loop Feedback	118	signed 32 bit	$100/2^{31}$	% max input
Velocity Loop				
Estimated Velocity	95	signed 32 bit	$1/K_{vel}$	counts/cycle
Commanded Velocity	96	signed 32 bit	$1/K_{vel}$	counts/cycle
Velocity Error	97	signed 32 bit	$1/K_{vel}$	counts/cycle
Velocity Loop Integral Term	98	signed 32 bit	$100/(2^{13}K_{out})$	% output
Velocity Loop Output	99	signed 16 bit	$100/2^{15}$	% output
Velocity Biquad Input	100	signed 32 bit	$1/K_{vel}$	counts/cycle
Tachometer	102	signed 16 bit	$100/2^{14}$	% max tachometer analog input
Status Registers				
Event Status	12	unsigned 16 bit	-	see **GetEventStatus**
Activity Status	13	unsigned 16 bit	-	see **GetActivityStatus**
Signal Status	14	unsigned 16 bit	-	see **GetSignalStatus**
Drive Status	56	unsigned 16 bit	-	see **GetDriveStatus**
Drive Fault Status	79	unsigned 16 bit	-	see **GetDriveFaultStatus**
Active Operating Mode	110	unsigned 16 bit	-	see **GetActiveOperating Mode**

Description (cont.)

Variable	Encoding	Type	Scaling	Units/Notes
Commutation/Phasing				
Active Motor Command	7	signed 16 bit	$100/2^{15}$	% output
Phase Angle	15	unsigned 32 bit	unity	counts or microsteps
Phase Offset	16	signed 32 bit	unity	counts
Phase Angle Scaled	29	unsigned 16 bit	$360/2^{15}$	degrees
Commutation Error	89	signed 32 bit	unity	counts (set during phase initialization or correction)
Commutation Error Cause	119	unsigned 16 bit		enumerated value, explanation below
Current Control				
Phase A Actual Current	31	signed 16 bit	$160/2^{15}$	% max leg current analog input
Phase B Actual Current	36	signed 16 bit	$160/2^{15}$	% max leg current analog input
d Component Reference	40	signed 16 bit	$160/2^{15}$	% max leg current analog input
d Component Error	41	signed 16 bit	$160/2^{15}$	% max leg current analog input
d Component Actual Current	42	signed 16 bit	$160/2^{15}$	% max leg current analog input
d Component Integral Term	44	signed 32 bit	$200/2^{15}$	% output
d Component Output	45	signed 16 bit	$100/2^{15}$	% output
q Component Reference	46	signed 16 bit	$160/2^{15}$	% max leg current analog input
q Component Error	47	signed 16 bit	$160/2^{15}$	% max leg current analog input
q Component Actual Current	48	signed 16 bit	$160/2^{15}$	% max leg current analog input
q Component Integral Term	50	signed 32 bit	$200/2^{15}$	% output
q Component Output	51	signed 16 bit	$100/2^{15}$	% output
Alpha Component Output	52	signed 16 bit	$100/2^{15}$	% output
Beta Component Output	53	signed 16 bit	$100/2^{15}$	% output
Leg A Current	69	signed 16 bit	$100/2^{15}$	% max leg current analog input
Leg B Current	70	signed 16 bit	$100/2^{15}$	% max leg current analog input
Leg C Current	71	signed 16 bit	$100/2^{15}$	% max leg current analog input
Leg D Current	72	signed 16 bit	$100/2^{15}$	% max leg current analog input

Description (cont.)

