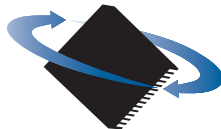


# **Juno Velocity & Torque Control IC Programming Reference**



**PERFORMANCE  
MOTION DEVICES**

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



---

## 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.

Copyright 1998–2019 by 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.

## 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 including 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

<b>1. The Juno MC78113 IC Family .....</b>	<b>7</b>
1.1 Introduction.....	7
1.2 Family Overview.....	8
<b>2. C-Motion .....</b>	<b>9</b>
2.1 Introduction .....	9
2.2 C-Motion Versions .....	9
2.3 Files.....	10
2.4 Using C-Motion .....	10
<b>3. Visual Basic Interface.....</b>	<b>13</b>
3.1 Introduction .....	13
3.2 Visual Basic Classes .....	13
<b>4. C# Interface .....</b>	<b>15</b>
4.1 Introduction .....	15
4.2 Visual C# Classes .....	15
<b>5. Script Interface.....</b>	<b>17</b>
5.1 Introduction .....	17
<b>6. Non-Volatile (NVRAM) Storage.....</b>	<b>19</b>
6.1 Introduction .....	19
<b>7. Instruction Reference .....</b>	<b>29</b>
7.1 How to Use This Reference .....	29
<b>8. Instruction Summary Tables .....</b>	<b>177</b>
8.1 Descriptions by Functional Category .....	177
8.2 Alphabetical Listing .....	179
8.3 Numerical Listing .....	182

*This page intentionally left blank.*

# 1. The Juno MC78113 IC Family

1

## *In This Chapter*



- Introduction
- Family Overview

## 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
<b>Motor Type &amp; Control Mode</b>					
Motor Type	Step motor	DC Brush	DC Brush	Brushless DC	Brushless DC
Velocity			✓		✓
Torque/current	✓	✓	✓	✓	✓
Position & outer loop			✓		✓
<b>Host Interface</b>					
Serial point-to-point	✓	✓	✓	✓	✓
Serial multi-drop			✓		✓
SPI			✓		✓
CANbus			✓		✓
<b>Command Input</b>					
Analog velocity or torque		✓	✓	✓	✓
SPI velocity or torque		✓	✓	✓	✓
Pulse & direction	✓		✓		✓
SPI position increment	✓				✓
<b>Motion I/O</b>					
Quadrature encoder input	✓ (MC74113 & MC74113N only)		✓	✓	✓
Hall sensor input				✓	✓
Tachometer input			✓		✓
AtRest input	✓				
FaultOut output	✓	✓	✓	✓	✓
HostInterrupt output	✓	✓	✓	✓	✓
<b>Amplifier Control</b>					
PWM High/Low	✓	✓	✓	✓	✓
PWM Sign/Magnitude	✓	✓	✓		
<b>DC Bus &amp; Safety</b>					
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," 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
PMDdecode.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
NIV\*.*	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 hAxisI;

// open interface to PMD Juno processor on COM1
PMDSetupAxisInterface_Serial(&hAxisI, PMDAxisI, I);
```

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(&hAxisI, 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
<b>PMDSerial_SetConfig</b>	<i>axis_handle.transport_data</i> <i>baudrate</i> <i>parity</i>	Used to set serial port configuration after PMDSetupAxisInterface_Serial.
<b>PMDSerial_SetProtocol</b>	<i>axis_handle.transport_data</i> <i>mode</i>	Used to set serial port mode after PMDSetupAxisInterface_Serial, required for multi-drop communication.
<b>PMDSerial_SetMultiDropAddress</b>	<i>axis_handle.transport_data</i> <i>address</i>	Used to set multi-drop address after PMDSetupAxisInterface_Serial, required for multi-drop communication.
<b>PMDCreateMultiDropHandle</b>	<i>dest_axis_handle</i> <i>src_axis_handle</i> <i>axis_number</i> <i>nodeID</i>	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.
<b>PMDSetupAxisInterface_Serial</b>	<i>axis_handle</i> , <i>axis_number</i> <i>port_number</i>	Used to setup an axis interface connection for communicating over a RS232 or RS485 serial bus.
<b>PMDSetupAxisInterface_CAN</b>	<i>axis_handle</i> , <i>axis_number</i> <i>board_number</i>	Used to setup an axis interface connection for communicating over a CAN bus.
<b>PMDSetupAxisInterface_SPI</b>	<i>axis_handle</i> <i>axis_number</i> <i>device</i>	Used to setup an axis interface connection for communicating over an SPI bus.
<b>PMDCloseAxisInterface</b>	<i>axis_handle</i>	Should be called to terminate an interface connection.
<b>PMDGetErrorMessage</b>	<i>ErrorCode</i>	Returns a character string representation of the corresponding PMD chip or C-Motion error code.
<b>GetCMotionVersion</b>	<i>MajorVersion</i> <i>MinorVersion</i>	Returns the major and minor version number of C-Motion.

This page intentionally left blank.

## 3. Visual Basic Interface

3

### 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

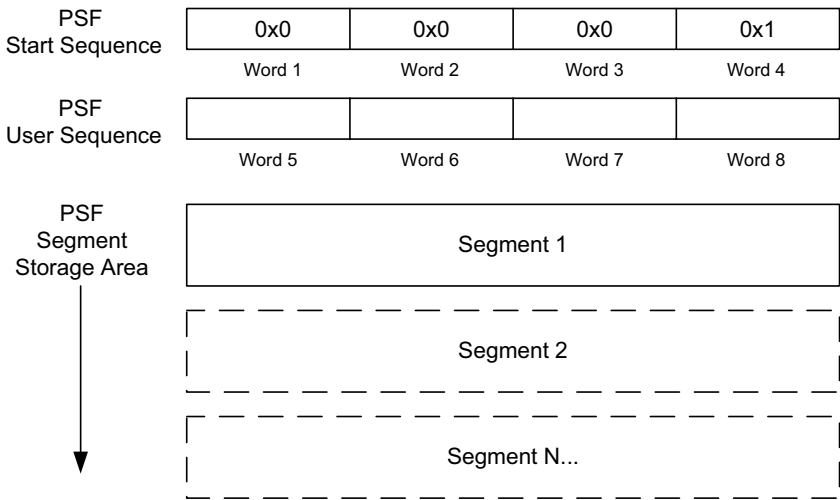
## 6.1 Introduction

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 utilizes 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

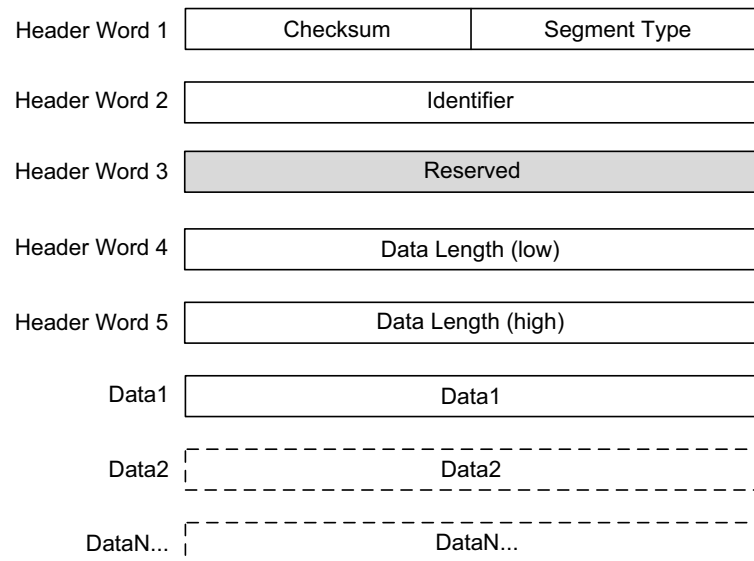
PSF (PMD Structured data Format) is a general purpose data storage format designed for use with non-volatile storage memory such as provided by Juno IC. PSF provides a method to store and label initialization information used by Juno during startup, as well as to allow user-defined storage in NVRAM.

[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.

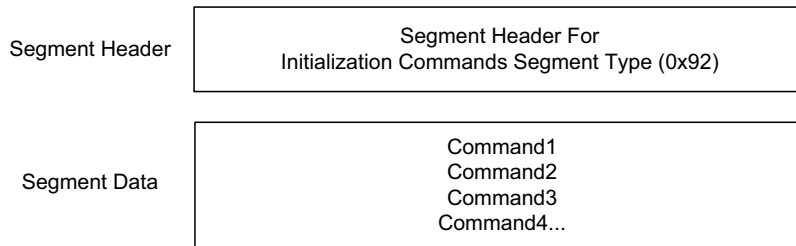
### 6.1.3 Pre-Defined Segment Types

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 6-3 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.

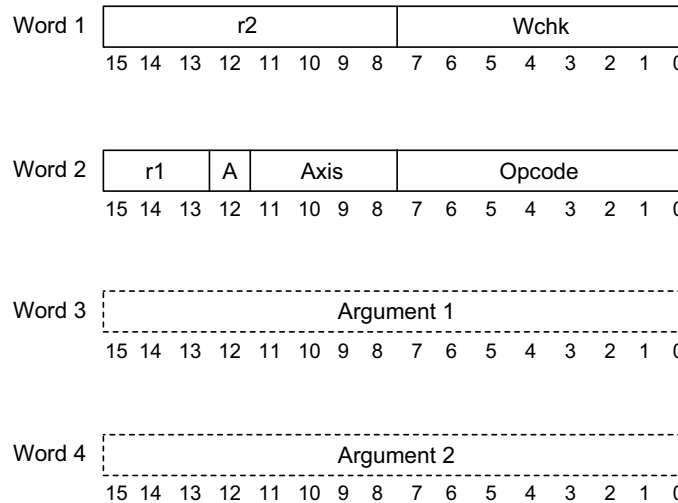
Segment Data Address	Mnemonic	Stored Code (in hex)	Comments
Data1	SetDriveFaultParameter 2 I	0x00EF	Checksum
Data2		0x0062	Axis (0) and opcode
Data3		0x0002	Argument 1: event handling mode
Data4		0x0001	Argument 2: automatic event recovery
Data5	ExecutionControl 0 256	0x001F	Checksum
Data6		0x0035	Axis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

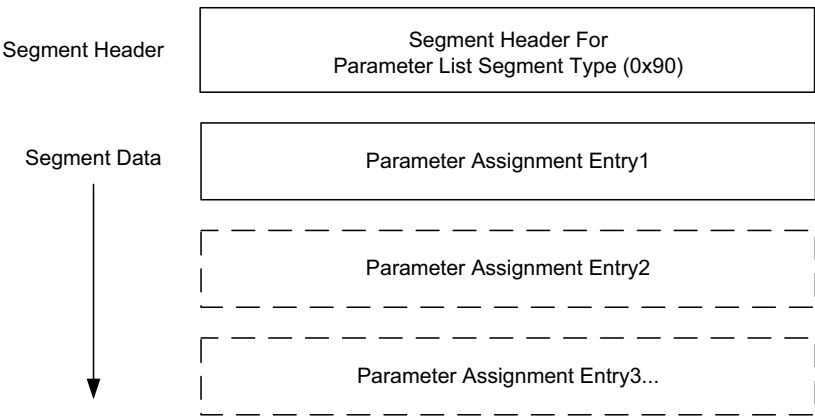


Figure 6-5:  
Parameter List  
Segment  
Format

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

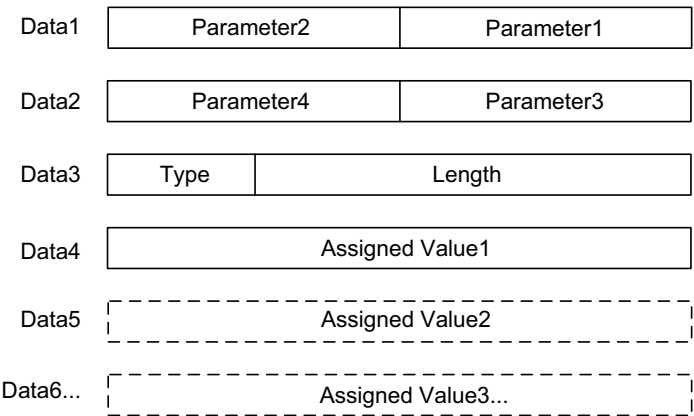


Figure 6-6:  
Format of  
Parameter  
Assignment  
Entry

[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 "axis.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	Addr	Word	Contents	Comments
0	0x0000	0	PSF Start Sequence	39	0x006c	"l"	
1	0x0000	0		40	0x0065	"e"	
2	0x0000	0		41	0x002E	."	
3	0x0001	1		42	0x0074	"t"	
4	0x0005	5	PSF User Sequence	43	0x0078	"x"	
5	0x0006	6		44	0x0074	"t"	
6	0x0007	7		45	0x4457	'W', 'D'	
7	0x0008	8		46	0x0000	nul, nul	
8	0x2D90	Chksum, seg. type	Parameter List Segment	47	0x000A	type, length	Assign WD = "2017-01-25"
9	0x0000	identifier		48	0x0032	"2"	
10	0x0000	reserved		49	0x0030	"0"	
11	0x002D	length low		50	0x0031	"1"	
12	0x0000	length high	Assign CN = "Init1"	51	0x0037	"7"	
13	0x4E43	'C', 'N'		52	0x002D	."	
14	0x0000	nul, nul		53	0x0030	"0"	
15	0x0005	type, length		54	0x0031	"1"	
16	0x0049	"l"	Assign CVER="1.2"	55	0x002D	."	Initialization Comments Segment
17	0x006E	"n"		56	0x0032	"2"	
18	0x0069	"i"		57	0x0035	"5"	
19	0x0074	"t"		58	0xB692	chksum, seg. type	
20	0x0031	"1"	Assign DESC = "test"	59	0x0000	identifier	
21	0x5643	'C', 'V'		60	0x0000	reserved	
22	0x5245	'E', 'R'		61	0x000C	length low	
23	0x0003	type, length		62	0x0000	length high	
24	0x0031	"1"	Assign FN = "file.txt"	63	0x00EF		SetDriveFault Parameter 2 1
25	0x002E	."		64	0x0062		
26	0x0032	"2"		65	0x0002		
27	0x4544	'D', 'E'		66	0x0001		
28	0x4353	'S', 'C'	ExecutionControl 0 256	67	0x001F		
29	0x0004	type, length		68	0x0035		
30	0x0074	"t"		69	0x0000		
31	0x0065	"e"		70	0x0000		
32	0x0073	"s"	SetOperatingMode 7	71	0x0100		
33	0x0074	"t"		72	0x00E8		
34	0x4E46	'F', 'N'		73	0x0065		
35	0x0000	nul, nul		74	0x0007		
36	0x0008	type, length					
37	0x0066	"f"					
38	0x0069	"i"					

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.

```
#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 6-8:**  
**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 [Section 6.1.1, “PMD Structured Data Format,” on page 19](#)

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.

This page intentionally left blank.

## 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:

<b>Name</b>	The instruction mnemonic is shown at the left, its hexadecimal code at the right.
<b>Motor Types</b>	The motor types to which this command applies. Supported motor types are printed in black; unsupported motor types for the command are greyed out.
<b>Arguments</b>	<p>There are two types of arguments: encoded-field and numeric.</p> <p>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 <i>italic</i>) is that shown in the generic syntax. Instance (in <i>italic</i>) is the mnemonic used to represent the data value. Encoding is the value assigned to the field for that instance.</p> <p>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.</p>
<b>Packet Structure</b>	<p>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.</p> <p>The hex code of the instruction is shown in boldface.</p> <p>Argument names are shown in their respective words or fields.</p> <p>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.</p>
<b>Description</b>	Describes what the instruction does and any special information relating to the instruction.
<b>Restrictions</b>	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.
<b>Errors</b>	Lists the error codes that may be returned by the instruction and what they mean in the context of the instruction.
<b>C-Motion API</b>	The syntax of the C function call in the PMD C-Motion library that implements this motion control IC command.
<b>Script API</b>	The syntax for the command in Pro-Motion scripts used for programming NVRAM.
<b>C# API</b>	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.
<b>Visual Basic API</b>	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.
<b>see</b>	Refers to related instructions.

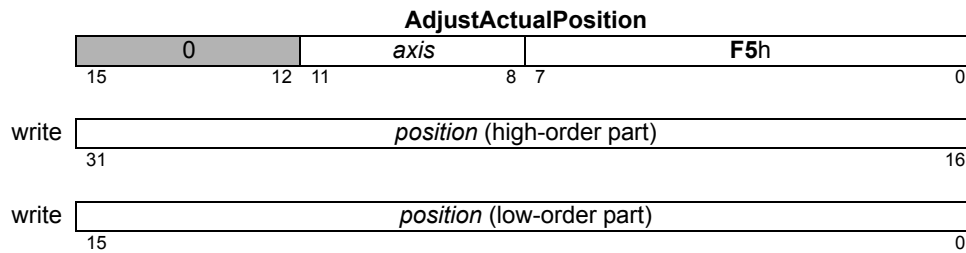
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

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

## Packet Structure



## 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. 59), [GetActualVelocity](#) (p. 41), [Set/GetActualPositionUnits](#) (p. 86), [Set/GetActualPosition](#) (p. 85)

## Motor Types

DC Brush	Brushless DC	Microstepping	
----------	--------------	---------------	--

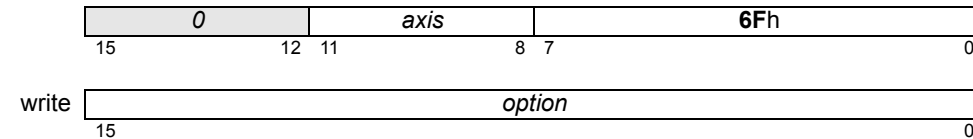
## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0
<i>option</i>	<i>leg currents</i>	0
	<i>analog command</i>	1
	<i>tachometer</i>	2

## Returned data

None

## Packet Structure



## 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. 87), [ReadAnalog](#) (p. 74), [NVRAM](#) (p. 71), [ExecutionControl](#) (p. 35)

## Motor Types

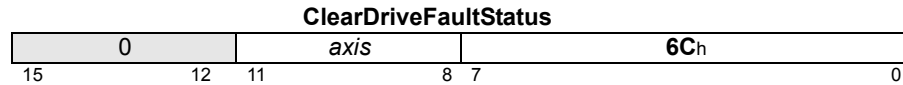
DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

## Packet

## Structure



## 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](#))



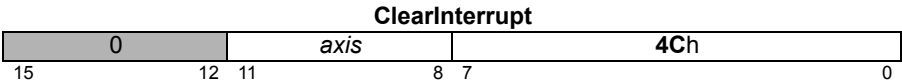
Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

Packet Structure



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. 150) 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. 131), **ResetEventStatus** (p. 81).

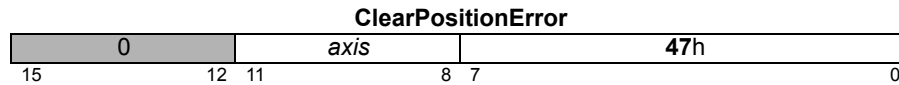
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

## Packet Structure



## 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. 59](#))

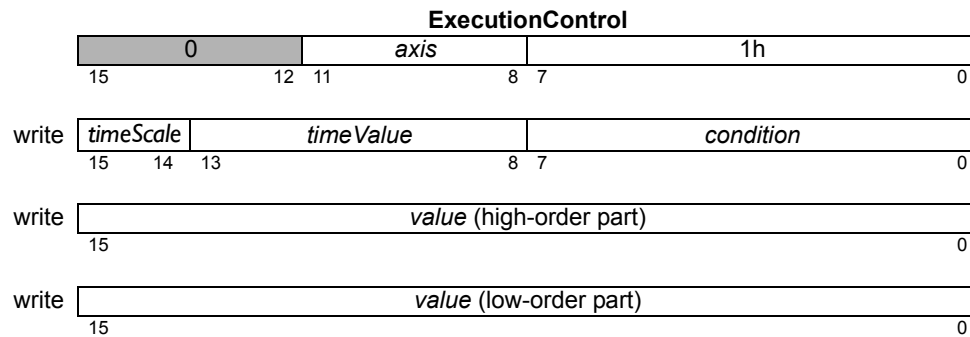
## Motor Types

Brush DC	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

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

## Packet Structure



## 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  $\mu$ s 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  $\sim$ *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  $63 \times 2^{22} = 264,241,152$ , 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  $8 + 256 \times 3 + 16384 \times 1 = 17160$

**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. 71), **GetEventStatus** (p. 52), **GetActivityStatus** (p. 40), **GetDriveStatus** (p. 48), **GetSignalStatus** (p. 63), **GetInstructionError** (p. 56)

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

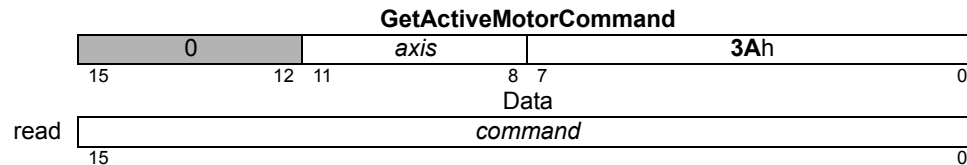
## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

## Returned data

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

## Packet Structure



### 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. 137), **Set/GetOperatingMode** (p. 143), **GetActiveOperatingMode** (p. 38)



<b>Description (cont.)</b>	<p>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.</p> <p>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.</p>
<b>Restrictions</b>	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.
<b>Errors</b>	None
<b>C-Motion API</b>	<pre>PMDresult <b>PMDGetActiveOperatingMode</b>(PMDAxisInterface <i>axis_intf</i>,  PMDuint16* <i>mode</i>);</pre>
<b>Script API</b>	<b>GetActiveOperatingMode</b>
<b>C# API</b>	<code>UInt16 mode = <b>PMDAxis.ActiveOperatingMode</b>;</code>
<b>Visual Basic API</b>	<code>UInt16 mode = <b>PMDAxis.ActiveOperatingMode</b></code>
<b>see</b>	<b>GetOperatingMode</b> (p. 143), <b>RestoreOperatingMode</b> (p. 82), <b>Set/GetEventAction</b> (p. 124), <b>SetDeceleration</b> (p. 112), <b>SetAcceleration</b> (p. 83)