Variable	Encoding	Type	Scaling	Units/Notes
Current Control (cont.)				
Alpha Component Current	73	signed 16 bit	$100/2^{15}$	% max leg current analog input
Beta Component Current	74	signed 16 bit	$100/2^{15}$	% max leg current analog input
Motor Output				
Bus Voltage	54	unsigned 16 bit	$100/2^{16}$	% bus voltage analog input
Temperature	55	unsigned 16 bit	$100/2^{15}$	% temperature analog input
Foldback Energy	68	unsigned 32 bit	see note below	A^2s
PWM A Output	75	signed 16 bit	$100/2^{15}$	% max output
PWM B Output	76	signed 16 bit	$100/2^{15}$	% max output
PWM C Output	77	signed 16 bit	$100/2^{15}$	% max output
Bus Current Supply	86	signed 16 bit	$100/2^{15}$	% max bus current analog input
Bus Current Return	87	signed 16 bit	$100/2^{15}$	% max leg current analog input
Analog Inputs				
Analog Raw Channel 0	20	unsigned 16 bit	$100/2^{16}$	% input
Analog Raw Channel 1	21	unsigned 16 bit	$100/2^{16}$	% input
Analog Raw Channel 2	22	unsigned 16 bit	$100/2^{16}$	% input
Analog Raw Channel 3	23	unsigned 16 bit	$100/2^{16}$	% input
Analog Raw Channel 4	24	unsigned 16 bit	$100/2^{16}$	% input
Analog Raw Channel 5	25	unsigned 16 bit	$100/2^{16}$	% input
Analog Raw Channel 6	26	unsigned 16 bit	$100/2^{16}$	% input
Analog Raw Channel 7	27	unsigned 16 bit	$100/2^{16}$	% input
Analog Raw Channel 8	111	unsigned 16 bit	$100/2^{16}$	% input
Analog Raw Channel 9	112	unsigned 16 bit	$100/2^{16}$	% input
Miscellaneous				
None	0	-	-	Terminates variable list
Motion Processor Time	8	unsigned 32 bit	unity	cycles

SetTraceVariable (cont.) B6h
GetTraceVariable B7h

Description (cont.)

K_{vel} and K_{out} above mean the raw values. K_{out} means either the velocity or position/outer loop parameter, as appropriate.

The foldback energy scaling factor is $t_c(i_{f\,s}/20480)^2 2^{15}$, where t_c is the current loop period of 51.2 x 10^{-6}s and $i_{f\,s}$ is the actual current when a leg current sensor is at full scale.

The Commutation Error Cause trace value indicates the reason for the first commutation error since the value was cleared. Reading the value, either with trace or by using **GetTraceValue**, clears it to zero. The error codes are:

Error Code	Encoding
No error	0
Phase correction too large	1
Invalid Hall state	2
— (Reserved)	3
Pulse phase initialization, signal/noise too low, or no movement	4
Pulse phase initialization, too much movement during ramp	5

The script inteface combines the traceAxis with the variableID in a single code argument as shown below. For example, to set the second trace variable to Active Motor Command (7) for axis 1 (0), code = 7*256 + 0 = 1792, so the command should be:

SetTraceVariable 1 1792

Errors

Invalid Parameter: Unrecognized variableID, trace axis or variableNumber out of range.

C-Motion API

```
PMDresult PMDSetTraceVariable(PMDAxisInterface axis_intf,
                              PMDuint16 variableNumber,
                              PMDAxis traceAxis,
                              PMDuint8 variableID);
PMDresult PMDGetTraceVariable(PMDAxisInterface axis_intf,
                              PMDuint16 variableNumber,
                              PMDAxis* traceAxis,
                              PMDuint8* variableID);
```

Script API

```
GetTraceVariable variableNumber
SetTraceVariable variableNumber code
where code = variableID*256 + traceAxis
```

C# API

```
PMDAxis.GetTraceVariable(PMDTraceVariableNumber VariableNumber,
                         ref PMDAxisNumber TraceAxis,
                         ref PMDTraceVariable variable);
PMDAxis.SetTraceVariable(PMDTraceVariableNumber VariableNumber,
                         PMDAxisNumber TraceAxis,
                         PMDTraceVariable variable);
```

Visual Basic API

```
PMDAxis.GetTraceVariable(ByVal VariableNumber As PMDTraceVariableNumber,
                         ByRef TraceAxis As PMDAxisNumber,
                         ByRef variable As PMDTraceVariable)
PMDAxis.SetTraceVariable(ByVal VariableNumber As PMDTraceVariableNumber,
                         ByVal TraceAxis As PMDAxisNumber,
                         ByVal variable As PMDTraceVariable)
```

see **SetTracePeriod** (p. 158), **SetTraceStart** (p. 159), **SetTraceStop** (p. 162), **GetTraceVariable** (p. 164)

SetVelocity 11h
GetVelocity 4Bh

Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Instance	Encoding		
axis	*Axis1*	0		
	Type	**Range**	**Scaling**	**Units**
velocity	signed 32 bits	-2^{31} *to* $2^{31}-1$	$1/2^{16}$	counts/cycle microsteps/cycle

Packet Structure

SetVelocity

	0	*axis*	**11**h
	15 12	11 8	7 0

write	*velocity* (high-order part)
	31 16
write	*velocity* (low-order part)
	15 0

GetVelocity

	0	*axis*	**4B**h
	15 12	11 8	7 0

read	*velocity* (high-order part)
	31 16
read	*velocity* (low-order part)
	15 0

Description

SetVelocity loads the maximum velocity register for the specified *axis*.