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

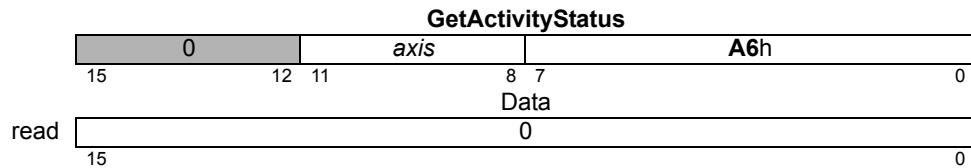
## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

## Returned Data

	Type	
<i>status</i>	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. 63), [GetDriveStatus](#) (p. 48)



Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

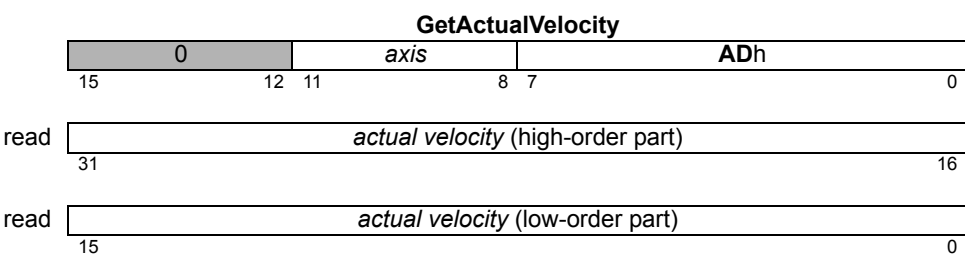
Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

Returned Data

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

Packet Structure



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. 85)

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

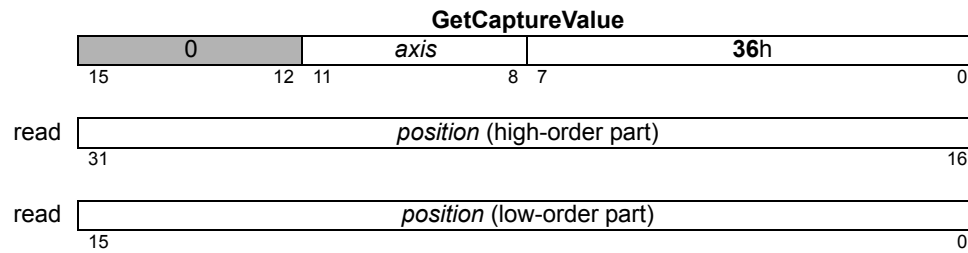
## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

## Returned data

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

## Packet Structure



## 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. 86), **GetActivityStatus** (p. 40)

Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

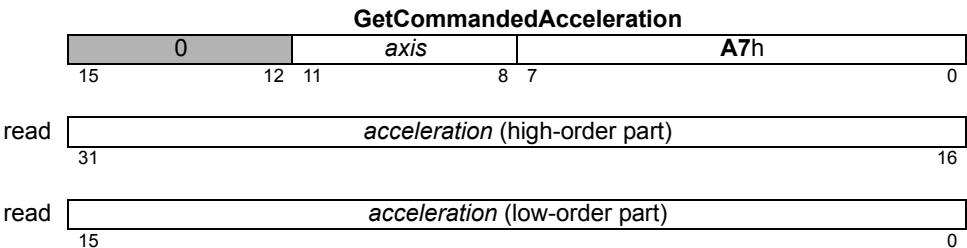
Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

Returned data

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

Packet Structure



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<sup>2</sup> 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. 113)

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

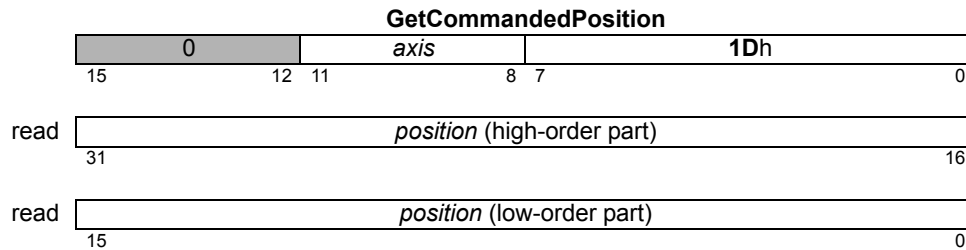
## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

## Returned data

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

## Packet Structure



## 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](#))

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

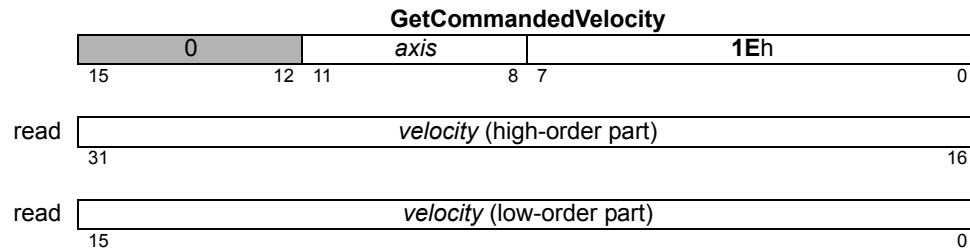
## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

## Returned data

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

## Packet Structure



## 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. 113)



**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. 141), **SetEventAction** (p. 124),  
**Set/GetDriveFaultParameter** (p. 115), **GetSPIMode** (p. 64)

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

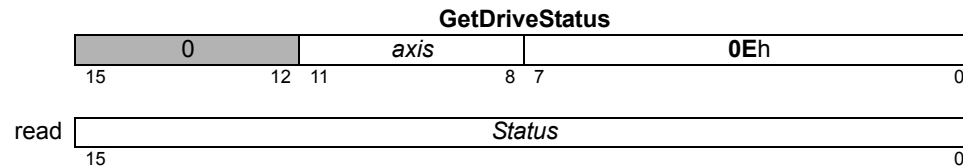
Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

## Returned data

Type
<i>status</i> unsigned 16 bits see below

## Packet

## Structure



## 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



<b>C-Motion API</b>	<pre>PMDresult <b>PMDGetDriveStatus</b>(PMDAxisInterface axis_intf,                               PMDuint16* status);</pre>
<b>Script API</b>	<b>GetDriveStatus</b>
<b>C# API</b>	<pre>UInt16 status = <b>PMDAxis.DriveStatus</b>;</pre>
<b>Visual Basic API</b>	<pre>UInt16 status = <b>PMDAxis.DriveStatus</b></pre>
<b>see</b>	<b>ExecutionControl</b> ( <a href="#">p. 35</a> ), <b>CalibrateAnalog</b> ( <a href="#">p. 31</a> ), <b>GetBufferReadIndex</b> ( <a href="#">p. 93</a> )

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

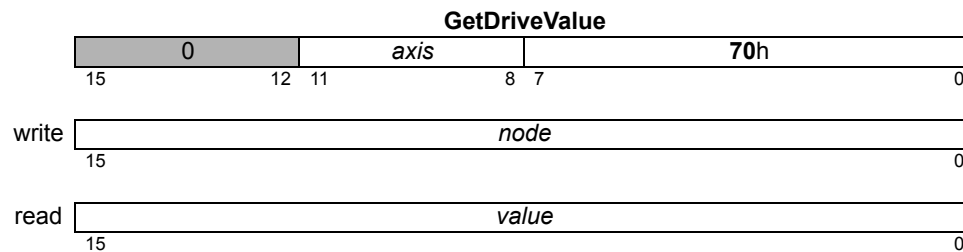
Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0
<i>node</i>	<i>Bus Voltage</i>	0
	<i>Temperature</i>	1
	<i>Bus Current Supply</i>	2
	<i>Bus Current Return</i>	3

## Returned data

	Type	Range/Scaling
<i>value</i>	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. 87](#)), **CalibrateAnalog** ([p. 31](#)), **SetDriveFaultParameter** ([p. 115](#))

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

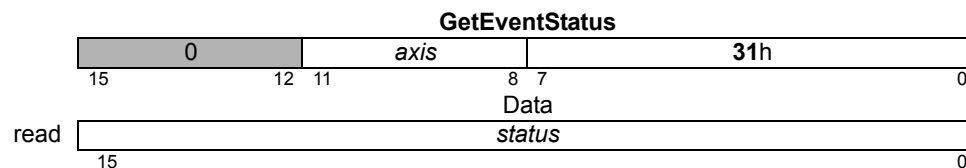
## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

## Returned data

	Type	
<i>status</i>	unsigned 16 bits	see below

## Packet Structure



## 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. 62](#)), **GetSignalStatus** ([p. 63](#)), **GetDriveStatus** ([p. 48](#)), **GetDriveFaultStatus** ([p. 46](#))

## Motor Types

	Brushless DC	Microstepping	
--	--------------	---------------	--

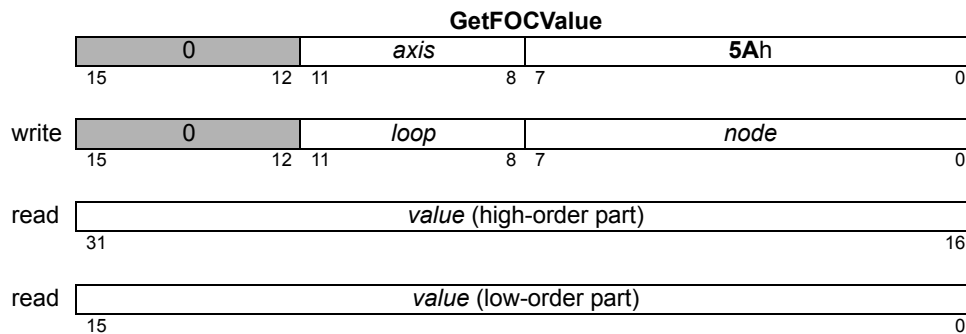
## Arguments

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

## Returned data

	Type	Range/Scaling
<i>value</i>	signed 32 bits	see below

## Packet Structure



## 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
<i>Reference (D,Q)</i>	$-2^{15}$ to $2^{15}-1$	$100/2^{14}$	% max current
<i>Feedback (D,Q)</i>	$-2^{15}$ to $2^{15}-1$	$100/2^{14}$	% max current
<i>Error (D,Q)</i>	$-2^{15}$ to $2^{15}-1$	$100/2^{14}$	% max current
<i>Integrator Contribution (D,Q)</i>	$-2^{31}$ to $2^{31}-1$	$100/2^{14}$	% PWM
<i>Output (D,Q)</i>	$-2^{15}$ to $2^{15}-1$	$100/2^{14}$	% PWM
<i>FOC Output (Alpha,Beta)</i>	$-2^{15}$ to $2^{15}-1$	$100/2^{14}$	% PWM
<i>Actual Current (A,B)</i>	$-2^{15}$ to $2^{15}-1$	$100/2^{14}$	% max current
<i>I<sup>2</sup>t Energy</i>	$-2^{31}$ 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  $\text{option} = 1 \times 256 + 6 = 262$ .

**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  $\text{option} = \text{loop} \times 256 + \text{node}$

**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. 105](#)), **Set/GetCurrentControlMode** ([p. 107](#)), **Set/GetFOC** ([p. 129](#))

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

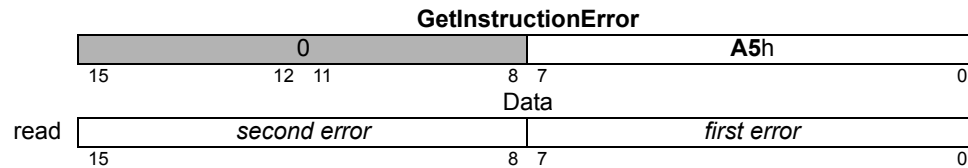
## Arguments

None

## Returned data

<i>error</i>	<b>Type</b> unsigned 16 bits	<b>Range</b> 0 to 35
--------------	---------------------------------	-------------------------

## Packet Structure



## 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	<code>PMDresult <b>PMDGetInstructionError</b> (PMDAxisInterface axis_intf, PMDuint16* error);</code>
Script API	<code>GetInstructionError</code>
C# API	<code>UInt16 error = <b>PMDAxis.InstructionError</b>;</code>
Visual Basic API	<code>UInt16 error = <b>PMDAxis.InstructionError</b></code>
see	<code>GetEventStatus</code> ( <a href="#">p. 52</a> ), <code>ResetEventStatus</code> ( <a href="#">p. 81</a> )

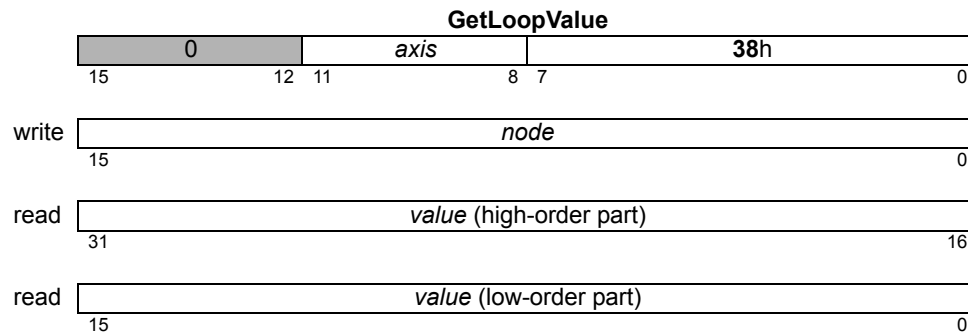
DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

Argument	Name Node	Instance	Encoding
		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
<i>value</i>	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

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

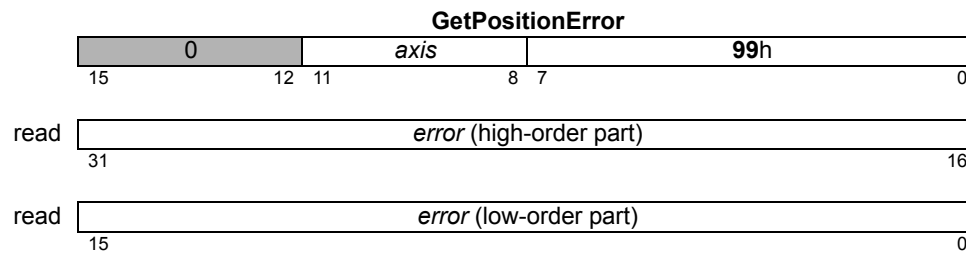
## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

## Returned data

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

## Packet Structure



## 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. 133](#))

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

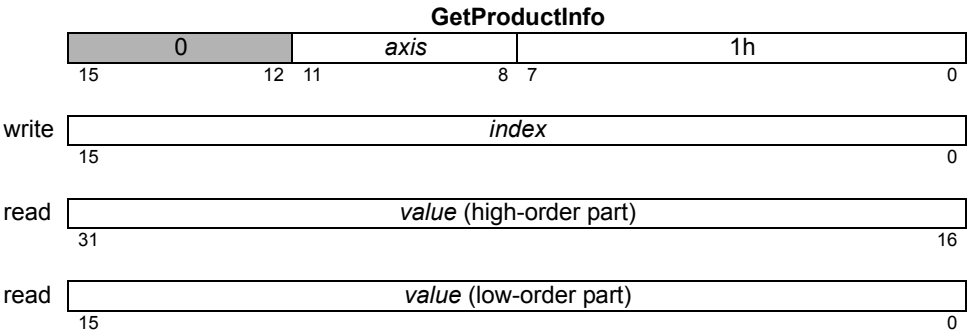
## Arguments

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

## Returned Data

value	Type
	unsigned 32 bits

## Packet Structure



## 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 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. 71](#)), SetBufferStart ([p. 95](#)), SetBufferLength ([p. 91](#)), ReadBuffer ([p. 75](#)), ReadBuffer16 ([p. 76](#)), GetVersion ([p. 69](#))

Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

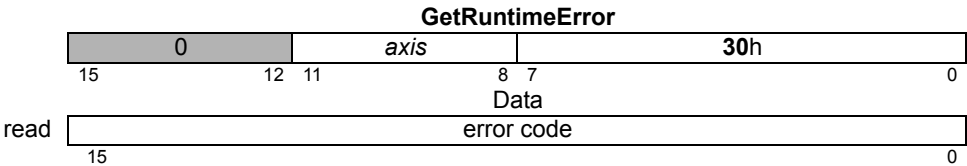
Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

Returned Data

Type	Range/scaling
unsigned 16 bits	see below

Packet Structure



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 occurs 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 occurred 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. 81)

**Motor Types**

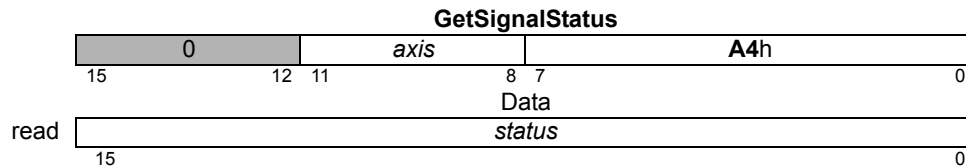
DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

**Arguments**

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

**Returned data**

Type
status
unsigned 16 bits

**Packet Structure****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. 154)) 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. 127) for more information. The bit definitions are as follows:

Description	Bit Number	Description	Bit Number
Encoder A	0	— (Reserved)	10
Encoder B	1	Positive Input	11
Encoder Index	2	— (Reserved)	12
— (Reserved)	3-6	/Enable	13
Hall A	7	FaultOut	14
Hall B	8	Direction Input	15
Hall C	9		

**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. 154)



## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

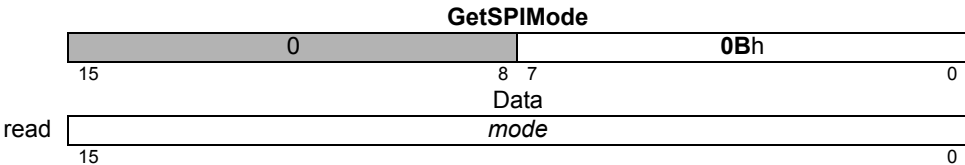
## Argument

None

## Returned Data

Name	Instance	Encoding
<i>mode</i>	<i>Host Command</i>	0
	<i>Direct</i>	8000h

## Packet Structure



## 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 \(p. 145\)](#), [SetDriveCommandMode \(p. 113\)](#), [SetLoop \(p. 133\)](#), [GetEventStatus \(p. 52\)](#), [SetEventAction \(p. 124\)](#), [GetDriveFaultStatus \(p. 46\)](#)

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

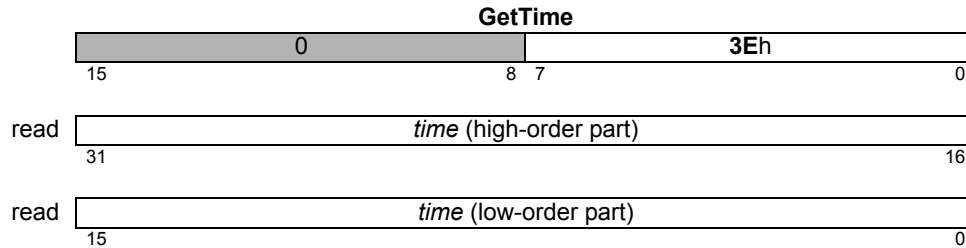
## Arguments

None

## Returned data

Name	Type	Range	Scaling	Units
<i>time</i>	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. 150](#))

Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

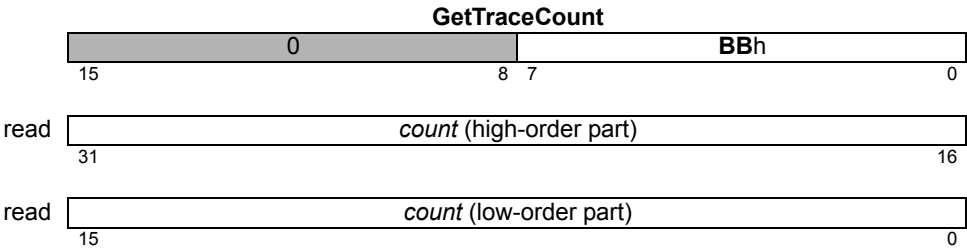
Arguments

None

Returned data

Name	Type	Range	Scaling	Units
<i>count</i>	unsigned 32 bits	0 to 2 <sup>32</sup> –1	unity	samples

Packet Structure



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. 67), **ReadBuffer** (p. 75), **Set/GetBufferLength** (p. 91), **Set/GetTraceMode** (p. 156), **Set/GetTraceStart** (p. 158), **Set/GetTraceStop** (p. 161),

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

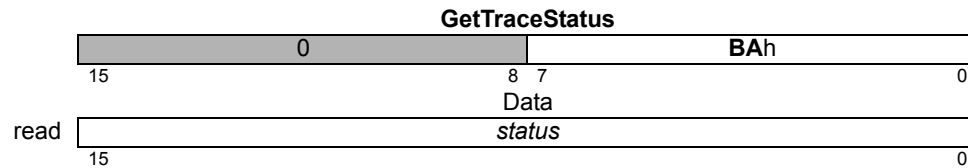
## Arguments

None

## Returned data

Name	Type
status	unsigned 16 bits

## Packet Structure



## 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 <b>ReadBuffer</b> 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. 158), **Set/GetTraceMode** (p. 156)

Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

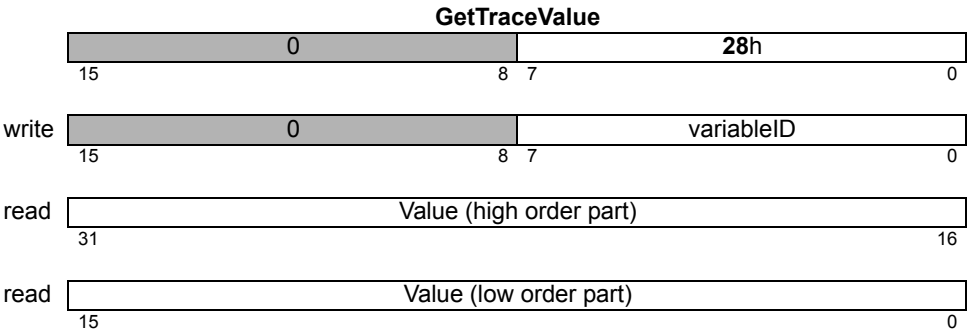
Arguments

Name	Type	Encoding
variableID	unsigned 8 bit	see below

Returned data

Value	Type	Range/Scaling
	32 bit	see below

Packet Structure



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. 163)

**Motor Types**

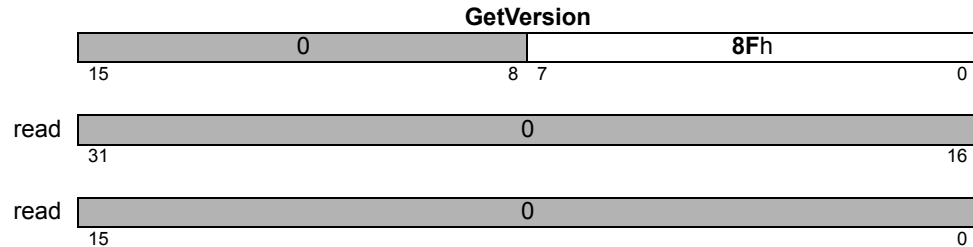
DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

**Arguments**

None

**Returned data**

Name	Type
<i>version</i>	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. 60)

## Motor Types

	Brushless DC		
--	--------------	--	--

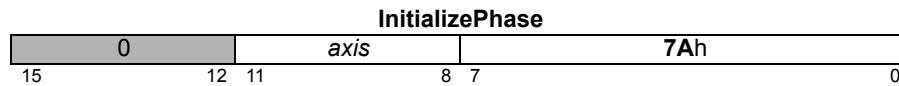
## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

## Returned data

None

## Packet Structure



## 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

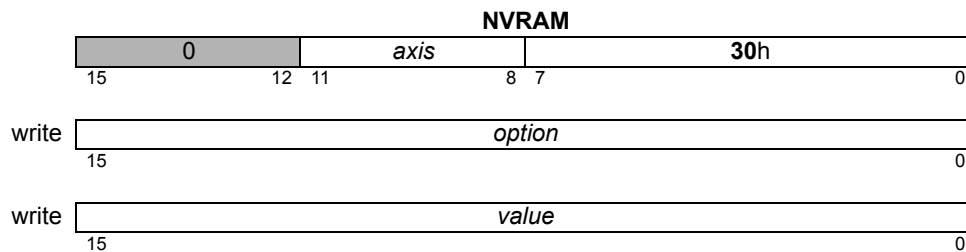
```
PMDAxis.InitializePhase()
```

## see

[GetActivityStatus](#) (p. 40), [GetEventStatus](#) (p. 52), [Set/GetCommutationMode](#) (p. 101)

Arguments	Name	Instance	Encoding
	<i>axis</i>	<i>Axis1</i>	0
	<i>option</i>	<i>NVRAM mode</i>	256
		<i>Erase NVRAM</i>	1
		<i>Write</i>	2
		<i>Block Write Begin</i>	3
		<i>Block Write End</i>	4
		<i>Skip</i>	8
	<i>value</i>	Type unsigned 16 bit	Range see below

### Packet Structure



### 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, meaning 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  $\sim$ HostSPIEnable signal.

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

The block write operation is concluded by sending a **BlockWriteEnd** command. 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. 77)

**Motor Types**

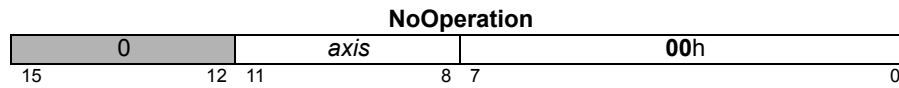
DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

**Arguments**

Name	Instance	Encoding
<i>axis</i>	Axis1	0

**Returned data**

None

**Packet Structure****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

Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

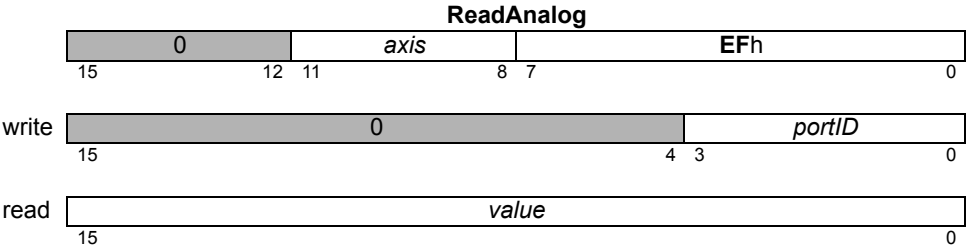
Arguments

Name	Instance	Encoding		
<i>axis</i>	Axis1	0		
Name	Type	Range	Scaling	Units
<i>portID</i>	unsigned 16 bits	0 to 10	unity	-

Returned data

	Type	Range	Scaling	Units
<i>value</i>	unsigned 16 bits	0 to 2 <sup>16</sup> -1	100/2 <sup>16</sup>	% input

Packet Structure



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

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

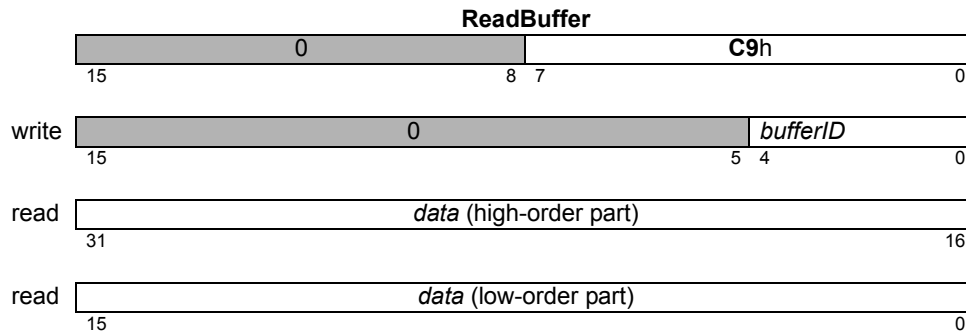
## Arguments

Name	Type	Range
<i>bufferID</i>	unsigned 16 bits	0 to 7

## Returned data

	Type	Range
<i>data</i>	signed 32 bits	$-2^{31}$ to $2^{31}-1$

## Packet Structure



## 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. 93), **Set/GetBufferStart** (p. 95), **Set/GetBufferLength** (p. 91)

Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

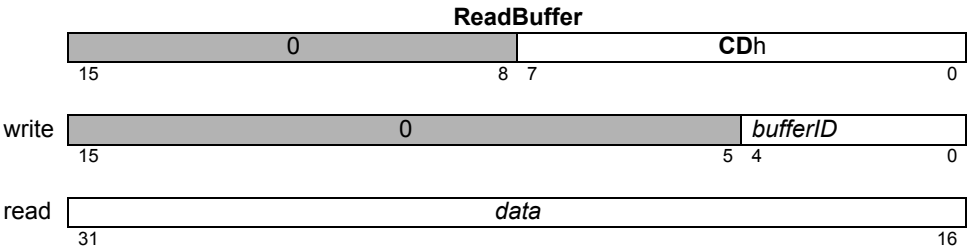
Arguments

Name	Type	Range
<i>bufferID</i>	unsigned 16 bits	0 to 7

Returned data

Type	Range
<i>data</i>	$-2^{15}$ 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

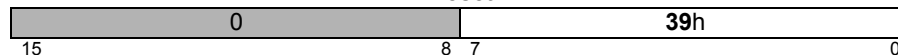
**Set/GetBufferReadIndex** (p. 93), **WriteBuffer** (p. 175), **Set/GetBufferStart** (p. 95), **Set/GetBufferLength** (p. 91)

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

None

None

## Reset



**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
<b>Interrupts</b>	
Interrupt Mask	0
<b>Commutation</b>	
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
<b>Current Control</b>	
Current 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
<b>Position/Outer Loop</b>	
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	7FFFFFFh
Outer Loop Output Lower Limit	-80000000h
<b>Encoder</b>	
Actual Position	0
Actual Position Units	motor dependent
Encoder Source	motor dependent
Encoder To Step Ratio	04000400h
<b>Motor Output</b>	
Operating Mode	0001h
Active Operating Mode	0001h
Output Mode	10
Motor Type	0
PWM Frequency	5000
PWM Limit	16384
PWM Dead Time	16879 <i>must be changed</i>
PWM Signal Sense	80FFh
PWM Refresh Period	1
PWM Refresh Time	32767 <i>must be changed</i>
PWM Current Sense Time	32767 <i>must be changed</i>
<b>Position Servo Loop Control</b>	
Sample Time	102
<b>Profile Generation</b>	
Acceleration	0
Deceleration	0
Profile Mode	1
Start Velocity	0
<b>Velocity Loop</b>	
Velocity Loop Kp	0
Velocity Loop Ki	0
Velocity Loop Integrator Sum Limit	1
Velocity Scalar	0
Velocity Error Limit	7FFFFFFh
Velocity Feedback Source	0
Deadband Upper Limit	0
Deadband Lower Limit	0
<b>RAM Buffer</b>	
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
<b>Safety</b>	
<i>Motion Error Event Action</i>	4
<i>Current Foldback Event Action</i>	7
<i>OvervoltageThreshold</i>	65535
<i>Undervoltage Threshold</i>	0
<i>OvertemperatureThreshold</i>	32767
<i>FaultOut Mask</i>	0600h
<i>Continuous Current Limit</i>	32768
<i>Energy Limit</i>	32768
<b>Status Registers and AxisOut Indicator</b>	
<i>Signal Sense</i>	0
<b>Traces</b>	
<i>Trace Mode</i>	0
<i>Trace Period</i>	1
<i>Trace Start</i>	0
<i>Trace Stop</i>	0
<i>Trace Variables</i>	all are 0
<i>Trace Trigger Values</i>	all are 0
<b>Miscellaneous</b>	
<i>CAN Mode</i>	C000h (see Notes)
<i>Serial Port Mode</i>	0004h (see Notes)

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

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

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

## Notes

See **Set/GetSampleTime** (p. 150) 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**

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

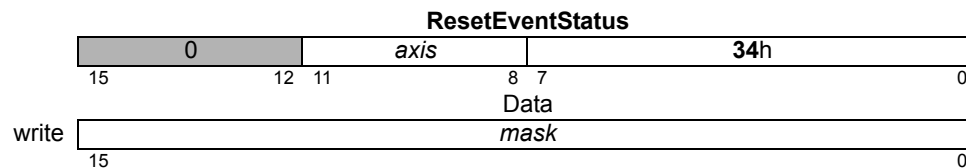
## Arguments

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

## Returned data

None

## Packet Structure



## 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);
```

## Visual Basic API

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

## see

**GetEventStatus** (p. 52)

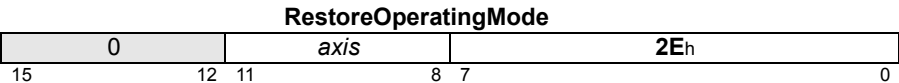
Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0

Packet Structure



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. 143), **Set/GetEventAction** (p. 124)

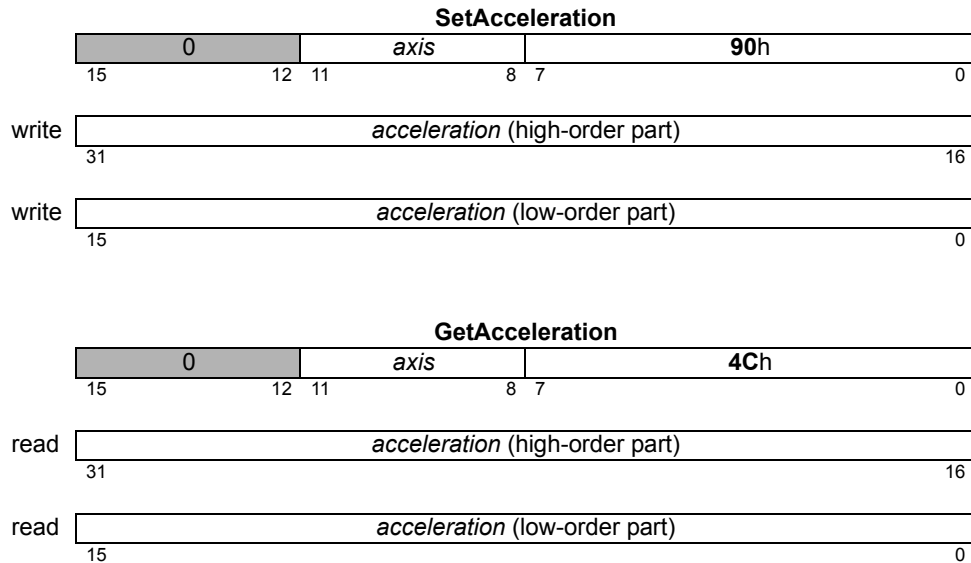
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

Name	Instance	Encoding			
<i>axis</i>	<i>Axis1</i>	0			
<i>acceleration</i>	<b>Type</b> unsigned 32 bits	<b>Range</b> 0 to $2^{31}-1$	<b>Scaling</b> $1/2^{16}$	<b>Units</b> counts/cycle <sup>2</sup> microsteps/cycle <sup>2</sup>	

## Packet Structure



## 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<sup>2</sup>, 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<sup>2</sup> or steps/cycle<sup>2</sup>.

## 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);
```

Script API	<code>GetAcceleration</code> <code>SetAcceleration</code> <i>acceleration</i>
C# API	<code>UInt32 acceleration = PMDAxis.Acceleration;</code> <code>PMDAxis.Acceleration = acceleration;</code>
Visual Basic API	<code>UInt32 acceleration = PMDAxis.Acceleration</code> <code>PMDAxis.Acceleration = acceleration</code>
see	<code>Set/GetDeceleration</code> ( <a href="#">p. 112</a> ), <code>Set/GetVelocity</code> ( <a href="#">p. 173</a> )

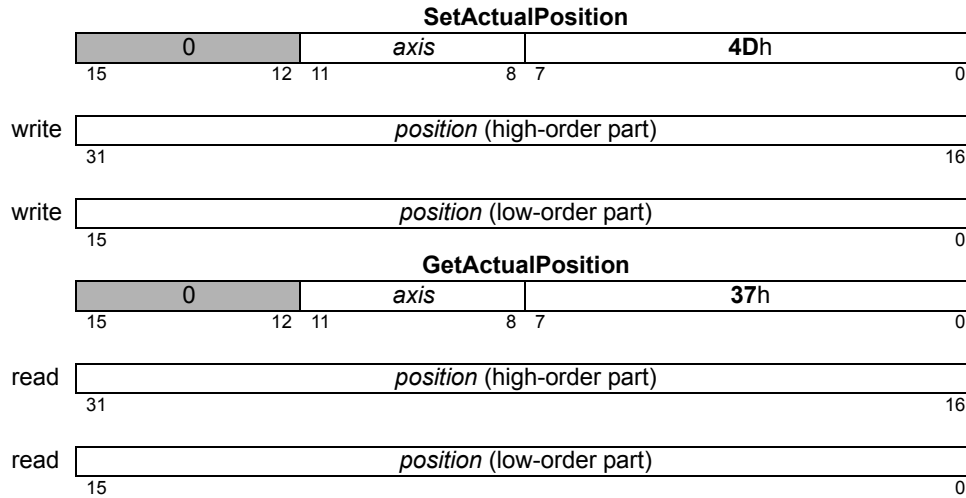
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

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

## Packet Structure



## 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. 59), **Set/GetActualPositionUnits** (p. 86), **AdjustActualPosition** (p. 30)

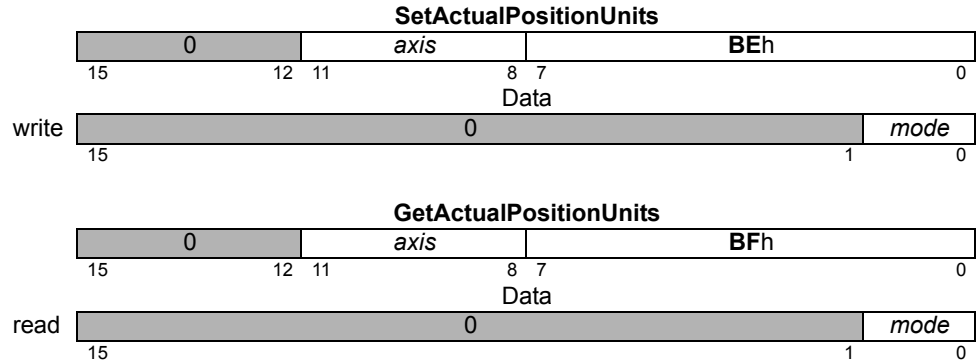
## Motor Types

		Microstepping
--	--	---------------

## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0
<i>mode</i>	<i>Counts</i>	0
	<i>Steps</i>	1

## Packet Structure



## 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. 85), **Set/GetEncoderToStepRatio** (p. 122), **AdjustActualPosition** (p. 30), **GetCaptureValue** (p. 42), **Set/GetTraceVariable** (p. 163)

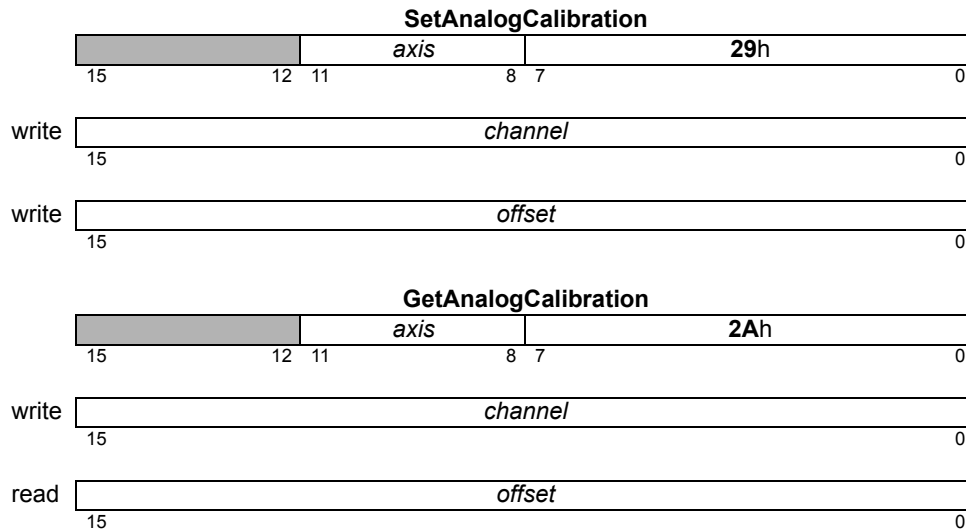
### Motor Types

DC Brush	Brushless DC	Microstepping	
----------	--------------	---------------	--

### Arguments

Name	Instance	Encoding			
<i>axis</i>	<i>Axis1</i>	0			
<i>channel</i>	<i>current leg A</i>	0			
	<i>current leg B</i>	1			
	<i>current leg C</i>	2			
	<i>current leg D</i>	3			
	<i>Analog command</i>	7			
	<i>Tachometer</i>	8			
<i>offset</i>	<b>Type</b> signed 16 bits	<b>Range</b> $-2^{15}$ to $2^{15}-1$	<b>Scaling</b> $100/2^{16}$	<b>Units</b> % input	

### Packet Structure



### 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 appropriate offsets.

**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 \(p. 74\)](#), [CalibrateAnalog \(p. 31\)](#)

### Motor Types

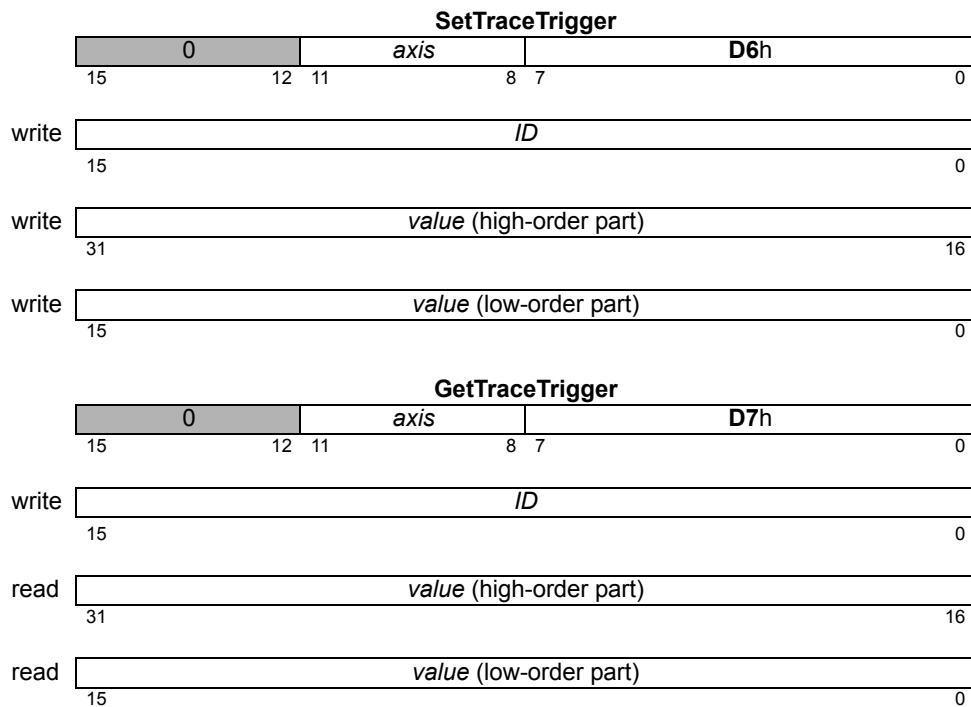
DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

### Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0
<i>ID</i>	<i>start</i>	256
	<i>stop</i>	257
	<i>stop delay</i>	258

*value* (see below)

### Packet Structure



### 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. 158). 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

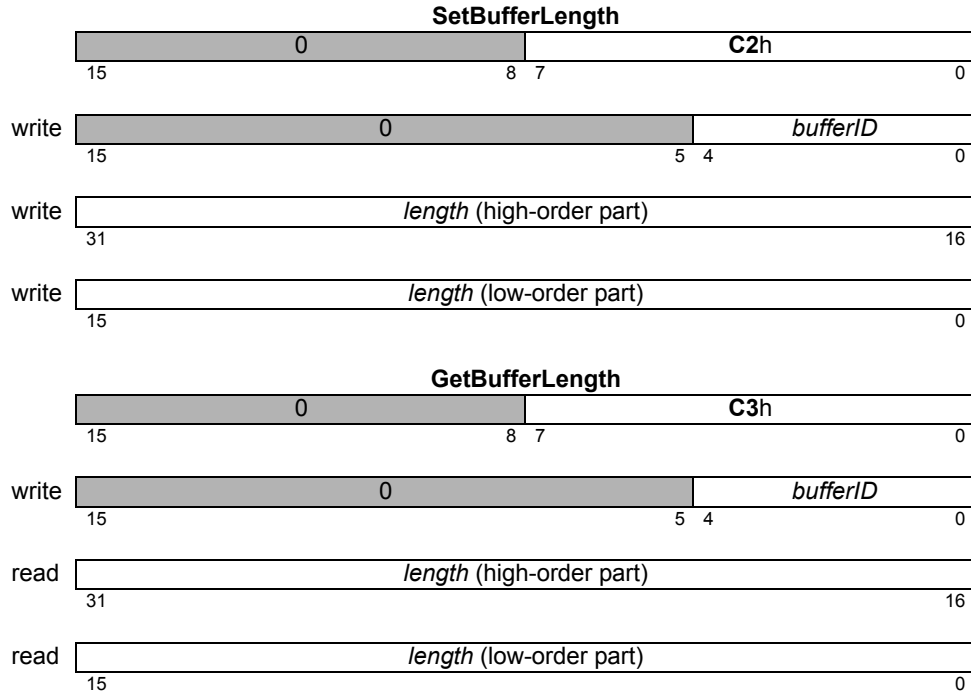
### Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