GetVelocity returns the contents of the maximum velocity register.

Scaling example: To load a velocity value of 1.750 counts/cycle, multiply by 65,536 (giving 114,688) and load the resultant number as a 32-bit number; giving 0001 in the high word and C000h in the low word. Numbers returned by **GetVelocity** must correspondingly be divided by 65,536 to convert to units of counts/cycle.

Restrictions

The velocity cannot be negative, except in the Velocity Contouring profile mode.

Errors

Invalid Parameter: Velocity too large for velocity scalar (would cause commanded scaled velocity overflow).
Move In Error: Attempt to change velocity from zero to nonzero without clearing an event that caused a stop.

C-Motion API

```
PMDresult PMDSetVelocity(PMDAxisInterface axis_intf,
                         PMDint32 velocity);
PMDresult PMDGetVelocity(PMDAxisInterface axis_intf,
                         PMDint32* velocity);
```

Script API

```
GetVelocity
SetVelocity velocity
```

C# API

```
Int32 velocity = PMDAxis.Velocity;
PMDAxis.Velocity = velocity;
```

Visual Basic API

```
Int32 velocity = PMDAxis.Velocity
PMDAxis.Velocity = velocity
```

see

Set/GetAcceleration (p. 84), **Set/GetDeceleration** (p. 113)

WriteBuffer C8h

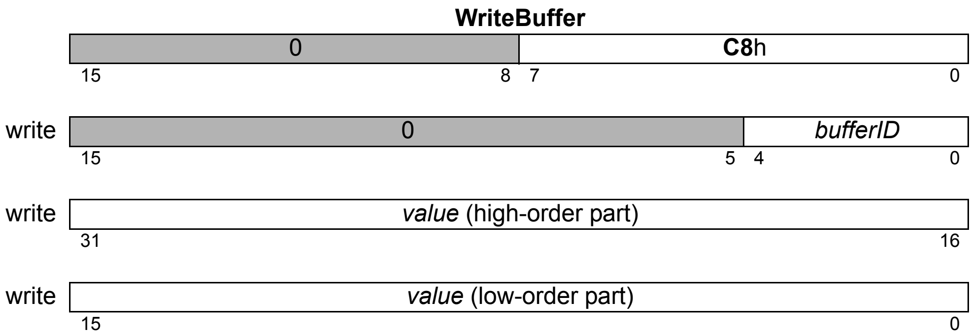
Motor Types

DC Brush	Brushless DC	Microstepping

Arguments

Name	Type	Range
bufferID	unsigned 16 bits	0 *to* 7
value	signed 32 bits	-2^{31} *to* $2^{31}-1$

Packet Structure



Description

WriteBuffer writes the 32-bit ***value*** into the location pointed to by the write buffer index in the specified buffer. After the contents have been written, the write index is incremented by 1. If the result is equal to the buffer length (set by **SetBufferLength**), the index is reset to zero (0).

Restrictions

WriteBuffer may only be used to write to RAM, it cannot write to buffers pointing to NVRAM.

Errors

Invalid Parameter: bufferID out of range
Trace Running: Attempt to write to trace buffer while trace is active.
Read-only Buffer: Attempt to write to NVRAM.
Block out of Bounds: Attempt to write to a zero-length buffer.

C-Motion API

```
PMDresult PMDWriteBuffer(PMDAxisInterface axis_intf,
                         PMDuint16 bufferID,
                         PMDint32 data);
```

Script API

```
WriteBuffer bufferID data
```

C# API

```
PMDAxis.WriteBuffer(Int16 bufferID, Int32 data);
```

Visual Basic API

```
PMDAxis.WriteBuffer(ByVal bufferID As Int16, ByVal data As Int32);
```

see

ReadBuffer (p. 76), **Set/GetBufferWriteIndex** (p. 98)

8. Instruction Summary Tables

8.1 Descriptions by Functional Category

Interrupts		Page
ClearInterrupt	Reset interrupt.	33
Set/GetInterruptMask	Set/Get interrupt event mask.	132

Motor Phase and Commutation		
Set/GetCommutationMode	Set/Get the commutation phasing mode.	102
Set/GetPhaseCorrectionMode	Set/Get phase correction method.	147
Set/GetCommutationParameter	Set/Get phase counts and other commutation parameters.	103
Set/GetPhaseParameter	Set/Get phase initialization parameters.	149
Set/GetPhaseInitializeMode	Set/Get phase initialization method.	148
InitializePhase	Perform phase initialization procedure.	71

Current Loops		
CalibrateAnalog	Determine offsets to zero analog inputs.	31
Set/GetAnalogCalibration	Set/Get analog offsets.	88
Set/GetCurrentControlMode	Set/Get current control mode (FOC or third leg floating).	108
Set/GetFOC	Set/Get parameters for current control.	130
GetFOCValue	Get value of current control state.	54

Digital Servo Filter		
ClearPositionError	Adjust commanded position to make error zero.	34
GetPositionError	Get actual position error.	60
Set/GetDriveCommandMode	Set/Get mode for commanding position, velocity, or torque	114
Set/GetLoop	Set/Get parameter for position/outer or velocity loop	134
GetLoopValue	Get value of position/outer or velocity loop state	58

Encoder		
AdjustActualPosition	Change the current encoder position by a specified offset.	30
Set/GetActualPosition	Set/Get the current encoder position.	86
Set/GetActualPositionUnits	Set/Get units of encoder position for step motors, counts or microsteps.	87
GetActualVelocity	Get the actual encoder velocity, without smoothing.	41
GetCaptureValue	Get the most recent index capture encoder position.	42
Set/GetEncoderSource	Set/Get the type of position feedback.	121
Set/GetEncoderToStepRatio	Set/Get the ratio of encoder counts to microsteps.	123

Motor Output		
GetActiveMotorCommand	Get the active commanded motor output	37
GetDriveValue	Read drive bus voltage, bus current, or temperature.	50
Set/GetMotorCommand	Set/Get the motor command if position/outer and velocity loops are disabled.	138
Set/GetMotorType	Set/Get the motor type.	142

Motor Output

Set/GetOutputMode	Set/Get the method of driving the motor amplifier.	146
Set/GetDrivePWM	Set/Get various PWM parameters, eg signal sense, frequency, and dead time.	119
Set/GetCurrentFoldback	Set/Get current foldback limits.	109
Set/GetCurrent	Set/Get current commands for driving step motors.	106
Set/GetCurrentLimit	Set/Get the maximum current that the velocity or position/outer loop may command.	111

Operating Mode and Event Control

Set/GetOperatingMode	Set/Get the static operating mode of an axis.	144
RestoreOperatingMode	Restore the active operating mode from the static operating mode of an axis.	83
GetActiveOperatingMode	Get the active operating mode of an axis.	38
Set/GetEventAction	Set/Get the response to events or other exceptional conditions.	125

Postion Servo Loop Control

Set/GetSampleTime	Set/Get the profile and servo loop sample time .	151
GetTime	Get the current IC time, in commutation periods.	66

Profile Generation

Set/GetAcceleration	Set/Get the maximum acceleration for the internal profile.	84
GetCommandedAcceleration	Get the current commanded profile acceleration.	43
GetCommandedPosition	Get the current commanded position.	44
GetCommandedVelocity	Get the current commanded (not scaled) velocity.	45
Set/GetDeceleration	Set/Get the maximum deceleration, if different from the maximum acceleration.	113
Set/GetVelocity	Set/Get the maximum velocity for the internal profile.	174

RAM Buffers

Set/GetBufferLength	Set/Get the length of a memory buffer.	92
Set/GetBufferReadIndex	Set/Get the index of the next read from a memory buffer.	94
Set/GetBufferStart	Set/Get the starting address of a memory buffer.	96
Set/GetBufferWriteIndex	Set/Get the index of the next write to a memory buffer.	98
ReadBuffer	Read a 32 bit double word from a RAM buffer.	76
ReadBuffer16	Read a 16 bit word from an NVRAM buffer.	77
WriteBuffer	Write a 32 bit double word to a RAM buffer.	176