### Arguments

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

### Packet Structure



### 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);
```

### 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. 93](#)), **Set/GetBufferStart** ([p. 95](#)), **Set/GetBufferWriteIndex** ([p. 97](#))

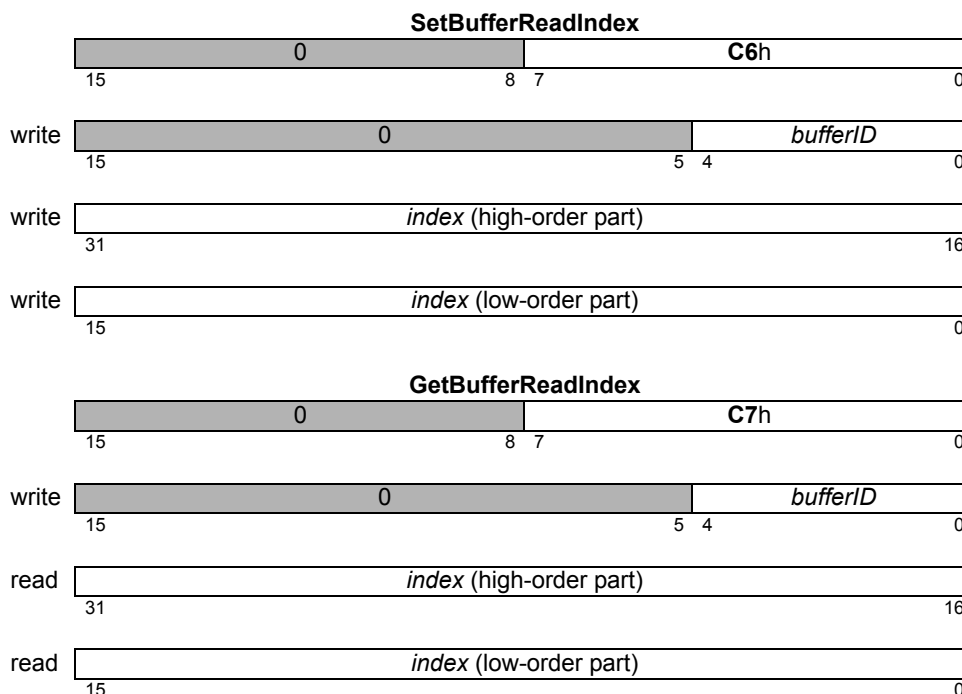
### Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

### Arguments

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

### Packet Structure



### 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. 91](#)), **Set/GetBufferStart** ([p. 95](#)), **Set/GetBufferWriteIndex** ([p. 97](#))

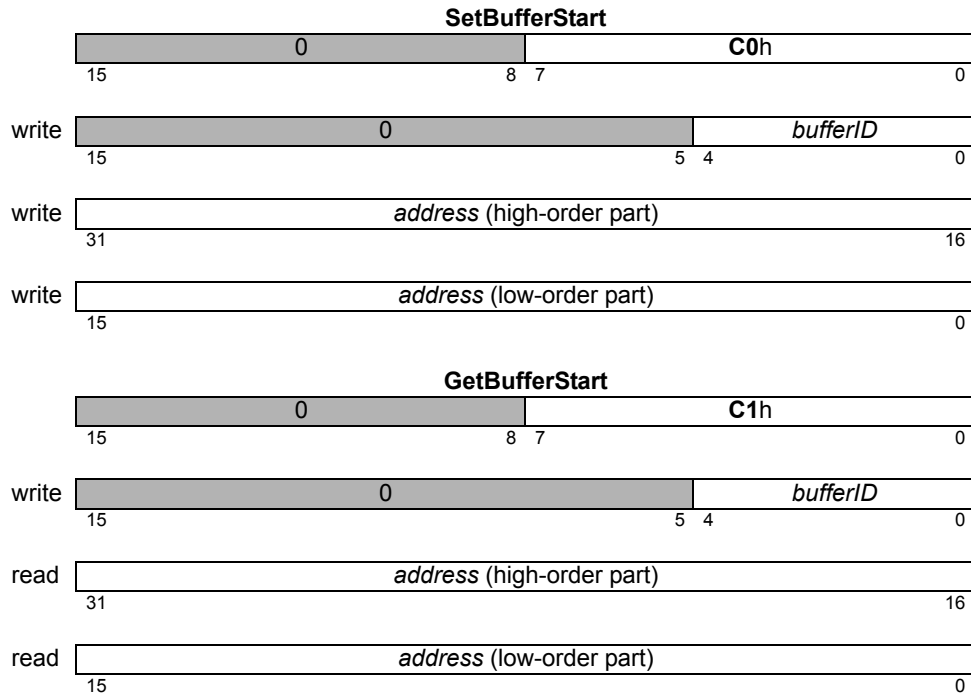
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

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

## Packet Structure



## 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 plus 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.



### 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. 91), **Set/GetBufferReadIndex** (p. 93), **Set/GetBufferWriteIndex** (p. 97)

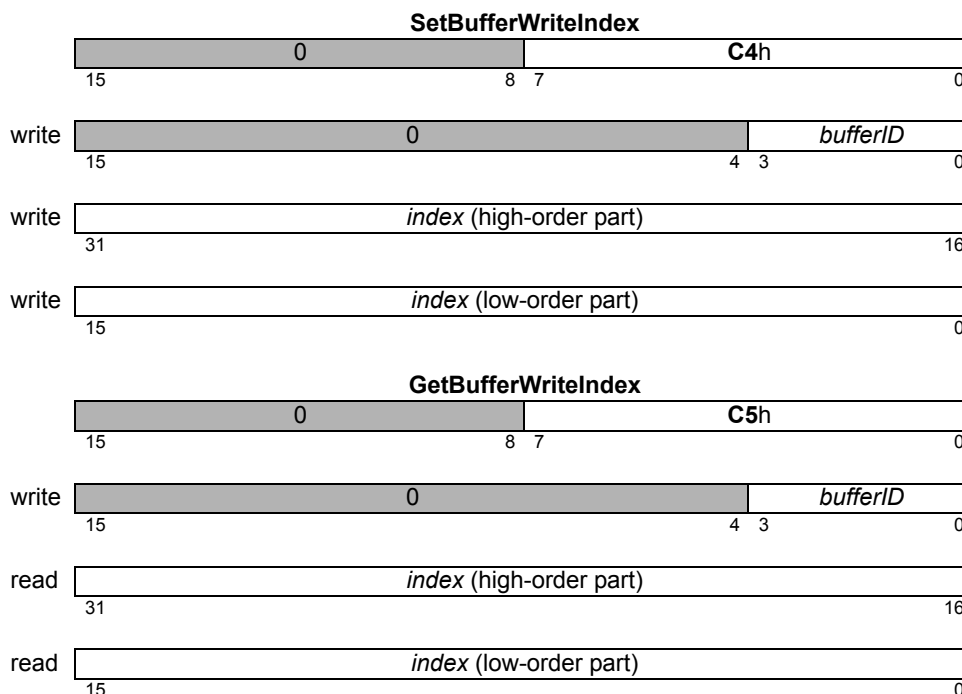
### Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

### Arguments

Name	Type	Range	Scaling	Units
<i>bufferID</i>	unsigned 16 bits	0 to 7	unity	-
<i>index</i>	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
```

### 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. 91](#)), **Set/GetBufferReadIndex** ([p. 93](#)), **Set/GetBufferStart** ([p. 95](#))

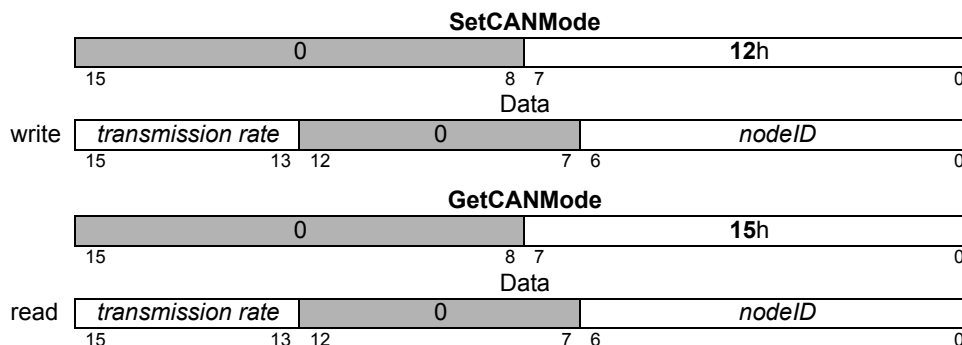
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

Name	Type	Encoding
<i>mode</i>	unsigned 16 bits	see below

## Packet Structure



## 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  $option = 2 \times 8192 + 3 = 16387$ .

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

Motor Types	<div> <div></div> <div>Brushless DC</div> <div></div> <div></div> </div>		
Arguments	Name	Instance	Encoding
	<i>axis</i>	<i>Axis1</i>	0
	<i>mode</i>	<i>Sinusoidal</i>	0
		<i>Hall-based</i>	1
Packet Structure	SetCommutationMode		
	0	axis	E2h
	151211870		
	Data		
write	0	mode	
	3110		
	GetCommutationMode		
	0	axis	E3h
	151211870		
	Data		
read	0	mode	
	3110		

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

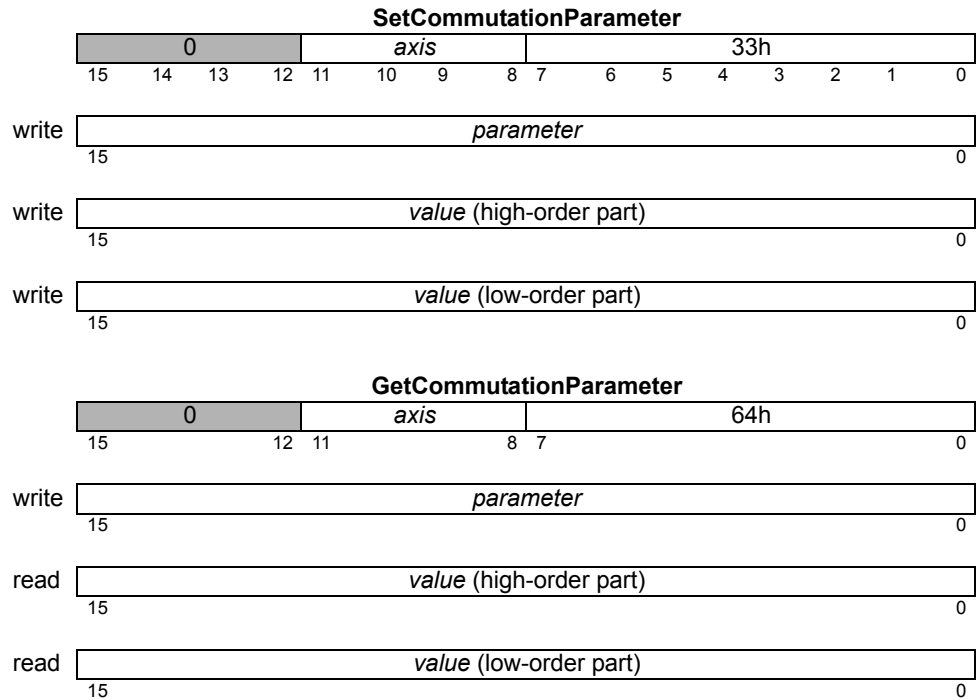
Motor Types

	Brushless DC	Microstepping
--	--------------	---------------

Arguments

Name	Instance	Encoding		
axis	Axis1	0		
parameter	phase counts	0		
	phase angle	1		
	phase offset	2		
	phase denominator	3		
value	Type	Range	Scaling/Units	
	unsigned 32-bits	0 to 2 <sup>31</sup> -1	counts	

Packet  
Structure



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**

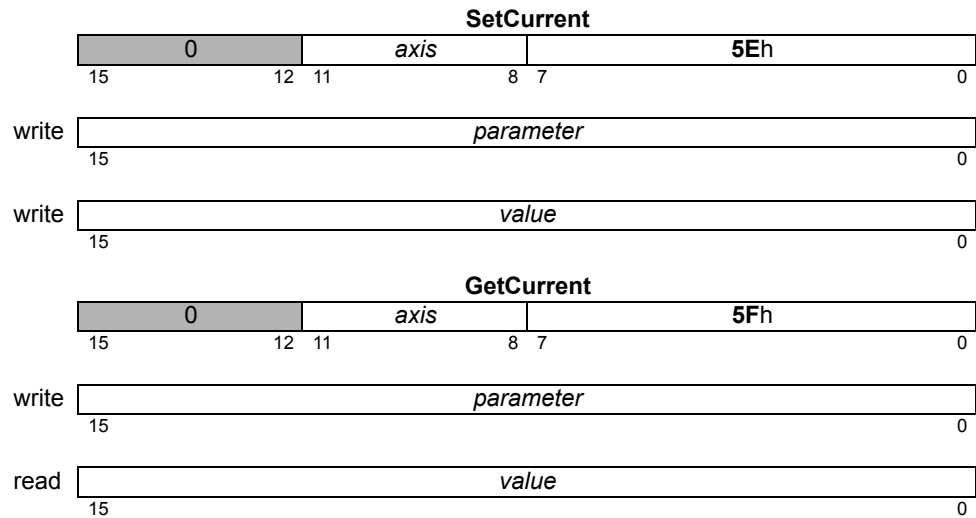
Motor Types

		Microstepping
--	--	---------------

Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0
<i>parameter</i>	<i>Holding Motor Limit</i>	0
	— (Reserved)	1
	<i>Drive Current</i>	2
<i>value</i>	Type	Range/Scaling
	unsigned 16-bit	see below

Packet  
Structure



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);
```

### 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. 150](#)), **SetMotorCommand** ([p. 137](#))

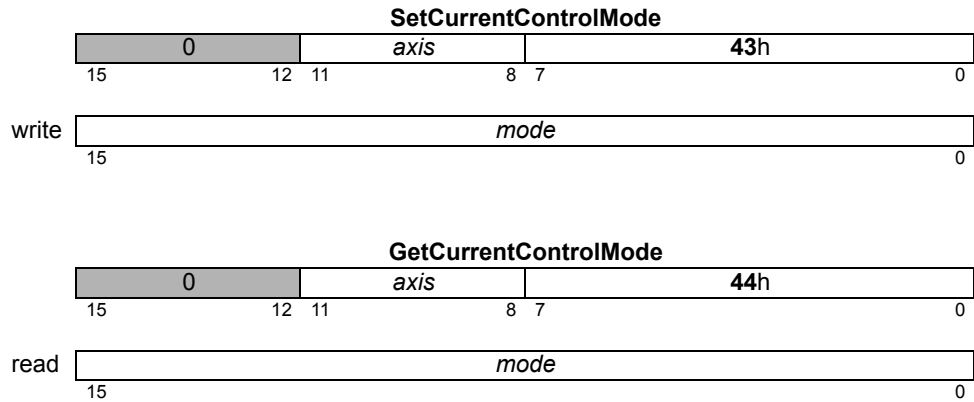
Motor Types

	Brushless DC	Microstepping	
--	--------------	---------------	--

Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0
<i>mode</i>	<i>reserved</i>	0
	<i>FOC</i>	1
	<i>Third leg floating</i>	2

Packet Structure



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
```

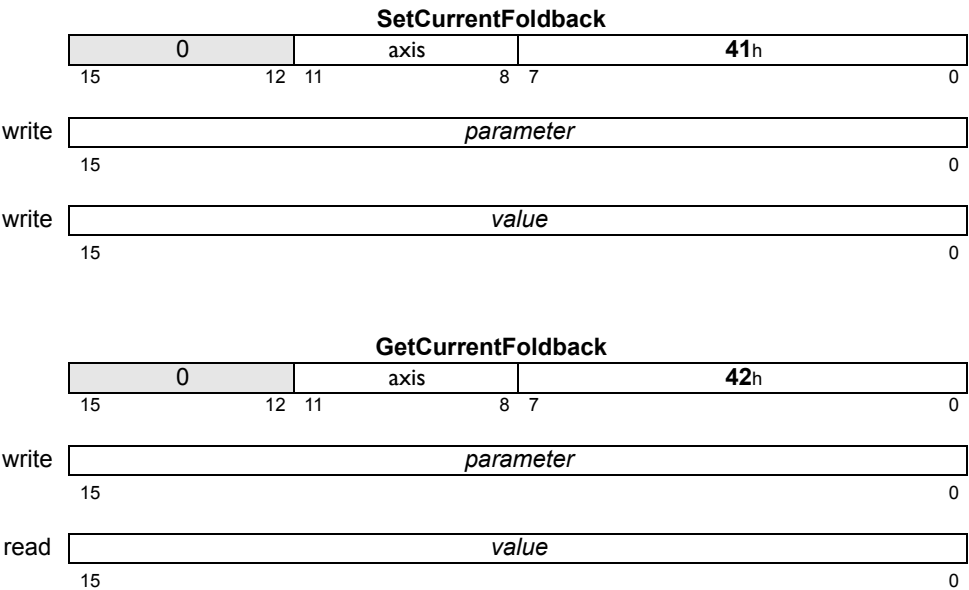
see

**GetFOCValue** (p. 54), **Get/SetFOC** (p. 129)

Motor Types	DC Brush	Brushless DC	Microstepping	
-------------	----------	--------------	---------------	--

Arguments	Name	Instance	Encoding
	<i>axis</i>	<i>Axis1</i>	0
	<i>parameter</i>	<i>Continuous Current Limit</i>	0
		<i>Energy Limit</i>	1
	<i>value</i>	Type unsigned 16-bit	Range/Scaling see below

Packet  
Structure



Description

**SetCurrentFoldback** is used to set various I<sup>2</sup>t 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<sup>2</sup>s. The scaling factor is  $2^{-31}/51.2\text{e-}6 \mu\text{s} / (\text{A/count})^2$ , where A/count is the current scaling factor and 51.2e-6  $\mu\text{s}$  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<sup>2</sup> energy above this setting begins. Once the accumulated excess I<sup>2</sup> 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<sup>2</sup> 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.

### 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. 81), **GetDriveStatus** (p. 48),  
**RestoreOperatingMode** (p. 82), **GetActiveOperatingMode** (p. 38)

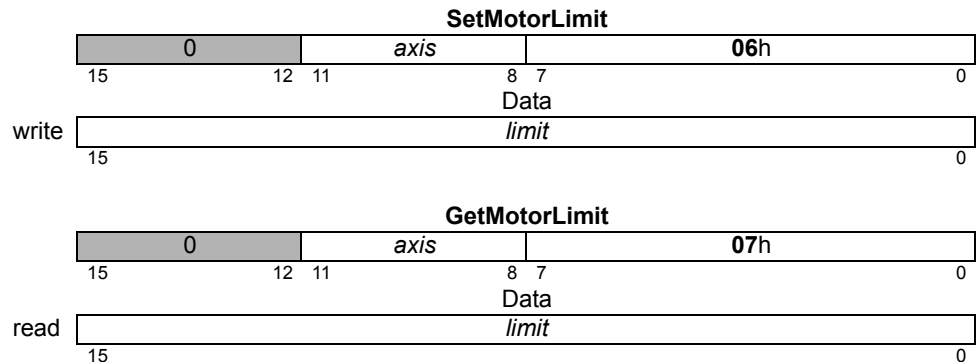
## Motor Types

DC Brush	Brushless DC		
----------	--------------	--	--

## Arguments

Name	Instance	Encoding			
<i>axis</i>	<i>Axis1</i>	0			
<i>limit</i>	Type unsigned 16 bits	Range 0 to 2 <sup>14</sup> −1	Scaling 100/2 <sup>15</sup>	Units % representable current	

## Packet Structure



## 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. 137](#)), **Set/GetOperatingMode** ([p. 143](#))



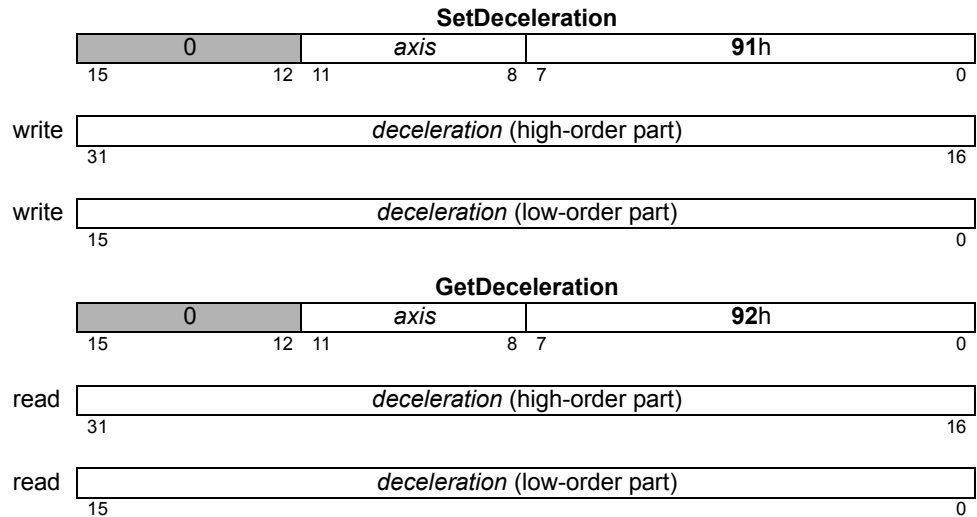
## Motor Types

### Arguments

	DC Brush	Brushless DC	Microstepping		
<b>Name</b>	<b>Instance</b>		<b>Encoding</b>		
<i>axis</i>	<i>Axis1</i>		0		
	<b>Type</b>		<b>Range</b>	<b>Scaling</b>	<b>Units</b>
<i>deceleration</i>	unsigned 32 bits		0 to $2^{31}-1$	$1/2^{16}$	counts/cycle <sup>2</sup> microsteps/cycle <sup>2</sup>