Drive

Set/GetDriveFaultParameter	Set/Get some drive safety parameters.	116
Set/GetFaultOutMask	Set/Get the event mask for driving the FaultOut signal.	128
GetDriveFaultStatus	Get a latched register showing some drive faults status.	46
GetDriveValue	Get some current drive state.	50
ClearDriveFaultStatus	Clear (zero) all drive fault bits.	32

Status Registers

GetActivityStatus	Get a register showing some current activity state.	40
GetDriveStatus	Get a register showing some current drive state.	48
GetEventStatus	Get a latched register showing some significant events.	52
GetSignalStatus	Get the current status of some input/output signals.	64
Set/GetSignalSense	Set/Get the logical sense of some input/output signals.	155

Status Registers		
ResetEventStatus	Clear (zero) some event bits.	82

Traces		
GetTraceCount	Get the number of trace values that have been stored.	67
Set/GetTraceMode	Set/Get the trace mode (one-time or rolling).	157
Set/GetTracePeriod	Set/Get the frequency of trace captures.	158
Set/GetTraceStart	Set/Get the condition that will start a trace.	159
Set/GetTraceStop	Set/Get the condition that will stop a trace.	162
GetTraceStatus	Get the trace status word.	68
Set/GetTraceVariable	Set/Get the set of quantities to save in a trace.	164
GetTraceValue	Get the current value of a traceable quantity.	69
Set/GetTraceTriggerValue	Set/Get a value to be used to determine trace start or stop.	90

Communications		
Set/GetCANMode	Set/Get the CANBus baud rate and node identifier.	100
GetInstructionError	Get and clear command error codes.	56
Set/GetSerialPortMode	Set/Get the serial port configuration.	153
GetSPIMode	Get the current SPI mode: host command or direct.	65
GetRuntimeError	Get and clear error codes not associated with a command.	63

Miscellaneous		
GetProductInfo	Get fixed configuration and version information.	61
ExecutionControl	Control some aspects of NVRAM IC initialization.	35
GetVersion	Legacy version command, returns zero.	70
NoOperation	Perform no operation, used to verify communications.	74
Reset	Reset IC.	78
NVRAM	Program non-volatile memory.	72
ReadAnalog	Read a raw analog input.	75

8.2 Alphabetical Listing

Get/Set instructions pairs are shown together on the same line of the table.

Instruction	Code	Instruction	Code	Page
AdjustActualPosition	F5h			30
CalibrateAnalog	6Fh			31
ClearDriveFaultStatus	6Ch			32
ClearInterrupt	ACh			33
ClearPositionError	47h			34
ExecutionControl	35h			35
GetActiveMotorCommand	3Ah			37
GetActiveOperatingMode	57h			38
GetActivityStatus	A6h			40
GetActualVelocity	ADh			41
GetCaptureValue	36h			42
GetCommandedAcceleration	A7h			43
GetCommandedPosition	1Dh			44

Instruction	Code	Instruction	Code	Page
GetCommandedVelocity	1Eh			45
GetDriveFaultStatus	6Dh			46
GetDriveStatus	0Eh			48
GetDriveValue	70h			50
GetEventStatus	31h			52
GetFOCValue	5Ah			54
GetInstructionError	A5h			56
GetLoopValue	38h			58
GetPositionError	99h			60
GetProductInfo	01h			61
GetRuntimeError	3Dh			63
GetSPIMode	0Bh			65
GetSignalStatus	A4h			64
GetTime	3Eh			66
GetTraceCount	BBh			67
GetTraceStatus	BAh			68
GetTraceValue	28h			69
GetVersion	8Fh			70
InitializePhase	7Ah			71
NVRAM	30h			72
NoOperation	00h			74
ReadAnalog	EFh			75
ReadBuffer	C9h			76
ReadBuffer16	CDh			77
Reset	39h			78
ResetEventStatus	34h			82
RestoreOperatingMode	2Eh			83
SetAcceleration	90h	**GetAcceleration**	4Ch	84
SetActualPosition	4Dh	**GetActualPosition**	37h	86
SetActualPositionUnits	BEh	**GetActualPositionUnits**	BFh	87
SetAnalogCalibration	29h	**GetAnalogCalibration**	2Ah	88
SetBufferLength	C2h	**GetBufferLength**	C3h	92
SetBufferReadIndex	C6h	**GetBufferReadIndex**	C7h	94
SetBufferStart	C0h	**GetBufferStart**	C1h	96
SetBufferWriteIndex	C4h	**GetBufferWriteIndex**	C5h	98
SetCANMode	12h	**GetCANMode**	15h	100
SetCommutationMode	E2h	**GetCommutationMode**	E3h	102
SetCommutationParameter	63h	**GetCommutationParameter**	64h	103
SetCurrent	5Eh	**GetCurrent**	5Fh	106
SetCurrentControlMode	43h	**GetCurrentControlMode**	44h	108
SetCurrentFoldback	41h	**GetCurrentFoldback**	42h	109
SetCurrentLimit	06h	**GetCurrentLimit**	07h	111
SetDeceleration	91h	**GetDeceleration**	92h	113
SetDriveCommandMode	7Eh	**GetDriveCommandMode**	7Fh	114
SetDriveFaultParameter	62h	**GetDriveFaultParameter**	60h	116
SetDrivePWM	23h	**GetDrivePWM**	24h	119
SetEncoderSource	DAh	**GetEncoderSource**	DBh	121
SetEncoderToStepRatio	DEh	**GetEncoderToStepRatio**	DFh	123
SetEventAction	48h	**GetEventAction**	49h	125
SetFOC	F6h	**GetFOC**	F7h	130
SetFaultOutMask	FBh	**GetFaultOutMask**	FCh	128

Instruction	Code	Instruction	Code	Page
SetInterruptMask	2Fh	**GetInterruptMask**	56h	132
SetLoop	78h	**GetLoop**	79h	134
SetMotorCommand	77h	**GetMotorCommand**	69h	138
SetMotorType	02h	**GetMotorType**	03h	142
SetOperatingMode	65h	**GetOperatingMode**	66h	144
SetOutputMode	E0h	**GetOutputMode**	6Eh	146
SetPhaseCorrectionMode	E8h	**GetPhaseCorrectionMode**	E9h	147
SetPhaseInitializeMode	E4h	**GetPhaseInitializeMode**	E5h	148
SetPhaseParameter	85h	**GetPhaseParameter**	86h	149
SetSampleTime	3Bh	**GetSampleTime**	3Ch	151
SetSerialPortMode	8Bh	**GetSerialPortMode**	8Ch	153
SetSignalSense	A2h	**GetSignalSense**	A3h	155
SetTraceMode	B0h	**GetTraceMode**	B1h	157
SetTracePeriod	B8h	**GetTracePeriod**	B9h	158
SetTraceStart	B2h	**GetTraceStart**	B3h	159
SetTraceStop	B4h	**GetTraceStop**	B5h	162
SetTraceTriggerValue	D6h	**GetTraceTriggerValue**	D7h	90
SetTraceVariable	B6h	**GetTraceVariable**	B7h	164
SetVelocity	11h	**GetVelocity**	4Bh	174
WriteBuffer	C8h			176