## Packet

### Structure



## 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<sup>2</sup> 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<sup>2</sup> or steps/cycle<sup>2</sup>

**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. 83), **Set/GetVelocity** (p. 173)

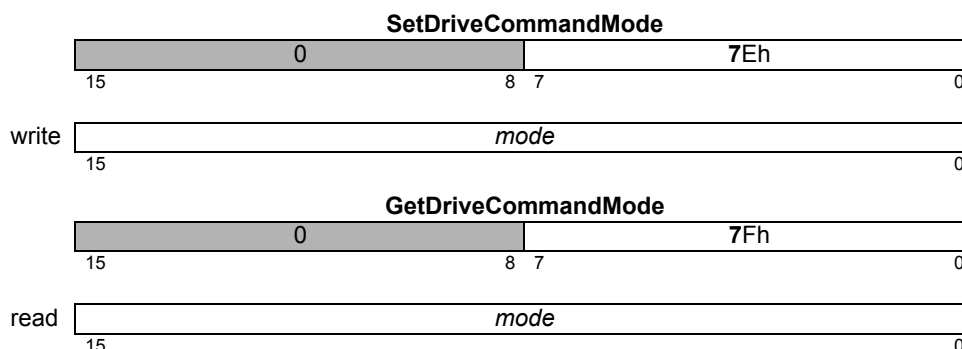
### Motor Type

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

### Arguments

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

### Packet Structure



### 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.

### 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. 83), **SetDeceleration** (p. 112), **SetLoop** (p. 133), **SetVelocity** (p. 173)

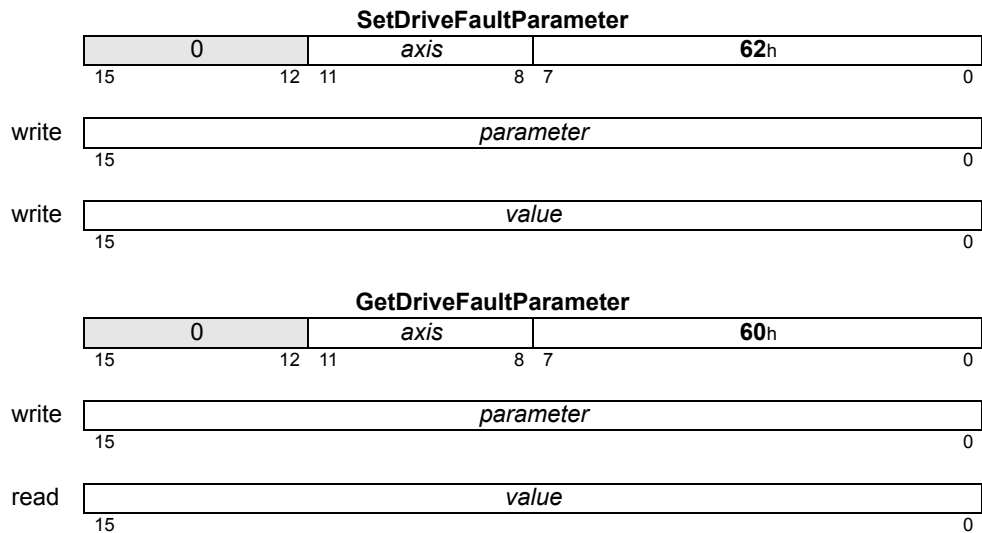
Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

Arguments

Name	Instance	Encoding	
<i>axis</i>	<i>Axis1</i>	0	
<i>parameter</i>	<i>Overvoltage Limit</i>	0	
	<i>Undervoltage Limit</i>	1	
	<i>Event Recovery Mode</i>	2	
	<i>Watchdog Limit</i>	3	
	<i>Temperature Limit</i>	4	
	<i>Temperature Hysteresis</i>	5	
	— (Reserved)	6,7	
	<i>Shunt voltage limit</i>	8	
	<i>Shunt duty</i>	9	
	<i>Bus current supply limit</i>	10	
<i>Bus current return limit</i>	11		
<i>value</i>	<b>Type</b>	<b>Range</b>	<b>Scaling</b>
	unsigned 16 bits	see below	see below

Packet  
Structure



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  $\sim$ 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  $\mu$ s, de-assert the  $\sim$ Enable signal, wait again for at least 150  $\mu$ s, and re-assert  $\sim$ Enable. After  $\sim$ 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  $\sim$ 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  $\mu$ s 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  $16 * 51.2 = 819 \mu$ s.

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.

### 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. 127), **GetDriveFaultStatus** (p. 46), **ClearDriveFaultStatus** (p. 32), **GetEventStatus** (p. 52), **ResetEventStatus** (p. 81), **SetEventAction** (p. 124)

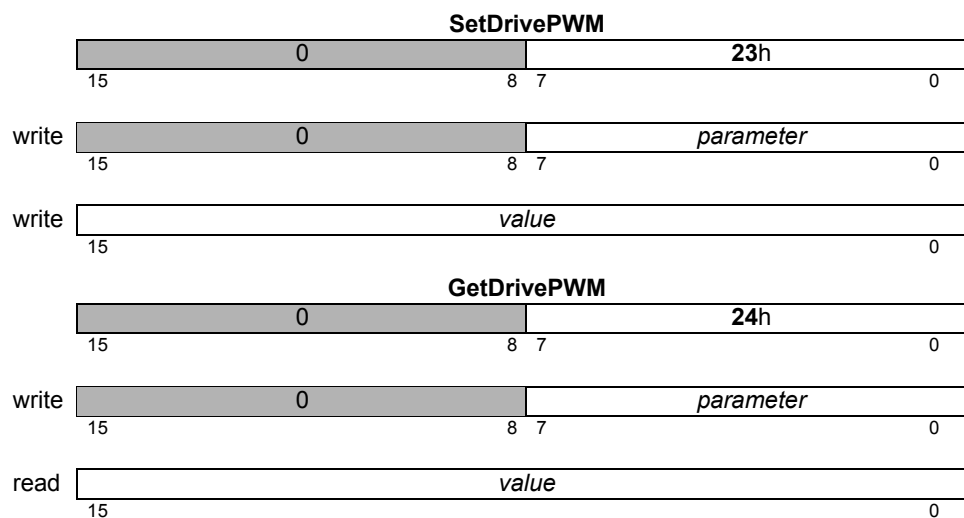
## Motor Type

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

Name	Instance	Encoding
<i>parameter</i>	Limit	0
	Dead Time	1
	Signal Sense	2
	Frequency	3
	Refresh Period	4
	Refresh Time	5
	Minimum Current Read Time	6
<i>value</i>	Type	Range/Scaling
	16-bit unsigned	see below

## Packet Structure



## 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

### 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)
```



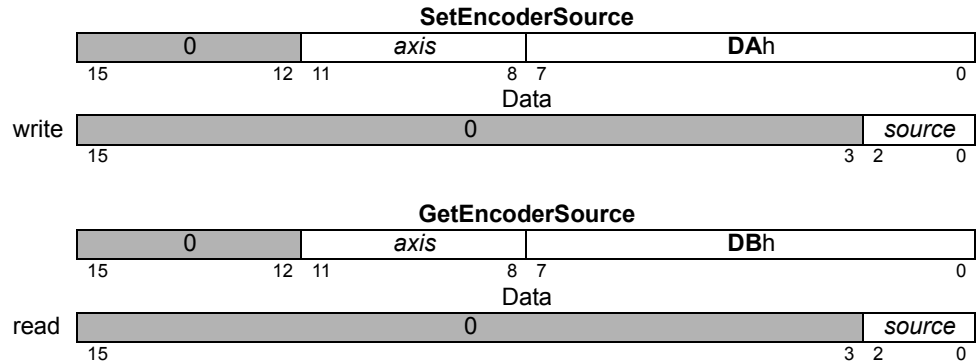
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

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

## Packet Structure



## 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
```

**C# API**

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

**Visual Basic  
API**

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

see

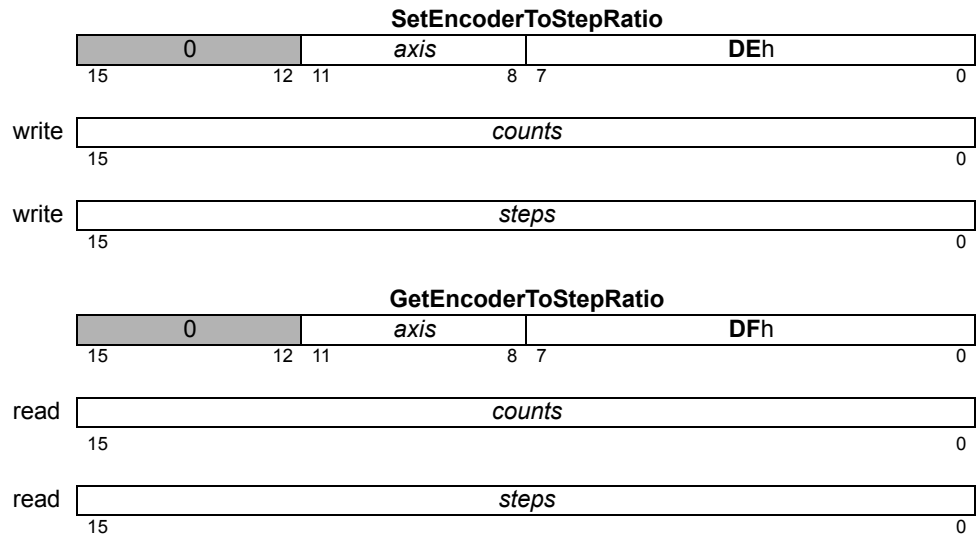
### Motor Types

		Microstepping
--	--	---------------

### Arguments

Name	Instance	Encoding			
<i>axis</i>	<i>Axis1</i>	0			
	<b>Type</b>	<b>Range</b>	<b>Scaling</b>	<b>Units</b>	
<i>counts</i>	unsigned 16 bits	1 to $2^{15}-1$	unity	counts	
<i>steps</i>	unsigned 16 bits	1 to $2^{15}-1$	unity	microsteps	

### Packet Structure



### 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);
```

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. 86)

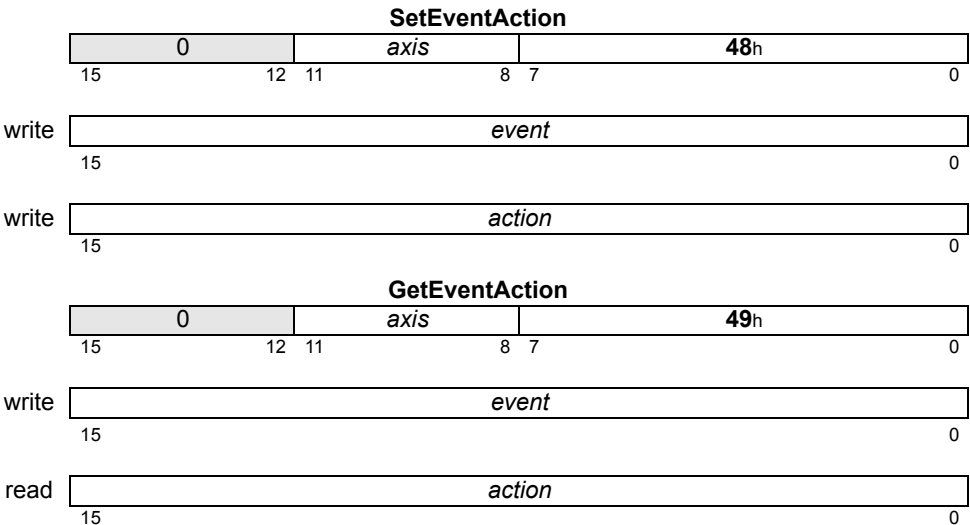
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

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

## Packet Structure



### 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 **Abrupt 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.

### 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. 82](#)), **Set/GetOperatingMode** ([p. 143](#))

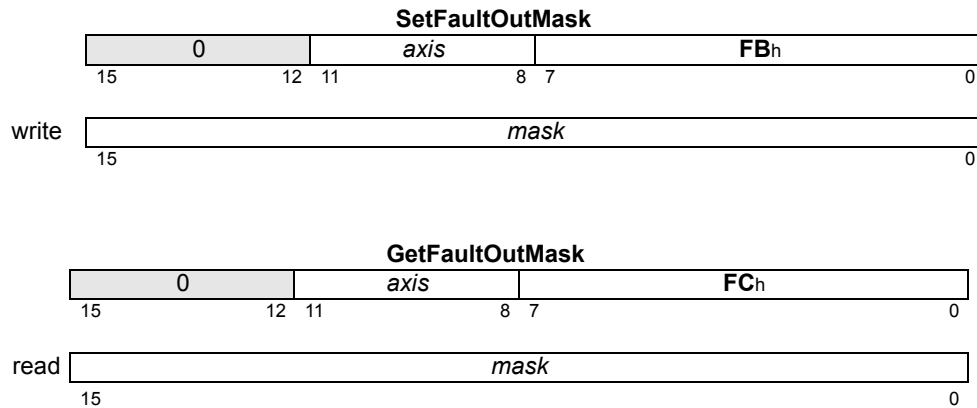
### Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

### Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0
<i>mask</i>	see below	bitmask

### Packet Structure



### 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*.



### 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

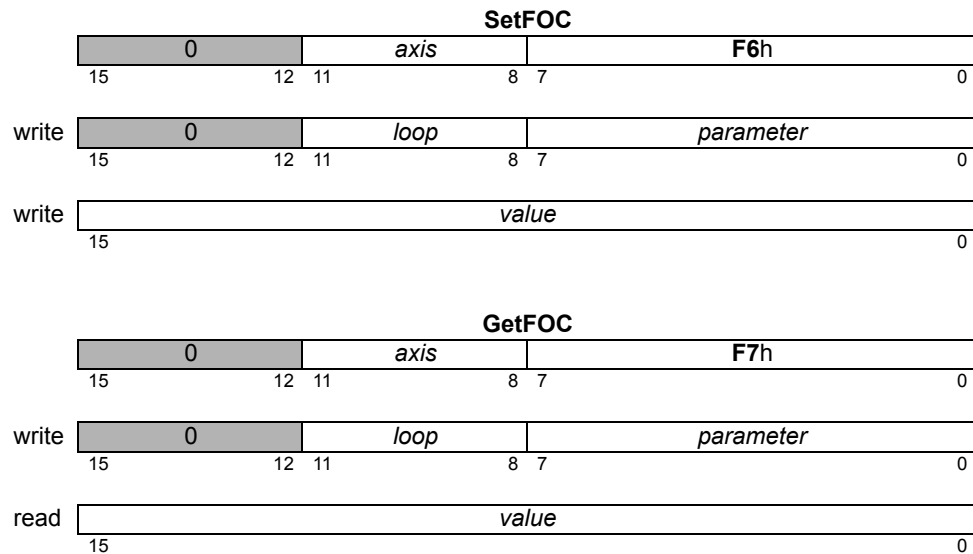
**Set/GetInterruptMask** ([p. 131](#))

## Motor Types

	Brushless DC	Microstepping
--	--------------	---------------

## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0
<i>loop</i>	<i>Direct(D)</i>	0
	<i>Quadrature(Q)</i>	1
	<i>Both(D and Q)</i>	2
<i>parameter</i>	<i>Proportional Gain (KpDQ)</i>	0
	<i>Integrator Gain (KiDQ)</i>	1
	<i>Integrator Sum Limit (ILimitDQ)</i>	2
<i>value</i>	<b>Type</b> unsigned 16 bits	<b>Range/Scaling</b> see below

Packet  
Structure

## 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
<i>Proportional Gain (KpDQ)</i>	0 to $2^{15}-1$	1/64	%output/%current
<i>Integrator Gain (KiDQ)</i>	0 to $2^{15}-1$	1/256	%output/%current/cycles
<i>Integrator Sum Limit (ILimitDQ)</i>	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  $2 \times 8192 = 16384 = 50\%$ .

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),  $\text{option} = 1 \times 256 + 1 = 257$ .

## 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. 107)

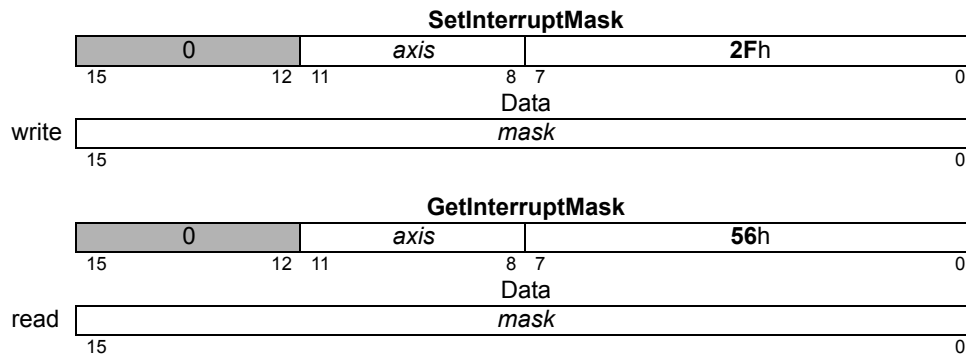
### Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

### Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0
<i>mask</i>	<i>Wrap-around</i>	0002h
	<i>Capture Received</i>	0008h
	<i>Motion Error</i>	0010h
	<i>Instruction Error</i>	0080h
	<i>Disabled</i>	0100h
	<i>Overtemperature Fault</i>	0200h
	<i>Drive Exception</i>	0400h
	<i>Commutation Error</i>	0800h
	<i>Current Foldback</i>	1000h
	<i>Runtime Error</i>	2000h

### Packet Structure



### 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. 127](#))

## Motor Types

DC Brush	Brushless DC	
----------	--------------	--

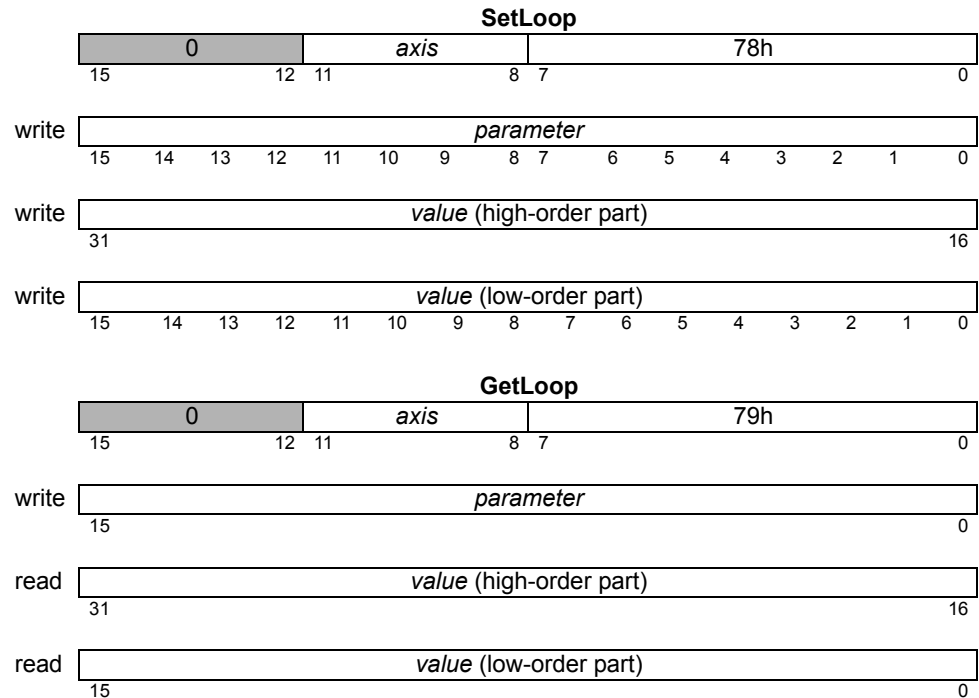
## Arguments

Name	Instance	Encoding
<i>axis</i>	Axis1	0
<i>parameter</i>		
	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

value	Type	Range/Scaling
	signed 32 bits	see below

## 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.

### 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. 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 `MotionEvent` 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 `MotionEvent` 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 its 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);
```

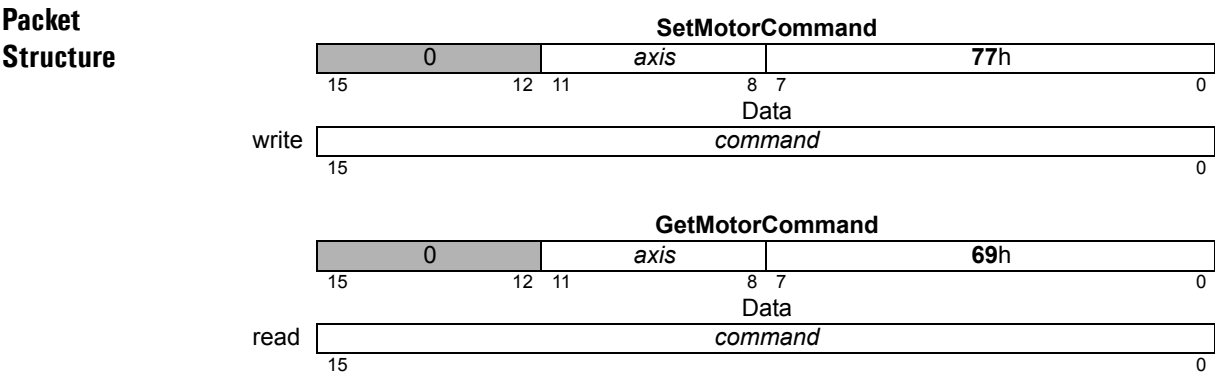
## Visual Basic API

```
Int32 value = PMDAxis.GetLoop(ByVal parameter As PMDLoop)
PMDAxis.SetLoop(ByVal parameter As PMDLoop, ByVal value As Int32)
```

## see

**SetEncoderSource** (p. 120), **SetDriveCommandMode** (p. 113), **SetSampleTime** (p. 150),  
**GetEventStatus** (p. 52), **SetEventAction** (p. 124)

Motor Types	<table><tr><td>DC Brush</td><td>Brushless DC</td><td>Microstepping</td><td colspan="2"></td></tr></table>					DC Brush	Brushless DC	Microstepping		
DC Brush	Brushless DC	Microstepping								
Arguments	Name	Instance	Encoding							
	<i>axis</i>	<i>Axis1</i>	0							
		Type	Range	Scaling	Units					
	<i>command</i>	signed 16 bits	$-2^{15}$ to $2^{15}-1$	$100/2^{15}$	% output					