8.3 Numerical Listing

Code	Instruction	Page	Code	Instruction	Page
00h	**NoOperation**	74	60h	**GetDriveFaultParameter**	116
01h	**GetProductInfo**	61	62h	**SetDriveFaultParameter**	116
02h	**SetMotorType**	142	63h	**SetCommutationParameter**	103
03h	**GetMotorType**	142	64h	**GetCommutationParameter**	103
06h	**SetCurrentLimit**	140	65h	**SetOperatingMode**	144
07h	**GetCurrentLimit**	140	66h	**GetOperatingMode**	144
0Bh	**GetSPIMode**	65	69h	**GetMotorCommand**	138
0Eh	**GetDriveStatus**	48	6Ch	**ClearDriveFaultStatus**	32
11h	**SetVelocity**	174	6Dh	**GetDriveFaultStatus**	46
12h	**SetCANMode**	100	6Eh	**GetOutputMode**	146
15h	**GetCANMode**	100	6Fh	**CalibrateAnalog**	31
1Dh	**GetCommandedPosition**	44	70h	**GetDriveValue**	50
1Eh	**GetCommandedVelocity**	45	77h	**SetMotorCommand**	138
23h	**SetDrivePWM**	119	78h	**SetLoop**	134
24h	**GetDrivePWM**	119	79h	**GetLoop**	134
28h	**GetTraceValue**	69	7Ah	**InitializePhase**	71
29h	**SetAnalogCalibration**	88	7Eh	**SetDriveCommandMode**	114
2Ah	**GetAnalogCalibration**	88	7Fh	**GetDriveCommandMode**	114
2Eh	**RestoreOperatingMode**	83	85h	**SetPhaseParameter**	149
2Fh	**SetInterruptMask**	132	86h	**GetPhaseParameter**	149
30h	**NVRAM**	72	8Bh	**SetSerialPortMode**	153
31h	**GetEventStatus**	52	8Ch	**GetSerialPortMode**	153
34h	**ResetEventStatus**	82	8Fh	**GetVersion**	70
35h	**ExecutionControl**	35	90h	**SetAcceleration**	84
36h	**GetCaptureValue**	42	91h	**SetDeceleration**	113
37h	**GetActualPosition**	86	92h	**GetDeceleration**	113
38h	**GetLoopValue**	58	99h	**GetPositionError**	60
39h	**Reset**	78	A2h	**SetSignalSense**	155
3Ah	**GetActiveMotorCommand**	37	A3h	**GetSignalSense**	155
3Bh	**SetSampleTime**	151	A4h	**GetSignalStatus**	64
3Ch	**GetSampleTime**	151	A5h	**GetInstructionError**	56
3Dh	**GetRuntimeError**	63	A6h	**GetActivityStatus**	40
3Eh	**GetTime**	66	A7h	**GetCommandedAcceleration**	43
41h	**SetCurrentFoldback**	109	ACh	**ClearInterrupt**	33
42h	**GetCurrentFoldback**	109	ADh	**GetActualVelocity**	41
43h	**SetCurrentControlMode**	108	B0h	**SetTraceMode**	157
44h	**GetCurrentControlMode**	108	B1h	**GetTraceMode**	157
47h	**ClearPositionError**	34	B2h	**SetTraceStart**	159
48h	**SetEventAction**	125	B3h	**GetTraceStart**	159
49h	**GetEventAction**	125	B4h	**SetTraceStop**	162
4Bh	**GetVelocity**	174	B5h	**GetTraceStop**	162
4Ch	**GetAcceleration**	84	B6h	**SetTraceVariable**	164
4Dh	**SetActualPosition**	86	B7h	**GetTraceVariable**	164
56h	**GetInterruptMask**	132	B8h	**SetTracePeriod**	158
57h	**GetActiveOperatingMode**	38	B9h	**GetTracePeriod**	158
5Ah	**GetFOCValue**	54	BAh	**GetTraceStatus**	68
5Eh	**SetCurrent**	106	BBh	**GetTraceCount**	67
5Fh	**GetCurrent**	106	BEh	**SetActualPositionUnits**	87

Code	Instruction	Page	Code	Instruction	Page
BFh	**GetActualPositionUnits**	87			
C0h	**SetBufferStart**	96			
C1h	**GetBufferStart**	96			
C2h	**SetBufferLength**	92			
C3h	**GetBufferLength**	92			
C4h	**SetBufferWriteIndex**	98			
C5h	**GetBufferWriteIndex**	98			
C6h	**SetBufferReadIndex**	94			
C7h	**GetBufferReadIndex**	94			
C8h	**WriteBuffer**	176			
C9h	**ReadBuffer**	76			
CDh	**ReadBuffer16**	77			
D6h	**SetTraceTriggerValue**	90			
D7h	**GetTraceTriggerValue**	90			
DAh	**SetEncoderSource**	121			
DBh	**GetEncoderSource**	121			
DEh	**SetEncoderToStepRatio**	123			
DFh	**GetEncoderToStepRatio**	123			
E0h	**SetOutputMode**	146			
E2h	**SetCommutationMode**	102			
E3h	**GetCommutationMode**	102			
E4h	**SetPhaseInitializeMode**	148			
E5h	**GetPhaseInitializeMode**	148			
E8h	**SetPhaseCorrectionMode**	147			
E9h	**GetPhaseCorrectionMode**	147			
EFh	**ReadAnalog**	75			
F5h	**AdjustActualPosition**	30			
F6h	**SetFOC**	130			
F7h	**GetFOC**	130			
FBh	**SetFaultOutMask**	128			
FCh	**GetFaultOutMask**	128			

This page intentionally left blank.

For additional information, or for technical assistance,
please contact PMD at (978) 266-1210.

You may also e-mail your request to support@pmdcorp.com

Visit our website at http://www.pmdcorp.com

Performance Motion Devices, Inc.
1 Technology Park Drive
Westford, MA 01886