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. 105](#)), `Set/GetCurrentLimit` ([p. 139](#)), `Set/GetOperatingMode` ([p. 143](#))

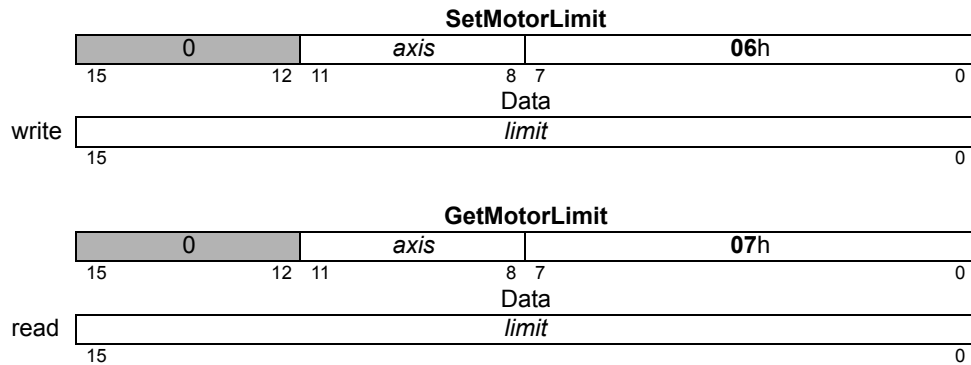
## Motor Types

DC Brush	Brushless DC		
----------	--------------	--	--

## Arguments

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

## Packet Structure



## 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. 137](#)), Set/GetOperatingMode ([p. 143](#))

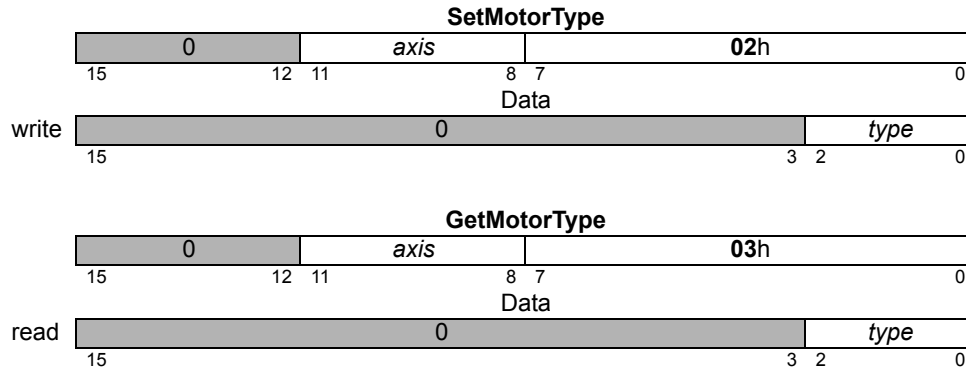
### Motor Types

#### Arguments

	DC Brush	Brushless DC	Microstepping
<b>Name</b>	<b>Instance</b>		
<i>axis</i>	<i>Axis1</i>		
<i>type</i>	<i>Brushless DC (3 phase)</i>		0
	— (Reserved)		1,2
	<i>Microstepping (2 phase)</i>		3
	— (Reserved)		4-6
	<i>DC Brush</i>		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
```

**C# API**                    `PMDMotorType type = PMDAxis.MotorType;`  
                             `PMDAxis.MotorType = type;`

**Visual Basic**            `PMDMotorType type = PMDAxis.MotorType`  
**API**                        `PMDAxis.MotorType = type`

**see**                        **Reset** ([p. 77](#))

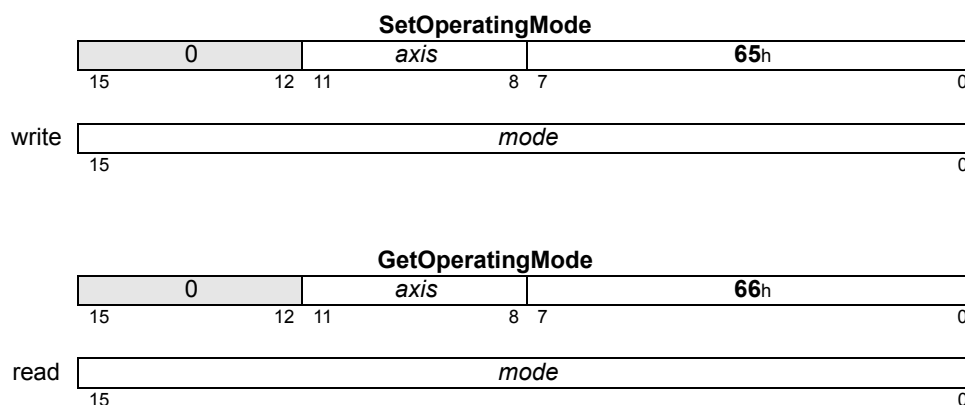
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0
	<b>Type</b>	<b>Range/Scaling</b>
<i>mode</i>	unsigned 16-bit	see below

## Packet Structure



## 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 <i>axis</i> processing, <i>axis</i> outputs in reset state. 1: <i>axis</i> active.
Motor Output Enabled	1	0: <i>axis</i> motor outputs disabled. 1: <i>axis</i> motor outputs enabled.
Current Control Enabled	2	0: <i>axis</i> current control bypassed. 1: <i>axis</i> current control active.
Velocity Loop Enabled	3	0: <i>axis</i> velocity loop bypassed 1: <i>axis</i> velocity loop active.
Position Loop Enabled	4	0: <i>axis</i> position loop bypassed. 1: <i>axis</i> 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.



### Description (cont.)

For example, to configure an axis for Torque mode, (trajectory, velocity, 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 occurred 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. 81)  
**RestoreOperatingMode** (p. 82), **SetDriveFaultParameter** (p. 115)

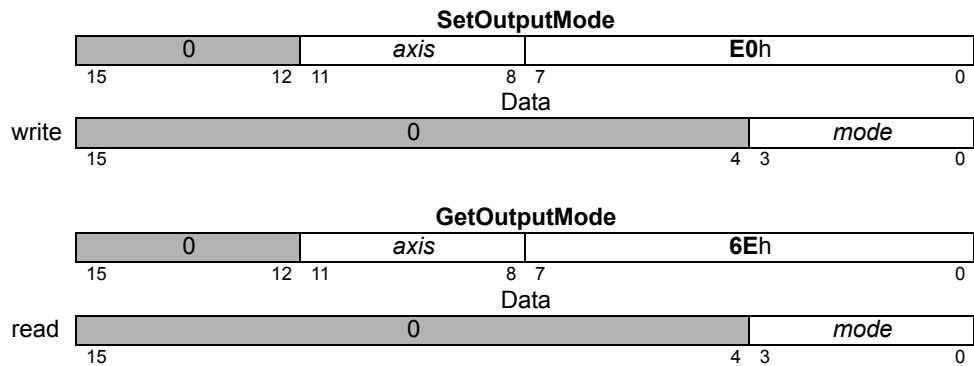
Motor Types

DC Brush	Brushless DC	Microstepping	
----------	--------------	---------------	--

Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0
<i>mode</i>	<i>PWM Sign Magnitude</i>	1
	— (Reserved)	2-6
	<i>PWM High/Low</i>	7
	— (Reserved)	8,9
	None	10

Packet  
Structure



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. 143)

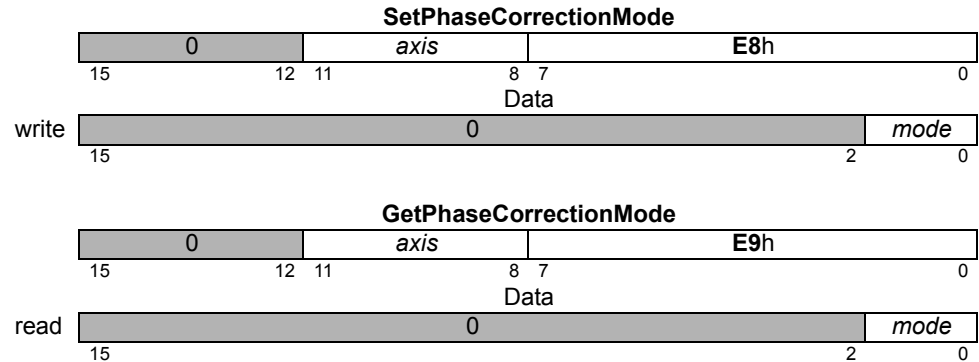
## Motor Types

	Brushless DC		
--	--------------	--	--

## Arguments

Name	Instance	Encoding
<i>axis</i>	<i>Axis1</i>	0
<i>mode</i>	<i>Disabled</i>	0
	<i>Index</i>	1
	<i>Hall</i>	2

## Packet Structure



## 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 \(p. 70\)](#)

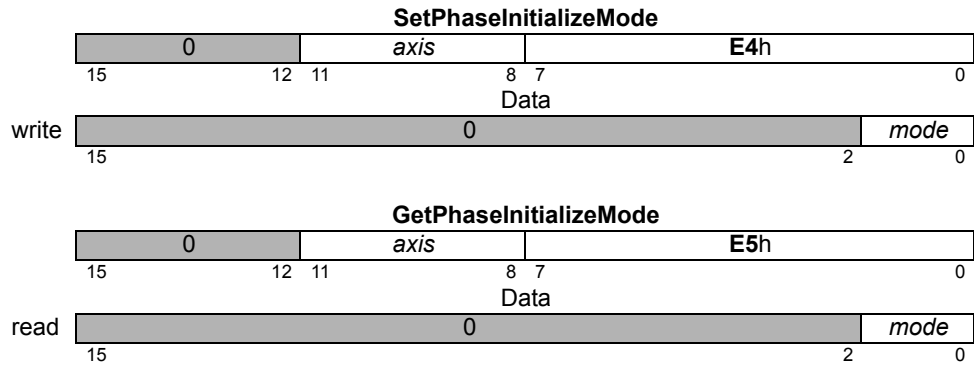
Motor Types

Table with 4 columns and 1 row. Row 1: [Empty], Brushless DC, [Empty], [Empty]

Arguments

Table with 3 columns: Name, Instance, Encoding. Row 1: axis, Axis1, 0. Row 2: mode, — (Reserved), 0. Row 3: Hall-based, 1. Row 4: Pulse, 2.

Packet Structure



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. 70), SetPhaseParameter (p. 148)

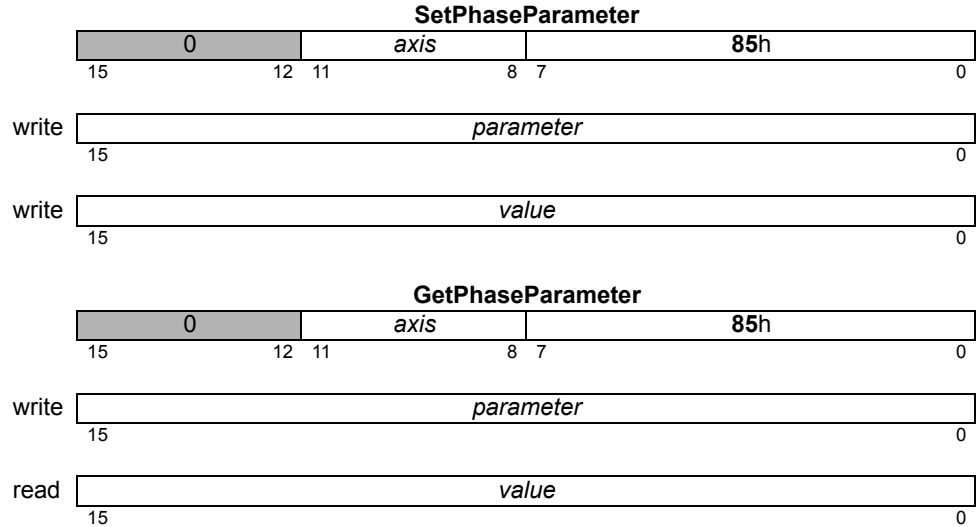
Motor Types

	Brushless DC	
--	--------------	--

Arguments

Name	Instance	Encoding	Type	Range	Scaling/Units
<i>axis</i>	<i>Axis1</i>	0			
<i>parameter</i>	ramp time	0			
	positive pulse time	1			
	negative pulse time	2			
	pulse command	3			
	— (Reserved)	4			
	ramp command	5			
<i>value</i>			unsigned 16bits	0 to 2 <sup>15</sup> –1	counts

Packet  
Structure



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.

<b>Description (cont.)</b>	<p>The ramp command is a non-negative value that is used as the motor command during the pull-in ramp.</p> <p>By default all phase parameters are zero, however phase initialization cannot possibly work in that state.</p> <p>The process of pulse phase initialization and how to set the various parameters is discussed in the <i>Juno Velocity and Torque IC User Guide</i>.</p> <p><b>GetPhaseParameter</b> is used to read the values set by SetPhaseParameter.</p>
<b>Errors</b>	Unrecognized parameter code, or value out of range.
<b>C-Motion API</b>	<pre>PMDresult PMDGetPhaseParameter (PMDAxisInterface axis_intf,                                 PMDuint16 parameter, PMDint16* value); PMDresult PMDSetPhaseParameter (PMDAxisInterface axis_intf,                                 PMDuint16 parameter, PMDint16 value);</pre>
<b>Script API</b>	<pre>GetPhaseParameter parameter SetPhaseParameter parameter value</pre>
<b>C# API</b>	<pre>Int32 value = PMDAxis.GetPhaseParameter(PMDPhaseParameter parameter); PMDAxis.SetPhaseParameter(PMDPhaseParameter parameter, Int32 value);</pre>
<b>Visual Basic API</b>	<pre>Int32 value = PMDAxis.GetPhaseParameter(ByVal parameter  As PMDPhaseParameter) PMDAxis.SetPhaseParameter(ByVal parameter As PMDPhaseParameter,                           ByVal value As Int32)</pre>
<b>see</b>	<b>InitializePhase</b> (p. 70), <b>SetPhaseInitializeMode</b> (p. 147)

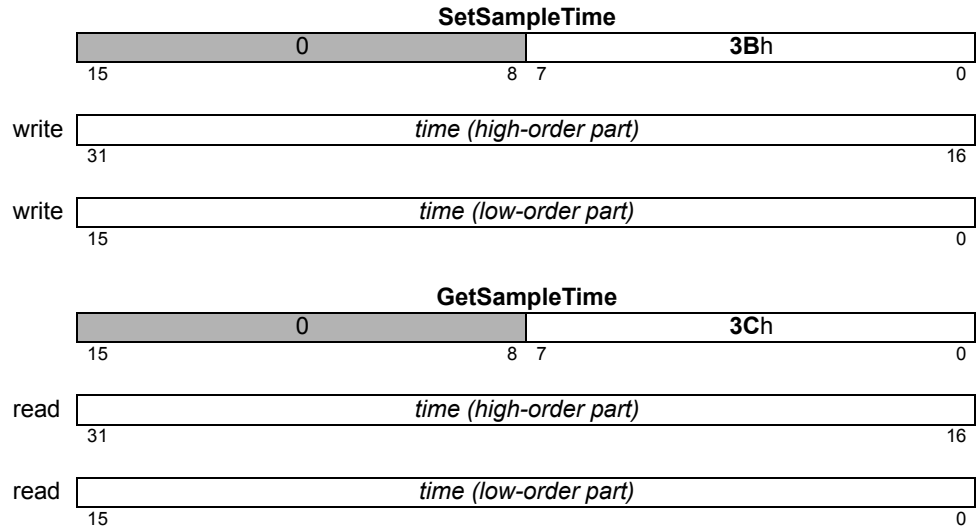
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

Name	Type	Range	Units
<i>time</i>	unsigned 32 bits	51 to 2 <sup>20</sup>	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 $\mu$ s	102.4 $\mu$ s	102.4 $\mu$ s	9.76 kHz

**GetSampleTime** returns the value of the sample time.

<b>Restrictions</b>	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.
<b>Errors</b>	<b>Invalid Parameter:</b> Argument out of range.
<b>C-Motion API</b>	<pre> PMDresult <b>PMDSetSampleTime</b>(PMDAxisInterface axis_intf,                              PMDuint32 time); PMDresult <b>PMDGetSampleTime</b>(PMDAxisInterface axis_intf,                              PMDuint32* time); </pre>
<b>Script API</b>	<pre> <b>GetSampleTime</b> <b>SetSampleTime</b> time </pre>
<b>C# API</b>	<pre> UINT32 time = <b>PMDAxis.SampleTime</b>; <b>PMDAxis.SampleTime</b> = time; </pre>
<b>Visual Basic API</b>	<pre> UINT32 time = <b>PMDAxis.SampleTime</b> <b>PMDAxis.SampleTime</b> = time </pre>
<b>see</b>	



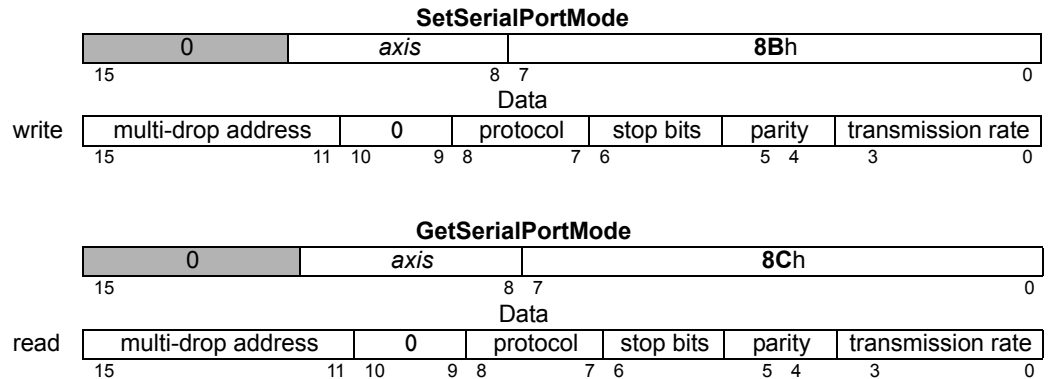
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

Name	Type	Encoding
<i>mode</i>	unsigned 16 bits	see below

## Packet Structure



## 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 arguments 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 =  $0 \times 2048 + 0 \times 128 + 1 \times 64 + 0 \times 16 + 4 = 68$ .

<b>Restrictions</b>	Multi-drop serial communication is not supported by all products, see the product user guide.
<b>Errors</b>	<b>Invalid Parameter:</b> Requested multi-drop protocol not supported.
<b>C-Motion API</b>	<pre> PMDresult <b>PMDSetSerialPortMode</b>(PMDAxisInterface axis_intf,                                 PMDuint8 baud,                                 PMDuint8 parity,                                 PMDuint8 stopBits,                                 PMDuint8 protocol,                                 PMDuint8 multiDropID); PMDresult <b>PMDGetSerialPortMode</b>(PMDAxisInterface axis_intf,                                 PMDuint8* baud,                                 PMDuint8* parity,                                 PMDuint8* stopBits,                                 PMDuint8* protocol,                                 PMDuint8* multiDropID); </pre>
<b>Script API</b>	<pre> <b>GetSerialPortMode</b> <b>SetSerialPortMode</b> mode where mode = MultiDropId*2048 + protocol*128 + StopBits*64 + parity*16 +             baud </pre>
<b>C# API</b>	<pre> PMDAxis.<b>GetSerialPortMode</b>(ref PMDSerialBaud baud,                            ref PMDSerialParity parity,                            ref PMDSerialStopBits StopBits,                            ref PMDSerialProtocol protocol,                            ref Byte MultiDropId); PMDAxis.<b>SetSerialPortMode</b>( PMDSerialBaud baud,                            PMDSerialParity parity,                            PMDSerialStopBits StopBits,                            PMDSerialProtocol protocol,                            Byte MultiDropId); </pre>
<b>Visual Basic API</b>	<pre> PMDAxis.<b>GetSerialPortMode</b>(ByRef baud As PMDSerialBaud,                            ByRef parity As PMDSerialParity,                            ByRef StopBits As PMDSerialStopBits,                            ByRef protocol As PMDSerialProtocol,                            ByRef MultiDropId As Byte) PMDAxis.<b>SetSerialPortMode</b>(ByVal baud As PMDSerialBaud,                            ByVal parity As PMDSerialParity,                            ByVal StopBits As PMDSerialStopBits,                            ByVal protocol As PMDSerialProtocol,                            ByVal MultiDropId As Byte) </pre>
<b>see</b>	

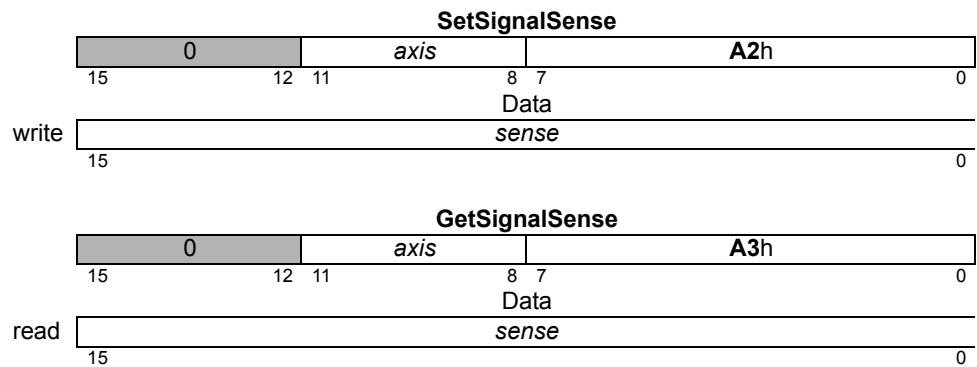
Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

Arguments

Name	Instance	Encoding	
<i>axis</i>	<i>Axis1</i>	0	
<i>sense</i>	<b>Indicator</b>	<b>Encoding</b>	<b>Bit Number</b>
	<i>EncoderA</i>	0001h	0
	<i>EncoderB</i>	0002h	1
	<i>Encoder Index</i>	0004h	2
	— (Reserved)		3-6
	<i>HallA</i>	0080h	7
	<i>HallB</i>	0100h	8
	<i>HallC</i>	0200h	9
	— (Reserved)		10
	<i>Pulse Input</i>	0800h	11
	<i>Motor Direction</i>	1000h	12
	— (Reserved)		13,14
	<i>Direction Input</i>	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.

<b>Description (cont.)</b>	<p>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.</p> <p>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.</p> <p><b>GetSignalSense</b> returns the value of the Signal Sense mask.</p>
<b>Restrictions</b>	<p>FaultOut and /Enable exist in the Signal Status register, but their sense is not controllable.</p> <p>Not all bits are implemented for all products. See the product user guide.</p>
<b>Errors</b>	None
<b>C-Motion API</b>	<pre>PMDresult PMDSetSignalSense(PMDAxisInterface axis_intf,                              PMDuint16 sense); PMDresult PMDGetSignalSense(PMDAxisInterface axis_intf,                              PMDuint16* sense);</pre>
<b>Script API</b>	<pre>GetSignalSense SetSignalSense sense</pre>
<b>C# API</b>	<pre>UInt16 sense = PMDAxis.SignalSense; PMDAxis.SignalSense = sense;</pre>
<b>Visual Basic API</b>	<pre>UInt16 sense = PMDAxis.SignalSense PMDAxis.SignalSense = sense</pre>
<b>see</b>	<b>GetSignalStatus</b> ( <a href="#">p. 63</a> )

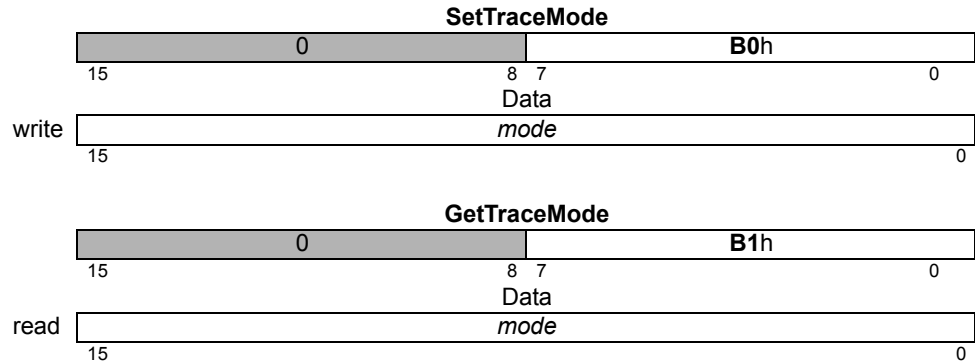
Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

Arguments

<b>Name</b>	<b>Instance</b>	<b>Encoding</b>
<i>mode</i>	<i>16-bit unsigned</i>	see below

Packet  
Structure



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. 67)

Motor Types	<div> <div>DC Brush</div> <div>Brushless DC</div> <div>Microstepping</div> </div>				
Arguments	<div> <div>Name</div> <div><i>period</i></div> </div>	<div> <div>Type</div> <div>unsigned 16 bits</div> </div>	<div> <div>Range</div> <div>1 to 2<sup>16</sup>–1</div> </div>	<div> <div>Scaling</div> <div>unity</div> </div>	<div> <div>Units</div> <div>cycles</div> </div>
Packet Structure	<div> <div> <div>SetTracePeriod</div> <div> <div> <div>0</div> <div>15</div> <div>8</div> <div>7</div> <div>0</div> </div> <div> <div>B8h</div> </div> </div> <div> <div>write</div> <div> <div>Data</div> <div><i>period</i></div> <div>15</div> <div>0</div> </div> </div> </div> <div> <div> <div>GetTracePeriod</div> <div> <div>0</div> <div>15</div> <div>8</div> <div>7</div> <div>0</div> </div> <div> <div>B9h</div> </div> </div> <div> <div>read</div> <div> <div>Data</div> <div><i>period</i></div> <div>15</div> <div>0</div> </div> </div> </div> </div>				
Description	<div> <div> <div>SetTracePeriod</div> <div>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.</div> </div> <div> <div>GetTracePeriod</div> <div>returns the value for the trace period.</div> </div> </div>				
Errors	<div>Invalid Parameter: Zero Period</div>				
C-Motion API	<div> <div> <div>PMDresult</div> <div>PMDSetTracePeriod</div> <div>(PMDAxisInterface</div> <div><i>axis_intf</i>,</div> <div>PMDuint16</div> <div><i>period</i>);</div> </div> <div> <div>PMDresult</div> <div>PMDGetTracePeriod</div> <div>(PMDAxisInterface</div> <div><i>axis_intf</i>,</div> <div>PMDuint16*</div> <div><i>period</i>);</div> </div> </div>				
Script API	<div> <div>GetTracePeriod</div> <div>SetTracePeriod</div> <div><i>period</i></div> </div>				
C# API	<div> <div> <div>UInt16</div> <div><i>period</i></div> <div>=</div> <div>PMDAxis.TracePeriod;</div> </div> <div> <div>PMDAxis.TracePeriod</div> <div>=</div> <div><i>period</i>;</div> </div> </div>				
Visual Basic API	<div> <div> <div>UInt16</div> <div><i>period</i></div> <div>=</div> <div>PMDAxis.TracePeriod</div> </div> <div> <div>PMDAxis.TracePeriod</div> <div>=</div> <div><i>period</i></div> </div> </div>				
see	<div> <div>Set/GetSampleTime</div> <div>(</div> <div><a href="#">p. 150</a></div> <div>,</div> <div>Set/GetTraceStart</div> <div>(</div> <div><a href="#">p. 158</a></div> <div>,</div> <div>Set/GetTraceStop</div> <div>(</div> <div><a href="#">p. 161</a></div> <div>)</div> </div>				

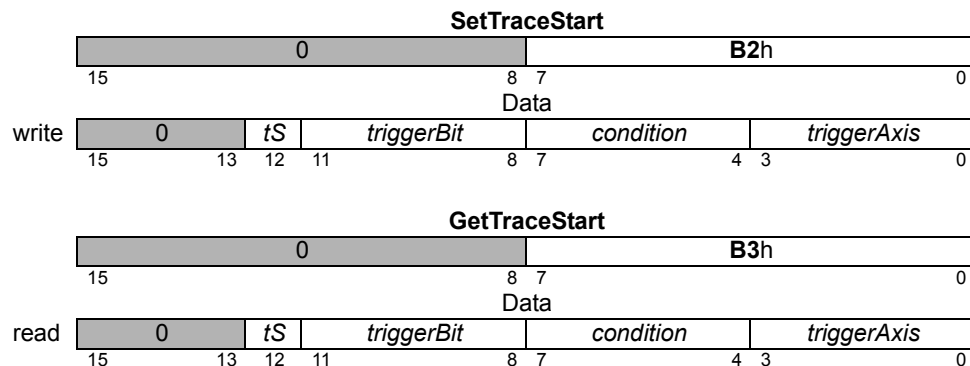
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

Name	Instance	Encoding
<i>triggerAxis</i>	<i>Axis1</i>	0
<i>condition</i>	<i>Immediate</i>	0
	— (Reserved)	1
	<i>Event Status</i>	2
	<i>Activity Status</i>	3
	<i>Signal Status</i>	4
	<i>Drive Status</i>	5
	— (Reserved)	6
	<i>Signed trace value greater than</i>	7
	<i>Signed trace value less than</i>	8
	<i>Unsigned trace value higher than</i>	9
	<i>Unsigned trace value lower than</i>	10
	<i>Trace value bitmask</i>	11
<i>triggerBit</i>	<i>Status Register Bit</i>	0 to 15
<i>triggerState (tS)</i>	<i>Triggering State of the Bit</i>	0 (value = 0) 1 (value = 1)

## Packet Structure



## 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).

### 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()*.

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. 91), **GetTraceCount** (p. 66), **Set/GetTraceMode** (p. 156),  
**Set/GetTracePeriod** (p. 157), **Set/GetTraceStop** (p. 161), **Set/GetTraceTriggerValue** (p. 89)

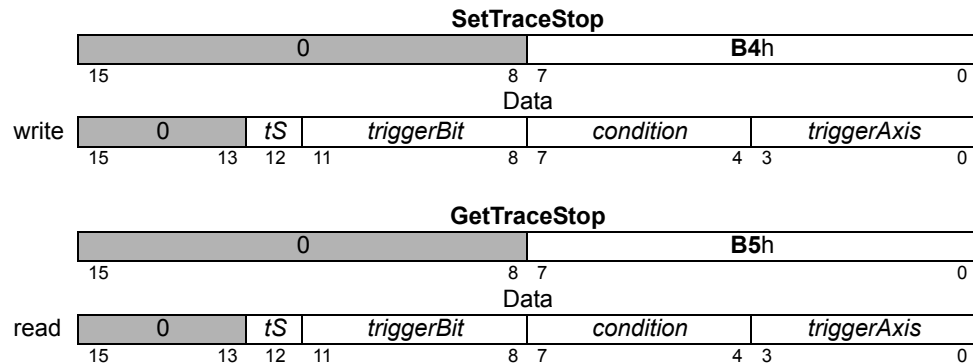
### Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

### Arguments

Name	Instance	Encoding
<i>triggerAxis</i>	<i>Axis1</i>	0
<i>condition</i>	<i>Immediate</i>	0
	<i>Next Update</i>	1
	<i>Event Status</i>	2
	<i>Activity Status</i>	3
	<i>Signal Status</i>	4
	<i>Drive Status</i>	5
	— (Reserved)	6
	<i>Signed trace value greater than</i>	7
	<i>Signed trace value less than</i>	8
	<i>Unsigned trace value higher than</i>	9
	<i>Unsigned trace value lower than</i>	10
	<i>Trace value bitmask</i>	11
<i>triggerBit</i>	<i>Status Register Bit</i>	0 to 15
<i>triggerState (tS)</i>	<i>Triggering State of the Bit</i>	0 (value = 0) 1 (value = 1)

### Packet Structure



### 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 158, 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.

### 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 + trigger-  
Axis
```

### 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. 66), **Set/GetTraceStart** (p. 158), **GetTraceStatus** (p. 67)

## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

Name	Instance	Encoding
<i>variableNumber</i>	<i>Variable1</i>	0
	<i>Variable2</i>	1
	<i>Variable3</i>	2
	<i>Variable4</i>	3
<i>traceAxis</i>	<i>Axis1</i>	0
<i>variableID</i>	<i>None</i>	0
	<i>Position Error</i>	1
	<i>Commanded Position</i>	2
	<i>Commanded Velocity</i>	3
	<i>Commanded Acceleration</i>	4
	<i>Actual Position</i>	5
	<i>Actual Velocity</i>	6
	<i>Active Motor Command</i>	7
	<i>Motion Processor Time</i>	8
	<i>Capture Value</i>	9
	<i>Position Loop Integrator Sum</i>	10
	<i>Position/Outer Loop Derivative Term</i>	11
	<i>Event Status</i>	12
	<i>Activity Status</i>	13
	<i>Signal Status</i>	14
	<i>Phase Angle</i>	15
	<i>Phase Offset</i>	16
	— (Reserved)	17-19
	<i>Analog Raw Channel 0</i>	20
	<i>Analog Raw Channel 1</i>	21
	<i>Analog Raw Channel 2</i>	22
	<i>Analog Raw Channel 3</i>	23
	<i>Analog Raw Channel 4</i>	24
	<i>Analog Raw Channel 5</i>	25
	<i>Analog Raw Channel 6</i>	26
	<i>Analog Raw Channel 7</i>	27
	— (Reserved)	28
	<i>Phase Angle Scaled</i>	29
	— (Reserved)	30
	<i>Phase A Actual Current</i>	31
	— (Reserved)	32-35
	<i>Phase B Actual Current</i>	36
	— (Reserved)	37-39
	<i>d Component Reference</i>	40
	<i>d Component Error</i>	41
	<i>d Component Actual Current</i>	42
	— (Reserved)	43
	<i>d Component Integral Term</i>	44
	<i>d Component Output</i>	45

Arguments (cont.)	variableID (cont.)	
	<i>q Component Reference</i>	46
	<i>q Component Error</i>	47
	<i>q Component Actual Current</i>	48
	— (Reserved)	49
	<i>q Component Integral Term</i>	50
	<i>q Component Output</i>	51
	<i>Alpha Component Output</i>	52
	<i>Beta Component Output</i>	53
	<i>Bus Voltage</i>	54
	<i>Temperature</i>	55
	<i>Drive Status</i>	56
	<i>Position/Outer Loop Integral Term</i>	57
	— (Reserved)	58-67
	<i>Foldback Energy</i>	68
	<i>Leg A Current</i>	69
	<i>Leg B Current</i>	70
	<i>Leg C Current</i>	71
	<i>Leg DCurrent</i>	72
	<i>Alpha Component Current</i>	73
	<i>Beta Component Current</i>	74
	<i>PWM A Output</i>	75
	<i>PWM B Output</i>	76
	<i>PWM C Output</i>	77
	— (Reserved)	78
	<i>Drive Fault Status</i>	79
	— (Reserved)	80-82
	<i>Actual Velocity</i>	83
	<i>Raw Encoder Reading</i>	84
	— (Reserved)	85
	<i>Bus Current Supply</i>	86
	<i>Bus Current Return</i>	87
	— (Reserved)	88
	<i>Commutation Error</i>	89
	— (Reserved)	90-94
	<i>Estimated Velocity</i>	95
	<i>Commanded Velocity</i>	96
	<i>Velocity Error</i>	97
	<i>Velocity Loop Integral Term</i>	98
	<i>Velocity Loop Output</i>	99
	<i>Velocity Biquad Input</i>	100
	<i>Analog Command Biquad Input</i>	101
	<i>Tachometer</i>	102
	<i>Analog Command</i>	103
	<i>Position/Outer loop Output</i>	104
	<i>SPI Direct Input</i>	105
	— (Reserved)	106,107
	<i>Internal Profile Position</i>	108
	<i>Internal Profile Velocity</i>	109
	<i>Active Operating Mode</i>	110
	<i>Analog Raw Channel 8</i>	111
	<i>Analog Raw Channel 9</i>	112
	— (Reserved)	113-116

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

### 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

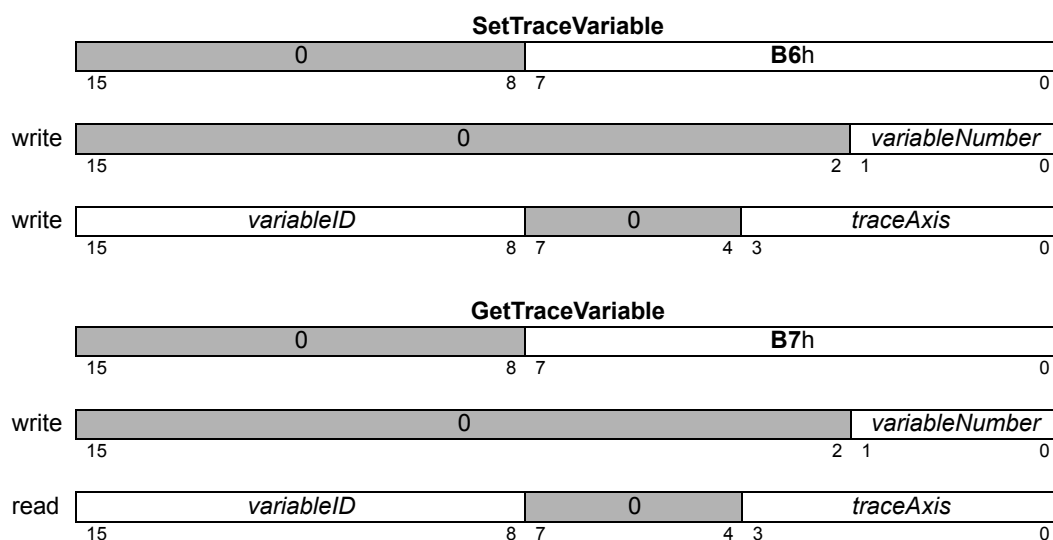
#### Analog Inputs

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



### 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
<b>Command Source</b>				
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 <sup>2</sup> or microsteps/cycle <sup>2</sup>
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
<b>Encoder</b>				
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
<b>Position/Outer Loop</b>				
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
<b>Velocity Loop</b>				
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
<b>Status Registers</b>				
Event Status	12	unsigned 16 bit	-	see <b>GetEventStatus</b>
Activity Status	13	unsigned 16 bit	-	see <b>GetActivityStatus</b>
Signal Status	14	unsigned 16 bit	-	see <b>GetSignalStatus</b>
Drive Status	56	unsigned 16 bit	-	see <b>GetDriveStatus</b>
Drive Fault Status	79	unsigned 16 bit	-	see <b>GetDriveFaultStatus</b>
Active Operating Mode	110	unsigned 16 bit	-	see <b>GetActiveOperating Mode</b>

### Description (cont.)

Variable	Encoding	Type	Scaling	Units/Notes
<b>Commutation/Phasing</b>				
Active Motor Command	7	signed 16 bit	100/2 <sup>15</sup>	% 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 <sup>15</sup>	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
<b>Current Control</b>				
Phase A Actual Current	31	signed 16 bit	160/2 <sup>15</sup>	% max leg current analog input
Phase B Actual Current	36	signed 16 bit	160/2 <sup>15</sup>	% max leg current analog input
d Component Reference	40	signed 16 bit	160/2 <sup>15</sup>	% max leg current analog input
d Component Error	41	signed 16 bit	160/2 <sup>15</sup>	% max leg current analog input
d Component Actual Current	42	signed 16 bit	160/2 <sup>15</sup>	% max leg current analog input
d Component Integral Term	44	signed 32 bit	200/2 <sup>15</sup>	% output
d Component Output	45	signed 16 bit	100/2 <sup>15</sup>	% output
q Component Reference	46	signed 16 bit	160/2 <sup>15</sup>	% max leg current analog input
q Component Error	47	signed 16 bit	160/2 <sup>15</sup>	% max leg current analog input
q Component Actual Current	48	signed 16 bit	160/2 <sup>15</sup>	% max leg current analog input
q Component Integral Term	50	signed 32 bit	200/2 <sup>15</sup>	% output
q Component Output	51	signed 16 bit	100/2 <sup>15</sup>	% output
Alpha Component Output	52	signed 16 bit	100/2 <sup>15</sup>	% output
Beta Component Output	53	signed 16 bit	100/2 <sup>15</sup>	% output
Leg A Current	69	signed 16 bit	100/2 <sup>15</sup>	% max leg current analog input
Leg B Current	70	signed 16 bit	100/2 <sup>15</sup>	% max leg current analog input
Leg C Current	71	signed 16 bit	100/2 <sup>15</sup>	% max leg current analog input
Leg D Current	72	signed 16 bit	100/2 <sup>15</sup>	% max leg current analog input

### Description (cont.)

Variable	Encoding	Type	Scaling	Units/Notes
<b>Current Control (cont.)</b>				
Alpha Component Current	73	signed 16 bit	100/2 <sup>15</sup>	% max leg current analog input
Beta Component Current	74	signed 16 bit	100/2 <sup>15</sup>	% max leg current analog input
<b>Motor Output</b>				
Bus Voltage	54	unsigned 16 bit	100/2 <sup>16</sup>	% bus voltage analog input
Temperature	55	unsigned 16 bit	100/2 <sup>15</sup>	% temperature analog input
Foldback Energy	68	unsigned 32 bit	see note below	A <sup>2</sup> s
PWM A Output	75	signed 16 bit	100/2 <sup>15</sup>	% max output
PWM B Output	76	signed 16 bit	100/2 <sup>15</sup>	% max output
PWM C Output	77	signed 16 bit	100/2 <sup>15</sup>	% max output
Bus Current Supply	86	signed 16 bit	100/2 <sup>15</sup>	% max bus current analog input
Bus Current Return	87	signed 16 bit	100/2 <sup>15</sup>	% max leg current analog input
<b>Analog Inputs</b>				
Analog Raw Channel 0	20	unsigned 16 bit	100/2 <sup>16</sup>	% input
Analog Raw Channel 1	21	unsigned 16 bit	100/2 <sup>16</sup>	% input
Analog Raw Channel 2	22	unsigned 16 bit	100/2 <sup>16</sup>	% input
Analog Raw Channel 3	23	unsigned 16 bit	100/2 <sup>16</sup>	% input
Analog Raw Channel 4	24	unsigned 16 bit	100/2 <sup>16</sup>	% input
Analog Raw Channel 5	25	unsigned 16 bit	100/2 <sup>16</sup>	% input
Analog Raw Channel 6	26	unsigned 16 bit	100/2 <sup>16</sup>	% input
Analog Raw Channel 7	27	unsigned 16 bit	100/2 <sup>16</sup>	% input
Analog Raw Channel 8	111	unsigned 16 bit	100/2 <sup>16</sup>	% input
Analog Raw Channel 9	112	unsigned 16 bit	100/2 <sup>16</sup>	% input
<b>Miscellaneous</b>				
None	0	-	-	Terminates variable list
Motion Processor Time	8	unsigned 32 bit	unity	cycles

### 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  $t_c(i_{fs}/20480)^{2^{15}}$ , where  $t_c$  is the current loop period of  $51.2 \times 10^{-6}$ s and  $i_{fs}$  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 interface 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 =  $7 \times 256 + 0 = 1792$ , 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 \(p. 157\)](#), [SetTraceStart \(p. 158\)](#), [SetTraceStop \(p. 161\)](#), [GetTraceVariable \(p. 163\)](#)

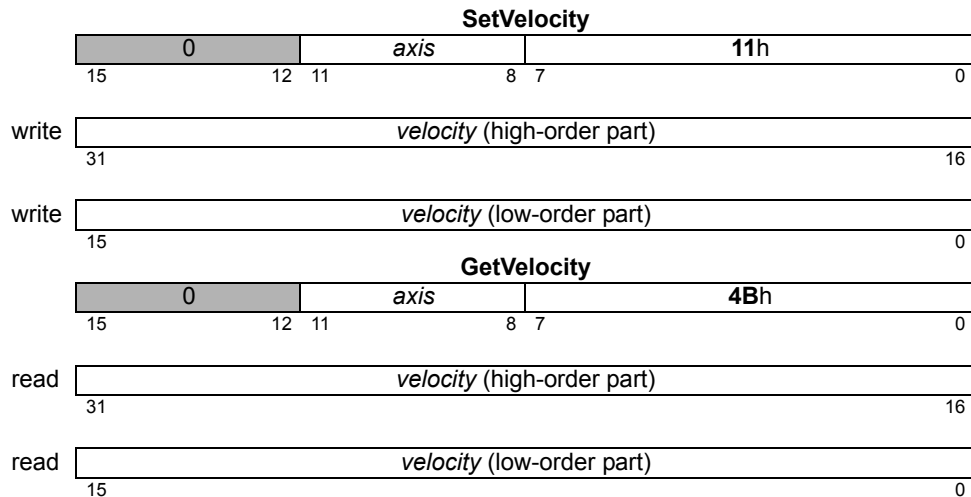
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

Name	Instance	Encoding			
<i>axis</i>	<i>Axis1</i>	0			
<i>velocity</i>	<b>Type</b> signed 32 bits	<b>Range</b> $-2^{31}$ to $2^{31}-1$	<b>Scaling</b> $1/2^{16}$	<b>Units</b> counts/cycle microsteps/cycle	

## Packet Structure



## 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. 83](#)), **Set/GetDeceleration** ([p. 112](#))

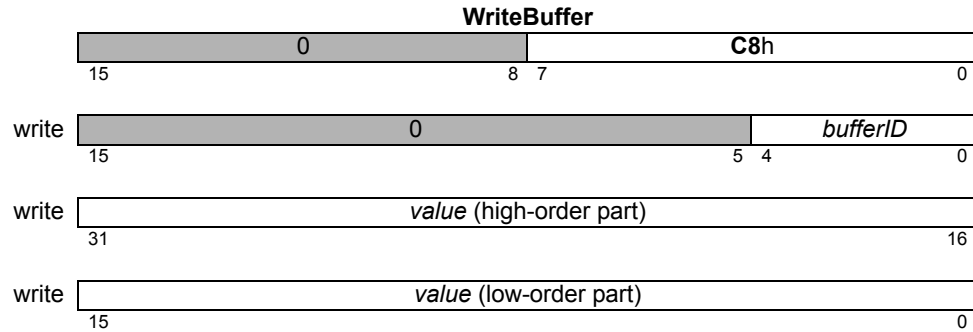
## Motor Types

DC Brush	Brushless DC	Microstepping
----------	--------------	---------------

## Arguments

Name	Type	Range
<i>bufferID</i>	unsigned 16 bits	0 to 7
<i>value</i>	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(ByteVal bufferID As Int16, ByteVal data As Int32);
```

## see

**ReadBuffer** (p. 76), **Set/GetBufferWriteIndex** (p. 98)



## 8. Instruction Summary Tables

### 8.1 Descriptions by Functional Category

Interrupts		Page
<b>ClearInterrupt</b>	Reset interrupt.	<a href="#">33</a>
<b>Set/GetInterruptMask</b>	Set/Get interrupt event mask.	<a href="#">132</a>
<b>Motor Phase and Commutation</b>		
<b>Set/GetCommutationMode</b>	Set/Get the commutation phasing mode.	<a href="#">102</a>
<b>Set/GetPhaseCorrectionMode</b>	Set/Get phase correction method.	<a href="#">147</a>
<b>Set/GetCommutationParameter</b>	Set/Get phase counts and other commutation parameters.	<a href="#">103</a>
<b>Set/GetPhaseParameter</b>	Set/Get phase initialization parameters.	<a href="#">149</a>
<b>Set/GetPhaseInitializeMode</b>	Set/Get phase initialization method.	<a href="#">148</a>
<b>InitializePhase</b>	Perform phase initialization procedure.	<a href="#">71</a>
<b>Current Loops</b>		
<b>CalibrateAnalog</b>	Determine offsets to zero analog inputs.	<a href="#">31</a>
<b>Set/GetAnalogCalibration</b>	Set/Get analog offsets.	<a href="#">88</a>
<b>Set/GetCurrentControlMode</b>	Set/Get current control mode (FOC or third leg floating).	<a href="#">108</a>
<b>Set/GetFOC</b>	Set/Get parameters for current control.	<a href="#">130</a>
<b>GetFOCValue</b>	Get value of current control state.	<a href="#">54</a>
<b>Digital Servo Filter</b>		
<b>ClearPositionError</b>	Adjust commanded position to make error zero.	<a href="#">34</a>
<b>GetPositionError</b>	Get actual position error.	<a href="#">60</a>
<b>Set/GetDriveCommandMode</b>	Set/Get mode for commanding position, velocity, or torque	<a href="#">114</a>
<b>Set/GetLoop</b>	Set/Get parameter for position/outer or velocity loop	<a href="#">134</a>
<b>GetLoopValue</b>	Get value of position/outer or velocity loop state	<a href="#">58</a>
<b>Encoder</b>		
<b>AdjustActualPosition</b>	Change the current encoder position by a specified offset.	<a href="#">30</a>
<b>Set/GetActualPosition</b>	Set/Get the current encoder position.	<a href="#">86</a>
<b>Set/GetActualPositionUnits</b>	Set/Get units of encoder position for step motors, counts or microsteps.	<a href="#">87</a>
<b>GetActualVelocity</b>	Get the actual encoder velocity, without smoothing.	<a href="#">41</a>
<b>GetCaptureValue</b>	Get the most recent index capture encoder position.	<a href="#">42</a>
<b>Set/GetEncoderSource</b>	Set/Get the type of position feedback.	<a href="#">121</a>
<b>Set/GetEncoderToStepRatio</b>	Set/Get the ratio of encoder counts to microsteps.	<a href="#">123</a>
<b>Motor Output</b>		
<b>GetActiveMotorCommand</b>	Get the active commanded motor output	<a href="#">37</a>
<b>GetDriveValue</b>	Read drive bus voltage, bus current, or temperature.	<a href="#">50</a>
<b>Set/GetMotorCommand</b>	Set/Get the motor command if position/outer and velocity loops are disabled.	<a href="#">138</a>
<b>Set/GetMotorType</b>	Set/Get the motor type.	<a href="#">142</a>

**Motor Output**

<b>Set/GetOutputMode</b>	Set/Get the method of driving the motor amplifier.	<a href="#">146</a>
<b>Set/GetDrivePWM</b>	Set/Get various PWM parameters, eg signal sense, frequency, and dead time.	<a href="#">119</a>
<b>Set/GetCurrentFoldback</b>	Set/Get current foldback limits.	<a href="#">109</a>
<b>Set/GetCurrent</b>	Set/Get current commands for driving step motors.	<a href="#">106</a>
<b>Set/GetCurrentLimit</b>	Set/Get the maximum current that the velocity or position/outer loop may command.	<a href="#">111</a>

**Operating Mode and Event Control**

<b>Set/GetOperatingMode</b>	Set/Get the static operating mode of an axis.	<a href="#">144</a>
<b>RestoreOperatingMode</b>	Restore the active operating mode from the static operating mode of an axis.	<a href="#">83</a>
<b>GetActiveOperatingMode</b>	Get the active operating mode of an axis.	<a href="#">38</a>
<b>Set/GetEventAction</b>	Set/Get the response to events or other exceptional conditions.	<a href="#">125</a>

**Position Servo Loop Control**

<b>Set/GetSampleTime</b>	Set/Get the profile and servo loop sample time .	<a href="#">151</a>
<b>GetTime</b>	Get the current IC time, in commutation periods.	<a href="#">66</a>

**Profile Generation**

<b>Set/GetAcceleration</b>	Set/Get the maximum acceleration for the internal profile.	<a href="#">84</a>
<b>GetCommandedAcceleration</b>	Get the current commanded profile acceleration.	<a href="#">43</a>
<b>GetCommandedPosition</b>	Get the current commanded position.	<a href="#">44</a>
<b>GetCommandedVelocity</b>	Get the current commanded (not scaled) velocity.	<a href="#">45</a>
<b>Set/GetDeceleration</b>	Set/Get the maximum deceleration, if different from the maximum acceleration.	<a href="#">113</a>
<b>Set/GetVelocity</b>	Set/Get the maximum velocity for the internal profile.	<a href="#">174</a>

**RAM Buffers**

<b>Set/GetBufferLength</b>	Set/Get the length of a memory buffer.	<a href="#">92</a>
<b>Set/GetBufferReadIndex</b>	Set/Get the index of the next read from a memory buffer.	<a href="#">94</a>
<b>Set/GetBufferStart</b>	Set/Get the starting address of a memory buffer.	<a href="#">96</a>
<b>Set/GetBufferWriteIndex</b>	Set/Get the index of the next write to a memory buffer.	<a href="#">98</a>
<b>ReadBuffer</b>	Read a 32 bit double word from a RAM buffer.	<a href="#">76</a>
<b>ReadBuffer16</b>	Read a 16 bit word from an NVRAM buffer.	<a href="#">77</a>
<b>WriteBuffer</b>	Write a 32 bit double word to a RAM buffer.	<a href="#">176</a>

**Drive**

<b>Set/GetDriveFaultParameter</b>	Set/Get some drive safety parameters.	<a href="#">116</a>
<b>Set/GetFaultOutMask</b>	Set/Get the event mask for driving the FaultOut signal.	<a href="#">128</a>
<b>GetDriveFaultStatus</b>	Get a latched register showing some drive faults status.	<a href="#">46</a>
<b>GetDriveValue</b>	Get some current drive state.	<a href="#">50</a>
<b>ClearDriveFaultStatus</b>	Clear (zero) all drive fault bits.	<a href="#">32</a>

**Status Registers**

<b>GetActivityStatus</b>	Get a register showing some current activity state.	<a href="#">40</a>
<b>GetDriveStatus</b>	Get a register showing some current drive state.	<a href="#">48</a>
<b>GetEventStatus</b>	Get a latched register showing some significant events.	<a href="#">52</a>
<b>GetSignalStatus</b>	Get the current status of some input/output signals.	<a href="#">64</a>
<b>Set/GetSignalSense</b>	Set/Get the logical sense of some input/output signals.	<a href="#">155</a>

**Status Registers**

<b>ResetEventStatus</b>	Clear (zero) some event bits.	<a href="#">82</a>
-------------------------	-------------------------------	--------------------

**Traces**

<b>GetTraceCount</b>	Get the number of trace values that have been stored.	<a href="#">67</a>
<b>Set/GetTraceMode</b>	Set/Get the trace mode (one-time or rolling).	<a href="#">157</a>
<b>Set/GetTracePeriod</b>	Set/Get the frequency of trace captures.	<a href="#">158</a>
<b>Set/GetTraceStart</b>	Set/Get the condition that will start a trace.	<a href="#">159</a>
<b>Set/GetTraceStop</b>	Set/Get the condition that will stop a trace.	<a href="#">162</a>
<b>GetTraceStatus</b>	Get the trace status word.	<a href="#">68</a>
<b>Set/GetTraceVariable</b>	Set/Get the set of quantities to save in a trace.	<a href="#">164</a>
<b>GetTraceValue</b>	Get the current value of a traceable quantity.	<a href="#">69</a>
<b>Set/GetTraceTriggerValue</b>	Set/Get a value to be used to determine trace start or stop.	<a href="#">90</a>

**Communications**

<b>Set/GetCANMode</b>	Set/Get the CANBus baud rate and node identifier.	<a href="#">100</a>
<b>GetInstructionError</b>	Get and clear command error codes.	<a href="#">56</a>
<b>Set/GetSerialPortMode</b>	Set/Get the serial port configuration.	<a href="#">153</a>
<b>GetSPIMode</b>	Get the current SPI mode: host command or direct.	<a href="#">65</a>
<b>GetRuntimeError</b>	Get and clear error codes not associated with a command.	<a href="#">63</a>

**Miscellaneous**

<b>GetProductInfo</b>	Get fixed configuration and version information.	<a href="#">61</a>
<b>ExecutionControl</b>	Control some aspects of NVRAM IC initialization.	<a href="#">35</a>
<b>GetVersion</b>	Legacy version command, returns zero.	<a href="#">70</a>
<b>NoOperation</b>	Perform no operation, used to verify communications.	<a href="#">74</a>
<b>Reset</b>	Reset IC.	<a href="#">78</a>
<b>NVRAM</b>	Program non-volatile memory.	<a href="#">72</a>
<b>ReadAnalog</b>	Read a raw analog input.	<a href="#">75</a>

## 8.2 Alphabetical Listing

Get/Set instructions pairs are shown together on the same line of the table.



Instruction	Code	Instruction	Code	Page
<b>AdjustActualPosition</b>	F5h			<a href="#">30</a>
<b>CalibrateAnalog</b>	6Fh			<a href="#">31</a>
<b>ClearDriveFaultStatus</b>	6Ch			<a href="#">32</a>
<b>ClearInterrupt</b>	ACh			<a href="#">33</a>
<b>ClearPositionError</b>	47h			<a href="#">34</a>
<b>ExecutionControl</b>	35h			<a href="#">35</a>
<b>GetActiveMotorCommand</b>	3Ah			<a href="#">37</a>
<b>GetActiveOperatingMode</b>	57h			<a href="#">38</a>
<b>GetActivityStatus</b>	A6h			<a href="#">40</a>
<b>GetActualVelocity</b>	ADh			<a href="#">41</a>
<b>GetCaptureValue</b>	36h			<a href="#">42</a>
<b>GetCommandedAcceleration</b>	A7h			<a href="#">43</a>
<b>GetCommandedPosition</b>	IDh			<a href="#">44</a>

Instruction	Code	Instruction	Code	Page
<b>GetCommandedVelocity</b>	1Eh			<a href="#">45</a>
<b>GetDriveFaultStatus</b>	6Dh			<a href="#">46</a>
<b>GetDriveStatus</b>	0Eh			<a href="#">48</a>
<b>GetDriveValue</b>	70h			<a href="#">50</a>
<b>GetEventStatus</b>	31h			<a href="#">52</a>
<b>GetFOCValue</b>	5Ah			<a href="#">54</a>
<b>GetInstructionError</b>	A5h			<a href="#">56</a>
<b>GetLoopValue</b>	38h			<a href="#">58</a>
<b>GetPositionError</b>	99h			<a href="#">60</a>
<b>GetProductInfo</b>	01h			<a href="#">61</a>
<b>GetRuntimeError</b>	3Dh			<a href="#">63</a>
<b>GetSPIMode</b>	0Bh			<a href="#">65</a>
<b>GetSignalStatus</b>	A4h			<a href="#">64</a>
<b>GetTime</b>	3Eh			<a href="#">66</a>
<b>GetTraceCount</b>	BBh			<a href="#">67</a>
<b>GetTraceStatus</b>	BAh			<a href="#">68</a>
<b>GetTraceValue</b>	28h			<a href="#">69</a>
<b>GetVersion</b>	8Fh			<a href="#">70</a>
<b>InitializePhase</b>	7Ah			<a href="#">71</a>
<b>NVRAM</b>	30h			<a href="#">72</a>
<b>NoOperation</b>	00h			<a href="#">74</a>
<b>ReadAnalog</b>	EFh			<a href="#">75</a>
<b>ReadBuffer</b>	C9h			<a href="#">76</a>
<b>ReadBuffer16</b>	CDh			<a href="#">77</a>
<b>Reset</b>	39h			<a href="#">78</a>
<b>ResetEventStatus</b>	34h			<a href="#">82</a>
<b>RestoreOperatingMode</b>	2Eh			<a href="#">83</a>
<b>SetAcceleration</b>	90h	<b>GetAcceleration</b>	4Ch	<a href="#">84</a>
<b>SetActualPosition</b>	4Dh	<b>GetActualPosition</b>	37h	<a href="#">86</a>
<b>SetActualPositionUnits</b>	BEh	<b>GetActualPositionUnits</b>	BFh	<a href="#">87</a>
<b>SetAnalogCalibration</b>	29h	<b>GetAnalogCalibration</b>	2Ah	<a href="#">88</a>
<b>SetBufferLength</b>	C2h	<b>GetBufferLength</b>	C3h	<a href="#">92</a>
<b>SetBufferReadIndex</b>	C6h	<b>GetBufferReadIndex</b>	C7h	<a href="#">94</a>
<b>SetBufferStart</b>	C0h	<b>GetBufferStart</b>	C1h	<a href="#">96</a>
<b>SetBufferWriteIndex</b>	C4h	<b>GetBufferWriteIndex</b>	C5h	<a href="#">98</a>
<b>SetCANMode</b>	12h	<b>GetCANMode</b>	15h	<a href="#">100</a>
<b>SetCommutationMode</b>	E2h	<b>GetCommutationMode</b>	E3h	<a href="#">102</a>
<b>SetCommutationParameter</b>	63h	<b>GetCommutationParameter</b>	64h	<a href="#">103</a>
<b>SetCurrent</b>	5Eh	<b>GetCurrent</b>	5Fh	<a href="#">106</a>
<b>SetCurrentControlMode</b>	43h	<b>GetCurrentControlMode</b>	44h	<a href="#">108</a>
<b>SetCurrentFoldback</b>	41h	<b>GetCurrentFoldback</b>	42h	<a href="#">109</a>
<b>SetCurrentLimit</b>	06h	<b>GetCurrentLimit</b>	07h	<a href="#">111</a>
<b>SetDeceleration</b>	91h	<b>GetDeceleration</b>	92h	<a href="#">113</a>
<b>SetDriveCommandMode</b>	7Eh	<b>GetDriveCommandMode</b>	7Fh	<a href="#">114</a>
<b>SetDriveFaultParameter</b>	62h	<b>GetDriveFaultParameter</b>	60h	<a href="#">116</a>
<b>SetDrivePWM</b>	23h	<b>GetDrivePWM</b>	24h	<a href="#">119</a>
<b>SetEncoderSource</b>	DAh	<b>GetEncoderSource</b>	DBh	<a href="#">121</a>
<b>SetEncoderToStepRatio</b>	DEh	<b>GetEncoderToStepRatio</b>	DFh	<a href="#">123</a>
<b>SetEventAction</b>	48h	<b>GetEventAction</b>	49h	<a href="#">125</a>
<b>SetFOC</b>	F6h	<b>GetFOC</b>	F7h	<a href="#">130</a>
<b>SetFaultOutMask</b>	FBh	<b>GetFaultOutMask</b>	FCh	<a href="#">128</a>

Instruction	Code	Instruction	Code	Page
<b>SetInterruptMask</b>	2Fh	<b>GetInterruptMask</b>	56h	<a href="#">132</a>
<b>SetLoop</b>	78h	<b>GetLoop</b>	79h	<a href="#">134</a>
<b>SetMotorCommand</b>	77h	<b>GetMotorCommand</b>	69h	<a href="#">138</a>
<b>SetMotorType</b>	02h	<b>GetMotorType</b>	03h	<a href="#">142</a>
<b>SetOperatingMode</b>	65h	<b>GetOperatingMode</b>	66h	<a href="#">144</a>
<b>SetOutputMode</b>	E0h	<b>GetOutputMode</b>	6Eh	<a href="#">146</a>
<b>SetPhaseCorrectionMode</b>	E8h	<b>GetPhaseCorrectionMode</b>	E9h	<a href="#">147</a>
<b>SetPhaseInitializeMode</b>	E4h	<b>GetPhaseInitializeMode</b>	E5h	<a href="#">148</a>
<b>SetPhaseParameter</b>	85h	<b>GetPhaseParameter</b>	86h	<a href="#">149</a>
<b>SetSampleTime</b>	3Bh	<b>GetSampleTime</b>	3Ch	<a href="#">151</a>
<b>SetSerialPortMode</b>	8Bh	<b>GetSerialPortMode</b>	8Ch	<a href="#">153</a>
<b>SetSignalSense</b>	A2h	<b>GetSignalSense</b>	A3h	<a href="#">155</a>
<b>SetTraceMode</b>	B0h	<b>GetTraceMode</b>	B1h	<a href="#">157</a>
<b>SetTracePeriod</b>	B8h	<b>GetTracePeriod</b>	B9h	<a href="#">158</a>
<b>SetTraceStart</b>	B2h	<b>GetTraceStart</b>	B3h	<a href="#">159</a>
<b>SetTraceStop</b>	B4h	<b>GetTraceStop</b>	B5h	<a href="#">162</a>
<b>SetTraceTriggerValue</b>	D6h	<b>GetTraceTriggerValue</b>	D7h	<a href="#">90</a>
<b>SetTraceVariable</b>	B6h	<b>GetTraceVariable</b>	B7h	<a href="#">164</a>
<b>SetVelocity</b>	11h	<b>GetVelocity</b>	4Bh	<a href="#">174</a>
<b>WriteBuffer</b>	C8h			<a href="#">176</a>

## 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
08h	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	88h	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	<b>GetActualPositionUnits</b>	<a href="#">87</a>			
C0h	<b>SetBufferStart</b>	<a href="#">96</a>			
C1h	<b>GetBufferStart</b>	<a href="#">96</a>			
C2h	<b>SetBufferLength</b>	<a href="#">92</a>			
C3h	<b>GetBufferLength</b>	<a href="#">92</a>			
C4h	<b>SetBufferWriteIndex</b>	<a href="#">98</a>			
C5h	<b>GetBufferWriteIndex</b>	<a href="#">98</a>			
C6h	<b>SetBufferReadIndex</b>	<a href="#">94</a>			
C7h	<b>GetBufferReadIndex</b>	<a href="#">94</a>			
C8h	<b>WriteBuffer</b>	<a href="#">176</a>			
C9h	<b>ReadBuffer</b>	<a href="#">76</a>			
CDh	<b>ReadBuffer16</b>	<a href="#">77</a>			
D6h	<b>SetTraceTriggerValue</b>	<a href="#">90</a>			
D7h	<b>GetTraceTriggerValue</b>	<a href="#">90</a>			
DAh	<b>SetEncoderSource</b>	<a href="#">121</a>			
DBh	<b>GetEncoderSource</b>	<a href="#">121</a>			
DEh	<b>SetEncoderToStepRatio</b>	<a href="#">123</a>			
DFh	<b>GetEncoderToStepRatio</b>	<a href="#">123</a>			
E0h	<b>SetOutputMode</b>	<a href="#">146</a>			
E2h	<b>SetCommutationMode</b>	<a href="#">102</a>			
E3h	<b>GetCommutationMode</b>	<a href="#">102</a>			
E4h	<b>SetPhaseInitializeMode</b>	<a href="#">148</a>			
E5h	<b>GetPhaseInitializeMode</b>	<a href="#">148</a>			
E8h	<b>SetPhaseCorrectionMode</b>	<a href="#">147</a>			
E9h	<b>GetPhaseCorrectionMode</b>	<a href="#">147</a>			
EFh	<b>ReadAnalog</b>	<a href="#">75</a>			
F5h	<b>AdjustActualPosition</b>	<a href="#">30</a>			
F6h	<b>SetFOC</b>	<a href="#">130</a>			
F7h	<b>GetFOC</b>	<a href="#">130</a>			
FBh	<b>SetFaultOutMask</b>	<a href="#">128</a>			
FC h	<b>GetFaultOutMask</b>	<a href="#">128</a>			

*This page intentionally left blank.*